Mathematics 18.337, Computer Science 6.338, SMA 5505

Applied Parallel Computing

Spring 2004

Lecturer: Alan Edelman[1]

MIT

[1]Department of Mathematics and Laboratory for Computer Science. Room 2-388, Massachusetts Institute of Technology, Cambridge, MA 02139, Email: edelman@math.mit.edu, http://math.mit.edu/~edelman

Contents

Lecture 1

Introduction

$Id: intro.tex,v 1.7 2004/02/16 21:31:14 drcheng Exp $

This book strives to study that elusive mix of theory and practice so important for understanding modern high performance computing. We try to cover it all, from the engineering aspects of computer science (parallel architectures, vendors, parallel languages, and tools) to the mathematical understanding of numerical algorithms, and also certain aspects from theoretical computer science.

Any understanding of the subject should begin with a quick introduction to the current scene in terms of machines, vendors, and performance trends. This sets a realistic expectation of maximal performance and an idea of the monetary price. Then one must quickly introduce at least one, though possibly two or three software approaches so that there is no waste in time in using a computer. Then one has the luxury of reviewing interesting algorithms, understanding the intricacies of how architecture influences performance and how this has changed historically, and also obtaining detailed knowledge of software techniques.

1.1 The machines

We begin our introduction with a look at machines available for high performance computation. We list four topics worthy of further exploration:

The **top500** list: We encourage readers to explore the data on http://www.top500.org. Important concepts are the types of machines currently on the top 500 and what benchmark is used. See Section 1.8 for a case study on how this might be done.

The "**grass-roots**" machines: We encourage readers to find out what machines that one can buy in the 10k–300k range. These are the sorts of machines that one might expect a small group or team might have available. Such machines can be built as a do-it-yourself project or they can be purchased as pre-built rack-mounted machines from a number of vendors. We created a web-site at MIT `http://beowulf.lcs.mit.edu/hpc.html` to track the machines available at MIT.

Special interesting architectures: At the time of writing, the Japanese Earth Simulator and the Virginia Tech Mac cluster are of special interest. It will not be long before they move into the next category:

Interesting historical architectures and trends: To get an idea of history, consider the popular 1990 Michael Crichton novel *Jurassic Park* and the 1993 movie. The novel has the dinosaur theme park controlled by a Cray vector supercomputer. The 1993 movie shows the CM-5 in the background of the control center and even mentions the Thinking Machines Computer, but you could easily miss it if you do not pay attention. The first decade of architecture is captured by the Crichton novel: vector supercomputers. We recommend the Cray supercomputers as an interesting examples. The second decade is characterized by **MPP**s: massively parallel supercomputers. We recommend the CM2 and CM5 for their historical interest. The third decade, that of the cluster, has seen a trend toward ease of availability, deployment and use. The first cluster dates back to 1994 consisting of 16 commodity computers connected by ethernet. Historically, the first beowulf is worthy of study.

When studying architectures, issues of interconnect, processor type and speed, and other nitty gritty issues arise. Sometimes low level software issues are also worthy of consideration when studying hardware.

1.2 The software

The three software models that we introduce quickly are

MPI—The message passing interface. This is the defacto standard for parallel computing though perhaps it is the lowest common denominator. We believe it was originally meant to be the high performance low-level language that libraries and compilers would reduce to. In fact, because it is portably and universally available it has become very much the language of parallel computing.

OpenMP—This less successful language (really language extension) has become popular for so-called shared memory implementations. Those are implementations where the user need not worry about the location of data.

Star-P—Our homegrown software that we hope will make parallel computing significantly easier. It is based on a server which currently uses a MATLAB front end, and either OCTAVE or MATLAB compute engines as well as library calls on the back end.

We also mention that there are any number of software libraries available for special purposes.

Mosix and **OpenMosix** are two technologies which allow for automatic load balancing between nodes of a Linux cluster. The difference between the two is that OpenMosix is released under the GNU Public License, while Mosix is proprietary software. Mosix and OpenMosix are installed as kernel patches (so it is the somewhat daunting task of patching, recompiling, and installing the patched Linux kernel). Once installed on a cluster, processes are automatically migrated from node to node to achieve load balancing. This allows for an exceptionally simple way to run embarrassingly parallel jobs, by simply backgrounding them with the ampersand (&) in the shell. For example:

```
#! /bin/sh
for i in 1 2 3 4 5 6 7 8 9 10
    do ./monte-carlo \$i &
done
wait
echo "All processes done."
```

Although all ten `monte-carlo` processes (each executing with a different command-line parameter) initially start on the same processor, the Mosix or OpenMosix system will automatically migrate the processes to different nodes of the cluster by capturing the entire state of the running program, sending it over the network, and restarting it from where it left off on a different node. Unfortunately, interprocess communication is difficult. It can be done through the standard Unix methods, for example, with sockets or via the file system.

The Condor Project, developed at the University of Wisconsin at Madison, is a batch queing system with the an interesting feature.

> [Condor can] effectively harness wasted CPU power from otherwise idle desktop workstations. For instance, Condor can be configured to only use desktop machines where

> the keyboard and mouse are idle. Should Condor detect that a machine is no longer available (such as a key press detected), in many circumstances Condor is able to transparently produce a checkpoint and migrate a job to a different machine which would otherwise be idle.[1]

Condor can run parallel computations across multiple Condor nodes using PVM or MPI, but (for now) using MPI requires dedicated nodes that cannot be used as desktop machines.

1.3 The Reality of High Performance Computing

There are probably a number of important issues regarding the reality of parallel computing that all too often is learned the hard way. You may not often find this in previously written textbooks.

Parallel computers may not give a speedup of p but you probably will be happy to be able to solve a larger problem in a reasonable amount of time. In other words if your computation can already be done on a single processor in a reasonable amount of time, you probably cannot do much better.

If you are deplying a machine, worry first about heat, power, space, and noise, not speed and performance.

1.4 Modern Algorithms

This book covers some of our favorite modern algorithms. One goal of high performance computing is very well defined, that is, to find faster solutions to larger and more complex problems. This area is a highly interdisciplinary area ranging from numerical analysis and algorithm design to programming languages, computer architectures, software, and real applications. The participants include engineers, computer scientists, applied mathematicians, and physicists, and many others.

We will concentrate on well-developed and more research oriented parallel algorithms in scientific and engineering that have “traditionally” relied on high performance computing, particularly parallel methods for differential equations, linear algebra, and simulations.

1.5 Compilers

[move to a software section?]

Most parallel languages are defined by adding parallel extensions to well-established sequential languages, such as C and Fortran. Such extensions allow user to specify various levels of parallelism in an application program, to define and manipulate parallel data structures, and to specify message passing among parallel computing units.

Compilation has become more important for parallel systems. The purpose of a compiler is not just to transform a parallel program in a high-level description into machine-level code. The role of compiler optimization is more important. We will always have discussions about: *Are we developing methods and algorithms for a parallel machine or are we designing parallel machines for algorithm and applications*? Compilers are meant to bridge the gap between algorithm design and machine architectures. Extension of compiler techniques to run time libraries will further reduce users' concern in parallel programming.

Software libraries are important tools in the use of computers. The purpose of libraries is to enhance the productivity by providing preprogrammed functions and procedures. Software

[1] http://www.cs.wisc.edu/condor/description.html

libraries provide even higher level support to programmers than high-level languages. Parallel scientific libraries embody expert knowledge of computer architectures, compilers, operating systems, numerical analysis, parallel data structures, and algorithms. They systematically choose a set of basic functions and parallel data structures, and provide highly optimized routines for these functions that carefully consider the issues of data allocation, data motion, load balancing, and numerical stability. Hence scientists and engineers can spend more time and be more focused on developing efficient computational methods for their problems. Another goal of scientific libraries is to make parallel programs portable from one parallel machine platform to another. Because of the lack, until very recently, of non-proprietary parallel programming standards, the development of portable parallel libraries has lagged far behind the need for them. There is good evidence now, however, that scientific libraries will be made more powerful in the future and will include more functions for applications to provide a better interface to real applications.

Due to the generality of scientific libraries, their functions may be more complex than needed for a particular application. Hence, they are less efficient than the best codes that take advantage of the special structure of the problem. So a programmer needs to learn how to use scientific libraries. A pragmatic suggestion is to use functions available in the scientific library to develop the first prototype and then to iteratively find the bottleneck in the program and improve the efficiency.

1.6 Scientific Algorithms

The multidisciplinary aspect of scientific computing is clearly visible in algorithms for scientific problems. A scientific algorithm can be either sequential or parallel. In general, algorithms in scientific software can be classified as graph algorithms, geometric algorithms, and numerical algorithms, and most scientific software calls on algorithms of all three types. Scientific software also makes use of advanced data structures and up-to-date user interfaces and graphics.

1.7 History, State-of-Art, and Perspective

The supercomputer industry started when Cray Research developed the vector-processor-powered Cray-1, in 1975. The massively parallel processing (MPP) industry emerged with the introduction of the CM-2 by Thinking Machines in 1985. Finally, 1994 brought the first Beowulf cluster, or "commidity supercomputing" (parallel computers built out of off-the-shelf components by independent vendors or do-it-your-selfers).

1.7.1 Things that are not traditional supercomputers

There have been a few successful distributed computing projects using volunteers across the internet. These projects represent the beginning attempts at "grid" computing.

- `Seti@home` uses thousands of Internet-connected computers to analyze radio telescope data in a search for extraterrestrial intelligence. The principal computation is FFT of the radio signals, and approximately 500,000 computers contribute toward a total computing effort of about 60 teraflops.

- `distributed.net` works on RSA Laboratories' RC5 cipher decryption contest and also searches for optimal Golumb rulers.

- `mersenne.org` searches for Mersenne primes, primes of the form $2^p - 1$.

- `chessbrain.net` is a distributed chess computer. On January 30, 2004, a total of 2,070 machines participated in a match against a Danish grandmaster. The game resulted in a draw by repetition.

Whereas the above examples of benevolent or harmless distributed computing, there are also other sorts of distributed computing which are frowned upon, either by the entertainment industry in the first example below, or universally in the latter two.

- Peer-to-peer file-sharing (the original Napster, followed by Gnutella, KaZaA, Freenet, and the like) can viewed as a large distributed supercomputer, although the resource being parallelized is storage rather than processing. KaZaA itself is notable because the client software contains an infamous hook (called Altnet) which allows a KaZaA corporate partner (Brilliant Digital Entertainment) to load and run arbitrary code on the client computer. Brilliant Digital has been quoted as saying they plan to use Altnet as "the next advancement in distributed bandwidth, storage and computing."[2] Altnet has so far only been used to distribute ads to KaZaA users.

- Distributed Denial of Service (DDoS) attacks harness thousands of machines (typically compromised through a security vulnerability) to attack an Internet host with so much traffic that it becomes too slow or otherwise unusable by legitimate users.

- Spam e-mailers have also increasingly turned to hundreds or thousands compromised machines to act as remailers, as a response to the practice of "blacklisting" machines thought to be spam mailers. The disturbing mode of attack is often a e-mail message containing a trojan attachment, which when executed opens a backdoor that the spammer can use to send more spam.

Although the class will not focus on these non-traditional supercomputers, the issues they have to deal with (communication between processors, load balancing, dealing with unreliable nodes) are similar to the issues that will be addressed in this class.

1.8 Analyzing the top500 List Using Excel

The following is a brief tutorial on analyzing data from the top500 website using Excel. Note that while the details provided by the top500 website will change from year to year, the ability for you to analyze this data should always be there. If the listed .xml file no longer exists, try searching the website yourself for the most current .xml file.

1.8.1 Importing the XML file

Open up Microsoft Office Excel 2003 on Windows XP. Click on the File menubar and then Open (or type `Ctrl-O`). As shown in Figure 1.1, for the the File name: text area, type in:

`http://www.top500.org/lists/2003/11/top500.200311.xml`

Click Open. A small window pops up asking how you would like to open the file. Leave the default as is and click OK (Figure 1.2).

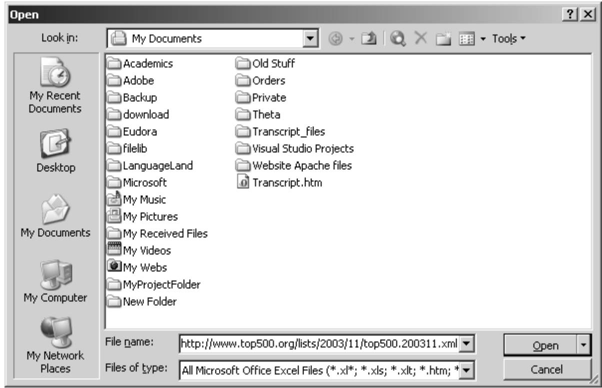


Figure 1.1: Open dialog

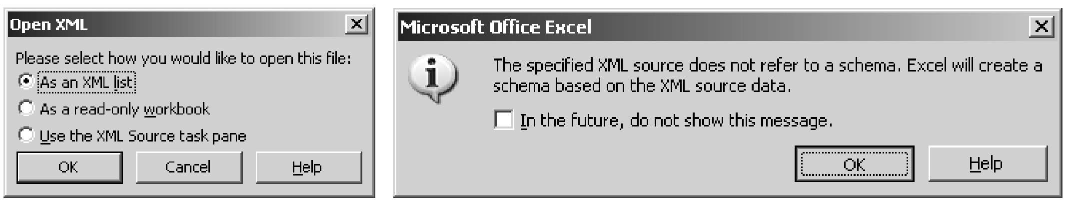
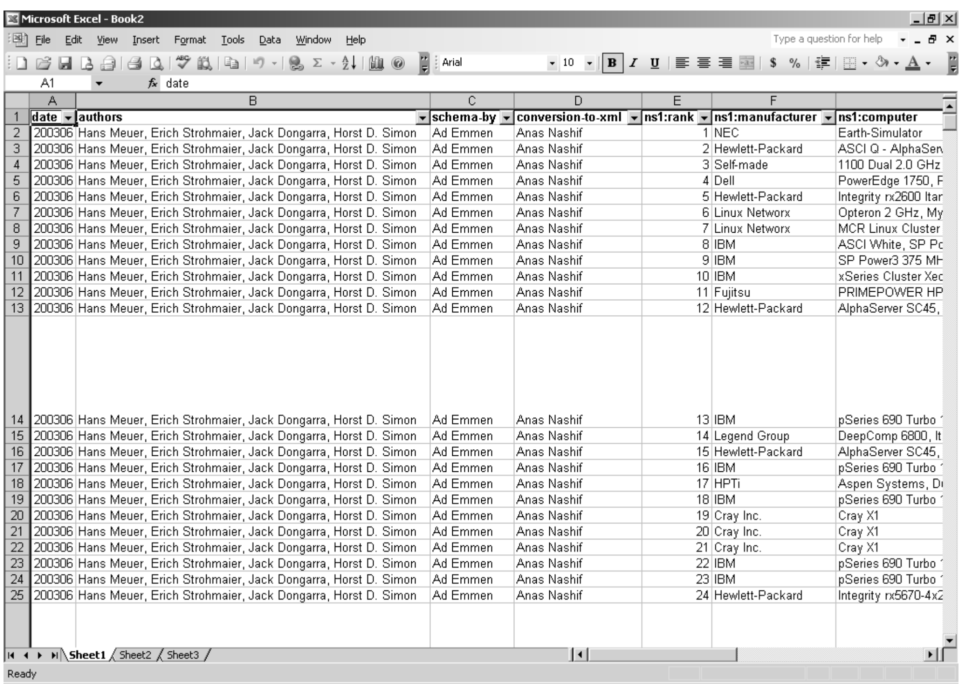
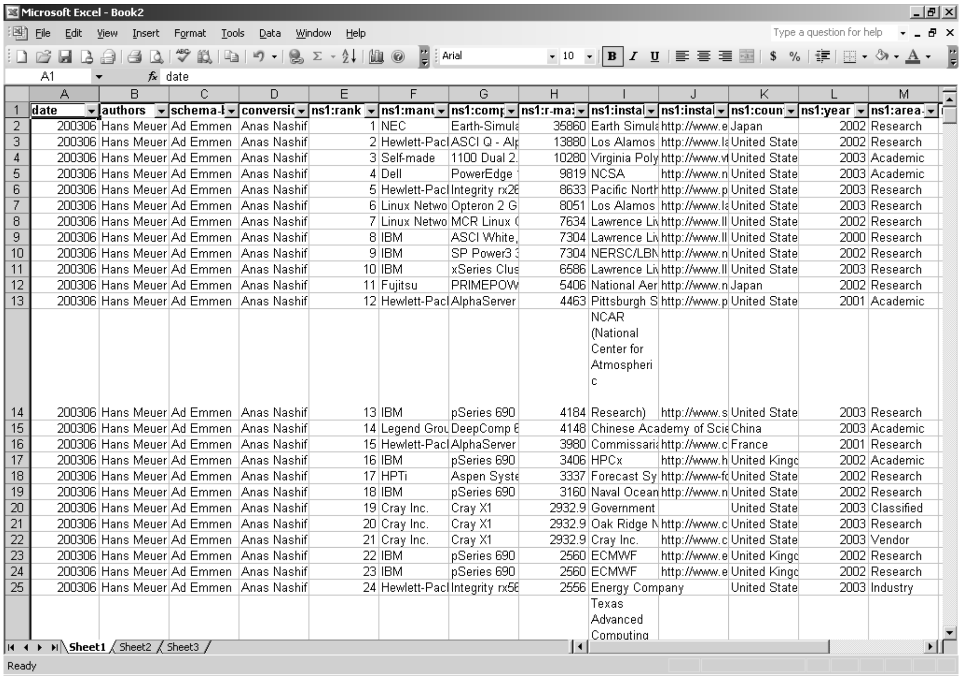


Figure 1.2: Opening the XML file



	date	authors	schema-by	conversion-to-xml	ns1:rank	ns1:manufacturer	ns1:computer
2	200306	Hans Meuer, Erich Strohmaier, Jack Dongarra, Horst D. Simon	Ad Emmen	Anas Nashif	1	NEC	Earth-Simulator
3	200306	Hans Meuer, Erich Strohmaier, Jack Dongarra, Horst D. Simon	Ad Emmen	Anas Nashif	2	Hewlett-Packard	ASCI Q - AlphaSen
4	200306	Hans Meuer, Erich Strohmaier, Jack Dongarra, Horst D. Simon	Ad Emmen	Anas Nashif	3	Self-made	1100 Dual 2.0 GHz
5	200306	Hans Meuer, Erich Strohmaier, Jack Dongarra, Horst D. Simon	Ad Emmen	Anas Nashif	4	Dell	PowerEdge 1750, F
6	200306	Hans Meuer, Erich Strohmaier, Jack Dongarra, Horst D. Simon	Ad Emmen	Anas Nashif	5	Hewlett-Packard	Integrity rx2600 Itar
7	200306	Hans Meuer, Erich Strohmaier, Jack Dongarra, Horst D. Simon	Ad Emmen	Anas Nashif	6	Linux Networx	Opteron 2 GHz, My
8	200306	Hans Meuer, Erich Strohmaier, Jack Dongarra, Horst D. Simon	Ad Emmen	Anas Nashif	7	Linux Networx	MCR Linux Cluster
9	200306	Hans Meuer, Erich Strohmaier, Jack Dongarra, Horst D. Simon	Ad Emmen	Anas Nashif	8	IBM	ASCI White, SP Pc
10	200306	Hans Meuer, Erich Strohmaier, Jack Dongarra, Horst D. Simon	Ad Emmen	Anas Nashif	9	IBM	SP Power3 375 MH
11	200306	Hans Meuer, Erich Strohmaier, Jack Dongarra, Horst D. Simon	Ad Emmen	Anas Nashif	10	IBM	xSeries Cluster Xec
12	200306	Hans Meuer, Erich Strohmaier, Jack Dongarra, Horst D. Simon	Ad Emmen	Anas Nashif	11	Fujitsu	PRIMEPOWER HP
13	200306	Hans Meuer, Erich Strohmaier, Jack Dongarra, Horst D. Simon	Ad Emmen	Anas Nashif	12	Hewlett-Packard	AlphaServer SC45,
14	200306	Hans Meuer, Erich Strohmaier, Jack Dongarra, Horst D. Simon	Ad Emmen	Anas Nashif	13	IBM	pSeries 690 Turbo
15	200306	Hans Meuer, Erich Strohmaier, Jack Dongarra, Horst D. Simon	Ad Emmen	Anas Nashif	14	Legend Group	DeepComp 6800, It
16	200306	Hans Meuer, Erich Strohmaier, Jack Dongarra, Horst D. Simon	Ad Emmen	Anas Nashif	15	Hewlett-Packard	AlphaServer SC45,
17	200306	Hans Meuer, Erich Strohmaier, Jack Dongarra, Horst D. Simon	Ad Emmen	Anas Nashif	16	IBM	pSeries 690 Turbo
18	200306	Hans Meuer, Erich Strohmaier, Jack Dongarra, Horst D. Simon	Ad Emmen	Anas Nashif	17	HPTi	Aspen Systems, Di
19	200306	Hans Meuer, Erich Strohmaier, Jack Dongarra, Horst D. Simon	Ad Emmen	Anas Nashif	18	IBM	pSeries 690 Turbo
20	200306	Hans Meuer, Erich Strohmaier, Jack Dongarra, Horst D. Simon	Ad Emmen	Anas Nashif	19	Cray Inc.	Cray X1
21	200306	Hans Meuer, Erich Strohmaier, Jack Dongarra, Horst D. Simon	Ad Emmen	Anas Nashif	20	Cray Inc.	Cray X1
22	200306	Hans Meuer, Erich Strohmaier, Jack Dongarra, Horst D. Simon	Ad Emmen	Anas Nashif	21	Cray Inc.	Cray X1
23	200306	Hans Meuer, Erich Strohmaier, Jack Dongarra, Horst D. Simon	Ad Emmen	Anas Nashif	22	IBM	pSeries 690 Turbo
24	200306	Hans Meuer, Erich Strohmaier, Jack Dongarra, Horst D. Simon	Ad Emmen	Anas Nashif	23	IBM	pSeries 690 Turbo
25	200306	Hans Meuer, Erich Strohmaier, Jack Dongarra, Horst D. Simon	Ad Emmen	Anas Nashif	24	Hewlett-Packard	Integrity rx5670-4x2

Figure 1.3: Loaded data



	A	B	C	D	E	F	G	H	I	J	K	L	M
1	date	authors	schema-l	conversic	ns1:rank	ns1:manu	ns1:comp	ns1:r-ma:	ns1:instal	ns1:instal	ns1:coun	ns1:year	ns1:area-
2	200306	Hans Meuer	Ad Emmen	Anas Nashif	1	NEC	Earth-Simula	35860	Earth Simula	http://www.e	Japan	2002	Research
3	200306	Hans Meuer	Ad Emmen	Anas Nashif	2	Hewlett-Pacl	ASCI Q - Alp	13880	Los Alamos	http://www.la	United State	2002	Research
4	200306	Hans Meuer	Ad Emmen	Anas Nashif	3	Self-made	1100 Dual 2.	10280	Virginia Poly	http://www.vt	United State	2003	Academic
5	200306	Hans Meuer	Ad Emmen	Anas Nashif	4	Dell	PowerEdge 1	9819	NCSA	http://www.n	United State	2003	Academic
6	200306	Hans Meuer	Ad Emmen	Anas Nashif	5	Hewlett-Pacl	Integrity rx26	8633	Pacific North	http://www.p	United State	2003	Research
7	200306	Hans Meuer	Ad Emmen	Anas Nashif	6	Linux Netwo	Opteron 2 G	8051	Los Alamos	http://www.la	United State	2003	Research
8	200306	Hans Meuer	Ad Emmen	Anas Nashif	7	Linux Netwo	MCR Linux (	7634	Lawrence Liv	http://www.ll	United State	2002	Research
9	200306	Hans Meuer	Ad Emmen	Anas Nashif	8	IBM	ASCI White,	7304	Lawrence Liv	http://www.ll	United State	2000	Research
10	200306	Hans Meuer	Ad Emmen	Anas Nashif	9	IBM	SP Power3 3	7304	NERSC/LBN	http://www.n	United State	2002	Research
11	200306	Hans Meuer	Ad Emmen	Anas Nashif	10	IBM	xSeries Clus	6586	Lawrence Liv	http://www.ll	United State	2003	Research
12	200306	Hans Meuer	Ad Emmen	Anas Nashif	11	Fujitsu	PRIMEPOW	5406	National Aer	http://www.n	Japan	2002	Research
13	200306	Hans Meuer	Ad Emmen	Anas Nashif	12	Hewlett-Pacl	AlphaServer	4463	Pittsburgh S	http://www.p	United State	2001	Academic
14	200306	Hans Meuer	Ad Emmen	Anas Nashif	13	IBM	pSeries 690	4184	NCAR (National Center for Atmospheri c Research)	http://www.s	United State	2003	Research
15	200306	Hans Meuer	Ad Emmen	Anas Nashif	14	Legend Grou	DeepComp 6	4148	Chinese Academy of Scie		China	2003	Academic
16	200306	Hans Meuer	Ad Emmen	Anas Nashif	15	Hewlett-Pacl	AlphaServer	3980	Commissaria	http://www.c	France	2001	Research
17	200306	Hans Meuer	Ad Emmen	Anas Nashif	16	IBM	pSeries 690	3406	HPCx	http://www.h	United Kingc	2002	Academic
18	200306	Hans Meuer	Ad Emmen	Anas Nashif	17	HPTi	Aspen Syste	3337	Forecast Sy	http://www-fc	United State	2002	Research
19	200306	Hans Meuer	Ad Emmen	Anas Nashif	18	IBM	pSeries 690	3160	Naval Ocean	http://www.n	United State	2002	Research
20	200306	Hans Meuer	Ad Emmen	Anas Nashif	19	Cray Inc.	Cray X1	2932.9	Government		United State	2003	Classified
21	200306	Hans Meuer	Ad Emmen	Anas Nashif	20	Cray Inc.	Cray X1	2932.9	Oak Ridge N	http://www.c	United State	2003	Research
22	200306	Hans Meuer	Ad Emmen	Anas Nashif	21	Cray Inc.	Cray X1	2932.9	Cray Inc.	http://www.c	United State	2003	Vendor
23	200306	Hans Meuer	Ad Emmen	Anas Nashif	22	IBM	pSeries 690	2560	ECMWF	http://www.e	United Kingc	2002	Research
24	200306	Hans Meuer	Ad Emmen	Anas Nashif	23	IBM	pSeries 690	2560	ECMWF	http://www.e	United Kingc	2002	Research
25	200306	Hans Meuer	Ad Emmen	Anas Nashif	24	Hewlett-Pacl	Integrity rx56	2556	Energy Company		United State	2003	Industry
									Texas Advanced Computing				

Figure 1.4: Filtering available for each column

1.8.2 Filtering

Click OK again. Something as in Figure 1.3 should show up.

If you don't see the bolded column titles with the arrows next to them, go to the menubar and select Data → Filter → Autofilter (make sure it is checked). Type `Ctrl-A` to highlight all the entries. Go to the menubar and select Format → Column → Width. You can put in any width you'd like, but 10 would work.

You can now filter the data in numerous ways. For example, if you want to find out all SGI installations in the US, you can click on the arrow in column K (country) and select United States. Then, click on the arrow in column F (manufacturer) and select SGI. The entire data set is now sorted to those select machines (Figure 1.5).

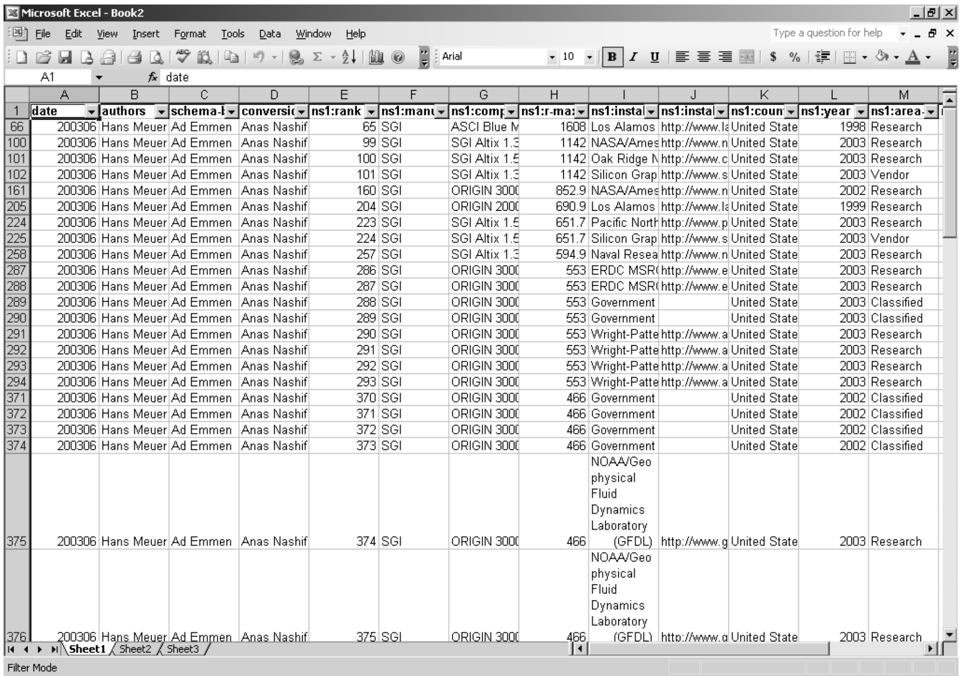
If you continue to click on other arrows, the selection will become more and more filtered. If you would like to start over, go to the menubar and select Data → Filter → Show All. Assuming that you've started over with all the data, we can try to see all machines in both Singapore and Malaysia. Click on the arrow in column K (country) again and select (Custom...). You should see something as in Figure 1.6.

On the top right side, pull down the arrow and select Singapore. On the lower left, pull down the arrow and select equals. On the lower right, pull down the arrow and select Malaysia (Figure 1.6.

Now click OK. You should see a blank filtered list because there are no machines that are in both Singapore and Malaysia (Figure 1.7).

If you want to find out all machines in Singapore or India, you have to start off with all the data again. You perform the same thing as before except that in the Custom AutoFilter screen, you should click on the Or toggle button. You should also type in India instead of Malaysia (Figure

[2]Brilliant Digital Entertainment's Annual Report (Form 10KSB), Filed with SEC April 1, 2002.



	date	authors	schema-l	conversic	ns1:rank	ns1:manu	ns1:comp	ns1:r-ma:	ns1:instal	ns1:instal	ns1:coun	ns1:year	ns1:area-
66	200306	Hans Meuer	Ad Emmen	Anas Nashif	65	SGI	ASCI Blue M	1608	Los Alamos	http://www.la	United State	1998	Research
100	200306	Hans Meuer	Ad Emmen	Anas Nashif	99	SGI	SGI Altix 1.3	1142	NASA/Ames	http://www.n	United State	2003	Research
101	200306	Hans Meuer	Ad Emmen	Anas Nashif	100	SGI	SGI Altix 1.5	1142	Oak Ridge N	http://www.c	United State	2003	Research
102	200306	Hans Meuer	Ad Emmen	Anas Nashif	101	SGI	SGI Altix 1.3	1142	Silicon Grap	http://www.s	United State	2003	Vendor
161	200306	Hans Meuer	Ad Emmen	Anas Nashif	160	SGI	ORIGIN 3000	852.9	NASA/Ames	http://www.n	United State	2002	Research
205	200306	Hans Meuer	Ad Emmen	Anas Nashif	204	SGI	ORIGIN 2000	690.9	Los Alamos	http://www.la	United State	1999	Research
224	200306	Hans Meuer	Ad Emmen	Anas Nashif	223	SGI	SGI Altix 1.5	651.7	Pacific North	http://www.p	United State	2003	Research
225	200306	Hans Meuer	Ad Emmen	Anas Nashif	224	SGI	SGI Altix 1.5	651.7	Silicon Grap	http://www.s	United State	2003	Vendor
258	200306	Hans Meuer	Ad Emmen	Anas Nashif	257	SGI	SGI Altix 1.3	594.9	Naval Resea	http://www.n	United State	2003	Research
287	200306	Hans Meuer	Ad Emmen	Anas Nashif	286	SGI	ORIGIN 3000	553	ERDC MSR(	http://www.e	United State	2003	Research
288	200306	Hans Meuer	Ad Emmen	Anas Nashif	287	SGI	ORIGIN 3000	553	ERDC MSR(	http://www.e	United State	2003	Research
289	200306	Hans Meuer	Ad Emmen	Anas Nashif	288	SGI	ORIGIN 3000	553	Government		United State	2003	Classified
290	200306	Hans Meuer	Ad Emmen	Anas Nashif	289	SGI	ORIGIN 3000	553	Government		United State	2003	Classified
291	200306	Hans Meuer	Ad Emmen	Anas Nashif	290	SGI	ORIGIN 3000	553	Wright-Patte	http://www.a	United State	2003	Research
292	200306	Hans Meuer	Ad Emmen	Anas Nashif	291	SGI	ORIGIN 3000	553	Wright-Patte	http://www.a	United State	2003	Research
293	200306	Hans Meuer	Ad Emmen	Anas Nashif	292	SGI	ORIGIN 3000	553	Wright-Patte	http://www.a	United State	2003	Research
294	200306	Hans Meuer	Ad Emmen	Anas Nashif	293	SGI	ORIGIN 3000	553	Wright-Patte	http://www.a	United State	2003	Research
371	200306	Hans Meuer	Ad Emmen	Anas Nashif	370	SGI	ORIGIN 3000	466	Government		United State	2002	Classified
372	200306	Hans Meuer	Ad Emmen	Anas Nashif	371	SGI	ORIGIN 3000	466	Government		United State	2002	Classified
373	200306	Hans Meuer	Ad Emmen	Anas Nashif	372	SGI	ORIGIN 3000	466	Government		United State	2002	Classified
374	200306	Hans Meuer	Ad Emmen	Anas Nashif	373	SGI	ORIGIN 3000	466	Government		United State	2002	Classified
375	200306	Hans Meuer	Ad Emmen	Anas Nashif	374	SGI	ORIGIN 3000	466	NOAA/Geo physical Fluid Dynamics Laboratory (GFDL)	http://www.g	United State	2003	Research
376	200306	Hans Meuer	Ad Emmen	Anas Nashif	375	SGI	ORIGIN 3000	466	NOAA/Geo physical Fluid Dynamics Laboratory (GFDL)	http://www.g	United State	2003	Research

Figure 1.5: All SGI installations

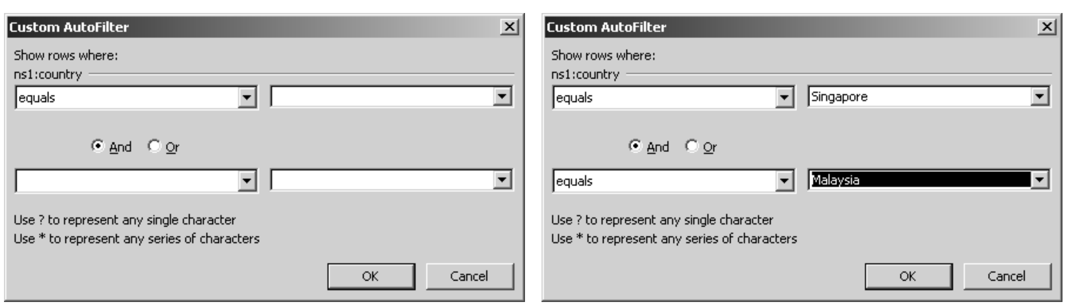


Figure 1.6: Custom filter

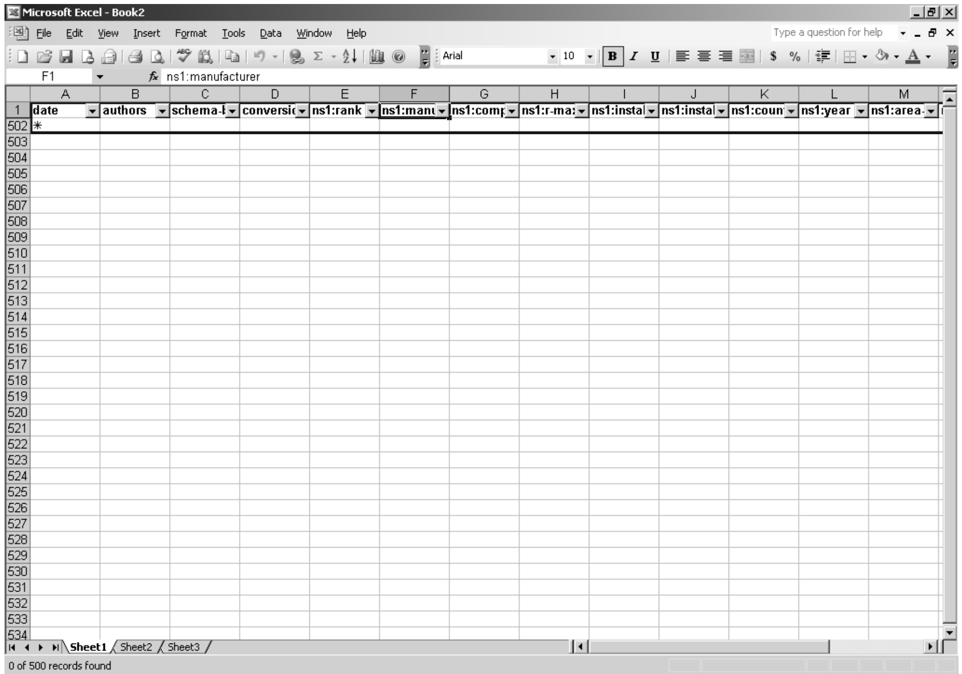


Figure 1.7: Machines in Singapore and Malaysia (none)

1.8).

Click OK and you should see something as in Figure 1.9.

1.8.3 Pivot Tables

Now let's try experimenting with a new feature called pivot tables. First, go to the menubar and click on Data → Filter → Show All to bring back all the entries. Type `Ctrl-A` to select all the entries. Go to the menubar and click on Data → PivotTable and PivotChart Report... Figure 1.10 shows the 3 wizard screens that you'll see. You can click Finish in the first screen unless you want to specify something different.

You get something as in Figure 1.11.

Let's try to find out the number of machines in each country. Scroll down the PivotTable Field List, find ns1:country, and use your mouse to drag it to where it says "Drop Row Fields Here".

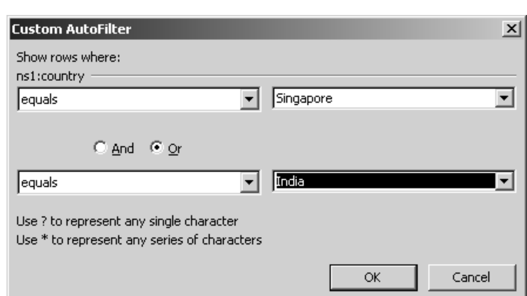


Figure 1.8: Example filter: Singapore or India

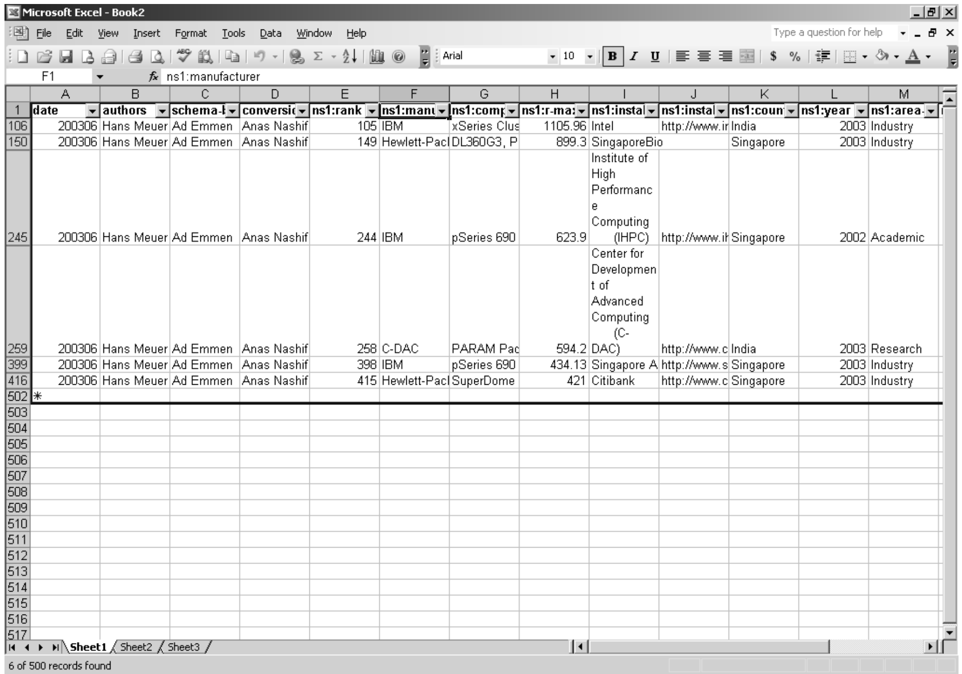


Figure 1.9: Machines in Singapore or India

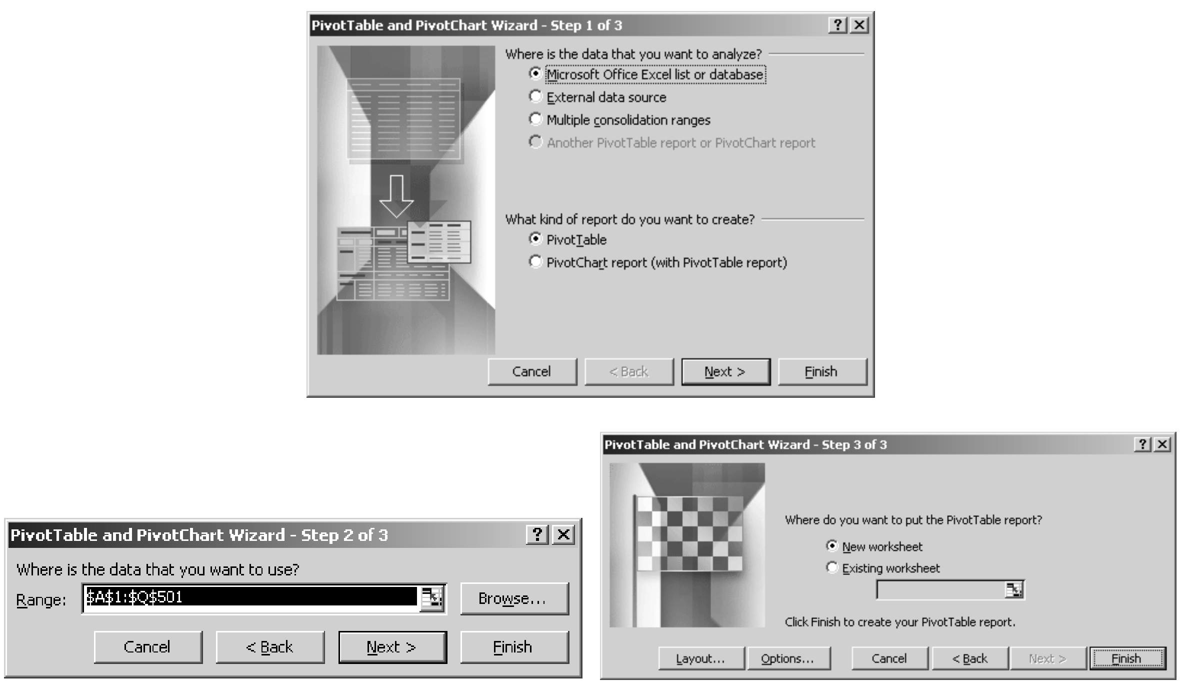


Figure 1.10: Pivot Table wizard

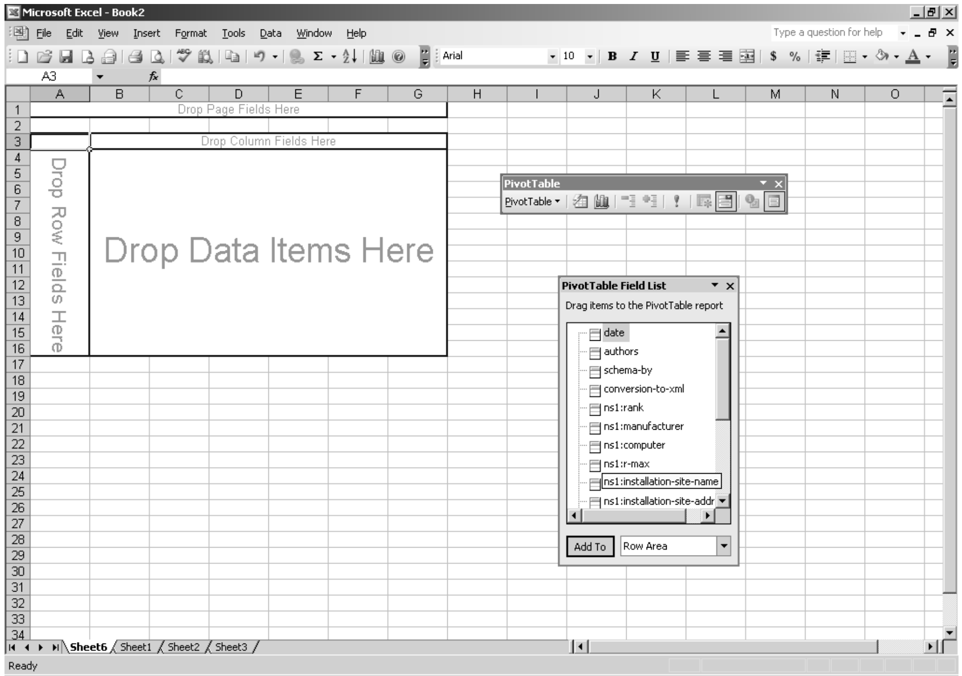


Figure 1.11: Empty Pivot Table

Now scroll down the PivotTable Field List again, find ns1:computer, and use your mouse to drag it to where it says "Drop Data Items Here" (Figure 1.12).

Notice that at the top left of the table it says "Count of ns1:computer". This means that it is counting up the number of machines for each individual country. Now let's fine the highest rank of each machine. Click on the gray ns1:country column heading and drag it back to the PivotTable Field List. Now from the PivotTable Field List, drag ns1:computer to the column where the country list was before. Click on the gray Count of ns1:computer column heading and drag it back to the PivotTable Field List. From the same list, drag ns1:rank to where the ns1:computer information was before (it says "Drop Data Items Here"). The upper left column heading should now be: Sum of ns1: rank. Double click on it and the resulting pop up is shown in Figure 1.13.

To find the highest rank of each machine, we need the data to be sorted by Min. Click on Min and click OK (Figure 1.14).

The data now shows the highest rank for each machine. Let's find the number of machines worldwide for each vendor. Following the same procedures as before, we replace the Row Fields and Data Items with ns1:manufacturer and ns1:computer, respectively. We see that the upper left column heading says "Count of ns1:computer", so we now it is finding the sum of the machines for each vendor. The screen should look like Figure 1.15.

Now let's find the minimum rank for each vendor. By now we should all be experts at this! The tricky part here is that we actually want the Max of the ranks. The screen you should get is shown in Figure 1.16.

Note that it is also possible to create 3D pivot tables. Start out with a blank slate by dragging everything back to the Pivot Table Field List. Then, from that list, drag ns1:country to where it says "Drop Row Fields Here". Drag ns1:manufacturer to where it says "Drop Column Fields Here". Now click on the arrow under the ns1:country title, click on Show All (which releases on the checks), Australia, Austria, Belarus, and Belgium. Click on the arrow under the ns1:manufacturer

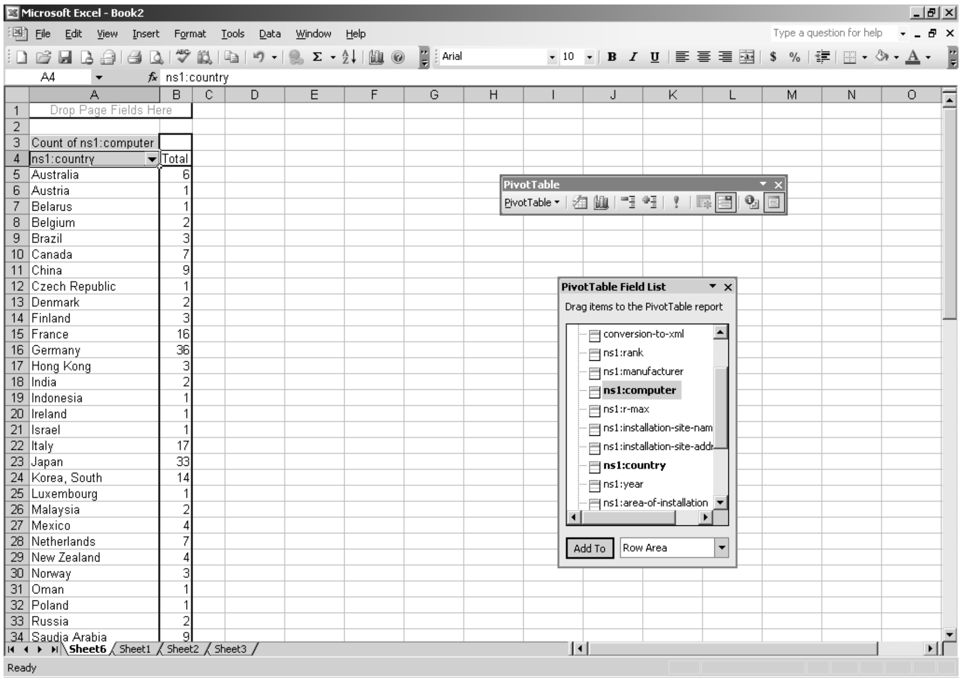


Figure 1.12: Number of machines in each country

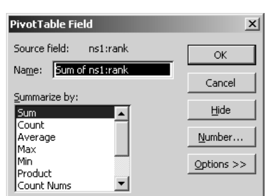


Figure 1.13: Possible functions on the rank

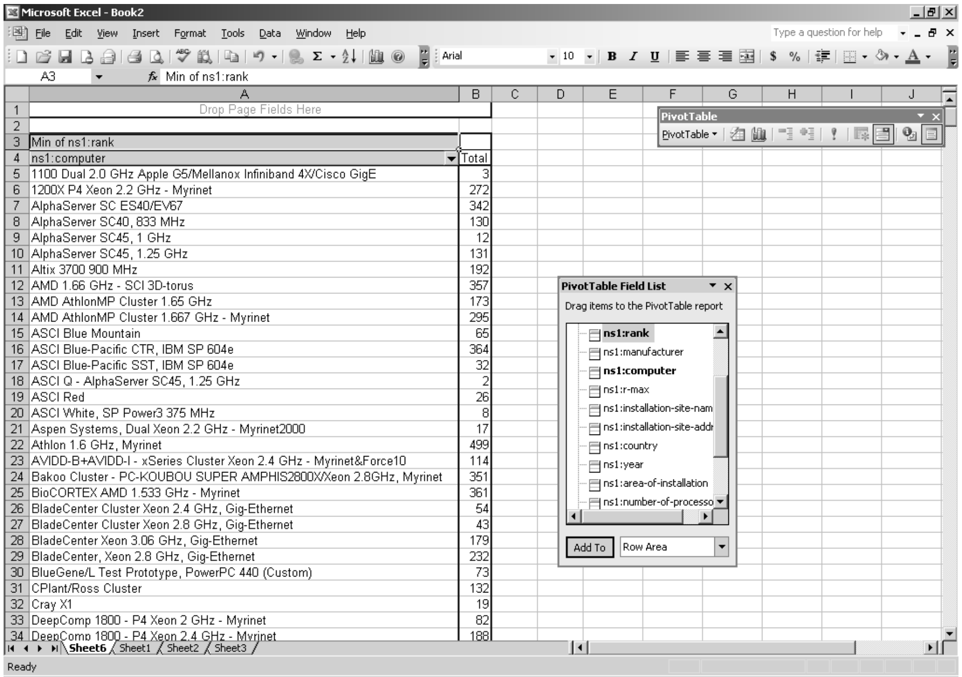


Figure 1.14: Minimum ranked machine from each country

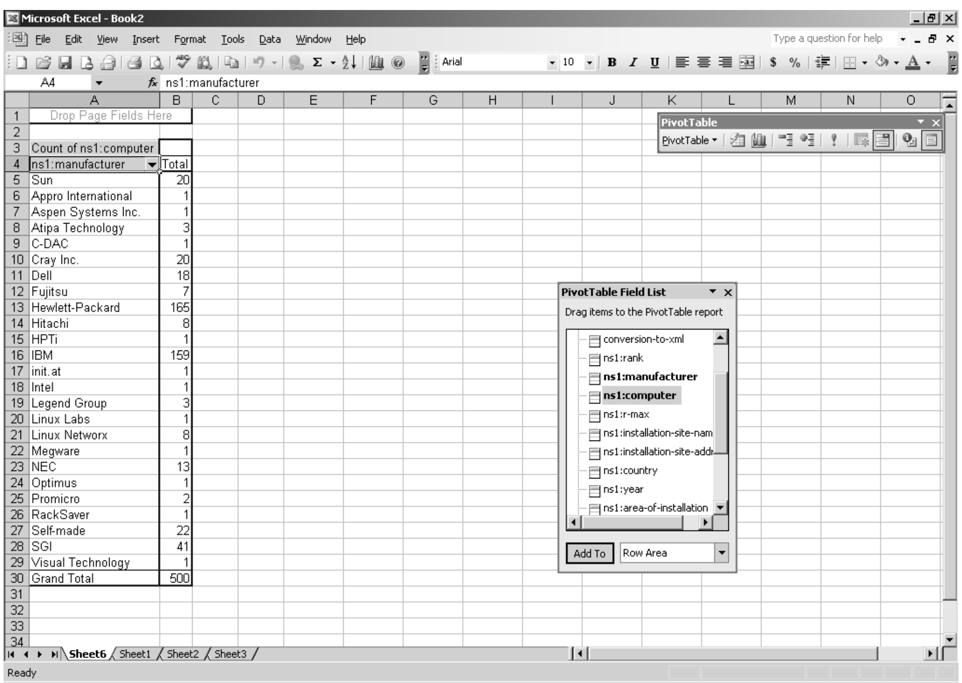


Figure 1.15: Count of machines for each vendor

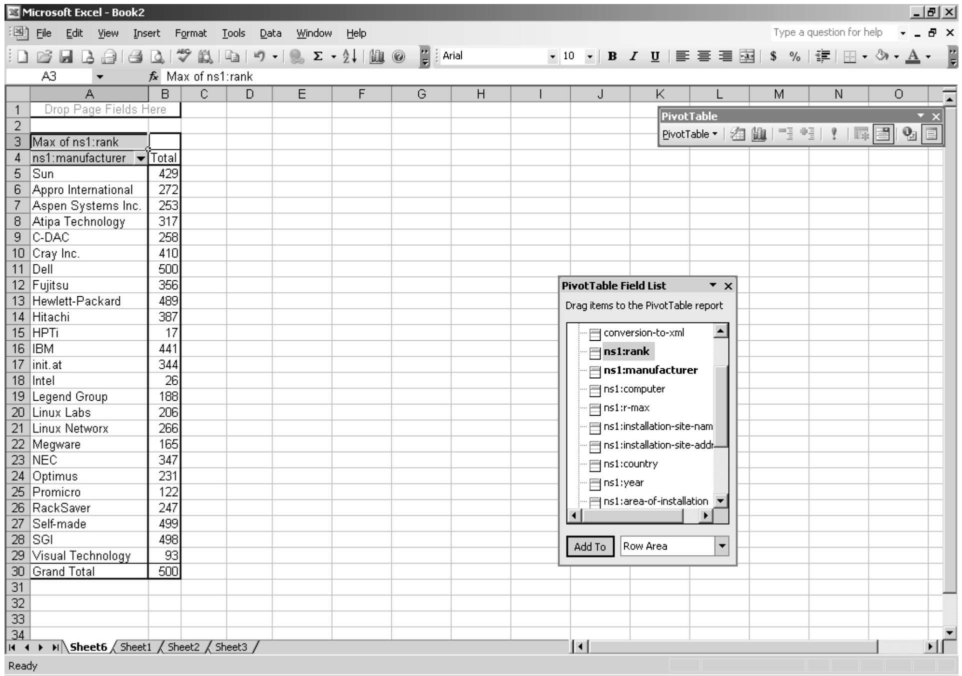


Figure 1.16: Minimum ranked machine from each vendor

title, click on Show All, Dell, Hewlett-Packward, and IBM. Drag ns1:rank from the Pivot Table Field List to where it says "Drop Data Items Here". Double-click on the Sum of ns1:rank and select Count instead of Sum. Finally, drag ns1:year to where it says "Drop Page Fields Here". You have successfully completed a 3D pivot table that looks like Figure 1.17.

As you can imagine, there are infinite possibilities for auto-filter and pivot table combinations. Have fun with it!

1.9 Parallel Computing: An Example

Here is an example of a problem that one can easily imagine being performed in parallel:

> **A rudimentary parallel problem:** Compute
>
> $$x_1 + x_2 + \ldots + x_P,$$
>
> where x_i is a floating point number and for purposes of exposition, let us assume P is a power of 2.

Obviously, the sum can be computed in $\log P$ (base 2 is understood) steps, simply by adding neighboring pairs recursively. (The algorithm has also been called *pairwise summation* and *cascade summation*). The data flow in this computation can be thought of as a binary tree.

(Illustrate on tree.)

Nodes represent values, either input or the result of a computation. Edges communicate values from their definition to their uses.

This is an example of what is often known as a *reduce* operation. We can replace the addition operation with any associative operator to generalize. (Actually, floating point addition is not

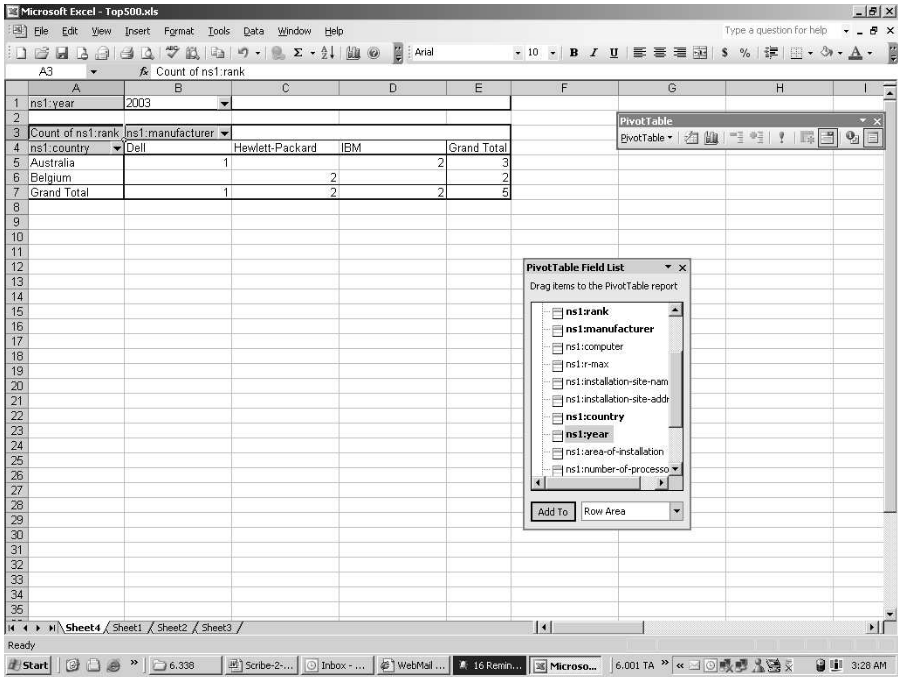


Figure 1.17: 3D pivot table

associative, leading to interesting numerical questions about the best order in which to add numbers. This is studied by Higham [49, See p. 788].)

There are so many questions though. How would you write a program to do this on P processors? Is it likely that you would want to have such a program on P processors? How would the data (the x_i) get to these processors in the first place? Would they be stored there or result from some other computation? It is far more likely that you would have $10000P$ numbers on P processors to add.

A correct parallel program is often not an efficient parallel program. A flaw in a parallel program that causes it to get the right answer *slowly* is known as a performance bug. Many beginners will write a working parallel program, obtain poor performance, and prematurely conclude that either parallelism is a bad idea, or that it is the machine that they are using which is slow.

What are the sources of performance bugs? We illustrate some of them with this little, admittedly contrived example. For this example, imagine four processors, numbered zero through three, each with its own private memory, and able to send and receive message to/from the others. As a simple approximation, assume that the time consumed by a message of size n words is $A + Bn$.

Three Bad Parallel Algorithms for Computing $1 + 2 + 3 + \cdots + 10^6$ on Four Processors and generalization

1. **Load imbalance**. One processor has too much to do, and the others sit around waiting for it. For our problem, we could eliminate all the communication by having processor zero do all the work; but we won't see any parallel speedup with this method!

2. **Excessive communication**.

   Processor zero has all the data which is divided into four equally sized parts each with a quarter of a million numbers. It ships three of the four parts to each of the other processors for proper load balancing. The results are added up on each processor, and then processor 0 adds the final four numbers together.

True, there is a tiny bit of load imbalance here, since processor zero does those few last additions. But that is nothing compared with the cost it incurs in shipping out the data to the other processors. In order to get that data out onto the network, it incurs a large cost that does not drop with the addition of more processors. (In fact, since the number of messages it sends grows like the number of processors, the time spent in the initial communication will actually increase.)

3. A **sequential bottleneck**. Let's assume the data are initially spread out among the processors; processor zero has the numbers 1 through 250,000, etc. Assume that the owner of i will add it to the running total. So there will be no load imbalance. But assume, further, that we are constrained to add the numbers in their original order! (Sounds silly when adding, but other algorithms require exactly such a constraint.) Thus, processor one may not begin its work until it receives the sum $0 + 1 + \cdots + 250,000$ from processor zero!

   We are thus requiring a sequential computation: our binary summation tree is maximally unbalanced, and has height 10^6. It is always useful to know the critical path—the length of the longest path in the dataflow graph—of the computation being parallelized. If it is excessive, change the algorithm!

   Some problems look sequential such as Fibonacci: $F_{k+1} = F_k + F_{k-1}$, but looks can be deceiving. Parallel prefix will be introduced later, which can be used to parallelize the Fibonacci computation.

1.10 Exercises

1. Compute the sum of 1 through $1,000,000$ using HPF. This amounts to a "hello world" program on whichever machine you are using. We are not currently aware of any free distributions of HPF for workstations, so your instructor will have to suggest a computer to use.

2. Download MPI to your machine and compute the sum of 1 through $1,000,000$ using C or Fortran with MPI. A number of MPI implementations may be found at `http://www.mcs.anl.gov/Projects/mpi/implementations.html` The easy to follow ten step quick start on `http://www.mcs.anl.gov/mpi/mpiinstall/node1.html#Node1` worked very easily on the MIT mathematics department's SUN network.

3. In HPF, generate $1,000,000$ real random numbers and sort them. (Use `RANDOM_NUMBER` and `GRADE_UP`.

4. (Extra Credit) Do the same in C or Fortran with MPI.

5. Set up the excessive communication situation described as the second bad parallel algorithm. Place the numbers 1 through one million in a vector on one processor. Using four processors see how quickly you can get the sum of the numbers 1 through one million.

Lecture 2

MPI, OpenMP, MATLAB*P

A parallel language must provide mechanisms for implementing parallel algorithms, i.e., to specify various levels of parallelism and define parallel data structures for distributing and sharing information among processors.

Most current parallel languages add parallel constructs for standard sequential languages. Different parallel languages provide different basic constructs. The choice largely depends on the parallel computing model the language means to support.

There are at least three basic parallel computation models other than vector and pipeline model: *data parallel*, *message passing*, and *shared memory task parallel*.

2.1 Programming style

Data parallel vs. message passing. Explicit communication can be somewhat hidden if one wants to program in a data parallel style; it's something like SIMD: you specify a single action to execute on all processors. Example: if `A`, `B` and `C` are matrices, you can write `C=A+B` and let the compiler do the hard work of accessing remote memory locations, partition the matrix among the processors, etc.

By contrast, explicit message passing gives the programmer careful control over the communication, but programming requires a lot of knowledge and is much more difficult (as you probably understood from the previous section).

There are uncountable other alternatives, still academic at this point (in the sense that no commercial machine comes with them bundled). A great deal of research has been conducted on multithreading; the idea is that the programmer expresses the computation graph in some appropriate language and the machine executes the graph, and potentially independent nodes of the graph can be executed in parallel. Example in Cilk (multithreaded C, developed at MIT by Charles Leiserson and his students):

```
thread int fib(int n)
{
    if (n<2)
        return n;
    else {
        cont int x, y;
        x = spawn fib (n-2);
        y = spawn fib (n-1);
```

```
            sync;
            return x+y;
        }
    }
```

Actually Cilk is a little bit different right now, but this is the way the program will look like when you read these notes. The whole point is that the two computations of `fib(n-1)` and `fib(n-2)` can be executed in parallel. As you might have expected, there are dozens of multithreaded languages (functional, imperative, declarative) and implementation techniques; in some implementations the thread size is a single-instruction long, and special processors execute this kind of programs. Ask Arvind at LCS for details.

Writing a good HPF compiler is difficult and not every manufacturer provides one; actually for some time TMC machines were the only machines available with it. The first HPF compiler for the Intel Paragon dates December 1994.

Why SIMD is not necessarily the right model for data parallel programming. Consider the following Fortran fragment, where `x` and `y` are vectors:

```
        where (x > 0)
             y = x + 2
        elsewhere
             y = -x + 5
endwhere
```

A SIMD machine might execute both cases, and discard one of the results; it does twice the needed work (see why? there is a single flow of instructions). This is how the CM-2 operated.

On the other hand, an HPF compiler for a SIMD machine can take advantage of the fact that there will be many more elements of `x` than processors. It can execute the `where` branch until there are no positive elements of `x` that haven't been seen, then it can execute the `elsewhere` branch until all other elements of `x` are covered. It can do this provided the machine has the ability to generate independent memory addresses on every processor.

Moral: even if the programming model is that there is one processor per data element, the programmer (and the compiler writer) must be aware that it's not true.

2.2 Message Passing

In the SISD, SIMD, and MIMD computer taxonomy, SISD machines are conventional uniprocessors, SIMD are single instruction stream machines that operate in parallel on ensembles of data, like arrays or vectors, and MIMD machines have multiple active threads of control (processes) that can operate on multiple data in parallel. How do these threads share data and how do they synchronize? For example, suppose two processes have a producer/consumer relationship. The producer generates a sequence of data items that are passed to the consumer, which processes them further. How does the producer deliver the data to the consumer?

If they share a common memory, then they agree on a location to be used for the transfer. In addition, they have a mechanism that allows the consumer to know when that location is full, *i.e.* it has a valid datum placed there for it by the producer, and a mechanism to read that location and change its state to empty. A full/empty bit is often associated with the location for this purpose. The hardware feature that is often used to do this is a "test-and-set" instruction that tests a bit in memory and simultaneously sets it to one. The producer has the obvious dual mechanisms.

Many highly parallel machines have been, and still are, just collections of independent computers on some sort of a network. Such machines can be made to have just about any data sharing and synchronization mechanism; it just depends on what software is provided by the operating system, the compilers, and the runtime libraries. One possibility, the one used by the first of these machines (The Caltech Cosmic Cube, from around 1984) is message passing. (So it's misleading to call these "message passing machines"; they are really multicomputers with message passing library software.)

From the point of view of the application, these computers can send a message to another computer and can receive such messages off the network. Thus, a process cannot touch any data other than what is in its own, private memory. The way it communicates is to send messages to and receive messages from other processes. Synchronization happens as part of the process, by virtue of the fact that both the sending and receiving process have to make a call to the system in order to move the data: the sender won't call send until its data is already in the send buffer, and the receiver calls receive when its receive buffer is empty and it needs more data to proceed.

Message passing systems have been around since the Cosmic Cube, about ten years. In that time, there has been a lot of evolution, improved efficiency, better software engineering, improved functionality. Many variants were developed by users, computer vendors, and independent software companies. Finally, in 1993, a standardization effort was attempted, and the result is the Message Passing Interface (MPI) standard. MPI is flexible and general, has good implementations on all the machines one is likely to use, and is almost certain to be around for quite some time. We'll use MPI in the course. On one hand, MPI is complicated considering that there are more than 150 functions and the number is still growing. But on the other hand, MPI is simple because there are only six basic functions: `MPI_Init`, `MPI_Finalize`, `MPI_Comm_rank`, `MPI_Comm_size`, `MPI_Send` and `MPI_Recv`.

In print, the best MPI reference is the handbook *Using MPI*, by William Gropp, Ewing Lusk, and Anthony Skjellum, published by MIT Press ISBN 0-262-57104-8.

The standard is on the World WIde Web. The URL is
http://www.mcs.anl.gov/mpi/mpi-report/mpi-report.html. An updated version is at
ftp://ftp.mcs.anl.gov/pub/mpi/mpi-1.jun95/mpi-report.ps

2.2.1 Who am I?

On the SP-2 and other multicomputers, one usually writes one program which runs on all the processors. In order to differentiate its behavior, (like producer and consumer) a process usually first finds out at runtime its rank within its process group, then branches accordingly. The calls

```
MPI_Comm_size(MPI_Comm comm, int *size)
```

sets `size` to the number of processes in the group specified by `comm` and the call

```
MPI_Comm_rank(MPI_Comm comm, int *rank)
```

sets `rank` to the rank of the calling process within the group (from 0 up to $n-1$ where n is the size of the group). Usually, the first thing a program does is to call these using `MPI_COMM_WORLD` as the communicator, in order to find out the answer to the *big* questions, "Who am I?" and "How many other 'I's are there?".

Okay, I lied. That's the second thing a program does. Before it can do anything else, it has to make the call

```
MPI_Init(int *argc, char ***argv)
```

where `argc` and `argv` should be pointers to the arguments provided by UNIX to main(). While we're at it, let's not forget that one's code needs to start with

```
#include "mpi.h"
```

The *last* thing the MPI code does should be

```
MPI_Finalize()
```

No arguments.

Here's an MPI multi-process "Hello World":

```
#include <stdio.h>
#include "mpi.h"

main(int argc, char** argv) {
  int i, myrank, nprocs;
  double a = 0, b = 1.1, c = 0.90;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  printf("Hello world!  This is process %d out of %d\n",myrank, nprocs);
  if (myrank == 0) printf("Some processes are more equal than others.");
  MPI_Finalize();
}  /*  main  */
```

which is compiled and executed on the SP-2 at Ames by

```
babbage1% mpicc -O3 example.c
babbage1% a.out -procs 2
```

and produces (on the standard output)

```
  0:Hello world!  This is process 0 out of 2
  1:Hello world!  This is process 1 out of 2
  0:Some processes are more equal than others.
```

Another important thing to know about is the MPI wall clock timer:

```
double MPI_Wtime()
```

which returns the time in seconds from some unspecified point in the past.

2.2.2 Sending and receiving

In order to get started, let's look at the two most important MPI functions, `MPI_Send` and `MPI_Recv`.

The call

```
MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest,
          int tag, MPI_Comm comm)
```

sends `count` items of data of type `datatype` starting at the location `buf`. In all message passing systems, the processes have identifiers of some kind. In MPI, the process is identified by its `rank`, an integer. The data is sent to the processes whose rank is `dest`. Possible values for `datatype` are `MPI_INT, MPI_DOUBLE, MPI_CHAR` *etc.* `tag` is an integer used by the programmer to allow the receiver to select from among several arriving messages in the `MPI_Recv`. Finally, `comm` is something called a communicator, which is essentially a subset of the processes. Ordinarily, message passing occurs within a single subset. The subset `MPI_COMM_WORLD` consists of all the processes in a single parallel job, and is predefined.

A receive call matching the send above is

```
MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,
         int tag, MPI_Comm comm, MPI_Status *status)
```

`buf` is where the data is placed once it arrives. `count`, an input argument, is the size of the buffer; the message is truncated if it is longer than the buffer. Of course, the receive has to be executed by the correct destination process, as specified in the `dest` part of the send, for it to match. `source` must be the rank of the sending process. The communicator and the tag must match. So must the datatype.

The purpose of the datatype field is to allow MPI to be used with heterogeneous hardware. A process running on a little-endian machine may communicate integers to another process on a big-endian machine; MPI converts them automatically. The same holds for different floating point formats. Type conversion, however, is not supported: an integer must be sent to an integer, a double to a double, *etc.*

Suppose the producer and consumer transact business in two word integer packets. The producer is process 0 and the consumer is process 1. Then the send would look like this:

```
int outgoing[2];
MPI_Send(outgoing, 2, MPI_INT, 1 100, MPI_COMM_WORLD)
```

and the receive like this:

```
MPI_Status stat;
int incoming[2];
MPI_Recv(incoming, 2, MPI_INT, 0 100, MPI_COMM_WORLD, &stat)
```

What if one wants a process to which several other processes can send messages, with service provided on a first-arrived, first-served basis? For this purpose, we don't want to specify the source in our receive, and we use the value `MPI_ANY_SOURCE` instead of an explicit source. The same is true if we want to ignore the tag: use `MPI_ANY_TAG`. The basic purpose of the status argument, which is an output argument, is to find out what the tag and source of such a received message are. `status.MPI_TAG` and `status.MPI_SOURCE` are components of the struct `status` of type int that contain this information after the `MPI_Recv` function returns.

This form of send and receive are "blocking", which is a technical term that has the following meaning. for the send, it means that `buf` has been read by the system and the data has been moved out as soon as the send returns. The sending process can write into it without corrupting the message that was sent. For the receive, it means that `buf` has been filled with data on return. (A call to `MPI_Recv` with no corresponding call to `MPI_Send` occurring elsewhere is a very good and often used method for hanging a message passing application.)

MPI implementations may use buffering to accomplish this. When send is called, the data are copied into a system buffer and control returns to the caller. A separate system process (perhaps using communication hardware) completes the job of sending the data to the receiver. Another implementation is to wait until a corresponding receive is posted by the destination process, then transfer the data to the receive buffer, and finally return control to the caller. MPI provides two variants of send, `MPI_Bsend` and `MPI_Ssend` that force the buffered or the rendezvous implementation. Lastly, there is a version of send that works in "ready" mode. For such a send, the corresponding receive must have been executed previously, otherwise an error occurs. On some systems, this may be faster than the blocking versions of send. All four versions of send have the same calling sequence.

NOTE: MPI allows a process to send itself data. Don't try it. On the SP-2, if the message is big enough, it doesn't work. Here's why. Consider this code:

```
if (myrank == 0)
   for(dest = 0; dest < size; dest++)
      MPI_Send(sendbuf+dest*count, count, MPI_INT, dest, tag, MPI_COMM_WORLD);
MPI_Recv(recvbuf, count, MPI_INT, 0, tag, MPI_COMM_WORLD, &stat);
```

The programmer is attempting to send data from process zero to all processes, including process zero; $4 \cdot \text{count}$ bytes of it. If the system has enough buffer space for the outgoing messages, this succeeds, but if it doesn't, then the send blocks until the receive is executed. But since control does not return from the blocking send to process zero, the receive never does execute. If the programmer uses buffered send, then this deadlock cannot occur. An error will occur if the system runs out of buffer space, however:

```
if (myrank == 0)
   for(dest = 0; dest < size; dest++)
      MPI_Bsend(sendbuf+dest*count, count, MPI_INT, dest, tag, MPI_COMM_WORLD);
MPI_Recv(recvbuf, count, MPI_INT, 0, tag, MPI_COMM_WORLD, &stat);
```

2.2.3 Tags and communicators

Tags are used to keep messages straight. An example will illustrate how they are used. Suppose each process in a group has one integer and one real value, and we wish to find, on process zero, the sum of the integers and the sum of the reals. Lets write this:

```
itag = 100;
MPI_Send(&intvar, 1, MPI_INT, 0, itag, MPI_COMM_WORLD);
ftag = 101;
MPI_Send(&floatvar, 1, MPI_FLOAT, 0, ftag, MPI_COMM_WORLD);
/****  Sends are done.   Receive on process zero   ****/
if (myrank == 0) {
   intsum = 0;
   for (kount = 0; kount < nprocs; kount++) {
      MPI_Recv(&intrecv, 1, MPI_INT, MPI_ANY_SOURCE, itag, MPI_COMM_WORLD, &stat);
      intsum += intrecv;
   }
   fltsum = 0;
   for (kount = 0; kount < nprocs; kount++) {
      MPI_Recv(&fltrecv, 1, MPI_FLOAT, MPI_ANY_SOURCE, ftag, MPI_COMM_WORLD, &stat);
      fltsum += fltrecv;
   }
```

```
}
```

It looks simple, but there are a lot of subtleties here! First, note the use of MPI_ANY_SOURCE in the receives. We're happy to receive the data in the order it arrives. Second, note that we use two different tag values to distinguish between the int and the float data. Why isn't the MPI_TYPE filed enough? Because MPI does not include the type as part of the message "envelope". The envelope consists of the source, destination, tag, and communicator, and these must match in a send-receive pair. Now the two messages sent to process zero from some other process are guaranteed to arrive in the order they were sent, namely the integer message first. But that does not mean that all of the integer message precede all of the float messages! So the tag is needed to distinguish them.

This solution creates a problem. Our code, as it is now written, sends off a lot of messages with tags 100 and 101, then does the receives (at process zero). Suppose we called a library routine written by another user before we did the receives. What if that library code uses the same message tags? Chaos results. We've "polluted" the tag space. Note, by the way, that synchronizing the processes before calling the library code does not solve this problem.

MPI provides communicators as a way to prevent this problem. The communicator is a part of the message envelope. So we need to change communicators while in the library routine. To do this, we use `MPI_Comm_dup`, which makes a new communicator with the same processes in the same order as an existing communicator. For example

```
void safe_library_routine(MPI_Comm oldcomm)
{
   MPI_Comm mycomm;
   MPI_Comm_dup(oldcomm, &mycomm);
   <library code using mycomm for communication>
   MPI_Comm_free(&mycomm);
}
```

The messages sent and received inside the library code cannot interfere with those send outside.

2.2.4 Performance, and tolerance

Try this exercise. See how long a message of length n bytes takes between the call time the send calls send and the time the receiver returns from receive. Do an experiment and vary n. Also vary the rank of the receiver for a fixed sender. Does the model

$$\mathrm{Elapsed\_Time}(n, r) = \alpha + \beta n$$

work? (r is the receiver, and according to this model, the cost is receiver independent.)

In such a model, the latency for a message is α seconds, and the bandwidth is $1/\beta$ bytes/second. Other models try to split α into two components. The first is the time actually spent by the sending processor and the receiving processor on behalf of a message. (Some of the per-byte cost is also attributed to the processors.) This is called the overhead. The remaining component of latency is the delay as the message actually travels through the machines interconnect network. It is ordinarily much smaller than the overhead on modern multicomputers (ones, rather than tens of microseconds).

A lot has been made about the possibility of improving performance by "tolerating" communication latency. To do so, one finds other work for the processor to do while it waits for a message to arrive. The simplest thing is for the programmer to do this explicitly. For this purpose, there are "nonblocking" versions of send and receive in MPI and other dialects.

Nonblocking send and receive work this way. A nonblock **send start** call initiates a send but returns before the data are out of the send buffer. A separate call to **send complete** then blocks the sending process, returning only when the data are out of the buffer. The same two-phase protocol is used for nonblocking receive. The **receive start** call returns right away, and the **receive complete** call returns only when the data are in the buffer.

The simplest mechanism is to match a nonblocking receive with a blocking send. To illustrate, we perform the communication operation of the previous section using nonblocking receive.

```
MPI_Request request;
MPI_IRecv(recvbuf, count, MPI_INT, 0, tag, MPI_COMM_WORLD, &request);
if (myrank == 0)
   for(dest = 0; dest < size; dest++)
      MPI_Send(sendbuf+dest*count, count, MPI_INT, dest, tag, MPI_COMM_WORLD);
MPI_Wait(&request, &stat);
```

`MPI_Wait` blocks until the nonblocking operation identified by the handle `request` completes. This code is correct regardless of the availability of buffers. The sends will either buffer or block until the corresponding **receive start** is executed, and all of these will be.

Before embarking on an effort to improve performance this way, one should first consider what the payoff will be. In general, the best that can be achieved is a two-fold improvement. Often, for large problems, it's the bandwidth (the βn term) that dominates, and latency tolerance doesn't help with this. Rearrangement of the data and computation to avoid some of the communication altogether is required to reduce the bandwidth component of communication time.

2.2.5 Who's got the floor?

We usually think of send and receive as the basic message passing mechanism. But they're not the whole story by a long shot. If we wrote codes that had genuinely different, independent, asynchronous processes that interacted in response to random events, then send and receive would be used to do all the work. Now consider computing a dot product of two identically distributed vectors. Each processor does a local dot product of its pieces, producing one scalar value per processor. Then we need to add them together and, probably, broadcast the result. Can we do this with send and receive? Sure. Do we want to? No. No because it would be a pain in the neck to write and because the MPI system implementor may be able to provide the two necessary, and generally useful collective operations (sum, and broadcast) for us in a more efficient implementation.

MPI has lots of these "collective communication" functions. (And like the sum operation, they often do computation as part of the communication.)

Here's a sum operation, on doubles. The variable `sum` on process `root` gets the sum of the variables `x` on all the processes.

```
double x, sum;
int root, count = 1;
MPI_Reduce(&x, &sum, count, MPI_DOUBLE, MPI_SUM, root, MPI_COMM_WORLD);
```

The fifth argument specifies the operation; other possibilities are `MPI_MAX, MPI_LAND, MPI_BOR,` ... which specify maximum, logical AND, and bitwise OR, for example.

The semantics of the collective communication calls are subtle to this extent: nothing happens except that a process stops when it reaches such a call, until *all* processes in the specified group reach it. Then the reduction operation occurs and the result is placed in the `sum` variable on the *root* processor. Thus, reductions provide what is called a *barrier* synchronization.

There are quite a few collective communication operations provided by MPI, all of them useful and important. We will use several in the assignment. To mention a few, MPI_Bcast broadcasts a vector from one process to the rest of its process group; MPI_Scatter sends different data from one process to each process in its a group; MPI_Gather is the inverse of a scatter: one process receives and concatenates data from all processes in its group; MPI_Allgather is like a gather followed by a broadcast: all processes receive the concatenation of data that are initially distributed among them; MPI_Reduce_scatter is like reduce followed by scatter: the result of the reduction ends up distributed among the process group. Finally, `MPI_Alltoall` implements a very general communication in which each process has a separate message to send to each member of the process group.

Often the process group in a collective communication is some subset of all the processors. In a typical situation, we may view the processes as forming a grid, let's say a 2d grid, for example. We may want to do a reduction operation within rows of the process grid. For this purpose, we can use MPI_Reduce, with a separate communicator for each row.

To make this work, each process first computes its coordinates in the process grid. MPI makes this easy, with

```
int nprocs, myproc, procdims[2], myproc_row, myproc_col;
MPI_Dims_create(nprocs, 2, procdims);
myproc_row = myrank / procdims[1];
myproc_col = myrank % procdims[1];
```

Next. one creates new communicators, one for each process row and one for each process column. The calls

```
MPI_Comm my_prow, my_pcol;
MPI_Comm_split(MPI_COMM_WORLD, myproc_row, 0, &my_prow);
MPI_Comm_split(MPI_COMM_WORLD, myproc_row, 0, &my_prow);
```

create them and

```
MPI_Comm_free(&my_prow);
MPI_Comm_free(&my_pcol);
```

free them. The reduce-in-rows call is then

```
MPI_Reduce(&x, &sum, count, MPI_DOUBLE, MPI_SUM, 0, my_prow);
```

which leaves the sum of the vectors `x` in the vector `sum` in the process whose rank in the group is zero: this will be the first process in the row. The general form is

```
MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm newcomm)
```

As in the example above, the group associated with `comm` is split into disjoint subgroups, one for every different value of `color`; the communicator for the subgroup that this process belongs to is returned in `newcomm`. The argument `key` determines the rank of this process within `newcomm`; the members are ranked according to their value of `key`, with ties broken using the rank in `comm`.

2.3 More on Message Passing

2.3.1 Nomenclature

- Performance:

  Latency: the time it takes to send a zero length message (overhead)
  Bandwidth: the rate at which data can be sent

- Synchrony:

  Synchronous: sending function returns when a matching receiving operation has been initiated at the destination.
  Blocking: sending function returns when the message has been safely copied.
  Non-blocking (asynchronous): sending function returns immediately. Message buffer must not be changed until it is safe to do so

- Miscellaneous:

  Interrupts: if enabled, the arrival of a message interrupts the receiving processor
  Polling: the act of checking the occurrence of an event
  Handlers: code written to execute when an interrupt occurs
  Critical sections: sections of the code which cannot be interrupted safely
  Scheduling: the process of determining which code to run at a given time
  Priority: a relative statement about the ranking of an object according to some metric

2.3.2 The Development of Message Passing

- Early Multicomputers:
  UNIVAC
  Goodyear MPP (SIMD.)
  Denelcor HEP (Shared memory)

- The Internet and Berkeley Unix (High latency, low bandwidth)

  Support for communications between computers is part of the operating system (sockets, ports,remote devices)
  Client/server applications appear

- Parallel Machines (lowest latency,high bandwidth - dedicated networks)

  Ncube
  Intel (Hypercubes, Paragon)
  Meiko (CS1,CS2)
  TMC (CM-5)

- Workstation clusters (lower latency, high bandwidth, popular networks)

  IBM (SP1,2) - optional proprietary network

- Software (libraries):

  CrOS, CUBIX, NX, Express
  Vertex
  EUI/MPL (adjusts to the machine operating system)
  CMMD

PVM (supports heterogeneous machines; dynamic machine configuration)
PARMACS, P4, Chamaleon
MPI (analgum of everything above and adjusts to the operating system)

- Systems:

  Mach
  Linda (object-based system)

2.3.3 Machine Characteristics

- Nodes:

  CPU, local memory,perhaps local I/O

- Networks:

  Topology: Hypercube,Mesh,Fat-Tree, other
  Routing: circuit, packet, wormhole, virtual channel random
  Bisection bandwidth (how much data can be sent along the net)
  Reliable delivery
  Flow Control
  "State" : Space sharing, timesharing, stateless

- Network interfaces: Dumb: Fifo, control registers
  Smart: DMA (Direct Memory Access) controllers
  Very Smart: handle protocol managememt

2.3.4 Active Messages

[Not yet written]

2.4 OpenMP for Shared Memory Parallel Programming

OpenMP is the current industry standard for shared-memory parallel programming directives. Jointly defined by a group of major computer hardware and software vendors, OpenMP Application Program Interface (API) supports shared-memory parallel programming in C/C++ and Fortran on Unix and Windows NT platforms. The OpenMP specifications are owned and managed by the OpenMP Architecture Review Board (ARB). For detailed information about OpenMP, we refer readers to its website at

http://www.openmp.org/.

OpenMP stands for Open specifications for Multi Processing. It specify a set of compiler directives, library routines and environment variables as an API for writing multi-thread applications in C/C++ and Fortran. With these directives or pragmas, multi-thread codes are generated by compilers. OpenMP is a shared momoery model. Threads communicate by sharing variables. The cost of communication comes from the synchronization of data sharing in order to protect data conficts.

Compiler pragmas in OpenMP take the following forms:

- for C and C++,

`#pragma omp` *construct [clause [clause]...]*,

- for Fortran,

`C$OMP` *construct [clause [clause]...]*
`!$OMP` *construct [clause [clause]...]*
`*$OMP` *construct [clause [clause]...]*.

For example, OpenMP is usually used to parallelize loops, a simple parallel C program with its loops split up looks like

```
void main()
{
  double Data[10000];
#pragma omp parallel for
  for (int i=0; i<10000; i++) {
    task(Data[i]);
  }
}
```

If all the OpenMP constructs in a program are compiler pragmas, then this program can be compiled by compilers that do not support OpenMP.

OpenMP's constructs fall into 5 categories, which are briefly introduced as the following:

1. **Parallel Regions**
   Threads are created with the "omp parallel" pragma. In the following example, a 8-thread *parallel region* is created:

```
double Data[10000];
omp_set_num_threads(8);
#pragma omp parallel
{
  int ID = omp_get_thread_num();
  task(ID,Data);
}
```

   Each thread calls `task(ID,Data)` for `ID` $= 0$ to 7.

2. **Work Sharing**
   The "for" work-sharing construct splits up loop iterations among the threads a parallel region. The pragma used in our first example is the short hand form of "omp parallel" and "omp for" pragmas:

```
void main()
{
  double Data[10000];
#pragma omp parallel
#pragma omp for
  for (int i=0; i<10000; i++) {
    task(Data[i]);
  }
}
```

"Sections" is another work-sharing construct which assigns different jobs (different pieces of code) to each thread in a parallel region. For example,

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    {
      job1();
    }
    #pragma omp section
    {
      job2();
    }
  }
}
```

Similarly, there is also a "parallel sections" construct.

3. **Data Environment**
   In the shared-memory programming model, global variables are shared among threads, which are file scope and static variables for C, and common blocks, save and module variables for Fortran. Automatic variables within a statement block are private, also stack variables in sub-programs called from parallel regions. Constructs and clauses are available to selectively change storage attributes.

   - The "shared" clause uses a shared memory model for the variable, that is, all threads share the same variable.
   - The "private" clause gives a local copy of the variable in each thread.
   - "firstprivate" is like "private", but the variable is initialized with its value before the thread creation.
   - "lastprivate" is like "private", but the value is passed to a global variable after the thread execution.

4. **Synchronization**
   The following constructs are used to support synchronization:

   - "critical" and "end critical" constructs define a *critical region*, where only one thread can enter at a time.
   - "atomic" construct defines a critical region that only contains one simple statement.
   - "barrier" construct is usually implicit, for example at the end of a parallel region or at the end of a "for" work-sharing construct. The barrier construct makes threads wait until all of them arrive.
   - "ordered" construct enforces the sequential order for a block of code.
   - "master" construct marks a block of code to be only executed by the master thread. The other threads just skip it.

- "single" construct marks a block of code to be executed only by one thread.
- "flush" construct denotes a sequence point where a thread tries to create a consistent view of memory.

5. **Runtime functions and environment variables**
   Besides constructs, there are clauses, library routines and environment variables in OpenMP. We list a few of the most important below, please refer to OpenMP's web site for more details.

   - The number of threads can be controlled using the runtime environment routines "omp_set_num_threads()", "omp_get_num_threads()", "omp_get_thread_num()", "omp_get_max_threads()".
   - The number of processors in the system is given by "omp_num_procs()".

2.5 STARP

Star-P is a set of extensions to Matlab aimed at making it simple to parallelize many common computations. A running instance of Star-P consists of a control Matlab process (connected to the Matlab notebook/desktop the user sees) and a slaved Matlab processes for each processor available to the system. It permits a matlab user to declare distributed matrix objects, whose contents are distributed in various ways across the processors available to Star-P, and to direct each processor to apply a matlab function to the portion of a distributed matrix it stores. It also supplies parallel versions of many of the dense (and some of the sparse) matrix operations which occur in scientific computing, such as eigenvector/eigenvalue computations.

This makes Star-P especially convenient for embarassingly parallel problems, as all one needs to do is:

1. Write a matlab function which carries out a part of the embarassingly parallel problem, parametrized such that it can take the inputs it needs as the rows or columns of a matrix.
2. Construct a distributed matrix containing the parameters needed for each slave process.
3. Tell each slave to apply the function to the portion of the matrix it has.
4. Combine the results from each process.

Now, some details. Star-P defines the variable `np` to store the number of slave processes it controls. On a machine with two apparent processors, for example, `np` would equal 2. Distributed matrices are declared in Star-P by appending `*p` to the dimension along which you want your matrix to be distributed. For example, the code below declares `A` to be a row-distributed matrix; each processor theoretically gets an equal chunk of rows from the matrix.

```
A = ones(100*p, 100);
```

To declare a column-distributed matrix, one simply does:

```
A = ones(100, 100*p);
```

Beware: if the number of processors you're working with does not evenly divide the size of the dimension you are distributing along, you may encounter unexpected results. Star-P also supports matrices which are distributed into blocks, by appending `*p` to both dimensions; we will not go into the details of this here.

After declaring `A` as a distributed matrix, simply evaluating it will yield something like:

```
A = ddense object: 100-by-100
```

This is because the elements of `A` are stored in the slave processors. To bring the contents to the control process, simply index A. For example, if you wanted to view the entire contents, you'd do `A(:,:)`.

To apply a Matlab function to the chunks of a distributed matrix, use the `mm` command. It takes a string containing a procedure name and a list of arguments, each of which may or may not be distributed. It orders each slave to apply the function to the chunk of each distributed argument (echoing non-distributed arguments) and returns a matrix containing the results of each appended together. For example, `mm('fft', A)` (with `A` defined as a 100-by-100 column distributed matrix) would apply the `fft` function to each of the 25-column chunks. Beware: the chunks must each be distributed in the same way, and the function must return chunks of the same size. Also beware: `mm` is meant to apply a function to chunks. If you want to compute the two-dimensional `fft` of `A` in parallel, do not use `mm('fft2', A)`; that will compute (in serial) the `fft2`s of each chunk of A and return them in a matrix. `eig(A)`, on the other hand, will apply the parallel algorithm for eigenstuff to `A`.

Communication between processors must be mediated by the control process. This incurs substantial communications overhead, as the information must first be moved to the control process, processed, then sent back. It also necessitates the use of some unusual programming idioms; one common pattern is to break up a parallel computation into steps, call each step using `mm`, then do matrix column or row swapping in the control process on the distributed matrix to move information between the processors. For example, given a matrix `B = randn(10, 2*p)` (on a Star-P process with two slaves), the command `B = B(:,[2,1])` will swap elements between the two processors.

Some other things to be aware of:

1. Star-P provides its functionality by overloading variables and functions from Matlab. This means that if you overwrite certain variable names (or define your own versions of certain functions), they will shadow the parallel versions. In particular, DO NOT declare a variable named `p`; if you do, instead of distributing matrices when `*p` is appended, you will multiply each element by your variable `p`.

2. `persistent` variables are often useful for keeping state across stepped computations. The first time the function is called, each persistent variable will be bound to the empty matrix `[]`; a simple `if` can test this and initialize it the first time around. Its value will then be stored across multiple invocations of the function. If you use this technique, make sure to `clear` those variables (or restart Star-P) to ensure that state isn't propagated across runs.

3. To execute a different computation depending on processor id, create a matrix `id = 1:np` and pass it as an argument to the function you call using `mm`. Each slave process will get a different value of `id` which it can use as a different unique identifier.

4. If `mm` doesn't appear to find your `m`-files, run the `mmpath` command (which takes one argument - the directory you want `mm` to search in).

Have fun!

Lecture 3

Parallel Prefix

3.1 Parallel Prefix

An important primitive for (data) parallel computing is the *scan operation*, also called *prefix sum* which takes an associated binary operator $\oplus$ and an ordered set $[a_1, \ldots, a_n]$ of n elements and returns the ordered set

$$[a_1, (a_1 \oplus a_2), \ldots, (a_1 \oplus a_2 \oplus \ldots \oplus a_n)].$$

For example,

$$\texttt{plus\_scan}([1, 2, 3, 4, 5, 6, 7, 8]) = [1, 3, 6, 10, 15, 21, 28, 36].$$

Notice that computing the scan of an n-element array requires $n-1$ serial operations.

Suppose we have n processors, each with one element of the array. If we are interested only in the last element b_n, which is the total sum, then it is easy to see how to compute it efficiently in parallel: we can just break the array recursively into two halves, and add the sums of the two halves, recursively. Associated with the computation is a complete binary tree, each internal node containing the sum of its descendent leaves. With n processors, this algorithm takes $O(\log n)$ steps. If we have only $p < n$ processors, we can break the array into p subarrays, each with roughly $\lceil n/p \rceil$ elements. In the first step, each processor adds its own elements. The problem is then reduced to one with p elements. So we can perform the $\log p$ time algorithm. The total time is clearly $O(n/p + \log p)$ and communication only occur in the second step. With an architecture like hypercube and fat tree, we can embed the complete binary tree so that the communication is performed directly by communication links.

Now we discuss a parallel method of finding *all* elements $[b_1, \ldots, b_n] = \oplus\texttt{\_scan}[a_1, \ldots, a_n]$ also in $O(\log n)$ time, assuming we have n processors each with one element of the array. The following is a Parallel Prefix algorithm to compute the scan of an array.

Function $\texttt{scan}([a_i])$:

1. Compute pairwise sums, communicating with the adjacent processor
   $c_i := a_{i-1} \oplus a_i$ (if i even)

2. Compute the even entries of the output by recursing on the size $\frac{n}{2}$ array of pairwise sums
   $b_i := \texttt{scan}([c_i])$ (if i even)

3. Fill in the odd entries of the output with a pairwise sum
   $b_i := b_{i-1} \oplus a_i$ (if i odd)

4. Return $[b_i]$.

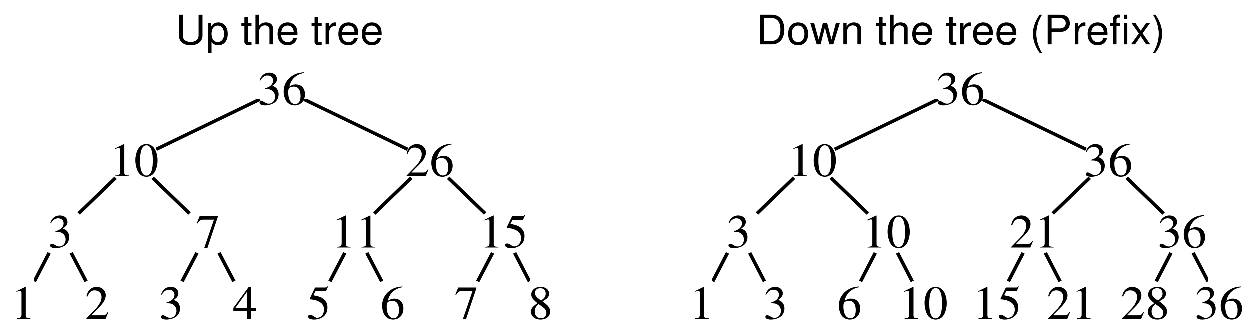


Figure 3.1: Action of the Parallel Prefix algorithm.

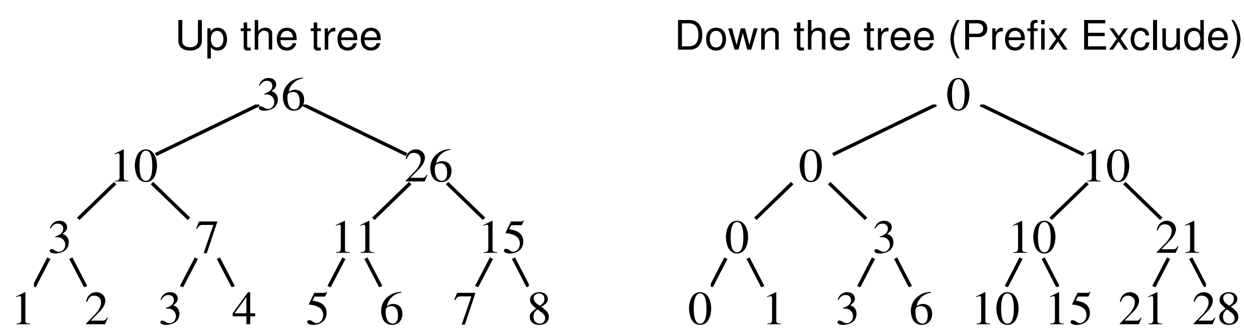


Figure 3.2: The Parallel Prefix Exclude Algorithm.

An example using the vector $[1, 2, 3, 4, 5, 6, 7, 8]$ is shown in Figure 3.1. Going up the tree, we simply compute the pairwise sums. Going down the tree, we use the updates according to points 2 and 3 above. For even position, we use the value of the parent node (b_i). For odd positions, we add the value of the node left of the parent node (b_{i-1}) to the current value (a_i).

We can create variants of the algorithm by modifying the update formulas 2 and 3. For example, the *excluded prefix sum*

$$[0, a_1, (a_1 \oplus a_2), \ldots, (a_1 \oplus a_2 \oplus \ldots \oplus a_{n-1})]$$

can be computed using the rule:

$$b_i := \texttt{excl\_scan}([c_i]) \quad \text{(if } i \text{ odd)}, \tag{3.1}$$

$$b_i := b_{i-1} \oplus a_{i-1} \quad \text{(if } i \text{ even)}. \tag{3.2}$$

Figure 3.2 illustrates this algorithm using the same input vector as before.

The total number of $\oplus$ operations performed by the Parallel Prefix algorithm is (ignoring a constant term of ± 1):

$$\begin{aligned} T_n &= \overbrace{\frac{n}{2}}^{\mathrm{I}} + \overbrace{T_{n/2}}^{\mathrm{II}} + \overbrace{\frac{n}{2}}^{\mathrm{III}} \\ &= n + T_{n/2} \\ &= 2n \end{aligned}$$

If there is a processor for each array element, then the number of parallel operations is:

$$\begin{aligned} T_n &= \overbrace{1}^{\mathrm{I}} + \overbrace{T_{n/2}}^{\mathrm{II}} + \overbrace{1}^{\mathrm{III}} \\ &= 2 + T_{n/2} \\ &= 2\lg n \end{aligned}$$

3.2 The "Myth" of $\lg n$

In practice, we usually do not have a processor for each array element. Instead, there will likely be many more array elements than processors. For example, if we have 32 processors and an array of 32000 numbers, then each processor should store a contiguous section of 1000 array elements. Suppose we have n elements and p processors, and define $k = n/p$. Then the procedure to compute the scan is:

1. At each processor i, compute a local scan serially, for n/p consecutive elements, giving result $[d_1^i, d_2^i, \ldots, d_k^i]$. Notice that this step vectorizes over processors.
2. Use the parallel prefix algorithm to compute

$$\texttt{scan}([d_k^1, d_k^2, \ldots, d_k^p]) = [b_1, b_2, \ldots, b_p]$$

3. At each processor $i > 1$, add b_{i-1} to all elements d_j^i.

The time taken for the will be

$$T = 2 \cdot \left(\begin{array}{c} \text{time to add and store} \\ n/p \text{ numbers serially} \end{array} \right) + 2 \cdot (\log p) \cdot \left(\begin{array}{c} \text{Communication time} \\ \text{up and down a tree,} \\ \text{and a few adds} \end{array} \right)$$

In the limiting case of $p \ll n$, the $\lg p$ message passes are an insignificant portion of the computational time, and the speedup is due solely to the availability of a number of processes each doing the prefix operation serially.

3.3 Applications of Parallel Prefix

3.3.1 Segmented Scan

We can extend the parallel scan algorithm to perform segmented scan. In *segmented scan* the original sequence is used along with an additional sequence of booleans. These booleans are used to identify the start of a new segment. Segmented scan is simply prefix scan with the additional condition the the sum starts over at the beginning of a new segment. Thus the following inputs would produce the following result when applying segmented plus scan on the array A and boolean array C.

$$\begin{aligned} A &= [1\ 2\ 3\ 4 \quad 5\ 6 \quad 7 \quad 8\ 9 \quad 10] \\ C &= [1\ 0\ 0\ 0 \quad 1\ 0 \quad 1 \quad 1\ 0 \quad 1\,] \\ \texttt{plus\_scan}(A, C) &= [\underline{1\ 3\ 6\ 10}\ \underline{5\ 11}\ \underline{7}\ \underline{8\ 17}\ \underline{10}\,] \end{aligned}$$

We now show how to reduce segmented scan to simple scan. We define an operator, $\bigoplus_2$, whose operand is a pair $\begin{pmatrix} x \\ y \end{pmatrix}$. We denote this operand as an element of the 2-element representation of A and C, where x and y are corresponding elements from the vectors A and C. The operands of the example above are given as:

$$\begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 2 \\ 0 \end{pmatrix} \begin{pmatrix} 3 \\ 0 \end{pmatrix} \begin{pmatrix} 4 \\ 0 \end{pmatrix} \begin{pmatrix} 5 \\ 1 \end{pmatrix} \begin{pmatrix} 6 \\ 0 \end{pmatrix} \begin{pmatrix} 7 \\ 1 \end{pmatrix} \begin{pmatrix} 8 \\ 1 \end{pmatrix} \begin{pmatrix} 9 \\ 0 \end{pmatrix} \begin{pmatrix} 10 \\ 1 \end{pmatrix}$$

The operator ($\bigoplus_2$) is defined as follows:

$$\begin{array}{c|cc} \bigoplus_2 & \begin{pmatrix} y \\ 0 \end{pmatrix} & \begin{pmatrix} y \\ 1 \end{pmatrix} \\ \hline \begin{pmatrix} x \\ 0 \end{pmatrix} & \begin{pmatrix} x \oplus y \\ 0 \end{pmatrix} & \begin{pmatrix} y \\ 1 \end{pmatrix} \\ \\ \begin{pmatrix} x \\ 1 \end{pmatrix} & \begin{pmatrix} x \oplus y \\ 1 \end{pmatrix} & \begin{pmatrix} y \\ 1 \end{pmatrix} \end{array}$$

As an exercise, we can show that the binary operator $\bigoplus_2$ defined above is associative and exhibits the segmenting behavior we want: for each vector A and each boolean vector C, let AC be the 2-element representation of A and C. For each binary associative operator $\oplus$, the result of $\bigoplus_2$_scan(AC) gives a 2-element vector whose first row is equal to the vector computed by segmented $\oplus$_scan(A, C). Therefore, we can apply the parallel scan algorithm to compute the segmented scan.

Notice that the method of assigning each segment to a separate processor may results in load imbalance.

3.3.2 Csanky's Matrix Inversion

The Csanky matrix inversion algorithm is representative of a number of the problems that exist in applying theoretical parallelization schemes to practical problems. The goal here is to create a matrix inversion routine that can be extended to a parallel implementation. A typical serial implementation would require the solution of $O(n^2)$ linear equations, and the problem at first looks unparallelizable. The obvious solution, then, is to search for a parallel prefix type algorithm.

Csanky's algorithm can be described as follows — the Cayley-Hamilton lemma states that for a given matrix x:

$$p(x) = \det(xI - A) = x^n + c_1 x^{n-1} + \ldots + c_n$$

where $c_n = \det(A)$, then

$$p(A) = 0 = A^n + c_1 A^{n-1} + \ldots + c_n$$

Multiplying each side by A^{-1} and rearranging yields:

$$A^{-1} = (A^{n-1} + c_1 A^{n-2} + \ldots + c_{n-1})/(-1/c_n)$$

The c_i in this equation can be calculated by Leverier's lemma, which relate the c_i to $s^k = \mathrm{tr}(A^k)$. The Csanky algorithm then, is to calculate the A^i by parallel prefix, compute the trace of each A^i, calculate the c_i from Leverier's lemma, and use these to generate A^{-1}.

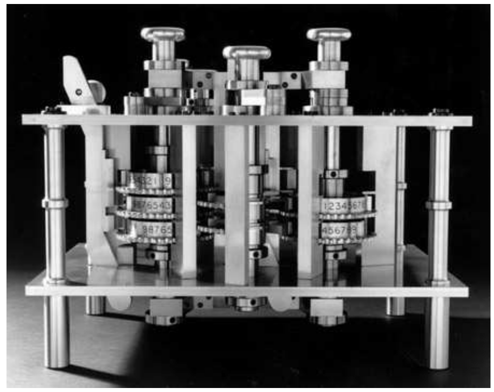


Figure 3.3: Babbage's Difference Engine, reconstructed by the Science Museum of London

While the Csanky algorithm is useful in theory, it suffers a number of practical shortcomings. The most glaring problem is the repeated multiplication of the A matix. Unless the coefficients of A are very close to 1, the terms of A^n are likely to increase towards infinity or decay to zero quite rapidly, making their storage as floating point values very difficult. Therefore, the algorithm is inherently unstable.

3.3.3 Babbage and Carry Look-Ahead Addition

Charles Babbage is considered by many to be the founder of modern computing. In the 1820s he pioneered the idea of mechanical computing with his design of a "Difference Engine," the purpose of which was to create highly accurate engineering tables.

A central concern in mechanical addition procedures is the idea of "carrying," for example, the overflow caused by adding two digits in decimal notation whose sum is greater than or equal to 10. Carrying, as is taught to elementary school children everywhere, is inherently serial, as two numbers are added left to right.

However, the carrying problem can be treated in a parallel fashion by use of parallel prefix. More specifically, consider:

$$\begin{array}{cccccc|l} & c_3 & c_2 & c_1 & c_0 & & \text{Carry} \\ & & a_3 & a_2 & a_1 & a_0 & \text{First Integer} \\ + & & b_3 & b_2 & b_1 & b_0 & \text{Second Integer} \\ \hline & s_4 & s_3 & s_2 & s_1 & s_0 & \text{Sum} \end{array}$$

By algebraic manipulation, one can create a transformation matrix for computing c_i from c_{i-1}:

$$\begin{pmatrix} c_i \\ 1 \end{pmatrix} = \begin{pmatrix} a_i + b_i & a_i b_i \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} c_{i-1} \\ 1 \end{pmatrix}$$

Thus, carry look-ahead can be performed by parallel prefix. Each c_i is computed by parallel prefix, and then the s_i are calculated in parallel.

3.4 Parallel Prefix in MPI

The MPI version of "parallel prefix" is performed by `MPI_Scan`. From Using MPI by Gropp, Lusk, and Skjellum (MIT Press, 1999):

> [`MPI_Scan`] is much like `MPI_Allreduce` in that the values are formed by combining values contributed by each process and that each process receives a result. The difference is that the result returned by the process with rank r is the result of operating on the input elements on processes with rank $0, 1, \dots, r$.

Essentially, `MPI_Scan` operates locally on a vector and passes a result to each processor. If the defined operation of `MPI_Scan` is `MPI_Sum`, the result passed to each process is the partial sum including the numbers on the current process.

`MPI_Scan`, upon further investigation, is not a true parallel prefix algorithm. It appears that the partial sum from each process is passed to the process in a serial manner. That is, the message passing portion of `MPI_Scan` does not scale as $\lg p$, but rather as simply p. However, as discussed in the Section 3.2, the message passing time cost is so small in large systems, that it can be neglected.

Lecture 4

Dense Linear Algebra

4.1 Dense Matrices

We next look at *dense* linear systems of the form $\mathbf{Ax} = \mathbf{b}$. Here A is a given $n \times n$ matrix and b is a given n-vector; we need to solve for the unknown n-vector x. We shall assume that A is a nonsingular matrix, so that for every b there is a unique solution $x = A^{-1}b$. Before solving dense linear algebra problems, we should define the terms sparse, dense, and structured.

Definition. (Wilkinson) A *sparse* matrix is a matrix with enough zeros that it is worth taking advantage of them.

Definition. A *structured* matrix has enough structure that it is worthwhile to use it.

For example, a Toeplitz Matrix is defined by $2n$ parameters. All entries on a diagonal are the same:

$$ToeplitzMatrix = \begin{bmatrix} 1 & 4 & & & & \\ 2 & 1 & 4 & & & \\ & 2 & 1 & 4 & & \\ \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\ & & & 2 & 1 & 4 \\ & & & & 2 & 1 \end{bmatrix}$$

Definition. A *dense* matrix is neither sparse nor structured.

These definitions are useful because they will help us identify whether or not there is any inherent parallelism in the problem itself. It is clear that a sparse matrix does indeed have an inherent structure to it that may conceivably result in performance gains due to parallelism. We will discuss ways to exploit this in the next chapter. The Toeplitz matrix also has some structure that may be exploited to realize some performance gains. When we formally identify these structures, a central question we seem to be asking is if it is it worth taking advantage of this? The answer, as in all of parallel computing, is: "it depends".

If $n = 50$, hardly. The standard $O(n^3)$ algorithm for ordinary matrices can solve $Tx = b$, ignoring its structure, in under one one-hundredth of one second on a workstation. On the other hand, for large n, it pays to use one of the "fast" algorithms that run in time $O(n^2)$ or even $O(n \log^2 n)$. What do we mean by "fast" in this case?

It certainly seems intuitive that the more we know about how the matrix entries are populated the better we should be able to design our algorithms to exploit this knowledge. However, for the purposes of our discussion in this chapter, this does not seem to help for dense matrices. This does not mean that any parallel dense linear algebra algorithm should be conceived or even code in a serial manner. What it does mean, however, is that we look particularly hard at what the matrix A represents in the practical application for which we are trying to solve the equation $\mathbf{Ax} = \mathbf{b}$. By examining the matrix carefully, we might indeed recognize some other less-obvious 'structure' that we might be able to exploit.

4.2 Applications

There are not many applications for large dense linear algebra routines, perhaps due to the "law of nature" below.

- "Law of Nature": Nature does not throw n^2 numbers at us haphazardly, therefore there are few dense matrix problems.

Some believe that there are no real problems that will turn up n^2 numbers to populate the $n \times n$ matrix without exhibiting some form of underlying structure. This implies that we should seek methods to identify the structure underlying the matrix. This becomes particularly important when the size of the system becomes large.

What does it mean to 'seek methods to identify the structure'? Plainly speaking that answer is not known not just because it is inherently difficult but also because prospective users of dense linear algebra algorithms (as opposed to developers of such algorithms) have not started to identify the structure of their A matrices. Sometimes identifying the structure might involve looking beyond traditional literature in the field.

4.2.1 Uncovering the structure from seemingly unstructured problems

For example, in communications and radar processing applications, the matrix A can often be modelled as being generated from another $n \times N$ matrix X that is in turn populated with independent, identically distributed Gaussian elements. The matrix A in such applications will be symettric and will then be obtained as $A = XX^T$ where $(.)^T$ is the transpose operator. At a first glance, it might seem as though this might not provide any structure that can be exploited besides the symmetry of A. However, this is not so. We simply have to dig a bit deeper.

The matrix $A = XX^T$ is actually a very well studied example in random matrix theory. Edelman has studied these types of problems in his thesis and what turns out to be important is that in solving $\mathbf{Ax} = \mathbf{b}$ we need to have a way of characterizing the condition number of A. For matrices, the condition number tells us how 'well behaved' the matrix is. If the condition number is very high then the numerical algorithms are likely to be unstable and there is little guarantee of numerical accuracy. On the other hand, when the condition number is close to 1, the numerical accurarcy is very high. It turns out that a mathematically precise characterization of the random condition number of A is possible which ends up depending on the dimensions of the matrix X. Specifically for a fixed n and large N (typically at least $10n$ is increased, the condition number of A will be fairly localized i.e. its distribution will not have long tails. On the other hand, when N is about the size of n the condition number distribution will not be localized. As a result when solving $x = A^{-1}b$ we will get poor numerical accuracy in our solution of x.

This is important to remember because, as we have described all along, a central feature in parallel computing is our need to distribute the data among different computing nodes (processors,clusters, etc) and to work on that chunk by itself as much as possible and then rely on inter-node

Year	Size of Dense System	Machine
1950's	≈ 100	
1991	55,296	
1992	75,264	Intel
1993	75,264	Intel
1994	76,800	CM
1995	128,600	Intel
1996	128,600	Intel
1997	235000	Intel ASCI Red
1998	431344	IBM ASCI Blue
1999	431344	IBM ASCI Blue
2000	431344	IBM ASCI Blue
2001	518096	IBM ASCI White-Pacific
2002	1041216	Earth Simulator Computer
2003	1041216	Earth Simulator Computer

Table 4.1: Largest Dense Matrices Solved

communication to collect and form our answer. If we did not pay attention to the condition number of A and correspondingly the condition number of *chunks* of A that reside on different processors, our numerical accuracy for the parralel computing task would suffer.

This was just one example of how even in a seemingly unstructured case, insights from another field, random matrix theory in this case, could potentially alter our impact or choice of algorithm design. Incidentally, even what we just described above has not been incorporated into any parallel applications in radar processing that we are aware of. Generally speaking, the design of efficient parallel dense linear algebra algorithms will have to be motivated by and modified based on specific applications with an emphasis on *uncovering the structure* even in seemingly unstructured problems. This, by definition, is something that only users of algorithms could do. Until then, an equally important task is to make dense linear algebra algorithms and libraries that run efficiently regardless of the underlying structure while we wait for the applications to develop.

While there are not too many everyday applications that require dense linear algebra solutions, it would be wrong to conclude that the world does not need large linear algebra libraries. Medium sized problems are most easily solved with these libraries, and the first pass at larger problems are best done with the libraries. Dense methods are the easiest to use, reliable, predictable, easiest to write, and work best for small to medium problems.

For large problems, it is not clear whether dense methods are best, but other approaches often require far more work.

4.3 Records

Table 4.1 shows the largest dense matrices solved. Problems that warrant such huge systems to be solved are typically things like the Stealth bomber and large Boundary Element codes[1]. Another application for large dense problems arise in the "methods of moments", electro-magnetic calculations used by the military.

[1]Typically this method involves a transformation using Greens Theorem from 3D to a dense 2D representation of the problems. This is where the large data sets are generated.

It is important to understand that space considerations, *not* processor speeds, are what bound the ability to tackle such large systems. Memory is the bottleneck in solving these large dense systems. Only a tiny portion of the matrix can be stored inside the computer at any one time. It is also instructive to look at how technological advances change some of these considerations.

For example, in 1996, the record setter of size $n = 128,600$ required $(2/3)n^3 = 1.4 \times 10^{15}$ arithmetic operations (or four times that many if it is a complex matrix) for its solution using Gaussian elimination. On a fast uniprocessor workstation in 1996 running at 140 MFlops/sec, that would take ten million seconds, about 16 and a half weeks; but on a large parallel machine, running at 1000 times this speed, the time to solve it is only 2.7 hours. The storage requirement was $8n^2 = 1.3 \times 10^13$ bytes, however. Can we afford this much main memory? Again, we need to look at it in historical perspective.

In 1996, the price was as low as \$10 per megabyte it would cost \$ 130 million for enough memory for the matrix. Today, however, the price for the memory is much lower. At 5 cents per megabyte, the memory for the same system would be \$650,000. The cost is still prohibitive, but much more realistic.

In contrast, the Earth Simulator which can solve a dense linear algebra system with $n = 1041216$ would require $(2/3)n^3 = 7.5 \times 10^{17}$ arithmetic operations (or four times that many if it is a complex matrix) for its solution using Gaussian elimination. For a 2.25 GHz Pentium 4 uniprocessor based workstation available today, at a speed of 3 GFlops/sec this would take 250 million seconds or roughly 414 weeks or about 8 years! On the Earth Simulator running at its maximum of 35.86 TFlops/sec or about 10000 times the speed of a desktop machine, this would only take about 5.8 hrs! The storage requirement for this machine would be $8n^2 = 8.7 \times 10^{14}$ bytes which at 5 cents a megabyte works out to about \$43.5 million. This is still equally prohibitive athough the figurative 'bang for the buck' keeps getting better.

As in 1996, the cost for the storage was not as high as we calculated. This is because in 1996, when most parallel computers were specially designed supercomputers, "out of core" methods were used to store the massive amount of data. In 2004, however, with the emergence of clusters as a viable and powerful supercomputing option, network storage capability and management becomes an equally important factor that adds to the cost and complexity of the parallel computer.

In general, however, Moore's law does indeed seem to be helpful because the cost per Gigabyte especially for systems with large storage capacity keeps getting lower. Concurrently the density of these storage media keeps increasing as well so that the amount of physical space needed to store these systems becomes smalller. As a result, we can expect that as storage systems become cheaper *and* denser, it becomes increasingly more practical to design and maintain parallel computers.

The accompanying figures show some of these trends in storage density, and price.

4.4 Algorithms, and mapping matrices to processors

There is a simple minded view of parallel dense matrix computation that is based on these assumptions:

- one matrix element per processor
- a huge number (n^2, or even n^3) of processors
- communication is instantaneous

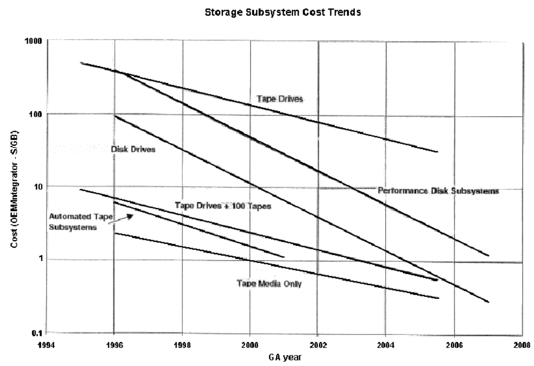


Figure 4.1: Storage sub system cost trends

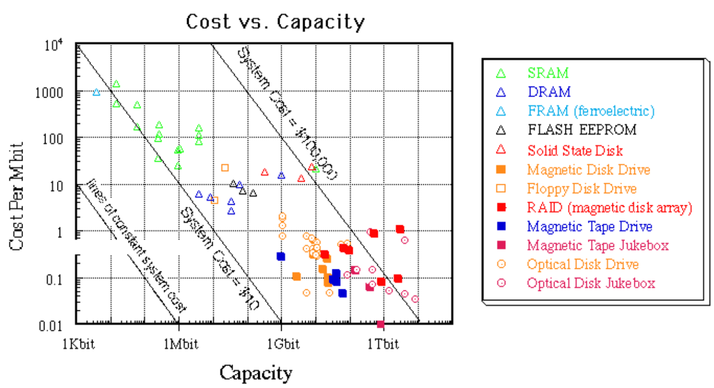


Figure 4.2: Trend in storage capacity

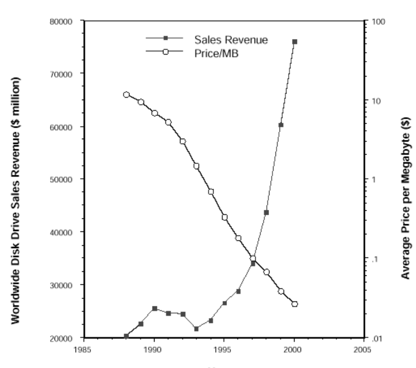


Figure 4.3: Average price per Mb cost trends

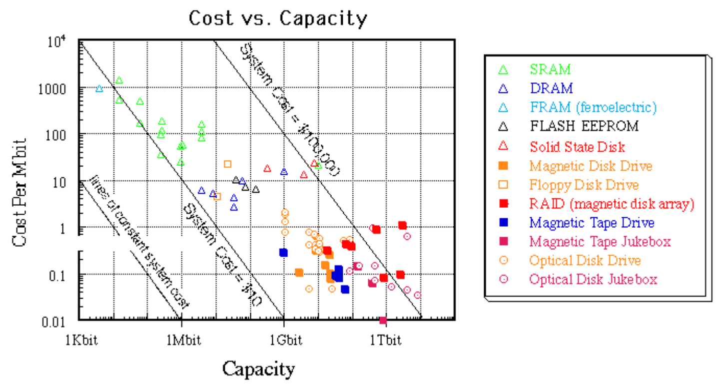


Figure 4.4: Storage density trends

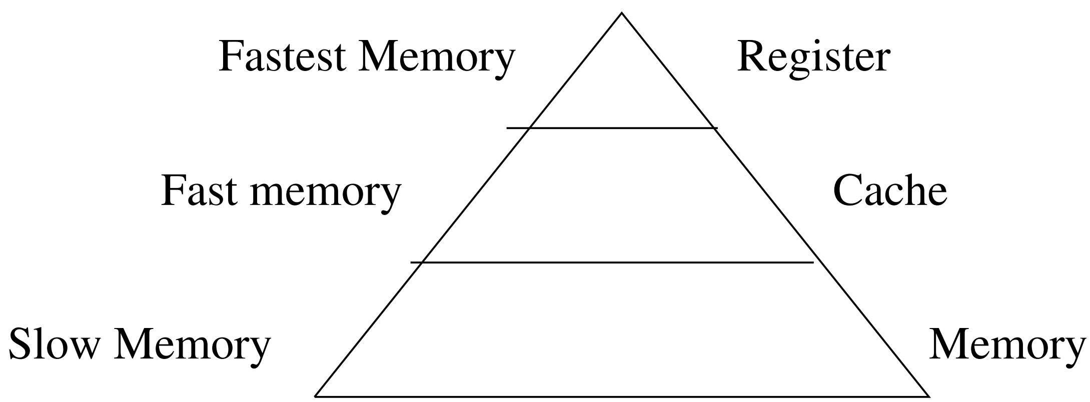


Figure 4.5: Matrix Operations on 1 processor

This is taught frequently in theory classes, but has no practical application. Communication cost is critical, and no one can afford n^2 processors when $n = 128,000$.[2]

In practical parallel matrix computation, it is essential to have large chunks of each matrix on each processor. There are several reasons for this. The first is simply that there are far more matrix elements than processors! Second, it is important to achieve *message vectorization*. The communications that occur should be organized into a small number of large messages, because of the high message overhead. Lastly, uniprocessor performance is heavily dependent on the nature of the local computation done on each processor.

4.5 The memory hierarchy

Parallel machines are built out of ordinary sequential processors. Fast microprocessors now can run far faster than the memory that supports them, and the gap is widening. The cycle time of a current microprocessor in a fast workstation is now in the 3 – 10 nanosecond range, while DRAM memory is clocked at about 70 nanoseconds. Typically, the memory bandwidth onto the processor is close to an order of magnitude less than the bandwidth required to support the computation.

[2]Biological computers have this many processing elements; the human brain has on the order of 10^{11} neurons.

To match the bandwidths of the fast processor and the slow memory, several added layers of memory hierarchy are employed by architects. The processor has registers that are as fast as the processing units. They are connected to an on-chip cache that is nearly that fast, but is small (a few ten thousands of bytes). This is connected to an off-chip level-two cache made from fast but expensive static random access memory (SRAM) chips. Finally, main memory is built from the least cost per bit technology, dynamic RAM (DRAM). A similar caching structure supports instruction accesses.

When LINPACK was designed (the mid 1970s) these considerations were just over the horizon. Its designers used what was then an accepted model of cost: the number of arithmetic operations. Today, a more relevant metric is the number of references to memory that miss the cache and cause a cache line to be moved from main memory to a higher level of the hierarchy. To write portable software that performs well under this metric is unfortunately a much more complex task. In fact, one cannot predict how many cache misses a code will incur by examining the code. One cannot predict it by examining the machine code that the compiler generates! The behavior of real memory systems is quite complex. But, as we shall now show, the programmer can still write quite acceptable code.

(We have a bit of a paradox in that this issue does not really arise on Cray vector computers. These computers have no cache. They have no DRAM, either! The whole main memory is built of SRAM, which is expensive, and is fast enough to support the full speed of the processor. The high bandwidth memory technology raises the machine cost dramatically, and makes the programmer's job a lot simpler. When one considers the enormous cost of software, this has seemed like a reasonable tradeoff.

Why then aren't parallel machines built out of Cray's fast technology? The answer seems to be that the microprocessors used in workstations and PCs have become as fast as the vector processors. Their usual applications do pretty well with cache in the memory hierarchy, without reprogramming. Enormous investments are made in this technology, which has improved at a remarkable rate. And so, because these technologies appeal to a mass market, they have simple priced the expensive vector machines out of a large part of their market niche.)

4.6 Single processor condiderations for dense linear algebra

If software is expected to perform optimally in a parallel computing environment, performance considerations of computation on a single processor must first be evaluated.

4.6.1 LAPACK and the BLAS

Dense linear algebra operations are critical to optimize as they are very compute bound. Matrix multiply, with its $2n^3$ operations involving $3n^2$ matrix elements, is certainly no exception: there is on $O(n)$ *reuse* of the data. If all the matrices fit in the cache, we get high performance. Unfortunately, we use supercomputers for big problems. The definition of "big" might well be "doesn't fit in the cache."

A typical old-style algorithm, which uses the SDOT routine from the BLAS to do the computation via inner product, is shown in Figure 4.6.

This method produces disappointing performance because too many memory references are needed to do an inner product. Putting it another way, if we use this approach we will get $O(n^3)$ cache misses.

Table 4.6.1 shows the data reuse characteristics of several different routines in the BLAS (for Basic Linear Algebra Subprograms) library.

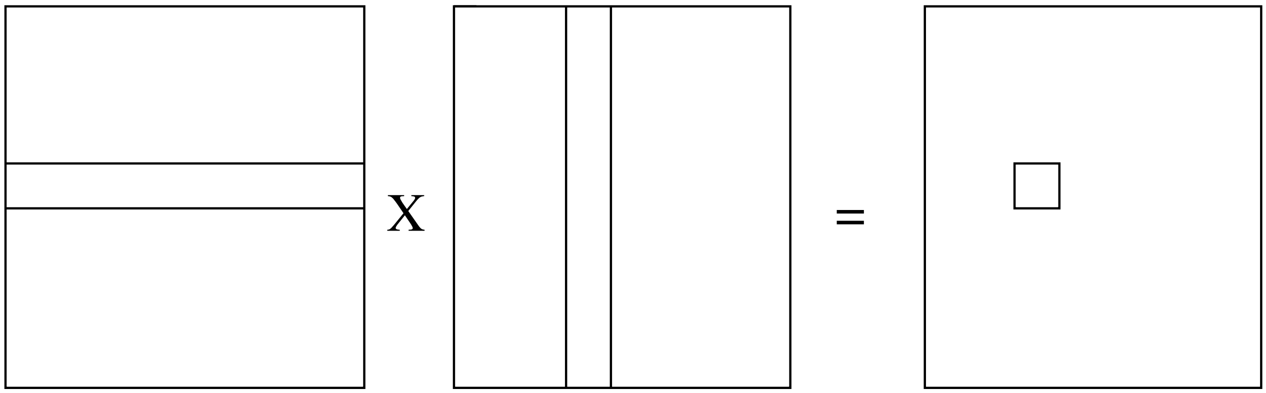


Figure 4.6: Matrix Multiply

Instruction	Operations	Memory Accesses (load/stores)	Ops /Mem Ref
BLAS1: SAXPY (Single Precision **Ax** Plus **y**)	2n	3n	$\frac{2}{3}$
BLAS1: SAXPY $\alpha =$ x . y	2n	2n	1
BLAS2: Matrix-vec $y = Ax + y$	$2n^2$	n^2	2
BLAS3: Matrix-Matrix $C = AB + C$	$2n^3$	$4n^2$	$\frac{1}{2}n$

Table 4.2: Basic Linear Algebra Subroutines (BLAS)

Creators of the LAPACK software library for dense linear algebra accepted the design challenge of enabling developers to write portable software that could minimize costly cache misses on the memory hierarchy of any hardware platform.

The LAPACK designers' strategy to achieve this was to have manufacturers write fast BLAS, especially for the BLAS3. Then, LAPACK codes call the BLAS. *Ergo*, LAPACK gets high performance. In reality, two things go wrong. Manufacturers dont make much of an investment in their BLAS. And LAPACK does other things, so Amdahl's law applies.

4.6.2 Reinventing dense linear algebra optimization

In recent years, a new theory has emerged for achieving optimized dense linear algebra computation in a portable fashion. The theory is based one of the most fundabmental principles of computer science, recursion, yet it escaped experts for many years.

The Fundamental Triangle

In section 5, the memory hierarchy, and its affect on performance, is discussed. Hardware architecture elements, such as the memory hierarchy, forms just one apex of *The Fundamental Triangle*, the other two represented by software algorithms and the compilers. The Fundamental Triangle is a model for thinking about the performance of computer programs. A comprehensive evaluation of performance cannot be acheived without thinking about these three components and their relationship to each other. For example, as was noted earlier algorithm designers cannot assume that memory is infinite and that communication is costless, they must consider how their algorithms they write will behave within the memory hierarchy. This section will show how a focus on the

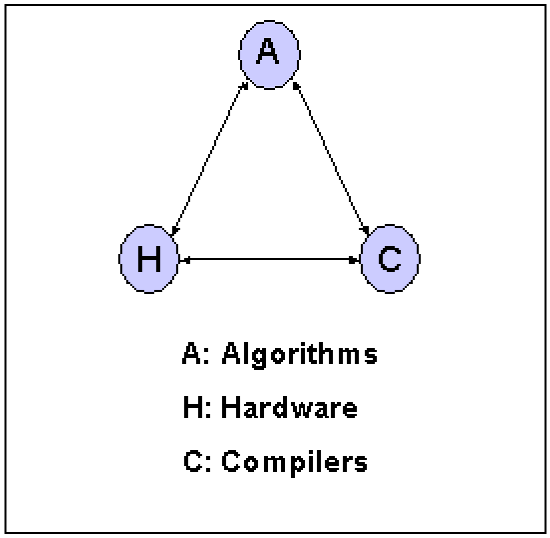


Figure 4.7: The Fundamental Triangle

interaction between algorithm and architecture can expose optimization possibilities. Figure 4.7 shows a graphical depiction of The Fundamental Triangle.

Examining dense linear algebra algorithms

Some scalar $a(i,j)$ algorithms may be expressed with square submatrix $A(I : * + NB - 1, J : J + NB - 1)$ algorithms. Also, dense matrix factorization is a BLAS level 3 computation consisting of a series of submatrix computations. Each submatrix computation is BLAS level 3, and each matrix operand in Level 3 is used multiple times. BLAS level 3 computation is $O(n^3)$ operations on $O(n^2)$ data. Therefore, in order to minimize the expense of moving data in and out of cache, the goal is to perform $O(n)$ operations per data movement, and amortize the expense over ther largest possible number of operations. The nature of dense linear algebra algorithms provides the opportunity to do just that, with the potential closeness of data within submatrices, and the frequent reuse of that data.

Architecture impact

The floating point arithmetic required for dense linear algebra computation is done in the L1 cache. Operands must be located in the L1 cache in order for multiple reuse of the data to yield peak performance. Moreover, operand data must map well into the L1 cache if reuse is to be possible. Operand data is represented using Fortran/C 2-D arrays. Unfortunately, the matrices that these 2-D arrays represent, and their submatrices, do not map well into L1 cache. Since memory is one dimensional, only one dimension of these arrays can be contiguous. For Fortran, the columns are contiguous, and for C the rows are contiguous.

To deal with this issue, this theory proposes that algorithms should be modified to map the input data from the native 2-D array representation to contiguous submatrices that can fit into the L1 cache.

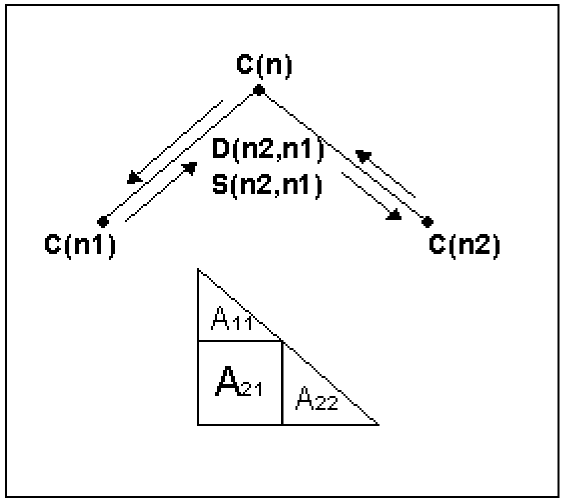


Figure 4.8: Recursive Submatrices

Blocking and Recursion

The principle of re-mapping data to form contiguous submatrices is known as blocking. The specific advantage blocking provides for minimizing data movement depends on the size of the block. A block becomes adventageous at minimizing data movement in and out of a level of the memory hierarchy when that entire block can fit in that level of the memory hierarcy in entirety. Therefore, for example, a certain size block would do well at minimizing register to cache data movement, and a different size block would do well at minimizing chach to memory data movement. However, the optimal size of these blocks is device dependent as it depends on the size of each level of the memory hierarchy. LAPACK does do some fixed blocking to improve performance, but its effectiveness is limited because the block size is fixed.

Writing dense linear algebra algorithms recursively enables automatic, variable blocking. Figure 4.8 shows how as the matrix is divided recursively into fours, blocking occurs naturally in sizes of $n, n/2, n/4...$ It is important to note that in order for these recursive blocks to be contiguous themselves, the 2-D data must be carefully mapped to one-dimensional storage memory. This data format is described in more detail in the next section.

The Recursive Block Format

The Recusive Block Format (RBF) maintains two dimensional data locality at every level of the one-dimensional tierd memory structure. Figure 4.9 shows the Recursive Block Format for a triangular matrix, an i right triangle of order N. Such a triange is converted to RBF by dividing each isoceles right triangle leg by two to get two smaller triangles and one "square"(rectangle).

Cholesky example

By utilizing the Recursive Block Format and by adopting a recursive strategy for dense linear algorithms, concise algorithms emerge. Figure 4.10 shows one node in the recursion tree of a recursive Cholesky algorithm. At this node, Cholesky is applied to a matrix of size n. Note that

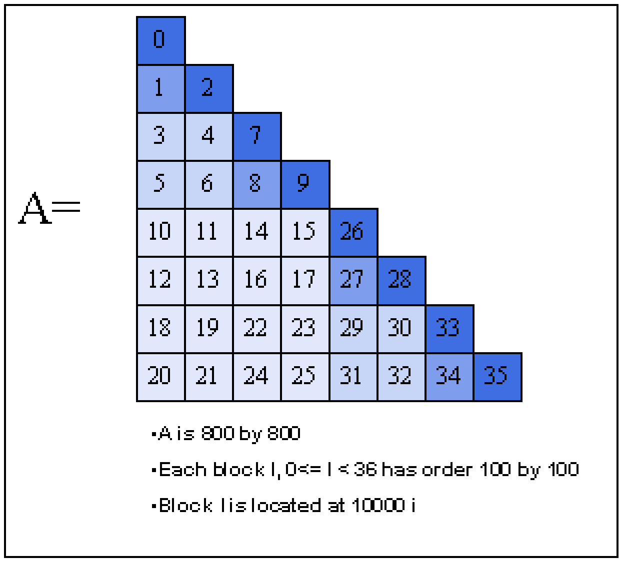


Figure 4.9: Recursive Block Format

n need not be the size of the original matrix, as this figure describes a node that could appear anywhere in the recursion tree, not just the root.

The lower triangular matrix below the Cholesky node describes the input matrix in terms of its recursive blocks, $A_{11}, A_{21}, and A_{22}$

- $n1$ is computed as $n1 = n/2$, and $n2 = n - n1$
- $C(n1)$ is computed recursively: Cholesky on submatrix A_{11}
- When $C(n1)$ has returned,L_{11} has been computed and it replaces A_{11}
- The DTRSM operation then computes $L_{21} = A_{21} L_{11T}{}^{-}1$
- L_{21} now replaces A_{21}
- The DSYRK operation uses L_{21} to do a rank $n1$ update of A_{22}
- $C(n2)$, Cholesky of the updated A_{22}, is now computed recursively, and L_{22} is returned

The BLAS operations (i.e.DTRSM and DSYRK) can be implemented using matrix multiply, and the operands to these operations are submatrices of A. This pattern generalizes to other dense linear algebra computations (i.e. general matrix factor, QR factorization). Every dense linear algebra algorithm calls the BLAS several times. Every one of the multiple BLAS calls has all of its matrix operands equal to the submatrices of the matrices, A,B, .. of the dense linear algebra algorithm. This pattern can be exploited to improve performance through the use of the Recursive Data Format.

A note on dimension theory

The reason why a theory such as the Recursive Data Format has utility for improving computational performance is because of the mis-match between the dimension of the data, and the dimension

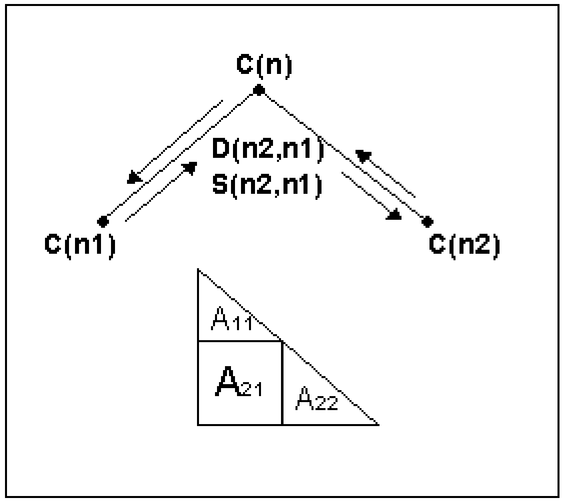


Figure 4.10: Recursive Cholesky

the hardware can represent. The laws of science and relate two and three dimensional objects. We live in a three dimensional world. However, computer storage is one dimensional. Moreover, mathmeticians have proved that it is not possible to maintain closeness between points in a neighborhood unless the two objects have the same dimension. Despite this negative theorem, and the limitations it implies on the relationship between data and available computer storage hardware, recursion provides a good approximation. Figure 4.11 shows this graphically via David Hilberts space filling curve.

4.7 Parallel computing considerations for dense linear algebra

Load Balancing:

We will use Gaussian elimination to demonstrate the advantage of cyclic distribution in dense linear algebra. If we carry out Gaussian elimination on a matrix with a one-dimensional block distribution, then as the computation proceeds, processors on the left hand side of the machine become idle after all their columns of the triangular matrices L and U have been computed. This is also the case for two-dimensional block mappings. This is poor load-balancing. With cyclic mapping, we balance the load much better.

In general, there are two methods to eliminate load imbalances:

- Rearrange the data for better load balancing (costs: communication).
- Rearrange the calculation: eliminate in unusual order.

So, should we convert the data from consecutive to cyclic order and from cyclic to consecutive when we are done? The answer is "no", and the better approach is to reorganize the algorithm rather than the data. The idea behind this approach is to regard matrix indices as a set (not necessarily ordered) instead of an ordered sequence.

In general if you have to rearrange the data, maybe you can rearrange the calculation.

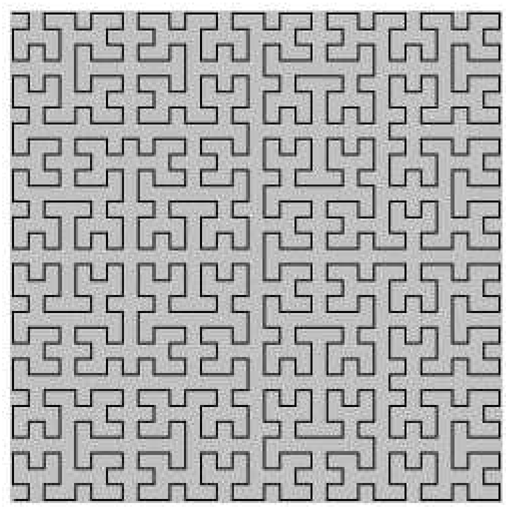


Figure 4.11: Hilbert Space Filling Curve

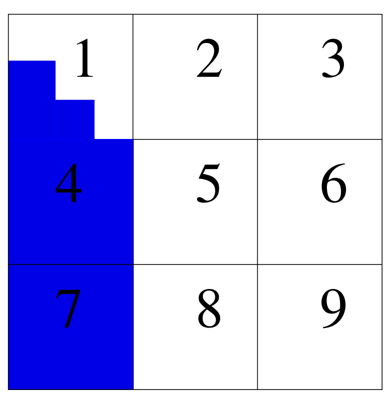


Figure 4.12: Gaussian elimination With Bad Load Balancing

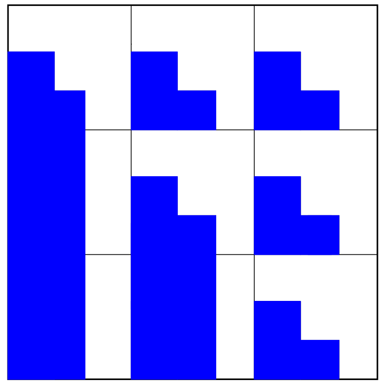


Figure 4.13: A stage in Gaussian elimination using cyclic order, where the shaded portion refers to the zeros and the unshaded refers to the non-zero elements

Lesson of Computation Distribution:

Matrix indices are a set (unordered), not a sequence (ordered). We have been taught in school to do operations in a linear order, but there is no mathematical reason to do this.

As Figure 4.9 demonstrates, we store data consecutively but do Gaussian elimination cyclicly. In particular, if the block size is 10×10, the pivots are 1, 11, 21, 31, …, 2, 22, 32, ….

We can apply the above reorganized algorithm in block form, where each processor does one block at a time and cycles through.

Here we are using all of our lessons, blocking for vectorization, and rearrangement of the calculation, not the data.

4.8 Better load balancing

In reality, the load balancing achieved by the two-dimensional cyclic mapping is not all that one could desire. The problem comes from the fact that the work done by a processor that owns A_{ij} is a function of i and j, and in fact grows *quadratically* with i. Thus, the cyclic mapping tends to overload the processors with a larger first processor index, as these tend to get matrix rows that are lower and hence more expensive. A better method is to map the matrix rows to the processor rows using some heuristic method to balance the load. Indeed, this is a further extension of the moral above – the matrix row and column indices do not come from any natural ordering of the equations and unknowns of the linear system – equation 10 has no special affinity for equations 9 and 11.

4.8.1 Problems

1. For performance analysis of the Gaussian elimination algorithm, one can ignore the operations performed outside of the inner loop. Thus, the algorithm is equivalent to

```
do k = 1, n
    do j = k,n
        do i = k,n
           a(i,j) = a(i,j) - a(i,k) * a(k,j)
        enddo
```

```
    enddo
enddo
```

The "owner" of $a(i,j)$ gets the task of the computation in the inner loop, for all $1 \leq k \leq \min(i,j)$.

Analyze the load imbalance that occurs in one-dimensional block mapping of the columns of the matrix: $n = bp$ and processor r is given the contiguous set of columns $(r-1)b+1, \ldots, rb$. (Hint: Up to low order terms, the average load per processor is $n^3/(3p)$ inner loop tasks, but the most heavily loaded processor gets half again as much to do.)

Repeat this analysis for the two-dimensional block mapping. Does this imbalance affect the scalability of the algorithm? Or does it just make a difference in the efficiency by some constant factor, as in the one-dimensional case? If so, what factor?

Finally, do an analysis for the two-dimensional cyclic mapping. Assume the $p = q^2$, and that $n = bq$ for some blocksize b. Does the cyclic method remove load imbalance completely?

Lecture 5

Sparse Linear Algebra

The solution of a linear system $Ax = b$ is one of the most important computational problems in scientific computing. As we shown in the previous section, these linear systems are often derived from a set of *differential equations*, by either finite difference or finite element formulation over a discretized mesh.

The matrix A of a discretized problem is usually very *sparse*, namely it has enough zeros that can be taken advantage of algorithmically. Sparse matrices can be divided into two classes: *structured sparse matrices* and *unstructured sparse matrices*. A structured matrix is usually generated from a structured regular grid and an unstructured matrix is usually generated from a non-uniform, unstructured grid. Therefore, sparse techniques are designed in the simplest case for structured sparse matrices and in the general case for unstructured matrices.

5.1 Cyclic Reduction for Structured Sparse Linear Systems

The simplest structured linear system is perhaps the tridiagonal system of linear equations $Ax = b$ where A is symmetric and positive definite and of form

$$A = \begin{pmatrix} b_1 & c_1 & & & \\ c_1 & b_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & c_{n-2} & b_{n-1} & c_{n-1} \\ & & & c_{n-1} & b_n \end{pmatrix}$$

For example, the finite difference formulation of the one dimensional model problems

$$-u''(x) + \sigma u(x) = f(x), \qquad 0 < x < 1, \sigma \geq 0 \tag{5.1}$$

subject to the boundary conditions $u(0) = u(1) = 0$, on a uniform discretization of spacing h yields of a triangular linear system of $n = 1/h$ variables, where $b_i = 2 + \sigma h^2$ and $c_i = -1$ for all $1 \leq i \leq n$.

Sequentially, we can solve a triangular linear system $Ax = b$ by factor A into $A = LDL^T$, where D is a diagonal matrix with diagonal $(d_1, d_2, ..., d_n)$ and L is of form

$$L = \begin{pmatrix} 1 & 0 & & \\ e_1 & 1 & 0 & \\ & \ddots & \ddots & \ddots \\ & & e_{n-1} & 1 \end{pmatrix}.$$

The factorization can be computed by the following simple algorithm.

Algorithm Sequential Tridiagonal Solver

1. $d_1 = b_1$
2. $e = c_1/d_1$
3. for $i = 2 : n$
   (a) $d_1 = b_i - e_{i-1}c_{i-1}$
   (b) if $i < n$ then $e_i = c_i/d_i$

The number float point operations is $3n$ upto a additive constant. With such factorization, we can then solve the tridiagonal linear system in additional $5n$ float point operations. However, this method is very reminiscent to the naive sequential algorithm for the prefix sum whose computation graph has a critical path of length $O(n)$. The cyclic reduction, developed by Golub and Hockney [**?**], is very similar to the parallel prefix algorithm presented in Section **??** and it reduces the length of dependency in the computational graph to the smallest possible.

The basic idea of the cyclic reduction is to first eliminate the odd numbered variables to obtain a tridiagonal linear system of $\lceil n/2 \rceil$ equations. Then we solve the smaller linear system recursively. Note that each variable appears in three equations. The elimination of the odd numbered variables gives a tridiagonal system over the even numbered variables as following:

$$c'_{2i-2}x_{2i-2} + b'_{2i}x_{2i} + c'_{2i}x_{2i+2} = f'_{2i},$$

for all $2 \leq i \leq n/2$, where

$$\begin{aligned} c'_{2i-2} &= -(c_{2i-2}c_{2i-1}/b_{2i-1}) \\ b'_{2i} &= (b_{2i} - c^2_{2i-1}/b_{2i-1} - c^2 2i/b_{2i+1}) \\ c'_{2i} &= c_{2i}c_{2i+1}/b_{2i+1} \\ f'_{2i} &= f_{2i} - c_{2i-1}f_{2i-1}/b_{2i-1} - c_{2i}f_{2i+1}/b_{2i+1} \end{aligned}$$

Recursively solving this smaller linear tridiagonal system, we obtain the value of x_{2i} for all $i = 1, ..., n/2$. We can then compute the value of x_{2i-1} by the simple equation:

$$x_{2i-1} = (f_{2i-1} - c_{2i-2}x_{2i-2} - c_{2i-1}x_{2i})/b_{2i-1}.$$

By simple calculation, we can show that the total number of float point operations is equal to $16n$ upto an additive constant. So the amount of total work is doubled compare with the sequential algorithm discussed. But the length of the critical path is reduced to $O(\log n)$. It is worthwhile to point out the the total work of the parallel prefix sum algorithm also double that of the sequential algorithm. Parallel computing is about the trade-off of parallel time and the total work. The discussion show that if we have n processors, then we can solve a tridiagonal linear system in $O(\log n)$ time.

When the number of processor p is much less than n, similar to prefix sum, we hybrid the cyclic reduction with the sequential factorization algorithm. We can show that the parallel float point operations is bounded by $16n(n + \log n)/p$ and the number of round of communication is bounded by $O(\log p)$. The communication pattern is the nearest neighbor.

Cyclic Reduction has been generalized to two dimensional finite difference systems where the matrix is a block tridiagonal matrix.

5.2 Sparse Direct Methods

Direct methods for solving sparse linear systems are important because of their generality and robustness. For linear systems arising in certain applications, such as linear programming and some structural engineering applications, they are the only feasible methods for numerical factorization.

5.2.1 LU Decomposition and Gaussian Elimination

The basis of direct methods for linear system is Gaussian Elimination, a process where we zero out certain entry of the original matrix in a systematically way. Assume we want to solve $Ax = b$ where A is a sparse $n \times n$ symmetric positive definite matrix. The basic idea of the direct method is to factor A into the product of triangular matrices $A = LL^T$. Such a procedure is called *Cholesky factorization*.

The first step of the Cholesky factorization is given by the following matrix fractorization:

$$A = \begin{pmatrix} d & v^T \\ v & C \end{pmatrix} = \begin{pmatrix} \sqrt{d} & 0 \\ v/\sqrt{d} & I \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & C - (vv^T)/d \end{pmatrix} \begin{pmatrix} \sqrt{d} & v^T/\sqrt{d} \\ 0 & I \end{pmatrix}$$

where v is $n-1 \times 1$ and C is $n-1 \times n-1$. Note that d is positive since A is positive definite. The term $C - \frac{vv^t}{d}$ is the Schur complement of A. This step is called elimination and the element d is the pivot. The above decomposition is now carried out on the Schur complement recursively. We therefore have the following algorithm for the Cholesky decomposition.

For $k = 1, 2, \ldots, n$
  $a(k,k) = \sqrt{a(k,k)}$
  $a(k+1:n,k) = \frac{a(k+1:n,k)}{a(k,k)}$
  $a(k+1:n,k+1:n) = a(k+1:n,k+1:n) - a(k+1:n,k)^T a(k+1:n,k)$
end

The entries on and below the diagonal of the resulting matrix are the entries of L. The main step in the algorithm is a rank 1 update to an $n-1 \times n-1$ block.

Notice that some fill-in may occur when we carry out the decomposition. i.e., L may be significantly less sparse than A. An important problem in direct solution to sparse linear system is to find a "good" ordering of the rows and columns of the matrix to reduce the amount of fill.

As we showed in the previous section, the matrix of the linear system generated by the finite element or finite difference formulation is associated with the graph given by the mesh. In fact, the nonzero structure of each matrix A can be represented by a graph, $G(A)$, where the rows are represented by a vertex and every nonzero element by an edge. An example of a sparse matrix and its corresponding graph is given in Figure 5.1. Note that nonzero entries are marked with a symbol, whereas zero entries are not shown.

The fill-in resulting from Cholesky factorization is also illustrated in Figure 5.1. The new graph $G^+(A)$ can be computed by looping over the nodes j, in order of the row operations, and adding edges between j's higher-numbered neighbors.

In the context of parallel computation, an important parameter the height of elimination tree, which is the number of parallel elimination steps need to factor with an unlimited number of processors. The *elimination tree* defined as follows from the fill-in calculation which was described above. Let $j > k$. Define $j >_L k$ if $l_{jk} \neq 0$ where l_{jk} is the (j,k) entry of L, the result of the decomposition. Let the parent of k, $p(k)$ =min$\{j : j >_L k\}$. This defines a tree since if $\beta >_L \alpha$, $\gamma >_L \alpha$ and $\gamma > \beta$ then $\gamma >_L \beta$. The elimination tree corresponding to our matrix is shown in Figure 5.2.

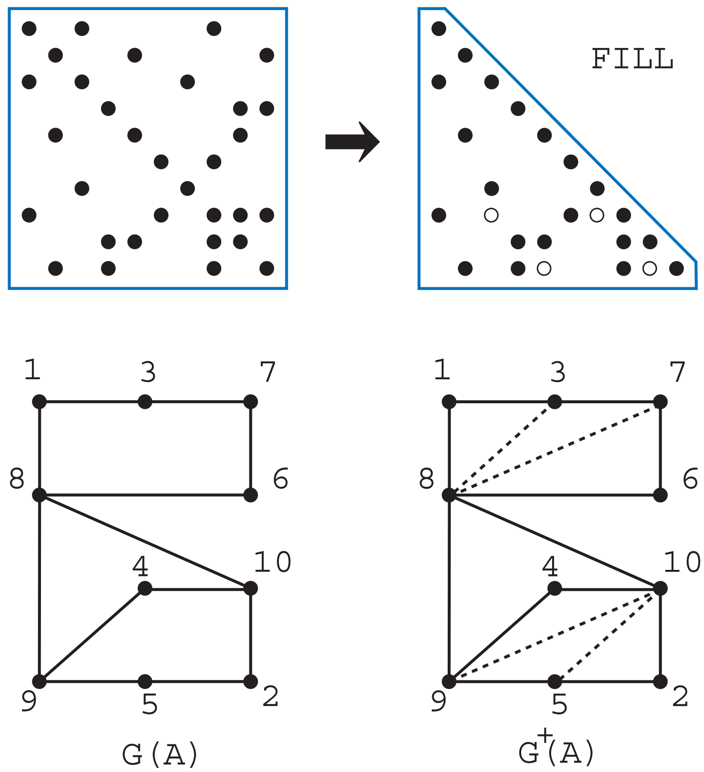


Figure 5.1: Graphical Representation of Fill-in

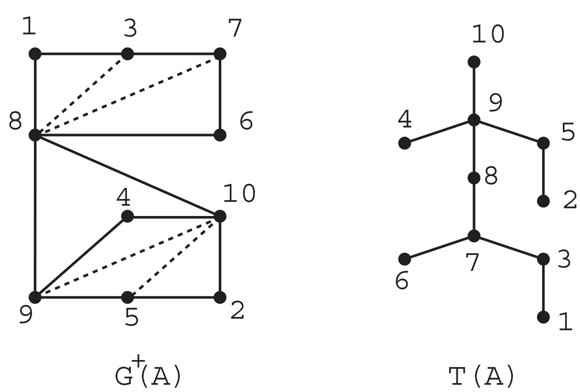


Figure 5.2: The Elimination Tree

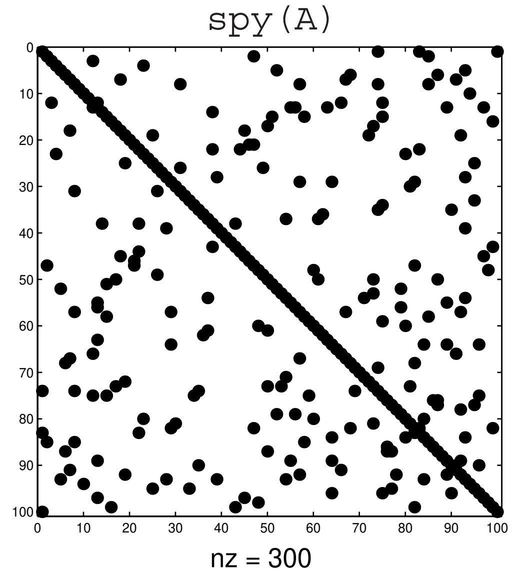


Figure 5.3: Sparsity Structure of Semi-Random Symmetric Matrix

The order of elimination determines both fill and elimination tree height. Unfortunately, but inevitably, finding the best ordering is NP-complete. Heuristics are used to reduce fill-in. The following lists some commonly used ones.

- Ordering by minimum degree (this is SYMMMD in Matlab)
- nested dissection
- Cuthill-McKee ordering.
- reverse Cuthill-McKee (SYMRCM)
- ordering by number of non-zeros (COLPERM or COLMMD)

These ordering heuristics can be investigated in Matlab on various sparse matrices. The simplest way to obtain a random sparse matrix is to use the command `A=sprand(n,m,f)`, where n and m denote the size of the matrix, and f is the fraction of nonzero elements. However, these matrices are not based on any physical system, and hence may not illustrate the effectiveness of an ordering scheme on a real world problem. An alternative is to use a database of sparse matrices, one of which is available with the command/package `ufget`.

Once we have a sparse matrix, we can view it's sparsity structure with the command `spy(A)`. An example with a randomly generated symmetric sparse matrix is given in Figure 5.3.

We now carry out Cholesky factorization of A using no ordering, and using SYMMMD. The sparsity structures of the resulting triangular matrices are given in Figure 5.4. As shown, using a heuristic-based ordering scheme results in significantly less fill in. This effect is usually more pronounced when the matrix arises from a physical problem and hence has some associated structure.

We now examine an ordering method called *nested dissection*, which uses vertex separators in a divide-and-conquer node ordering for sparse Gaussian elimination. Nested dissection [37, 38, 59] was originally a sequential algorithm, pivoting on a single element at a time, but it is an attractive

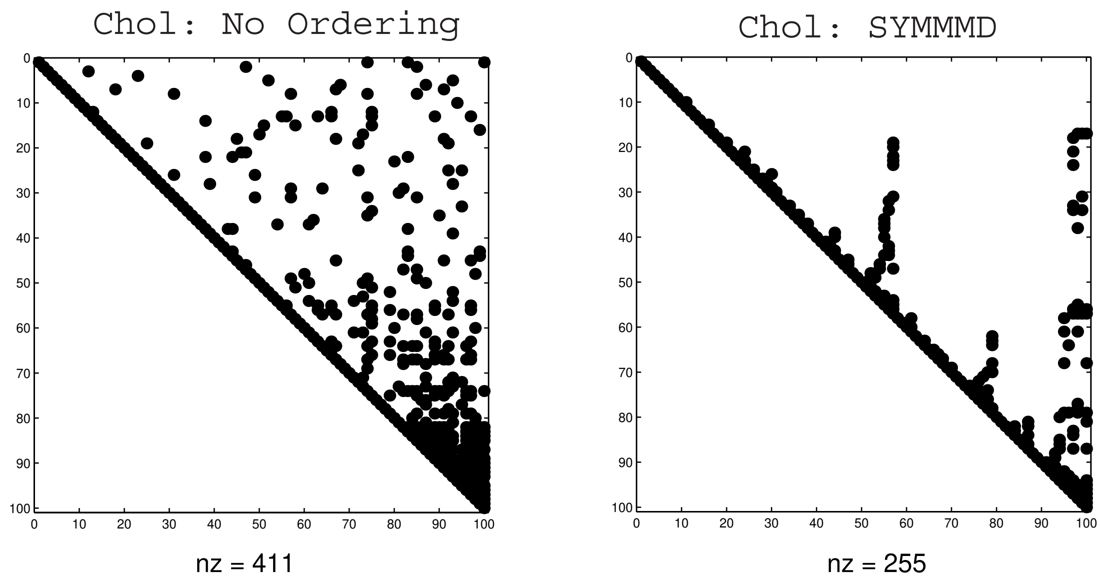


Figure 5.4: Sparsity Structure After Cholesky Factorization

parallel ordering as well because it produces blocks of pivots that can be eliminated independently in parallel [9, 29, 40, 61, 74].

Consider a regular finite difference grid. By dissecting the graph along the center lines (enclosed in dotted curves), the graph is split into four independent graphs, each of which can be solved in parallel.

The connections are included only at the end of the computation in an analogous way to domain decomposition discussed in earlier lectures. Figure 5.6 shows how a single domain can be split up into two roughly equal sized domains **A** and **B** which are independent and a smaller domain **C** that contains the connectivity.

One can now recursively order **A** and **B**, before finally proceeding to **C**. More generally, begin by recursively ordering at the leaf level and then continue up the tree. The question now arises as to how much fill is generated in this process. A recursion formula for the fill F generated for such

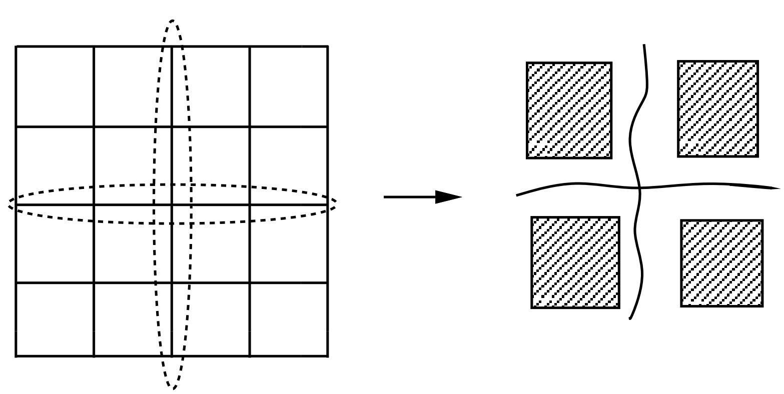


Figure 5.5: Nested Dissection

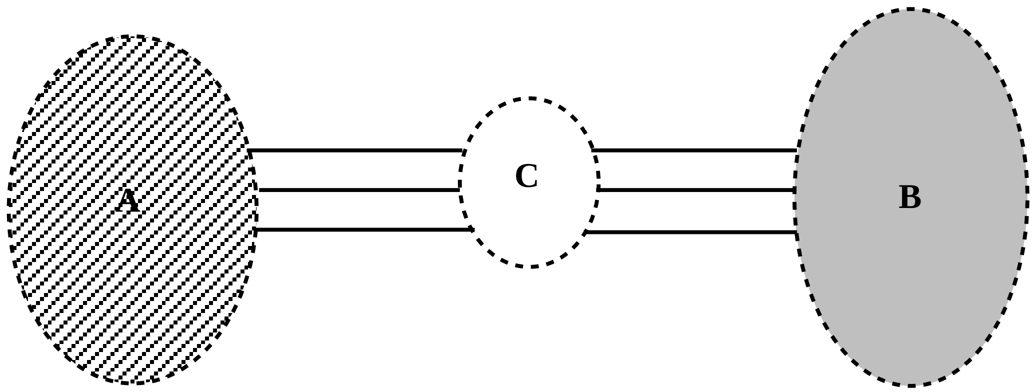


Figure 5.6: Vertex Separators

a 2-dimension nested dissection algorithm is readily derived.

$$F(n) = 4F(\frac{n}{2}) + \frac{(2\sqrt{n})^2}{2} \tag{5.2}$$

This yields upon solution

$$F(n) = 2n\log(n) \tag{5.3}$$

In an analogous manner, the elimination tree height is given by:

$$H(n) = H(\frac{n}{2}) + 2\sqrt{n} \tag{5.4}$$

$$H(n) = const \times \sqrt{n} \tag{5.5}$$

Nested dissection can be generalized to three dimensional regular grid or other classes of graphs that have small separators. We will come back to this point in the section of graph partitioning.

5.2.2 Parallel Factorization: the Multifrontal Algorithm

Nested dissection and other heuristics give the ordering. To factor in parallel, we need not only find a good ordering in parallel, but also to perform the elimination in parallel. To achieve better parallelism and scalability in elimination, a popular approach is to modify the algorithm so that we are performing a rank k update to an $n-k \times n-k$ block. The basic step will now be given by

$$A = \begin{pmatrix} D & V^T \\ V & C \end{pmatrix} = \begin{pmatrix} L_D & 0 \\ VL_D^{-T} & I \end{pmatrix} \begin{pmatrix} I & 0 \\ 0 & C - VD^{-1}V^T \end{pmatrix} \begin{pmatrix} L_D^T & L_D^{-1}V^T \\ 0 & I \end{pmatrix}$$

where C is $n-k \times n-k$, V is $n-k \times k$ and $D = L_D L_D^T$ is $k \times k$. D can be written in this way since A is positive definite. Note that $VD^{-1}V^T = (VL_D^{-T})(L_D^{-1}V^T)$.

The elimination tree shows where there is parallelism since we can "go up the separate branches in parallel." i.e. We can update a column of the matrix using only the columns below it in the elimination tree. This leads to the multifrontal algorithm. The sequential version of the algorithm is given below. For every column j there is a block $\bar{U}_j$ (which is equivalent to $VD^{-1}V^T$).

$$\bar{U}_j = -\sum_k \begin{pmatrix} l_{jk} \\ l_{i_1k} \\ \vdots \\ l_{i_rk} \end{pmatrix} (l_{jk}\ l_{i_1k}\ \ldots\ l_{i_rk})$$

where the sum is taken over all descendants of j in the elimination tree. $j, i_1, i_2, \ldots, i_r$ are the indices of the non-zeros in column j of the Cholesky factor.

For $j = 1, 2, \ldots, n$. Let $j, i_1, i_2, \ldots, i_r$ be the indices of the non-zeros in column j of L. Let $c_1, \ldots, c_s$ be the children of j in the elimination tree. Let $\bar{U} = U_{c_1} \updownarrow \ldots \updownarrow U_{c_s}$ where the U_i's were defined in a previous step of the algorithm. $\updownarrow$ is the extend-add operator which is best explained by example. Let

$$R = \begin{array}{c c} & \begin{array}{cc} 5 & 8 \end{array} \\ \begin{array}{c} 5 \\ 8 \end{array} & \begin{pmatrix} p & q \\ u & v \end{pmatrix} \end{array}, S = \begin{array}{c c} & \begin{array}{cc} 5 & 9 \end{array} \\ \begin{array}{c} 5 \\ 9 \end{array} & \begin{pmatrix} w & x \\ y & z \end{pmatrix} \end{array}$$

(The rows of R correspond to rows 5 and 8 of the original matrix etc.) Then

$$R \updownarrow S = \begin{array}{c c} & \begin{array}{ccc} 5 & 8 & 9 \end{array} \\ \begin{array}{c} 5 \\ 8 \\ 9 \end{array} & \begin{pmatrix} p+w & q & x \\ u & v & 0 \\ y & 0 & z \end{pmatrix} \end{array}$$

Define

$$F_j = \begin{pmatrix} a_{jj} & \ldots & a_{ji_r} \\ \vdots & \ddots & \\ a_{i_rj} & \ldots & a_{i_ri_r} \end{pmatrix} \updownarrow \bar{U}$$

(This corresponds to $C - VD^{-1}V^T$)
Now factor F_j

$$\begin{pmatrix} l_{jj} & 0 & \ldots & 0 \\ l_{i_1j} & & & \\ \vdots & & I & \\ l_{i_rj} & & & \end{pmatrix} \begin{pmatrix} 1 & 0 & \ldots & 0 \\ 0 & & & \\ \vdots & & U_j & \\ 0 & & & \end{pmatrix} \begin{pmatrix} l_{jj} & l_{i_1j} & \ldots & l_{i_rj} \\ 0 & & & \\ \vdots & & I & \\ 0 & & & \end{pmatrix}$$

(Note that U_j has now been defined.)

We can use various BLAS kernels to carry out this algorithm. Recently, Kumar and Karypis have shown that direct solver can be parallelized efficiently. They have designed a parallel algorithm for factorization of sparse matrices that is more scalable than any other known algorithm for this problem. They have shown that our parallel Cholesky factorization algorithm is asymptotically as scalable as any parallel formulation of dense matrix factorization on both mesh and hypercube architectures. Furthermore, their algorithm is equally scalable for sparse matrices arising from two- and three-dimensional finite element problems.

They have also implemented and experimentally evaluated the algorithm on a 1024-processor nCUBE 2 parallel computer and a 1024-processor Cray T3D on a variety of problems. In structural engineering problems (Boeing-Harwell set) and matrices arising in linear programming (NETLIB set), the preliminary implementation is able to achieve 14 to 20 GFlops on a 1024-processor Cray T3D.

In its current form, the algorithm is applicable only to Cholesky factorization of sparse symmetric positive definite (SPD) matrices. SPD systems occur frequently in scientific applications and are the most benign in terms of ease of solution by both direct and iterative methods. However, there are many applications that involve solving large sparse linear systems which are not SPD. An efficient parallel algorithm for a direct solution to non-SPD sparse linear systems will be extremely valuable because the theory of iterative methods is far less developed for general sparse linear systems than it is for SPD systems.

5.3 Basic Iterative Methods

These methods will focus on the solution to the linear system $Ax = b$ where $A \in \mathcal{R}^{n \times n}$ and $x, b \in \mathcal{R}^n$, although the theory is equally valid for systems with complex elements.

The basic outline of the iterative methods is as follows: Choose some initial guess, x_o, for the solution vector. Generate a series of solution vectors, $\{x_1, x_2, \ldots, x_k\}$, through an iterative process taking advantage of previous solution vectors.

Define x^* as the true (optimal) solution vector. Each iterative solution vector is chosen such that the absolute error, $e_i = \|x^* - x_i\|$, is decreasing with each iteration for some defined norm. Define also the residual error, $r_i = \|b - Ax_i\|$, at each iteration. These error quantities are clearly related by a simple transformation through A.

$$r_i = b - Ax_i = Ax^* - Ax_i = Ae_i$$

5.3.1 SuperLU-dist

SuperLU-dist is an iterative and approximate method for solving $Ax = b$. This simple algorithm eliminates the need for pivoting. The elimination of pivoting enhances parallel implementations due to the high communications overhead that pivoting imposes. The basic SuperLU-dist algorithm is as follows:

Algorithm: SuperLU-dist

1. $r = b - A * x$
2. $backerr = max_i(\frac{r_i}{(|A| * |x| + |b|)_i})$
3. if $(backerr < \epsilon)$ or $(backerr > \frac{lasterr}{2})$ then stop
4. solve: $L * U * dx = r$
5. $x = x + dx$
6. $lasterr = backerr$
7. loop to step 1

In this algorithm, x, L, and U are approximate while r is exact. This procedure usually converges to a reasonable solution after only 0-3 iterations and the error is on the order of 10^{-n} after n iterations.

5.3.2 Jacobi Method

Perform the matrix decomposition $A = D - L - U$ where D is some diagonal matrix, L is some strictly lower triangular matrix and U is some strictly upper triangular matrix.

Any solution satisfying $Ax = b$ is also a solution satisfying $Dx = (L + U)x + b$. This presents a straightforward iteration scheme with small computation cost at each iteration. Solving for the solution vector on the right-hand side involves inversion of a diagonal matrix. Assuming this inverse exists, the following iterative method may be used.

$$x_i = D^{-1}(L + U)\,x_{i-1} + D^{-1}b$$

This method presents some nice computational features. The inverse term involves only the diagonal matrix, D. The computational cost of computing this inverse is minimal. Additionally, this may be carried out easily in parallel since each entry in the inverse does not depend on any other entry.

5.3.3 Gauss-Seidel Method

This method is similar to Jacobi Method in that any solution satisfying $Ax = b$ is now a solution satisfying $(D - L)x = Ub$ for the $A = D - L - U$ decomposition. Assuming an inverse exists, the following iterative method may be used.

$$x_i = (D - L)^{-1}Ux_{i-1} + (D - L)^{-1}$$

This method is often stable in practice but is less easy to parallelize. The inverse term is now a lower triangular matrix which presents a bottleneck for parallel operations.

This method presents some practical improvements over the Jacobi method. Consider the computation of the j^{th} element of the solution vector x_i at the i^{th} iteration. The lower triangular nature of the inverse term demonstrates only the information of the $(j + 1)^{th}$ element through the n^{th} elements of the previous iteration solution vector x_{i-1} are used. These elements contain information not available when the j^{th} element of x_{i-1} was computed. In essence, this method updates using only the most recent information.

5.3.4 Splitting Matrix Method

The previous methods are specialized cases of Splitting Matrix algorithms. These algorithms utilize a decomposition $A = M - N$ for solving the linear system $Ax = b$. The following iterative procedure is used to compute the solution vector at the i^{th} iteration.

$$Mx_i = Nx_{i-1} + b$$

Consider the computational tradeoffs when choosing the decomposition.

- cost of computing M^{-1}
- stability and convergence rate

It is interesting the analyze convergence properties of these methods. Consider the definitions of absolute error, $e_i = x^* - x_i$, and residual error, $r_i = Ax_i - b$. An iteration using the above algorithm yields the following.

$$\begin{aligned} x_1 &= M^{-1}Nx_0 + M^{-1}b \\ &= M^{-1}(M-A)x_0 + M^{-1}b \\ &= x_0 + M^{-1}r_0 \end{aligned}$$

A similar form results from considering the absolute error.

$$\begin{aligned} x^* &= x_0 + e_0 \\ &= x_0 + A^{-1}r_0 \end{aligned}$$

This shows that the convergence of the algorithm is in some way improved if the M^{-1} term approximates A^{-1} with some accuracy. Consider the amount of change in the absolute error after this iteration.

$$\begin{aligned} e_1 &= A^{-1}r_0 - M^{-1}r_0 \\ &= e_0 - M^{-1}Ae_0 \\ &= M^{-1}Ne_0 \end{aligned}$$

Evaluating this change for a general iteration shows the error propagation.

$$e_i = \left(M^{-1}N\right)^i e_0$$

This relationship shows a bound on the error convergence. The largest eigenvalue, or spectral eigenvalue, of the matrix $M^{-1}N$ determines the rate of convergence of these methods. This analysis is similar to the solution of a general difference equation of the form $x_k = Ax_{k-1}$. In either case, the spectral radius of the matrix term must be less than 1 to ensure stability. The method will converge to 0 faster if all the eigenvalue are clustered near the origin.

5.3.5 Weighted Splitting Matrix Method

The splitting matrix algorithm may be modified by including some scalar weighting term. This scalar may be likened to the free scalar parameter used in practical implementations of Newton's method and Steepest Descent algorithms for optimization programming. Choose some scalar, w, such that $0 < w < 1$, for the following iteration.

$$\begin{aligned} x_i &= (1-w)x_0 + w\left(x_0 + M^{-1}v_0\right) \\ &= x_0 + wM^{-1}v_0 \end{aligned}$$

5.4 Red-Black Ordering for parallel Implementation

The concept of ordering seeks to separate the nodes of a given domain into subdomains. Red-black ordering is a straightforward way to achieve this. The basic concept is to alternate assigning a "color" to each node. Consider the one- and two-dimensional examples on regular grids..

The iterative procedure for these types of coloring schemes solves for variables at nodes with a certain color, then solves for variables at nodes of the other color. A linear system can be formed with a block structure corresponding to the color scheme.

$$\begin{bmatrix} \text{BLACK} & \text{MIXED} \\ \text{MIXED} & \text{RED} \end{bmatrix}$$

This method can easily be extended to include more colors. A common practice is to choose colors such that no nodes has neighbors of the same color. It is desired in such cases to minimize the number of colors so as to reduce the number of iteration steps.

5.5 Conjugate Gradient Method

The Conjugate Gradient Method is the most prominent iterative method for solving sparse symmetric positive definite linear systems. We now examine this method from parallel computing perspective. The following is a copy of a pseudocode for the conjugate gradient algorithm.
Algorithm: Conjugate Gradient

1. $\underline{x_0} = 0, \quad r_0 = b - A\underline{x_0} = b$
2. do $m = 1$, to n steps
   (a) if $m = 1$, then $p_1 = r_0$
   else
   $\beta = r_{m-1}^T r_{m-1} / r_{m-2}^T r_{m-2}$
   $p_m = r_{m-1} + \beta p_{m-1}$
   endif
   (b) $\alpha_m = r_{m-1}^T r_{m-1} / p_m^T A p_m$
   (c) $x_m = x_{m-1} + \alpha_m p_m$
   (d) $r_m = r_{m-1} - \alpha_m A p_m$

When A is symmetric positive definite, the solution of $Ax = b$ is equivalent to find a solution to the following quadratic minimization problem.

$$\min_{\underline{x}} \phi(x) = \frac{1}{2} x^T A x - x^T b.$$

In this setting, $r_0 = -\nabla\phi$ and $p_i^T A p_j = 0$, i.e., p_i^T and p_j are *conjugate* with respect to A.
How many iterations shall we perform and how to reduce the number of iterations?

Theorem 5.5.1 *Suppose the condition number is $\kappa(A) = \lambda_{max}(A)/\lambda_{min}(A)$, since A is Symmetric Positive Definite, $\forall x_0$, suppose x^* is a solution to $Ax = b$, then*

$$||x^* - x_m||_A \leq 2||x^* - x_0||_A (\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1})^m,$$

where $||V||_A = V^T A V$

Therefore, $||e_m|| \leq 2||e_0|| \cdot (\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1})^m$.

Another high order iterative method is Chebyshev iterative method. We refer interested readers to the book by Own Axelsson (Iterative Solution Methods, Cambridge University Press). Conjugate gradient method is a special Krylov subspace method. Other examples of Krylov subspace are GMRES (Generalized Minimum Residual Method) and Lanczos Methods.

5.5.1 Parallel Conjugate Gradient

Within each iteration of the conjugate gradient algorithm a single matrix-vector product must be taken. This calculation represents a bottleneck and the performance of conjugate gradient can be improved by parallelizing this step.

First, the matrix (A), vector (x), and solution vector (y) are laid out by rows across multiple processors as shown in Figure 5.7.

The algorithm for the distributed calculation is then simple: On each processor j, broadcast $x(j)$ and then compute $y(j) = A(j,:) * x$.

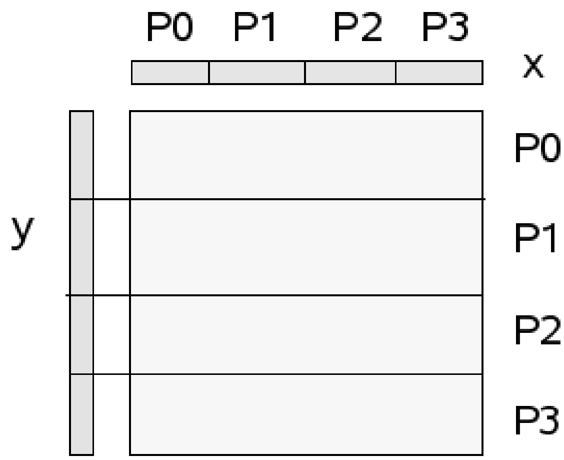


Figure 5.7: Example distribution of A, x, and b on four processors

5.6 Preconditioning

Preconditioning is important in reducing the number of iterations needed to converge in many iterative methods. Put more precisely, preconditioning makes iterative methods possible in practice. Given a linear system $Ax = b$ a parallel preconditioner is an invertible matrix C satisfying the following:

1. The inverse C^{-1} is relatively easy to compute. More precisely, after preprocessing the matrix C, solving the linear system $Cy = b'$ is much easier than solving the system $Ax = b$. Further, there are fast parallel solvers for $Cy = b'$.

2. Iterative methods for solving the system $C^{-1}Ax = C^{-1}b$, such as, conjugate gradient[1] should converge much more quickly than they would for the system $Ax = b$.

Generally a preconditioner is intended to reduce $\kappa(A)$.

Now the question is: *how to choose a preconditioner C?* There is no definite answer to this. We list some of the popularly used preconditioning methods.

- The basic splitting matrix method and SOR can be viewed as preconditioning methods.

- **Incomplete factorization preconditioning**: the basic idea is to first choose a good "sparsity pattern" and perform factorization by Gaussian elimination. The method rejects those fill-in entries that are either small enough (relative to diagonal entries) or in position outside the sparsity pattern. In other words, we perform an approximate factorization L^*U^* and use this product as a preconditioner. One effective variant is to perform block incomplete factorization to obtain a preconditioner.

  The incomplete factorization methods are often effective when tuned for a particular application. The methods also suffer from being too highly problem-dependent and the condition number usually improves by only a constant factor.

[1]In general the matrix $C^{-1}A$ is not symmetric. Thus the formal analysis uses the matrix LAL^T where $C^{-1} = LL^T$ [?].

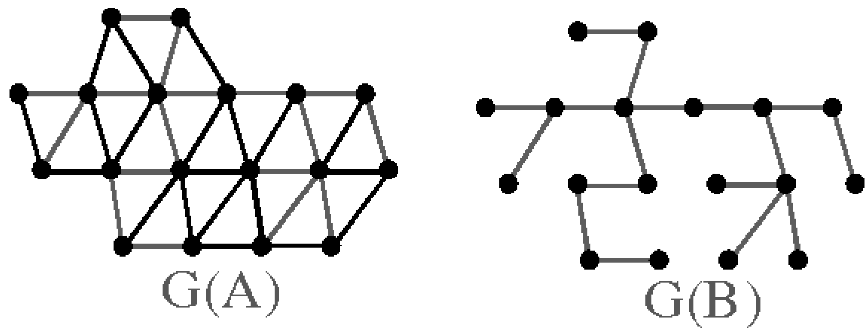


Figure 5.8: Example conversion of the graph of matrix A (G(A)) to a subgraph (G(B))

- **Subgraph preconditioning**: The basic idea is to choose a subgraph of the graph defined by the matrix of the linear system so that the linear system defined by the subgraph can be solved efficiently and the edges of the original graph can be embedded in the subgraph with small congestion and dilation, which implies small condition number of the preconditioned matrix. In other words, the subgraph can "support" the original graph. An example of converting a graph to a subgraph is shown in Figure 5.8.

  The subgraph can be factored in $O(n)$ space and time and applying the preconditioner takes $O(n)$ time per iteration.

- **Block diagonal preconditioning**: The observation of this method is that a matrix in many applications can be naturally partitioned in the form of a 2×2 blocks

  $$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

  Moreover, the linear system defined by A_{11} can be solved more efficiently. Block diagonal preconditioning chooses a preconditioner with format

  $$C = \begin{pmatrix} B_{11} & 0 \\ 0 & B_{22} \end{pmatrix}$$

  with the condition that B_{11} and B_{22} are symmetric and

  $$\alpha_1 A_{11} \leq B_{11} \leq \alpha_2 A_{11}$$
  $$\beta_1 A_{22} \leq B_{22} \leq \beta_2 A_{22}$$

  Block diagonal preconditioning methods are often used in conjunction with domain decomposition technique. We can generalize the 2-block formula to multi-blocks, which correspond to multi-region partition in the domain decomposition.

- **Sparse approximate inverses**: Sparse approximate inverses (B^{-1}) of A can be computed such that $A \approx B^{-1}$. This inverse is computed explicitly and the quantity $||B^{-1}A - I||_F$ is minimized in parallel (by columns). This value of B^{-1} can then be used as a preconditioner. This method has the advantage of being very parallel, but suffers from poor effectiveness in some situations.

5.7 Symmetric Supernodes

The following Section on Symmetric Supernodes is an edited excerpt from "A Supernodal Approach to Sparse Partial Pivoting," by Demmel, Eisenstat, Gilbert, Li, and Liu.

The idea of a supernode is to group together columns with the same nonzero structure, so they can be treated as a dense matrix for storage and computation. In the factorization $A = LL^T$ (or $A = LDL^T$), a supernode is a range $(r : s)$ of columns of L with the same nonzero structure below the diagonal; that is, $L(r : s, r : s)$ is full lower triangular and every row of $L(s : n, r : s)$ is either full or zero.

All the updates from columns of a supernode are summed into a dense vector before the sparse update is performed. This reduces indirect addressing and allows the inner loops to be unrolled. In effect, a sequence of col-col updates is replaced by a supernode-column (sup-col) update. The sup-col update can be implemented using a call to a standard dense Level 2 BLAS matrix-vector multiplication kernel. This idea can be further extended to supernode-supernode (sup-sup) updates, which can be implemented using a Level 3 BLAS dense matrix-matrix kernel. This can reduce memory traffic by an order of magnitude, because a supernode in the cache can participate in multiple column updates. Ng and Peyton reported that a sparse Cholesky algorithm based on sup-sup updates typically runs 2.5 to 4.5 times as fast as a col-col algorithm.

To sum up, supernodes as the source of updates help because of the following:

1. The inner loop (over rows) has no indirect addressing. (Sparse Level 1 BLAS is replaced by dense Level 1 BLAS.)

2. The outer loop (over columns in the supernode) can be unrolled to save memory references. (Level 1 BLAS is replaced by Level 2 BLAS.)

   Supernodes as the destination of updates help because of the following:

3. Elements of the source supernode can be reused in multiple columns of the destination supernode to reduce cache misses. (Level 2 BLAS is replaced by Level 3 BLAS.)

Supernodes in sparse Cholesky can be determined during symbolic factorization, before the numeric factorization begins. However, in sparse LU, the nonzero structure cannot be predicted before numeric factorization, so we must identify supernodes on the fly. Furthermore, since the factors L and U are no longer transposes of each other, we must generalize the definition of a supernode.

5.7.1 Unsymmetric Supernodes

There are several possible ways to generalize the symmetric definition of supernodes to unsymmetric factorization. We define $F = L + U - I$ to be the *filled matrix* containing both L and U.

T1 Same row and column structures: A supernode is a range $(r : s)$ of columns of L and rows of U, such that the diagonal block $F(r : s, r : s)$ is full, and outside that block all the columns of L in the range have the same structure and all the rows of U in the range have the same structure. T1 supernodes make it possible to do sup-sup updates, realizing all three benefits.

T2 Same column structure in L: A supernode is a range $(r : s)$ of columns of L with triangular diagonal block full and the same structure below the diagonal block. T2 supernodes allow sup-col updates, realizing the first two benefits.

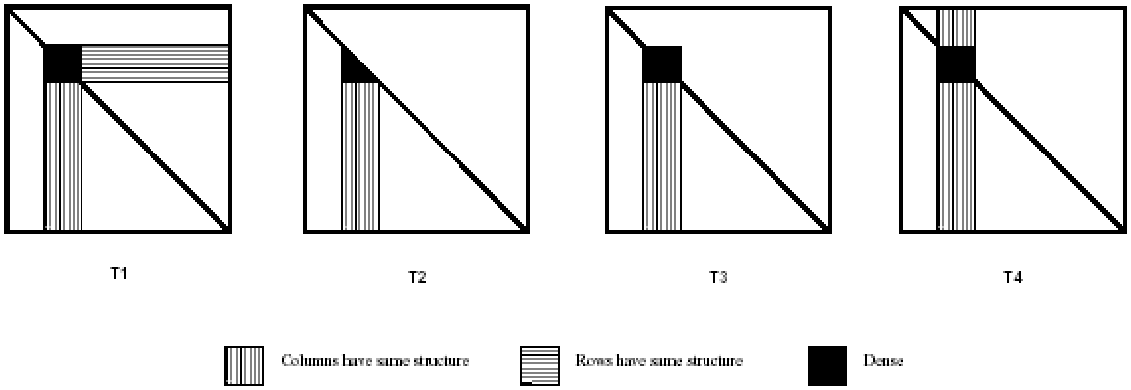


Figure 5.9: Four possible types of unsymmetric supernodes.

T3 Same column structure in L, full diagonal block in U: A supernode is a range $(r : s)$ of columns of L and U, such that the diagonal block $F(r : s, r : s)$ is full, and below the diagonal block the columns of L have the same structure. T3 supernodes allow sup-col updates, like T2. In addition, if the storage for a supernode is organized as for a two-dimensional (2-D) array (for Level 2 or 3 BLAS calls), T3 supernodes do not waste any space in the diagonal block of U.

T4 Same column structure in L and U: A supernode is a range $(r : s)$ of columns of L and U with identical structure. (Since the diagonal is nonzero, the diagonal block must be full.) T4 supernodes allow sup-col updates, and also simplify storage of L and U.

T5 Supernodes of A^TA: A supernode is a range $(r : s)$ of columns of L corresponding to a Cholesky supernode of the symmetric matrix A^TA. T5 supernodes are motivated by the observation that (with suitable representations) the structures of L and U in the unsymmetric factorization $PA = LU$ are contained in the structure of the Cholesky factor of A^TA. In unsymmetric LU, these supernodes themselves are sparse, so we would waste time and space operating on them. Thus we do not consider them further.

Figure 5.9 is a schematic of definitions T1 through T4.

Supernodes are only useful if they actually occur in practice. We reject T4 supernodes as being too rare to make up for the simplicity of their storage scheme. T1 supernodes allow Level 3 BLAS updates, but we can get most of their cache advantage with the more common T2 or T3 supernodes by using supernode-panel updates. Thus we conclude that either T2 or T3 is best by our criteria.

Figure 5.10 shows a sample matrix and the nonzero structure of its factors with no pivoting. Using definition T2, this matrix has four supernodes: $\{1, 2\}$, $\{3\}$, $\{4, 5, 6\}$, and $\{7, 8, 9, 10\}$. For example, in columns 4, 5, and 6 the diagonal blocks of L and U are full, and the columns of L all have nonzeros in rows 8 and 9. By definition T3, the matrix has five supernodes: $\{1, 2\}$, $\{3\}$, $\{4, 5, 6\}$, $\{7\}$, and $\{8, 9, 10\}$. Column 7 fails to join $\{8, 9, 10\}$ as a T3 supernode because u_{78} is zero.

5.7.2 The Column Elimination Tree

Since our definition requires the columns of a supernode to be contiguous, we should get larger supernodes if we bring together columns of L with the same nonzero structure. But the column ordering is fixed, for sparsity, before numeric factorization; what can we do?

$$
\begin{pmatrix}
1 & \bullet & \bullet & & & \bullet & & & & \\
\bullet & 2 & & \bullet & & & & \bullet & & \\
& & 3 & & & & \bullet & \bullet & & \\
& & & 4 & \bullet & \bullet & & & \bullet & \\
& & & \bullet & & & \bullet & & & \\
\bullet & & & & & 6 & & & \bullet & \\
& & & & & & 7 & & \bullet & \bullet \\
\bullet & & \bullet & & & & & & & \\
& & & \bullet & & \bullet & & & 9 & \\
& & \bullet & & & & \bullet & & \bullet & 10
\end{pmatrix}
\qquad
\begin{pmatrix}
1 & \bullet & \bullet & & & \bullet & & & & \\
\bullet & 2 & \bullet & \bullet & & \bullet & & \bullet & & \\
& & 3 & & & & \bullet & \bullet & & \\
& & & 4 & \bullet & \bullet & & & \bullet & \\
& & & \bullet & 5 & \bullet & \bullet & & \bullet & \\
\bullet & \bullet & \bullet & \bullet & \bullet & 6 & \bullet & \bullet & \bullet & \\
& & & & & & 7 & & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & 8 & \bullet & \bullet \\
& & & \bullet & \bullet & \bullet & \bullet & \bullet & 9 & \bullet \\
& & \bullet & & & & \bullet & \bullet & \bullet & 10
\end{pmatrix}
$$

Original matrix A — Factors $F = L + U - I$

Figure 5.10: A sample matrix and its LU factors. Diagonal elements a_{55} and a_{88} are zero.

$$
\begin{array}{c}
1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10
\end{array}
\begin{pmatrix}
s_1 & s_1 & u_3 & & & u_6 & & & & \\
s_1 & s_1 & u_3 & u_4 & & u_6 & & u_8 & & \\
& & s_2 & & & & u_7 & u_8 & & \\
& & & s_3 & s_3 & s_3 & & & u_9 & \\
& & & s_3 & s_3 & s_3 & u_7 & & u_9 & \\
s_1 & s_1 & s_2 & s_3 & s_3 & s_3 & u_7 & u_8 & u_9 & \\
& & & & & & s_4 & & s_4 & s_4 \\
s_1 & s_1 & s_2 & s_3 & s_3 & s_3 & s_4 & s_4 & s_4 & s_4 \\
& & & s_3 & s_3 & s_3 & s_4 & s_4 & s_4 & s_4 \\
& & s_2 & & & & s_4 & s_4 & s_4 & s_4
\end{pmatrix}
$$

Figure 5.11: Supernodal structure $bydefinitionT2$ of the factors of the sample matrix.

```
for column j = 1 to n do
    f = A(:, j);
    Symbolic factorization: determine which supernodes of L will update f;
    Determine whether j belongs to the same supernode as j − 1;
    for each updating supernode (r:s) < j in topological order do
        Apply supernode-column update to column j:
            f(r:s) = L(r:s, r:s)^-1 · f(r:s);  /* Now f(r:s) = U(r:s, j) */
            f(s + 1:n) = f(s + 1:n) − L(s + 1:n, r:s) · f(r:s);
    end for;
    Pivot: interchange f(j) and f(m), where |f(m)| = max |f(j:n)|;
    Separate L and U: U(1:j, j) = f(1:j);   L(j:n, j) = f(j:n);
    Scale: L(j:n, j) = L(j:n, j)/L(j, j);
    Prune symbolic structure based on column j;
end for;
```

Figure 5.12: LU factorization with supernode-column updates

In symmetric Cholesky factorization, one type of supernodes - the "fundamental" supernodes - can be made contiguous by permuting the matrix (symmetrically) according to a postorder on its elimination tree. This postorder is an example of what Liu calls an equivalent reordering, which does not change the sparsity of the factor. The postordered elimination tree can also be used to locate the supernodes before the numeric factorization.

We proceed similarly for the unsymmetric case. Here the appropriate analogue of the symmetric elimination tree is the *column elimination tree*, or column etree for short. The vertices of this tree are the integers 1 through n, representing the columns of A. The column etree of A is the (symmetric) elimination tree of the column intersection graph of A, or equivalently the elimination tree of A^TA provided there is no cancellation in computing A^TA. See Gilbert and Ng for complete definitions. The column etree can be computed from A in time almost linear in the number of nonzeros of A.

Just as a postorder on the symmetric elimination tree brings together symmetric supernodes, we expect a postorder on the column etree to bring together unsymmetric supernodes. Thus, before we factor the matrix, we compute its column etree and permute the matrix columns according to a postorder on the tree.

5.7.3 Relaxed Supernodes

For most matrices, the average size of a supernode is only about 2 to 3 columns (though a few supernodes are much larger). A large percentage of supernodes consist of only a single column, many of which are leaves of the column etree. Therefore merging groups of columns at the fringe of the etree into *artificial supernodes* regardless of their row structures can be beneficial. A parameter r controls the granularity of the merge. A good merge rule is: node i is merged with its parent node j when the subtree rooted at j has at most r nodes. In practice, the best values of r are generally between 4 and 8 and yield improvements in running time of 5% to 15%.

Artificial supernodes are a special case of relaxed supernodes. They allow a small number of zeros in the structure of any supernode, thus relaxing the condition that the columns must have strictly nested structures.

5.7.4 Supernodal Numeric Factorization

Now we show how to modify the col-col algorithm to use sup-col updates and supernode-panel updates. This section describes the numerical computation involved in the updates.

Supernode-Column Updated

Figure 5.12 sketches the sup-col algorithm. The only difference from the col-col algorithm is that all the updates to a column from a single supernode are done together. Consider a supernode $(r:s)$ that updates column j. The coefficients of the updates are the values from a segment of column j of U, namely $U(r:s,j)$. The nonzero structure of such a segment is particularly simple: all the nonzeros are contiguous, and follow all the zeros. Thus, if k is the index of the first nonzero row in $U(r:s,j)$, the updates to column j from supernode $(r:s)$ come from columns k through s. Since the supernode is stored as a dense matrix, these updates can be performed by a dense lower triangular solve (with the matrix $L(k:s,k:s)$) and a dense matrix-vector multiplication (with the matrix $L(s+1:n,k:s)$). The symbolic phase determines the value of k, that is, the position of the first nonzero in the segment $U(r:s,j)$.

The advantages of using sup-col updates are similar to those in the symmetric case. Efficient Level 2 BLAS matrix-vector kernels can be used for the triangular solve and matrix-vector multiply. Furthermore, all the updates from the supernodal columns can be collected in a dense vector before doing a single scatter into the target vector. This reduces the amount of indirect addressing.

Supernode-Panel Updates

We can improve the sup-col algorithm further on machines with a memory hierarchy by changing the data access pattern. The data we are accessing in the inner loop (lines 5-9 of Figure 5.12) include the destination column j and all the updating supernodes $(r:s)$ to the left of column j. Column j is accessed many times, while each supernode $(r:s)$ is used only once. In practice, the number of nonzero elements in column j is much less than that in the updating supernodes. Therefore, the access pattern given by this loop provides little opportunity to reuse cached data. In particular, the same supernode $(r:s)$ may be needed to update both columns j and $j+1$. But when we factor the (j+1)th column (in the next iteration of the outer loop), we will have to fetch supernode $(r:s)$ again from memory, instead of from cache (unless the supernodes are small compared to the cache).

Panels

To exploit memory locality, we factor several columns (say w of them) at a time in the outer loop, so that one updating supernode $(r:s)$ can be used to update as many of the w columns as possible. We refer to these w consecutive columns as a *panel* to differentiate them from a supernode, the row structures of these columns may not be correlated in any fashion, and the boundaries between panels may be different from those between supernodes. The new method requires rewriting the doubly nested loop as the triple loop shown in Figure 5.13.

The structure of each sup-col update is the same as in the sup-col algorithm. For each supernode $(r:s)$ to the left of column j, if $u_{kj}\neq 0$ for some $r\leq k\leq s$, then $u_{ij}\neq 0$ for all $k\leq i\leq s$. Therefore, the nonzero structure of the panel of U consists of dense column segments that are row-wise separated by supernodal boundaries, as in Figure 5.13. Thus, it is sufficient for the symbolic factorization algorithm to record only the first nonzero position of each column segment.

```
for column j = 1 to n step w do
    Symbolic factor: determine which supernodes will update any of L(:, j:j + w − 1);
    for each updating supernode (r:s) < j in topological order do
        for column jj = j to j + w − 1 do
            Apply supernode-column update to column jj;
        end for;
    end for;
    Inner factorization:
        Apply the sup-col algorithm on columns and supernodes within the panel;
end for;
```

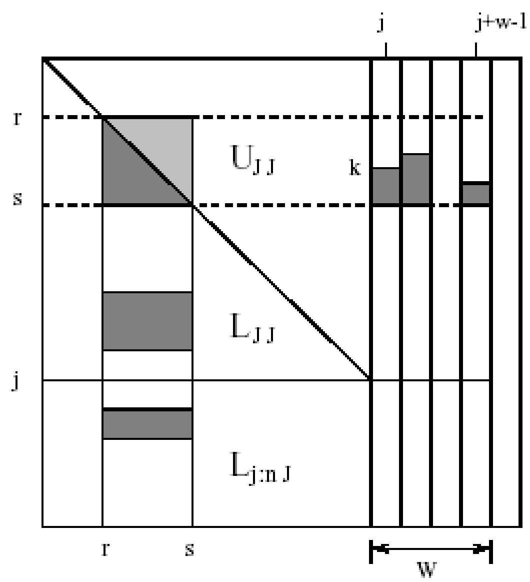


Figure 5.13: The supernode-panel algorithm, with columnwise blocking. $J = 1 : j - 1$

As detailed in section 4.4, symbolic factorization is applied to all the columns in a panel at once, over all the updating supernodes, before the numeric factorization step.

In dense factorization, the entire supernode-panel update in lines 3-7 of Figure 5.13 would be implemented as two Level 3 BLAS calls: a dense triangular solve with w right-hand sides, followed by a dense matrix-matrix multiply. In the sparse case, this is not possible, because the different sup-col updates begin at different positions k within the supernode, and the submatrix $U(r : s, j : j + w - 1)$ is not dense. Thus the sparse supernode-panel algorithm still calls the Level 2 BLAS. However, we get similar cache benefits to those from the Level 3 BLAS, at the cost of doing the loop reorganization ourselves. Thus we sometimes call the kernel of this algorithm a "BLAS-$2\frac{1}{2}$" method.

In the doubly nested loop (lines 3-7 of Figure 5.13), the ideal circumstance is that all w columns in the panel require updates from supernode $(r : s)$. Then this supernode will be used w times before it is forced out of the cache. There is a trade-off between the value of w and the size of the cache. For this scheme to work efficiently, we need to ensure that the nonzeros in the w columns do not cause cache thrashing. That is, we must keep w small enough so that all the data accessed in this doubly nested loop fit in cache. Otherwise, the cache interference between the source supernode and the destination panel can offset the benefit of the new algorithm.

5.8 Efficient sparse matrix algorithms

5.8.1 Scalable algorithms

By a *scalable* algorithm for a problem, we mean one that maintains efficiency bounded away from zero as the number p of processors grows and the size of the data structures grows roughly linearly in p.

Notable efforts at analysis of the scalability of dense matrix computations include those of Li and Coleman [58] for dense triangular systems, and Saad and Schultz [85]; Ostrouchov, *et al.* [73], and George, Liu, and Ng [39] have made some analyses for algorithms that map matrix columns to processors. Rothberg and Gupta [81] is a important paper for its analysis of the effect of caches on sparse matrix algorithms.

Consider any distributed-memory computation. In order to assess the communication costs analytically, it s useful to employ certain abstract lower bounds. Our model assumes that machine topology is given. It assumes that memory consists of the memories local to processors. It assumes that the communication channels are the edges of a given undirected graph $G = (W, L)$, and that processor–memory units are situated at some, possibly all, of the vertices of the graph. The model includes hypercube and grid-structured message-passing machines, shared-memory machines having physically distributed memory (the Tera machine) as well as tree-structured machines like a CM-5.

Let $V \subseteq W$ be the set of all processors and L be the set of all communication links.

We assume identical links. Let β be the inverse bandwidth (slowness) of a link in seconds per word. (We ignore latency in this model; most large distributed memory computations are bandwidth limited.)

We assume that processors are identical. Let ϕ be the inverse computation rate of a processor in seconds per floating-point operation. Let β_0 be the rate at which a processor can send or receive data, in seconds per word. We expect that β_0 and β will be roughly the same.

A distributed-memory computation consists of a set of processes that exchange information by sending and receiving messages. Let M be the set of all messages communicated. For $m \in M$,

$|m|$ denotes the number of words in m. Each message m has a source processor $src(m)$ and a destination processor $dest(m)$, both elements of V.

For $m \in M$, let $d(m)$ denote the length of the path taken by m from the source of the message m to its destination. We assume that each message takes a certain path of links from its source to its destination processor. Let $p(m) = (\ell_1, \ell_2, \ldots, \ell_{d(m)})$ be the path taken by message m. For any link $\ell \in L$, let the set of messages whose paths utilize ℓ, $\{m \in M \mid \ell \in p(m)\}$, be denoted $M(\ell)$.

The following are obviously lower bounds on the completion time of the computation. The first three bounds are computable from the set of message M, each of which is characterized by its size and its endpoints. The last depends on knowledge of the paths $p(M)$ taken by the messages.

1. (Average flux)

$$\frac{\sum_{m \in M} |m| \cdot d(m)}{|L|} \cdot \beta.$$

   This is the total flux of data, measured in word-hops, divided by the machine's total communication bandwidth, L/β.

2. (Bisection width) Given $V_0, V_1 \subseteq W$, V_0 and V_1 disjoint, define

$$sep(V_0, V_1) \equiv \min |\{L' \subseteq L \mid L' \text{ is an edge separator of } V_0 \text{ and } V_1\}|$$

   and

$$flux(V_0, V_1) \equiv \sum_{\{m \in M \mid src(m) \in V_i, dest(m) \in V_{1-i}\}} |m|.$$

   The bound is

$$\frac{flux(V_0, V_1)}{sep(V_0, V_1)} \cdot \beta.$$

   This is the number of words that cross from one part of the machine to the other, divided by the bandwidth of the wires that link them.

3. (Arrivals/Departures (also known as node congestion))

$$\max_{v \in V} \sum_{dest(m) = v} |m| \beta_0;$$

$$\max_{v \in V} \sum_{src(m) = v} |m| \beta_0.$$

   This is a lower bound on the communication time for the processor with the most traffic into or out of it.

4. (Edge contention)

$$\max_{\ell \in L} \sum_{m \in M(\ell)} |m| \beta.$$

   This is a lower bound on the time needed by the most heavily used wire to handle all its traffic.

Of course, the actual communication time may be greater than any of the bounds. In particular, the communication resources (the wires in the machine) need to be scheduled. This can be done dynamically or, when the set of messages is known in advance, statically. With detailed knowledge of the schedule of use of the wires, better bounds can be obtained. For the purposes of analysis of algorithms and assignment of tasks to processors, however, we have found this more realistic approach to be unnecessarily cumbersome. We prefer to use the four bounds above, which depend only on the integrated (i.e. time-independent) information M and, in the case of the edge-contention bound, the paths $p(M)$. In fact, in the work below, we won't assume knowledge of paths and we won't use the edge contention bound.

5.8.2 Cholesky factorization

We'll use the techniques we've introduced to analyze alternative distributed memory implementations of a very important computation, Cholesky factorization of a symmetric, positive definite (SPD) matrix A. The factorization is $A = LL^T$ where L is lower triangular; A is given, L is to be computed.

The algorithm is this:

1. $L := A$
2. `for` $k = 1$ `to` N `do`
3. $L_{kk} := \sqrt{L_{kk}}$
4. `for` $i = k + 1$ `to` N `do`
5. $L_{ik} := L_{ik} L_{kk}^{-1}$
6. `for` $j = k + 1$ `to` N `do`
7.   `for` $i = j$ `to` N `do`
8. $L_{ij} := L_{ij} - L_{ik} L_{jk}^T$

We can let the elements L_{ij} be scalars, in which case this is the usual or "point" Cholesky algorithm. Or we can take L_{ij} to be a block, obtained by dividing the rows into contiguous subsets and making the same decomposition of the columns, so that diagonal blocks are square. In the block case, the computation of $\sqrt{L_{kk}}$ (Step 3) returns the (point) Cholesky factor of the SPD block L_{kk}. If A is sparse (has mostly zero entries) then L will be sparse too, although less so than A. In that case, only the non-zero entries in the sparse factor L are stored, and the multiplication/division in lines 5 and 8 are omitted if they compute zeros.

Mapping columns

Assume that the columns of a dense symmetric matrix of order N are mapped to processors cyclically: column j is stored in processor $map(j) \equiv j \bmod p$. Consider communication costs on two-dimensional grid or toroidal machines. Suppose that p is a perfect square and that the machine is a $\sqrt{p} \times \sqrt{p}$ grid. Consider a mapping of the computation in which the operations in line 8 are performed by processor $map(j)$. After performing the operations in line 5, processor $map(k)$ must send column k to all processors $\{map(j) \mid j > k\}$.

Let us fix our attention on 2D grids. There are $L = 2p + O(1)$ links. A column can be broadcast from its source to all other processors through a spanning tree of the machine, a tree of total length p reaching all the processors. Every matrix element will therefore travel over $p - 1$ links, so the total information flux is $(1/2)N^2 p$ and the average flux bound is $(1/4)N^2 \beta$.

Type of Bound	Lower bound
Arrivals	$\frac{1}{4}N^2\beta_0$
Average flux	$\frac{1}{4}N^2\beta$

Table 5.1: Communication Costs for Column-Mapped Full Cholesky.

Type of Bound	Lower bound
Arrivals	$\frac{1}{4}N^2\beta\left(\frac{1}{p_r}+\frac{1}{p_c}\right)$
Edge contention	$N^2\beta\left(\frac{1}{p_r}+\frac{1}{p_c}\right)$

Table 5.2: Communication Costs for Torus-Mapped Full Cholesky.

Only $O(N^2/p)$ words leave any processor. If $N \gg p$, processors must accept almost the whole $(1/2)N^2$ words of L as arriving columns. The bandwidth per processor is β_0, so the arrivals bound is $(1/2)N^2\beta_0$ seconds. If $N \approx p$ the bound drops to half that, $(1/4)N^2\beta_0$ seconds. We summarize these bounds for 2D grids in Table 5.1.

We can immediately conclude that this is a nonscalable distributed algorithm. We may not take $p > \frac{N\phi}{\beta}$ and still achieve high efficiency.

Mapping blocks

Dongarra, Van de Geijn, and Walker [26] have shown that on the Intel Touchstone Delta machine ($p = 528$), mapping blocks is better than mapping columns in LU factorization. In such a mapping, we view the machine as an $p_r \times p_c$ grid and we map elements A_{ij} and L_{ij} to processor $(mapr(i), mapc(j))$. We assume a cyclic mappings here: $mapr(i) \equiv i \bmod p_r$ and similarly for $mapc$.

The analysis of the preceding section may now be done for this mapping. Results are summarized in Table 5.2. With p_r and p_c both $O(\sqrt{p})$, the communication time drops like $O(p^{-1/2})$. With this mapping, the algorithm is scalable even when $\beta \gg \phi$. Now, with $p = O(N^2)$, both the compute time and the communication lower bounds agree; they are $O(N)$. Therefore, we remain efficient when storage per processor is $O(1)$. (This scalable algorithm for distributed Cholesky is due to O'Leary and Stewart [72].)

5.8.3 Distributed sparse Cholesky and the model problem

In the sparse case, the same holds true. To see why this must be true, we need only observe that most of the work in sparse Cholesky factorization takes the form of the factorization of dense submatrices that form during the Cholesky algorithm. Rothberg and Gupta demonstrated this fact in their work in 1992 – 1994.

Unfortunately, with naive cyclic mappings, block-oriented approaches suffer from poor balance of the computational load and modest efficiency. Heuristic remapping of the block rows and columns can remove load imbalance as a cause of inefficiency.

Several researchers have obtained excellent performance using a block-oriented approach, both on fine-grained, massively-parallel SIMD machines [23] and on coarse-grained, highly-parallel MIMD machines [82]. A block mapping maps rectangular blocks of the sparse matrix to processors. A 2-D mapping views the machine as a 2-D $p_r \times p_c$ processor grid, whose members are denoted $p(i,j)$. To date, the 2-D cyclic (also called torus-wrap) mapping has been used: block L_{ij} resides at processor $p(i \bmod p_r, j \bmod p_c)$. All blocks in a given block row are mapped to the same row of processors, and all elements of a block column to a single processor column. Communication volumes grow as the square root of the number of processors, versus linearly for the 1-D mapping; 2-D mappings also asymptotically reduce the critical path length. These advantages accrue even when the underlying machine has some interconnection network whose topology is not a grid.

A 2-D cyclic mapping, however, produces significant load imbalance that severely limits achieved efficiency. On systems (such as the Intel Paragon) with high interprocessor communication bandwidth this load imbalance limits efficiency to a greater degree than communication or want of parallelism.

An alternative, heuristic 2-D block mapping succeeds in reducing load imbalance to a point where it is no longer the most serious bottleneck in the computation. On the Intel Paragon the block mapping heuristic produces a roughly 20% increase in performance compared with the cyclic mapping.

In addition, a scheduling strategy for determining the order in which available tasks are performed adds another 10% improvement.

5.8.4 Parallel Block-Oriented Sparse Cholesky Factorization

In the block factorization approach considered here, matrix blocks are formed by dividing the columns of the $n \times n$ matrix into N contiguous subsets, $N \leq n$. The identical partitioning is performed on the rows. A block L_{ij} in the sparse matrix is formed from the elements that fall simultaneously in row subset i and column subset j.

Each block L_{ij} has an owning processor. The owner of L_{ij} performs all block operations that update L_{ij} (this is the "owner-computes" rule for assigning work). Interprocessor communication is required whenever a block on one processor updates a block on another processor.

Assume that the processors can be arranged as a grid of p_r rows and p_c columns. In a *Cartesian product* (CP) mapping, $map(i,j) = p(RowMap(i), ColMap(j))$, where $RowMap : \{0..N-1\} \rightarrow \{0..p_r - 1\}$, and $ColMap : \{0..N-1\} \rightarrow \{0..p_c - 1\}$ are given mappings of rows and columns to processor rows and columns. We say that *map* is *symmetric Cartesian* (SC) if $p_r = p_c$ and $RowMap = ColMap$. The usual 2-D cyclic mapping is SC.[2]

5.9 Load balance with cyclic mapping

Any CP mapping is effective at reducing communication. While the 2-D cyclic mapping is CP, unfortunately it is not very effective at balancing computational load. Experiment and analysis show that the cyclic mapping produces particularly poor load balance; moreover, some serious load balance difficulties must occur for any SC mapping. Improvements obtained by the use of nonsymmetric CP mappings are discussed in the following section.

[2] See [82] for a discussion of *domains*, portions of the matrix mapped in a 1-D manner to further reduce communication.

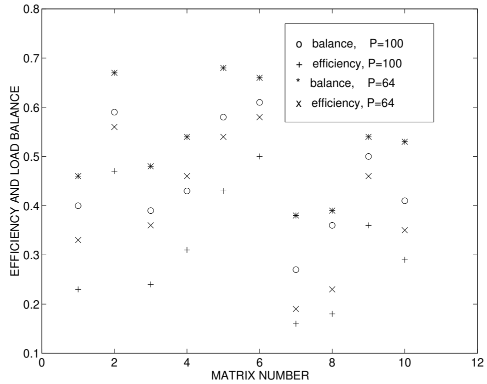


Figure 5.14: Efficiency and overall balance on the Paragon system ($B = 48$).

Our experiments employ a set of test matrices including two dense matrices (DENSE1024 and DENSE2048), two 2-D grid problems (GRID150 and GRID300), two 3-D grid problems (CUBE30 and CUBE35), and 4 irregular sparse matrices from the Harwell-Boeing sparse matrix test set [27]. Nested dissection or minimum degree orderings are used. In all our experiments, we choose $p_r = p_c = \sqrt{P}$, and we use a block size of 48. All Mflops measurements presented here are computed by dividing the operation counts of the best known sequential algorithm by parallel runtimes. Our experiments were performed on an Intel Paragon, using hand-optimized versions of the Level-3 BLAS for almost all arithmetic.

5.9.1 Empirical Load Balance Results

We now report on the efficiency and load balance of the method. Parallel efficiency is given by $t_{seq}/(P \cdot t_{par})$, where t_{par} is the parallel runtime, P is the number of processors, and t_{seq} is the runtime for the same problem on one processor. For the data we report here, we measured t_{seq} by factoring the benchmark matrices using our parallel algorithm on one processor. The overall balance of a distributed computation is given by $work_{total}/(P \cdot work_{max})$, where $work_{total}$ is the total amount of work performed in the factorization, P is the number of processors, and $work_{max}$ is the maximum amount of work assigned to any processor. Clearly, overall balance is an upper bound on efficiency.

Figure 1 shows efficiency and overall balance with the cyclic mapping. Observe that load balance and efficiency are generally quite low, and that they are well correlated. Clearly, load balance alone is not a perfect predictor of efficiency. Other factors limit performance. Examples include interprocessor communication costs, which we measured at 5% — 20% of total runtime, long critical paths, which can limit the number of block operations that can be performed concurrently, and poor scheduling, which can cause processors to wait for block operations on other processors to complete. Despite these disparities, the data indicate that load imbalance is an important contributor to reduced efficiency.

We next measured load imbalance among rows of processors, columns of processors, and diagonals of processors. Define $work[i, j]$ to be the runtime due to updating of block L_{ij} by its owner. To approximate runtime, we use an empirically calibrated estimate of the form work = operations + ω · block-operations; on the Paragon, $\omega = 1,000$.

Define $RowWork[i]$ to be the aggregate work required by blocks in row i: $RowWork[i] = \sum_{j=0}^{N-1} work[i, j]$. An analogous definition applies for $ColWork$, the aggregate column work. Define

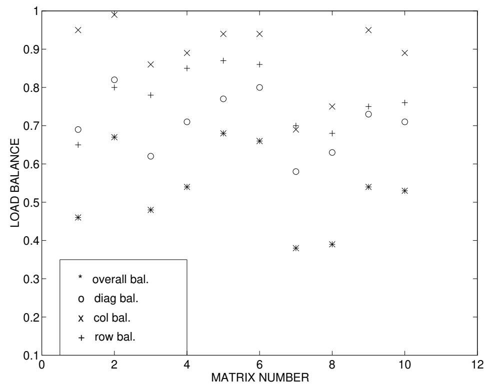


Figure 5.15: Efficiency bounds for 2-D cyclic mapping due to row, column and diagonal imbalances ($P = 64, B = 48$).

row balance by $work_{total}/p_r \cdot work_{rowmax}$, where $work_{rowmax} = \max_r \sum_{i:RowMap[i]=r} RowWork[i]$. This row balance statistic gives the best possible overall balance (and hence efficiency), obtained only if there is perfect load balance within each processor row. It isolates load imbalance due to an overloaded processor row caused by a poor row mapping. An analogous expression gives column balance, and a third analogous expression gives diagonal balance. (Diagonal d is made up of the set of processors $p(i,j)$ for which $(i-j) \bmod p_r = d$.) While these three aggregate measures of load balance are only upper bounds on overall balance, the data we present later make it clear that improving these three measures of balance will in general improve the overall load balance.

Figure 2 shows the row, column, and diagonal balances with a 2-D cyclic mapping of the benchmark matrices on 64 processors. Diagonal imbalance is the most severe, followed by row imbalance, followed by column imbalance.

These data can be better understood by considering dense matrices as examples (although the following observations apply to a considerable degree to sparse matrices as well). Row imbalance is due mainly to the fact that $RowWork[i]$, the amount of work associated with a row of blocks, increases with increasing i. More precisely, since $work[i,j]$ increases linearly with j and the number of blocks in a row increases linearly with i, it follows that $RowWork[i]$ increases quadratically in i. Thus, the processor row that receives the last block row in the matrix receives significantly more work than the processor row immediately following it in the cyclic ordering, resulting in significant row imbalance. Column imbalance is not nearly as severe as row imbalance. The reason, we believe, is that while the work associated with blocks in a column increases linearly with the column number j, the number of blocks in the column *decreases* linearly with j. As a result, $ColWork[j]$ is neither strictly increasing nor strictly decreasing. In the experiments, row balance is indeed poorer than column balance. Note that the reason for the row and column imbalance is not that the 2-D cyclic mapping is an SC mapping; rather, we have significant imbalance because the mapping functions $RowMap$ and $ColMap$ are each poorly chosen.

To better understand diagonal imbalance, one should note that blocks on the diagonal of the matrix are mapped exclusively to processors on the main diagonal of the processor grid. Blocks just below the diagonal are mapped exclusively to processors just below the main diagonal of the processor grid. These diagonal and sub-diagonal blocks are among the most work-intensive blocks in the matrix. In sparse problems, moreover, the diagonal blocks are the only ones that are guaranteed to be dense. (For the two dense test matrices, diagonal balance is not significantly worse than row balance.) The remarks we make about diagonal blocks and diagonal processors apply to *any* SC mapping, and do not depend on the use of a cyclic function $RowMap(i) = i \bmod p_r$.

5.10 Heuristic Remapping

Nonsymmetric CP mappings, which map rows independently of columns, are a way to avoid diagonal imbalance that is automatic with SC mappings. We shall choose the row mapping $RowMap$ to maximize the row balance, and independently choose $ColMap$ to maximize column balance. Since the row mapping has no effect on the column balance, and vice versa, we may choose the row mapping in order to maximize row balance independent of the choice of column mapping.

The problems of determining $RowMap$ and $ColMap$ are each cases of a standard NP-complete problem, *number partitioning* [36], for which a simple heuristic is known to be good [3]. This heuristic obtains a row mapping by considering the block rows in some predefined sequence. For each processor row, it maintains the total work for all blocks mapped to that row. The algorithm iterates over block rows, mapping a block row to the processor row that has received the least work thus far. We have experimented with several different sequences, the two best of which we now describe.

The **Decreasing Work (DW)** heuristic considers rows in order of decreasing work. This is a standard approach to number partitioning; that small values toward the end of the sequence allow the algorithm to lessen any imbalance caused by large values encountered early in the sequence.

The **Increasing Depth (ID)** heuristic considers rows in order of increasing depth in the elimination tree. In a sparse problem, the work associated with a row is closely related to its depth in the elimination tree.

The effect of these schemes is dramatic. If we look at the three aggregate measures of load balance, we find that these heuristics produce row and column balance of 0.98 or better, and diagonal balance of 0.93 or better, for test case BCSSTK31, which is typical. With ID as the row mapping we have produced better than a 50% improvement in overall balance, and better than a 20% improvement in performance, on average over our test matrices, with $P = 100$. The DW heuristic produces only slightly less impressive improvements The choice of column mapping, as expected, is less important. In fact, for our test suite, the cyclic column mapping and ID row mapping gave the best mean performance. [4]

We also applied these ideas to four larger problems: DENSE4096, CUBE40, COPTER2 (a helicopter rotor blade model, from NASA) and 10FLEET (a linear programming formulation of an airline fleet assignment problem, from Delta Airlines). On 144 and 196 processors the heuristic (increasing depth on rows and cyclic on columns) again produces a roughly 20% performance improvement over a cyclic mapping. Peak performance of 2.3 Gflops for COPTER2 and 2.7 Gflops for 10FLEET were achieved; for the model problems CUBE40 and DENSE4096 the speeds were 3.2 and 5.2 Gflops.

In addition to the heuristics described so far, we also experimented with two other approaches to improving factorization load balance. The first is a subtle modification of the original heuristic. It begins by choosing some column mapping (we use a cyclic mapping). This approach then iterates over rows of blocks, mapping a row of blocks to a row of processors so as to minimize the amount of work assigned to any one *processor*. Recall that the earlier heuristic attempted to minimize the aggregate work assigned to an entire row of processors. We found that this alternative heuristic produced further large improvements in overall balance (typically 10-15% better than that of our

[3]Number partitioning is a well studied NP-complete problem. The objective is to distribute a set of numbers among a fixed number of bins so that the maximum sum in any bin is minimized.

[4]Full experimental data has appeared in another paper [83].

original heuristic). Unfortunately, realized performance did not improve. This result indicates that load balance is not the most important performance bottleneck once our original heuristic is applied.

A very simple alternate approach reduces imbalance by performing cyclic row and column mappings on a processor grid whose dimensions p_c and p_r are relatively prime; this reduces diagonal imbalance. We tried this using 7×9 and 9×11 processor grids (using one fewer processor that for our earlier experiments with $P = 64$ and $P = 100$.) The improvement in performance is somewhat lower than that achieved with our earlier remapping heuristic (17% and 18% mean improvement on 63 and 99 processors versus 20% and 24% on 64 and 100 processors). On the other hand, the mapping needn't be computed.

5.11 Scheduling Local Computations

The next questions to be addressed, clearly, are: (i) what is the most constraining bottleneck after our heuristic is applied, and (ii) can this bottleneck be addressed to further improve performance?

One potential remaining bottleneck is communication. Instrumentation of our block factorization code reveals that on the Paragon system, communication costs account for less than 20% of total runtime for all problems, even on 196 processors. The same instrumentation reveals that most of the processor time not spent performing useful factorization work is spent idle, waiting for the arrival of data.

We do not believe that the idle time is due to insufficient parallelism. Critical path analysis for problem BCSSTK15 on 100 processors, for example, indicates that it should be possible to obtain nearly 50% higher performance than we are currently obtaining. The same analysis for problem BCSSTK31 on 100 processors indicates that it should be possible to obtain roughly 30% higher performance. We therefore suspected that the scheduling of tasks by our code was not optimal.

To that end we tried alternative scheduling policies. They are:

FIFO. Tasks are initiated in the order in which the processor discovers that they are ready.

Destination depth. Ready tasks initiated in order of the destination block's elimination tree depth.

Source depth. Ready tasks initiated in order of the source block's elimination tree depth.

For the FIFO policy, a queue of ready tasks is used, while for the others, a heap is used. We experimented with 64 processors, using BSCCST31, BCSSTK33, and DENSE2048. Both priority-based schemes are better than FIFO; destination depth seems slightly better than source depth. We observed a slowdown of 2% due to the heap data structure on BCSSTK33; the destination priority scheme then improved performance by 15% for a net gain of 13%. For BSCCST31 the net gain was 8%. For DENSE2048, however, there was no gain. This improvement is encouraging. There may be more that can be achieved through the pursuit of a better understanding of the scheduling question.

Lecture 6

Parallel Machines

A parallel computer is a connected configuration of processors and memories. The choice space available to a computer architect includes the network topology, the node processor, the address-space organization, and the memory structure. These choices are based on the parallel computation model, the current technology, and marketing decisions.

No matter what the pace of change, it is impossible to make intelligent decisions about parallel computers right now without some knowledge of their architecture. For more advanced treatment of computer architecture we recommend Kai Hwang's *Advanced Computer Architecture* and *Parallel Computer Architecture* by Gupta, Singh, and Culler.

One may gauge what architectures are important today by the Top500 Supercomputer[1] list published by Meuer, Strohmaier, Dongarra and Simon. The secret is to learn to read between the lines. There are three kinds of machines on **The November 2003 Top 500 list**:

- Distributed Memory Multicomputers (MPPs)

- Constellation of Symmetric Multiprocessors (SMPs)

- Clusters (NOWs and Beowulf cluster)

Vector Supercomputers, Single Instruction Multiple Data (SIMD) Machines and SMPs are no longer present on the list but used to be important in previous versions.

How can one simplify (and maybe grossly oversimplify) the current situation? Perhaps by pointing out that the world's fastest machines are mostly clusters. Perhaps it will be helpful to the reader to list some of the most important machines first sorted by type, and then by highest rank in the top 500 list. We did this in 1997 and also 2003.

[1] http://www.top500.org

Machine	Top 500 First Rank (1996)
Distributed Memory	
Hitachi/Tsukuba CP-PACS	1
Fujitsu NWT	2
Hitachi SR2201	3
Intel XP/S	4
Cray T3D	7
Fujitsu VPP500	8
IBM SP2	14
TMC CM-5	21
Hitachi S-3800	56
Intel Delta	120
Parsytec GC Power Plus	230
Meiko CS-2	438
IBM 9076	486
KSR2-80	491

Machine	Top 500 First Rank (1996)
SMP Arrays	
SGI Power Challenge Array	95
SMP	
SGI Origin 2000	211
Convex SPP1200	264
SGI Power Challenge	288
Digital AlphaServer 8400	296
Vector Machines	
NEC SX	17
Cray YMP	54
SIMD Machines	
TMC CM-200	196

Machine	Top 500 First Rank (2003)
Cluster	
ASCI Q - HP AlphaServer SC45	2
X - Selfmade Apple G5 Cluster	3
Tungsten - Dell PowerEdge Cluster	4
Mpp2 - HP Integrity rx2600 Itanium2 Cluster	5
Lightning - Linux Networx Opteron Cluster	6
MCR Linux Xeon Cluster	7
IBM/Quadrics xSeries Xeon Cluster	10
PSC - HP AlphaServer SC45	12
Legend DeepComp 6800 Itanium Cluster	14
CEA - HP AlphaServer SC45	15
Aspen Systems Dual Xeon Cluster	17
IBM xSeries Xeon Cluster	22
HP Integrity rx5670-4x256	25
Dell-Cray PowerEdge 1750 Cluster	26

Machine	Top 500 First Rank (2003)
Distributed Memory (MPPs)	
NEC Earth-Simulator	1
IBM ASCI White	8
IBM Seaborg SP Power3	9
NCAR - IBM pSeries 690 Turbo	13
HPCx - IBM pSeries 690 Turbo	16
NAVOCEANO - IBM pSeries 690 Turbo	18
US Govt. Cray X1	19
ORNL - Cray X1	20
Cray Inc. Cray X1	21
ECMWF - IBM pSeries 690 Turbo	23
ECMWF - IBM pSeries 690 Turbo	24
Intel - ASCI Red	27
ORNL - IBM pSeries 690 Turbo	28
IBM Canada pSeries 690 Turbo	29
Canstellation of SMPs	
Fujitsu PRIMEPOWER HPC2500	11
NEC SX-5/128M8	88
HP Integrity Superdome/HFabric	117
Sun Fire 15k/6800 Cluster	151

The trend is clear to anyone who looks at the list. Distributed memory machines are on the way out and cluster computers are now the dominant force in supercomputers.

Distributed Memory Multicomputers:

Remembering that a computer is a processor and memory, really a processor with cache and memory, it makes sense to call a set of such "computers" linked by a network a *multicomputer*. Figure 6.1 shows 1) a basic computer which is just a processor and memory and also 2) a fancier computer where the processor has cache, and there is auxiliary disk memory. To the right, we picture 3) a three processor multicomputer. The line on the right is meant to indicate the network.

These machines are sometimes called distributed memory multiprocessors. We can further distinguish between DMM's based on how each processor addresses memory. We call this the private/shared memory issue:

> **Private versus shared memory on distributed memory machines:** It is easy to see that the simplest architecture allows each processor to address only its own

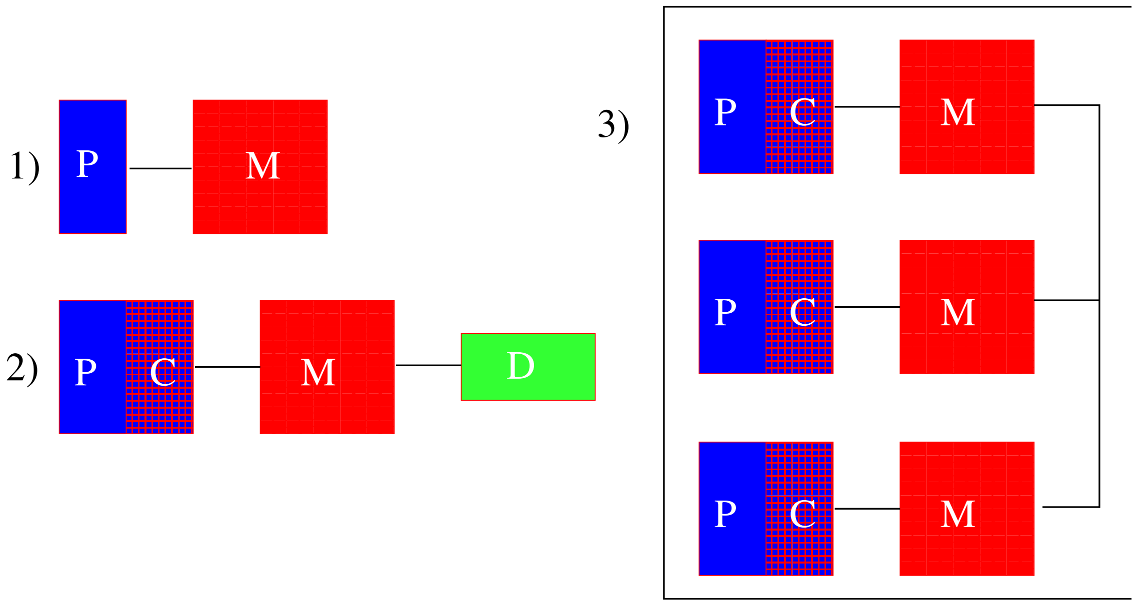


Figure 6.1: 1) A computer 2) A fancier computer 3)A multicomputer

memory. When a processor wants to read data from another processor's memory, the owning processor must send the data over the network to the processor that wants it. Such machines are said to have *private memory.* A close analog is that in my office, I can easily find information that is located on my desk (my memory) but I have to make a direct request via my telephone (ie., I must dial a phone number) to find information that is not in my office. And, I have to hope that the other person is in his or her office, waiting for the phone to ring. In other words, I need the co-operation of the other active party (the person or processor) to be able to read or write information located in another place (the office or memory).

The alternative is a machine in which every processor can directly address every instance of memory. Such a machine is said to have a *shared address space*, and sometimes informally *shared memory*, though the latter terminology is misleading as it may easily be confused with machines where the memory is physically shared. On a shared address space machine, each processor can load data from or store data into the memory of any processor, without the active cooperation of that processor. When a processor requests memory that is not local to it, a piece of hardware intervenes and fetches the data over the network. Returning to the office analogy, it would be as if I asked to view some information that happened to not be in my office, and some special assistant actually dialed the phone number for me without my even knowing about it, and got a hold of the special assistant in the other office, (and these special assistants never leave to take a coffee break or do other work) who provided the information.

Most distributed memory machines have private addressing. One notable exception is the Cray T3D and the Fujitsu VPP500 which have shared physical addresses.

Clusters (NOWs and Beowulf Cluster):

Clusters are built from independent computers integrated through an after-market network. The idea of providing COTS (Commodity off the shelf) base systems to satisfy specific computational requirements evolved as a market reaction to MPPs with the thought the cost might be cheaper. Clusters were considered to have slower communications compared to the specialized machines but they have caught up fast and now outperform most specialized machines.

NOWs stands for Network of Workstations. Any collection of workstations is likely to be networked together: this is the cheapest way for many of us to work on parallel machines given that the networks of workstations already exist where most of us work.

The first Beowulf cluster was built by Thomas Sterling and Don Becker at the Goddard Space Flight Center in Greenbelt Maryland, which is a cluster computer consisting of 16 DX4 processors connected by Ethernet. They named this machine Beowulf (a legendary Geatish warrior and hero of the Old English poem *Beowulf*). Now people use "Beowulf cluster" to denote a cluster of PCs or workstations interconnected by a private high-speed network, which is dedicated to running high-performance computing tasks. Beowulf clusters usually run a free-software operating system like Linux or FreeBSD, though windows Beowulfs exist.

Central Memory Symmetric Multiprocessors (SMPs) and Constellation of SMPs:

Notice that we have already used the word "shared" to refer to the shared address space possible in in a distributed memory computer. Sometimes the memory hardware in a machine does not obviously belong to any one processor. We then say the memory is *central*, though some authors may use the word "shared." Therefore, for us, the central/distributed distinction is one of system architecture, while the shared/private distinction mentioned already in the distributed context refers to addressing.

> **"Central" memory contrasted with distributed memory:** We will view the physical memory architecture as distributed if the memory is packaged with the processors in such a way that some parts of the memory are substantially "farther" (in the sense of lower bandwidth or greater access latency) from a processor than other parts. If all the memory is nearly equally expensive to access, the system has central memory. The vector supercomputers are genuine central memory machines. A network of workstations has distributed memory.

Microprocessor machines known as *symmetric multiprocessors* (SMP) are becoming typical now as mid-sized compute servers; this seems certain to continue to be an important machine design. On these machines each processor has its own cache while the main memory is central. There is no one "front-end" or "master" processor, so that every processor looks like every other processor. This is the "symmetry." To be precise, the symmetry is that every processor has equal access to the operating system. This means, for example, that each processor can independently prompt the user for input, or read a file, or see what the mouse is doing.

The microprocessors in an SMP themselves have caches built right onto the chips, so these caches act like distributed, low-latency, high-bandwidth memories, giving the system many of the important performance characteristics of distributed memory. Therefore if one insists on being precise, it is not all of the memory that is central, merely the main memory. Such systems are said to have non-uniform memory access (NUMA).

A big research issue for shared memory machines is the cache coherence problem. All fast processors today have caches. Suppose the cache can contain a copy of any memory location in the machine. Since the caches are distributed, it is possible that P2 can overwrite the value of x in P2's own cache and main memory, while P1 might not see this new updated value if P1 only

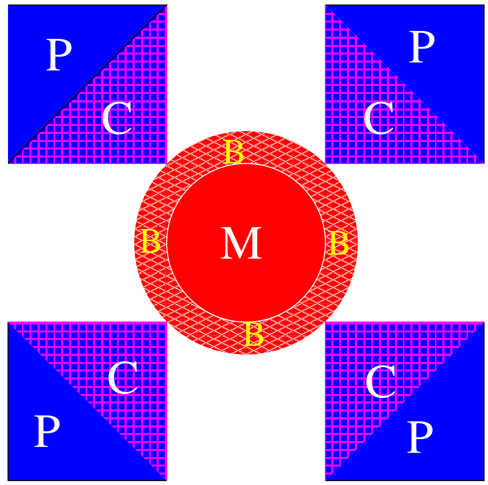


Figure 6.2: A four processor SMP (B denotes the bus between the central memory and the processor's cache

looks at its own cache. Coherent caching means that when the write to x occurs, any cached copy of x will be tracked down by the hardware and invalidated – *i.e.* the copies are thrown out of their caches. Any read of x that occurs later will have to go back to its home memory location to get its new value. Maintenance of cache coherence is expensive for scalable shared memory machines. Today, only the HP Convex machine has scalable, cache coherent, shared memory. Other vendors of scalable, shared memory systems (Kendall Square Research, Evans and Sutherland, BBN) have gone out of the business. Another, Cray, makes a machine (the Cray T3E) in which the caches can only keep copies of local memory locations.

SMPs are often thought of as not scalable (performance peaks at a fairly small number of processors), because as you add processors, you quickly saturate the bus connecting the memory to the processors.

Whenever anybody has a collection of machines, it is always natural to try to hook them up together. Therefore any arbitrary collection of computers can become one big distributed computer. When all the nodes are the same, we say that we have a homogeneous arrangement. It has recently become popular to form high speed connections between SMPs, known as *Constellations of SMPs* or *SMP arrays*. Sometimes the nodes have different architectures creating a heterogeneous situation. Under these circumstances, it is sometimes necessary to worry about the explicit format of data as it is passed from machine to machine.

SIMD machines:
In the late 1980's, there were debates over SIMD versus MIMD. (Either pronounced as SIM-dee/MIM-dee or by reading the letters es-eye-em-dee/em-eye-em-dee.) These two acronyms coined by Flynn in his classification of machines refer to **Single Instruction Multiple Data** and **Multiple Instruction Multiple Data**. The second two letters, "MD" for multiple data, refer to the ability to work on more than one operand at a time. The "SI" or "MI" refer to the ability of a processor to issue instructions of its own. Most current machines are MIMD machines. They are

built from microprocessors that are designed to issue instructions on their own. One might say that each processor has a brain of its own and can proceed to compute anything it likes independent of what the other processors are doing. On a SIMD machine, every processor is executing the same instruction, an add say, but it is executing on different data.

SIMD machines need not be as rigid as they sound. For example, each processor had the ability to not store the result of an operation. This was called em context. If the contest was false, the result was not stored, and the processor appeared to not execute that instruction. Also the CM-2 had the ability to do indirect addressing, meaning that the physical address used by a processor to load a value for an add, say, need not be constant over the processors.

The most important SIMD machines were the Connection Machines 1 and 2 produced by Thinking Machines Corporation, and the MasPar MP-1 and 2. The SIMD market received a serious blow in 1992, when TMC announced that the CM-5 would be a MIMD machine.

Now the debates are over. MIMD has won. The prevailing theory is that because of the tremendous investment by the personal computer industry in commodity microprocessors, it will be impossible to stay on the same steep curve of improving performance using any other processor technology. *"No one will survive the attack of the killer micros!"* said Eugene Brooks of the Lawrence Livermore National Lab. He was right. The supercomputing market does not seem to be large enough to allow vendors to build their own custom processors. And it is not realistic or profitable to build an SIMD machine out of these microprocessors. Furthermore, MIMD is more flexible than SIMD; there seem to be no big enough market niches to support even a single significant vendor of SIMD machines.

> **A close look at the SIMD argument:**
> In some respects, SIMD machines are faster from the communications viewpoint. They can communicate with minimal latency and very high bandwidth because the processors are always in synch. The Maspar was able to do a circular shift of a distributed array, or a broadcast, in less time than it took to do a floating point addition. So far as we are aware, no MIMD machine in 1996 has a latency as small as the 24 μsec overhead required for one hop in the 1988 CM-2 or the 8 μsec latency on the Maspar MP-2.
>
> Admitting that certain applications are more suited to SIMD than others, we were among many who thought that SIMD machines ought to be cheaper to produce in that one need not devote so much chip real estate to the ability to issue instructions. One would not have to replicate the program in every machine's memory. And communication would be more efficient in SIMD machines. Pushing this theory, the potentially fastest machines (measured in terms of raw performance if not total flexibility) should be SIMD machines. In its day, the MP-2 was the world's most cost-effective machine, as measured by the NAS Parallel Benchmarks. These advantages, however, do not seem to have been enough to overcome the relentless, amazing, and wonderful performance gains of the "killer micros".

Continuing with the Flynn classification (for historical purposes) **Single Instruction Single Data** or SISD denotes the sequential (or Von Neumann) machines that are on most of our desktops and in most of our living rooms. (Though most architectures show some amount of parallelism at some level or another.) Finally, there is **Multiple Instruction Single Data** or MISD, a class which seems to be without any extant member although some have tried to fit systolic arrays into this ill-fitting suit.

There have also been hybrids; the PASM Project (at Purdue University) has investigated the problem of running MIMD applications on SIMD hardware! There is, of course, some performance penalty.

Vector Supercomputers:

A vector computer today is a central, shared memory MIMD machine in which every processor has some pipelined arithmetic units and has vector instructions in its repertoire. A vector instruction is something like "add the 64 elements of vector register 1 to the 64 elements of vector register 2", or "load the 64 elements of vector register 1 from the 64 memory locations at addresses $x, x+10, x+20, \ldots, x+630$." Vector instructions have two advantages: fewer instructions fetched, decoded, and issued (since one instruction accomplishes a lot of computation), and predictable memory accesses that can be optimized for high memory bandwidth. Clearly, a single vector processor, because it performs identical operations in a vector instruction, has some features in common with SIMD machines. If the vector registers have p words each, then a vector processor may be viewed as an SIMD machine with shared, central memory, having p processors.

Hardware Technologies and Supercomputing:

Vector supercomputes have very fancy integrated circuit technology (bipolar ECL logic, fast but power hungry) in the processor and the memory, giving very high performance compared with other processor technologies; however, that gap has now eroded to the point that for most applications, fast microprocessors are within a factor of two in performance. Vector supercomputer processors are expensive and require unusual cooling technologies. Machines built of gallium arsenide, or using Josephson junction technology have also been tried, and none has been able to compete successfully with the silicon, CMOS (complementary, metal-oxide semiconductor) technology used in the PC and workstation microprocessors. Thus, from 1975 through the late 1980s, supercomputers were machines that derived their speed from uniprocessor performance, gained through the use of special hardware technologies; now supercomputer technology is the same as PC technology, and parallelism has become the route to performance.

6.0.1 More on private versus shared addressing

Both forms of addressing lead to difficulties for the programmer. In a shared address system, the programmer must insure that any two processors that access the same memory location do so in the correct order: for example, processor one should not load a value from location N until processor zero has stored the appropriate value there (this is called a "true" or "flow" dependence); in another situation, it may be necessary that processor one not store a new value into location N before processor zero loads the old value (this is an "anti" dependence); finally, if multiple processors write to location N, its final value is determined by the last writer, so the order in which they write is significant (this is called "output" dependence). The fourth possibility, a load followed by another load, is called an "input" dependence, and can generally be ignored. Thus, the programmer can get incorrect code do to "data races". Also, performance bugs due to too many accesses to the same location (the memory bank that holds a given location becomes the sequential bottleneck) are common. [2]

The big problem created by private memory is that the programmer has to distribute the data. "Where's the matrix?" becomes a key issue in building a LINPACK style library for private memory

[2]It is an important problem of the "PRAM" model used in the theory of parallel algorithms that it does not capture this kind of performance bug, and also does not account for communication in NUMA machines.

machines. And communication cost, whenever there is NUMA, is also a critical issue. It has been said that the three most important issues in parallel algorithms are "locality, locality, and locality".[3]

One factor that complicates the discussion is that a layer of software, at the operating system level or just above it, can provide virtual shared addressing on a private address machine by using interrupts to get the help of an owning processor when a remote processor wants to load or store data to its memory. A different piece of software can also segregate the shared address space of a machine into chunks, one per processor, and confine all loads and stores by a processor to its own chunk, while using private address space mechanisms like message passing to access data in other chunks. (As you can imagine, hybrid machines have been built, with some amount of shared and private memory.)

6.0.2 Programming Model

The programming model used may seem to be natural for one style of machine; data parallel programming seems to be a SIMD shared memory style, and message passing seems to favor distributed memory MIMD.

Nevertheless, it is quite feasible to implement data parallelism on distributed memory MIMD machines. For example, on the Thinking Machines CM-5, a user can program in CM-Fortran an array data parallel language, or program in node programs such as C and Fortran with message passing system calls, in the style of MIMD computing. We will discuss the pros and cons of SIMD and MIMD models in the next section when we discuss parallel languages and programming models.

6.0.3 Machine Topology

The two things processors need to do in parallel machines that they do not do when all alone are communication (with other processors) and coordination (again with other processors). Communication is obviously needed: one computes a number that the other requires, for example. Coordination is important for sharing a common pool of resources, whether they are hardware units, files, or a pool of work to be performed. The usual mechanisms for coordination, moreover, involve communication.

Parallel machines differ in their underlying hardware for supporting message passing and data routing.

In a shared memory parallel machine, communication between processors is achieved by access to common memory locations. Access to the common memory is supported by a switch network that connects the processors and memory modules. The set of proposed switch network for shared parallel machines includes crossbars and multistage networks such as the butterfly network. One can also connect processors and memory modules by a bus, and this is done when the number of processors is limited to ten or twenty.

An interconnection network is normally used to connect the nodes of a multicomputer as well. Again, the the network topology varies from one machine to another. Due to technical limitations, most commercially used topologies are of small node degree. Commonly used network topologies include (not exclusively) linear arrays, ring, hierarchical rings, two or three dimension grids or tori, hypercubes, fat trees.

The performance and scalability of a parallel machine in turn depend on the network topology. For example, a two dimensional grid of p nodes has diameter $\sqrt{p}$, on the other hand, the diameter of a hypercube or fat tree of p nodes is $\log p$. This implies that the number of physical steps to

[3]For those too young to have suffered through real estate transactions, the old adage in that business is that the three most important factors in determining the value of a property are "location, location, and location".

send a message from a processor to its most distant processor is $\sqrt{p}$ and $\log p$, respectively, for 2D grid and hypercube of p processors. The node degree of a 2D grid is 4, while the degree of a hypercube is $\log p$. Another important criterion for the performance of a network topology is its *bisection bandwidth*, which is the minimum communication capacity of a set of links whose removal partitions the network into two equal halves. Assuming unit capacity of each direct link, a 2D and 3D grid of p nodes has bisection bandwidth $\sqrt{p}$ and $p^{2/3}$ respectively, while a hypercube of p nodes has bisection bandwidth $\Theta(p/\log p)$. (See FTL page 394)

There is an obvious cost / performance trade-off to make in choosing machine topology. A hypercube is much more expensive to build than a two dimensional grid of the same size. An important study done by Bill Dally at Caltech showed that for randomly generated message traffic, a grid could perform better and be cheaper to build. Dally assumed that the number of data signals per processor was fixed, and could be organized into either four "wide" channels in a grid topology or $\log n$ "narrow" channels (in the first hypercubes, the data channels were bit-serial) in a hypercube. The grid won, because too the average utilization of the hypercube channels was too low: the wires, probably the most critical resource in the parallel machine, were sitting idle. Furthermore, the work on routing technology at Caltech and elsewhere in the mid 80's resulted in a family of hardware routers that delivered messages with very low latency even though the length of the path involved many "hops" through the machines. For the earliest multicomputers used "store and forward" networks, in which a message sent from A through B to C was copied into and out of the memory of the intermediate node B (and any others on the path): this causes very large latencies that grew in proportion to the number of hops. Later routers, including those used in todays networks, have a "virtual circuit" capability that avoids this copying and results in small latencies.

Does topology make any real difference to the performance of parallel machines in practice? Some may say "yes" and some may say "no". Due to the small size (less than 512 nodes) of most parallel machine configurations and large software overhead, it is often hard to measure the performance of interconnection topologies at the user level.

6.0.4 Homogeneous and heterogeneous machines

Another example of cost / performance trade-off is the choice between tightly coupled parallel machines and workstation clusters, workstations that are connected by fast switches or ATMs. The networking technology enables us to connect heterogeneous machines (including supercomputers) together for better utilization. Workstation clusters may have better cost/efficient trade-offs and are becoming a big market challenger to "main-frame" supercomputers.

A parallel machine can be *homogeneous* or *heterogeneous*. A homogeneous parallel machine uses identical node processors. Almost all tightly coupled supercomputers are homogeneous. Workstation clusters may often be heterogeneous. The Cray T3D is in some sense a heterogeneous parallel system which contains a vector parallel computer C90 as the front end and the massively parallel section of T3D. (The necessity of buying the front-end was evidently not a marketing plus: the T3E does not need one.) A future parallel system may contains a cluster of machines of various computation power from workstations to tightly coupled parallel machines. The scheduling problem will inevitably be much harder on a heterogeneous system because of the different speed and memory capacity of its node processors.

More than 1000 so called supercomputers have been installed worldwide. In US, parallel machines have been installed and used at national research labs (Los Almos National Laboratory, Sandia National Labs, Oak Ridge National Laboratory, Lawrence Livermore National Laboratory, NASA Ames Research Center, US Naval Research Laboratory, DOE/Battis Atomic Power Laboratory, etc) supercomputing centers (Minnesota Supercomputer Center, Urbana-Champaign NCSA, Pittsburgh Supercomputing Center, San Diego Supercomputer Center, etc) US Government, and commercial companies (Ford Motor Company, Mobil, Amoco) and major universities. Machines from different supercomputing companies look different, are priced differently, and are named differently. Here are the names and birthplaces of some of them.

- Cray T3E (MIMD, distributed memory, 3D torus, uses Digital Alpha microprocessors), C90 (vector), Cray YMP, from Cray Research, Eagan, Minnesota.
- Thinking Machine CM-2 (SIMD, distributed memory, almost a hypercube) and CM-5 (SIMD and MIMD, distributed memory, Sparc processors with added vector units, fat tree) from Thinking Machines Corporation, Cambridge, Massachusetts.
- Intel Delta, Intel Paragon (mesh structure, distributed memory, MIMD), from Intel Corporations, Beaverton, Oregon. Based on Intel i860 RISC, but new machines based on the P6. Recently sold world's largest computer (over 6,000 P6 processors) to the US Dept of Energy for use in nuclear weapons stockpile simulations.
- IBM SP-1, SP2, (clusters, distributed memory, MIMD, based on IBM RS/6000 processor), from IBM, Kingston, New York.
- MasPar, MP-2 (SIMD, small enough to sit next to a desk), by MasPar, Santa Clara, California.
- KSR-2 (global addressable memory, hierarchical rings, SIMD and MIMD) by Kendall Square, Waltham, Massachusetts. Now out of the business.
- Fujitsu VPX200 (multi-processor pipeline), by Fujitsu, Japan.
- NEC SX-4 (multi-processor vector, shared and distributed memory), by NEC, Japan.
- Tera MTA (MPP vector, shared memory, multithreads, 3D torus), by Tera Computer Company, Seattle, Washington. A novel architecture which uses the ability to make very fast context switches between threads to hide latency of access to the memory.
- Meiko CS-2HA (shared memory, multistage switch network, local I/O device), by Meiko Concord, Massachusetts and Bristol UK.
- Cray-3 (gallium arsenide integrated circuits, multiprocessor, vector) by Cray Computer Corporation, Colorado Spring, Colorado. Now out of the business.

6.0.5 Distributed Computing on the Internet and Akamai Network

Examples of distributed computing on the internet:

- seti@home: "a scientific experiment that uses Internet-connected computers to downloads and analyzes radio telescope data". When we input this item, its performance is 26.73 Teraflops per second.
- Distributed.net: to use the idle processing time of its thousands member computers to solve computationally intensive problems. Its computing power now is equivalent to that of "more than 160000 PII 266MHz computers".

- Parabon: recycle computer's idle time for bioinformatic computations.
- Google Compute: Runs as part of the Google Toolbar within a user's browser. Detects spare cycles on the machine and puts them to use solving scientific problems selected by Google.

Akamai Network consists of thousands servers spread globally that cache web pages and route traffic away from congested areas. This idea was originated by Tom Leighton and Danny Lewin at MIT.

Lecture 7

FFT

7.1 FFT

The *Fast Fourier Transform* is perhaps the most important subroutine in scientific computing. It has applications ranging from multiplying numbers and polynomials to image and signal processing, time series analysis, and the solution of linear systems and PDEs. There are tons of books on the subject including two recent worderful ones by Charles van Loand and Briggs.

The discrete Fourier transform of a vector x is $y = F_n x$, where F_n is the $n \times n$ matrix whose entry $(F_n)_{jk} = e^{-2\pi ijk/n}$, $j, k = 0 \ldots n-1$. It is nearly always a good idea to use 0 based notation (as with the C programming language) in the context of the discrete Fourier transform. The negative exponent corresponds to Matlab's definition. Indeed in matlab we obtain `fn=fft(eye(n))`.

A good example is

$$F_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix}.$$

Sometimes it is convenient to denote $(F_n)_{jk} = \omega_n^{jk}$, where $\omega_n = e^{-2\pi/n}$.

The Fourier matrix has more interesting properties than any matrix deserves to have. It is symmetric (but not Hermitian). It is Vandermonde (but not ill-conditioned). It is unitary except for a scale factor ($\frac{1}{\sqrt{n}} F_n$ is unitary). In two ways the matrix is connected to group characters: the matrix itself is the character table of the finite cyclic group, and the eigenvectors of the matrix are determined from the character table of a multiplicative group.

The trivial way to do the Fourier transform is to compute the matrix-vector multiply requiring n^2 multipilications and roughly the same number of additions. Cooley and Tukey gave the first $O(n \log n)$ time algorithm (actually the algorithm may be found in Gauss' work) known today as the FFT algorithm. We shall assume that $n = 2^p$.

The Fourier matrix has the simple property that if Π_n is an unshuffle operation, then

$$F_n \Pi_n^T = \begin{pmatrix} F_{n/2} & D_n F_{n/2} \\ F_{n/2} & -D_n F_{n/2} \end{pmatrix}, \tag{7.1}$$

where D_n is the diagonal matrix $\mathrm{diag}(1, \omega_n, \ldots, \omega_n^{n/2-1})$.

One DFT algorithm is then simply: 1) unshuffle the vector 2) recursively apply the FFT algorithm to the top half and the bottom half, then combine elements in the top part with corresponding elements in the bottom part ("the butterfly") as prescribed by the matrix $\begin{pmatrix} I & D_n \\ I & -D_n \end{pmatrix}$.

Everybody has their favorite way to visualize the FFT algorithm. For us, the right way is to think of the data as living on a hypercube. The algorithm is then, permute the cube, perform the FFT on a pair of opposite faces, and then perform the butterfly, along edges across the dimension connecting the opposite faces.

We now repeat the three steps of the recursive algorithm in index notation:

- Step 1: $i_{d-1} \dots i_1 i_0 \to i_0 i_{d-1} \dots i_1$
- Step 2: $i_0 i_{d-1} \dots i_1 \to i_0 \text{fft}(i_{d-1} \dots i_1)$
- Step 3: $\to \bar{i_0} \text{fft}(i_{d-1} \dots i_1)$

Here Step 1 is a data permutation, Step 2 refters to two FFTs, and Step 3 is the butterfly on the high order bit.

In conventional notation:

$$y_j = (F_n x)_j = \sum_{k=0}^{n-1} \omega_n^{jk} x_k$$

can be cut into the even and the odd parts:

$$y_j = \sum_{k=0}^{m-1} \omega_n^{2jk} x_{2k} + \omega_n^j \left(\sum_{k=0}^{m-1} \omega_n^{2jk} x_{2k+1} \right) \; ;$$

since $\omega_n^2 = \omega_m$, the two sums are just $\text{FFT}(x_{\text{even}})$ and $\text{FFT}(x_{\text{odd}})$. With this remark (see Fig. 1),

$$\begin{array}{rclcl} y_j & = & \sum_{k=0}^{m-1} \omega_m^{jk} x_{2k} & + & \omega_n^j \left(\sum_{k=0}^{m-1} \omega_m^{jk} x_{2k+1} \right) \\ y_{j+m} & = & \sum_{k=0}^{m-1} \omega_m^{jk} x_{2k} & - & \omega_n^j \left(\sum_{k=0}^{m-1} \omega_m^{jk} x_{2k+1} \right) \end{array} .$$

Then the algorithm keeps recurring; the entire "communication" needed for an FFT on a vector of length 8 can be seen in Fig. 2

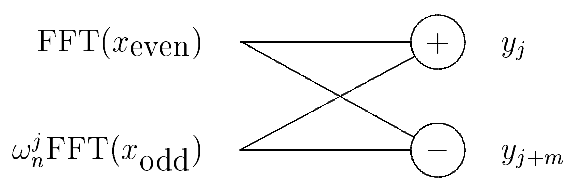


Figure 7.1: Recursive block of the FFT.

The number of operations for an FFT on a vector of length n equals to twice the number for an FFT on length $n/2$ plus $n/2$ on the top level. As the solution of this recurrence, we get that the total number of operations is $\frac{1}{2} n \log n$.

Now we analyze the data motion required to perform the FFT. First we assume that to each processor one element of the vector x is assigned. Later we discuss the "real-life" case when the number of processors is less than n and hence each processor has some subset of elements. We also discuss how FFT is implemented on the CM-2 and the CM-5.

The FFT always goes from high order bit to low order bit, i.e., there is a fundamental asymmetry that is not evident in the figures below. This seems to be related to the fact that one can obtain a subgroup of the cyclic group by alternating elements, but not by taking, say, the first half of the elements.

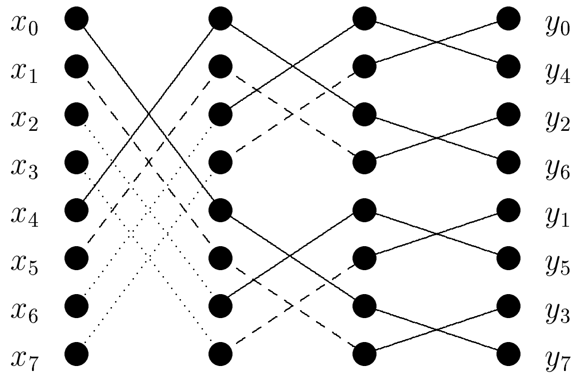


Figure 7.2: FFT network for 8 elements. (Such a network is not built in practice)

7.1.1 Data motion

Let $i_p i_{p-1} \ldots i_2 i_1 i_0$ be a bit sequence. Let us call $i_0 i_1 i_2 \ldots i_{p-1} i_p$ the **bit reversal** of this sequence. The important property of the FFT network is that if the i-th input is assigned to the i-th processor for $i \le n$, then the i-th output is found at the processor with address *the bit-reverse of* i. Consequently, if the input is assigned to processors with bit-reversed order, then the output is in standard order. The inverse FFT reverses bits in the same way.

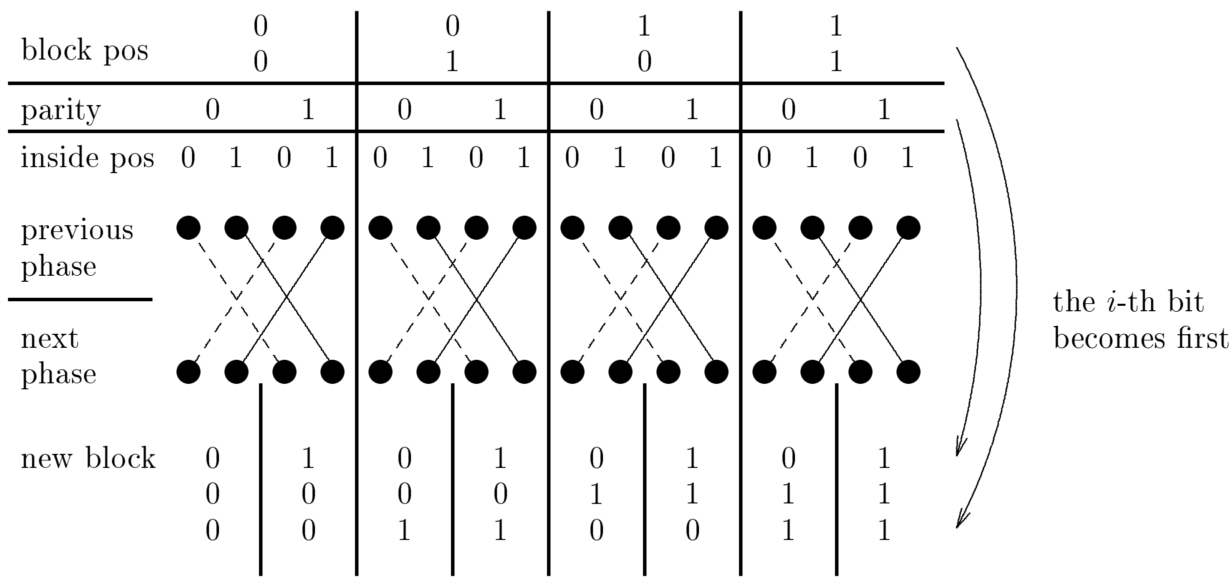


Figure 7.3: The output of FFT is in bit-reversed order.

To see why FFT reverses the bit order, let us have a look at the i-th segment of the FFT network (Fig. 3). The input is divided into parts and the current input (top side) consists of FFT's of these parts. One "block" of the input consists of the same fixed output element of all the parts. The $i - 1$ most significant bits of the input address determine this output element, while the least significant bits the the part of the original input whose transforms are at this level.

The next step of the FFT computes the Fourier transform of twice larger parts; these consist

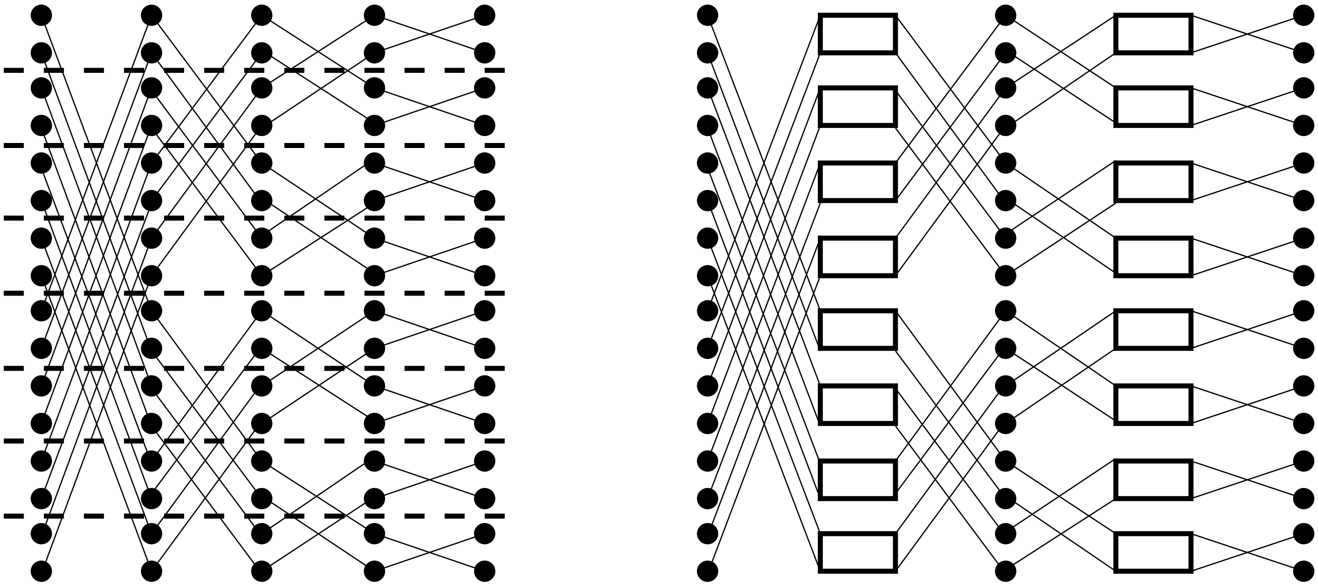


Figure 7.4: Left: more than one element per processor. Right: one box is a 4×4 matrix multiply.

of an "even" and an "odd" original part. Parity is determined by the i-th most significant bit.

Now let us have a look at one unit of the network in Fig. 1; the two inputs correspond to the same even and odd parts, while the two outputs are the possible "farthest" vector elements, they differ in the most significant bit. What happens is that the i-th bit jumps first and becomes most significant (see Fig. 3).

Now let us follow the data motion in the entire FFT network. Let us assume that the i-th input element is assigned to processor i. Then after the second step a processor with binary address $i_p i_{p-1} i_{p-2} \ldots i_1 i_0$ has the $i_{p-1} i_p i_{p-2} \ldots i_1 i_0$-th data, the second bit jumps first. Then the third, fourth, ..., p-th bits all jump first and finally that processor has the $i_0 i_1 i_2 \ldots i_{p-1} i_p$-th output element.

7.1.2 FFT on parallel machines

In a realistic case, on a parallel machine some bits in the input address are local to one processor. The communication network can be seen in Fig. 4, left. FFT requires a large amount of communication; indeed it has fewer operations per communication than usual dense linear algebra. One way to increase this ratio is to *combine some layers* into one, as in Fig. 4, right. If s consecutive layers are combined, in one step a $2^s \times 2^s$ matrix multiplication must be performed. Since matrix multiplication vectorizes and usually there are optimized routines to do it, such a step is more efficient than communicating all small parts. Such a modified algorithm is called the *High Radix* FFT.

The FFT algorithm on the CM-2 is basically a High Radix FFT. However, on the CM-5 data motion is organized in a different way. The idea is the following: if s bits of the address are local to one processor, the last s phases of the FFT do not require communication. Let 3 bits be local to one processor, say. On the CM-5 the following data rearrangement is made: the data from the

$$i_p i_{p-1} i_{p-2} \ldots i_3 | i_2 i_1 i_0 \text{ -th}$$

processor is moved to the

$$i_2 i_1 i_0 i_{p-3} i_{p-4} \dots i_3 | i_p i_{p-1} i_{p-2} \text{ -th!}$$

This data motion can be arranged in a clever way; after that the next 3 steps are local to processors. Hence the idea is to perform all communication at once *before* the actual operations are made.

[Not yet written: The six step FFT]

[Not yet written: FFTW]

[Good idea: Use the picture in our paper first to illustrate the notation]

7.1.3 Exercises

1. Verify equation (**??**).

2. Just for fun, find out about the FFT through Matlab.

   We are big fans of the `phone` command for those students who do not already have a good physical feeling for taking Fourier transforms. This command shows (and plays if you have a speaker!) the signal generated when pressing a touch tone telephone in the United States and many other countries. In the old days, when a pushbutton phone was broken apart, you could see that pressing a key depressed one lever for an entire row and another lever for an entire column. (For example, pressing 4 would depress the lever corresponding to the second row and the lever corresponding to the first column.)

   To look at the FFT matrix, in a way, `plot(fft(eye(7)));axis('square')`.

3. In Matlab use the `flops` function to obtain a flops count for FFT's for different power of 2 size FFT's. Make you input complex. Guess a flop count of the form $a + bn + c\log n + dn\log n$. Remembering that Matlab's \ operator solves least squares problems, find a, b, c and d. Guess whether Matlab is counting flops or using a formula.

7.2 Matrix Multiplication

Everyone thinks that to multiply two 2-by-2 matrices requires 8 multiplications. However, Strassen gave a method, which requires only 7 multiplications! Actually, compared to the 8 multiplies and 4 adds of the traditional way, Strassen's method requires only 7 multiplies but 18 adds. Nowadays when multiplication of numbers is as fast as addition, this does not seem so important. However when we think of block matrices, matrix multiplication is very slow compared to addition. Strassen's method will give an $O(n^{2.8074})$ algorithm for matrix multiplication, in a recursive way very similar to the FFT.

First we describe Strassen's method for two block matrices:

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \times \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} = \begin{pmatrix} P_1 + P_4 - P_5 + P_7 & P_3 + P_5 \\ P_2 + P_4 & P_1 + P_3 - P_2 + P_6 \end{pmatrix}$$

where

$$\begin{aligned} P_1 &= (A_{1,1} + A_{1,2})(B_{1,1} + B_{2,2}) \ , \\ P_2 &= (A_{2,1} + A_{2,2})B_{1,1} \ , \\ P_3 &= A_{1,1}(B_{1,2} - B_{2,2}) \ , \\ P_4 &= A_{2,2}(B_{2,1} - B_{1,1}) \ , \\ P_5 &= (A_{1,1} + A_{1,2})B_{2,2} \ , \end{aligned}$$

$$
\begin{aligned}
P_6 &= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}) \ , \\
P_7 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}) \ .
\end{aligned}
$$

If, as in the FFT algorithm, we assume that $n = 2^p$, the matrix multiply of two n-by-n matrices calls 7 multiplications of $(n/2)$-by-$(n/2)$ matrices. Hence the time required for this algorithm is $O(n^{\log_2 7}) = O(n^{2.8074})$. Note that Strassen's idea can further be improved (of course, with the loss that several additions have to be made and the constant is impractically large) the current such record is an $O(n^{2.376})$-time algorithm.

A final note is that, again as in the FFT implementations, we do not recur and use Strassen's method with 2-by-2 matrices. For some sufficient p, we stop when we get $2^p \times 2^p$ matrices and use direct matrix multiply which vectorizes well on the machine.

7.3 Basic Data Communication Operations

We conclude this section by list the set of basic data communication operations that are commonly used in a parallel program.

- **Single Source Broadcast**:
- **All-to-All Broadcast**:
- **All-to-All Personalized Communication**:
- **Array Indexing or Permutation**: There are two types of array indexing: the *left* array indexing and the *right* array indexing.
- **Polyshift**: SHIFT and EOSHIFT.
- **Sparse Gather and Scatter**:
- **Reduction and Scan**:

Lecture 8

Domain Decomposition

Domain decomposition is a term used by at least two different communities. Literally, the words indicate the partitioning of a region. As we will see in Chapter **??** of this book, an important computational geometry problem is to find good ways to partition a region. This is not what we will discuss here.

In scientific computing, domain decomposition refers to the technique of solving partial differential equations using subroutines that solve problems on subdomains. Originally, a domain was a contiguous region in space, but the idea has generalized to include any useful subset of the discretization points. Because of this generalization, the distinction between domain decomposition and multigrid has become increasingly blurred.

Domain decomposition is an idea that is already useful on serial computers, but it takes on a greater importance when one considers a parallel machine with, say, a handful of very powerful processors. In this context, domain decomposition is a parallel divide-and-conquer approach to solving the PDE.

To guide the reader, we quickly summarize the choice space that arises in the domain decomposition literature. As usual a domain decomposition problem starts as a continuous problem on a region and is disretized into a finite problem on a discrete domain.

We will take as our model problem the solution of the elliptic equation $\nabla^2 u = f$, where on a region Ω which is the union of at least subdomains Ω_1 and Ω_2. ∇^2 is the Laplacian operator, defined by $\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$. Domain decomposition ideas tend to be best developed for elliptic problems, but may be applied in more general settings.

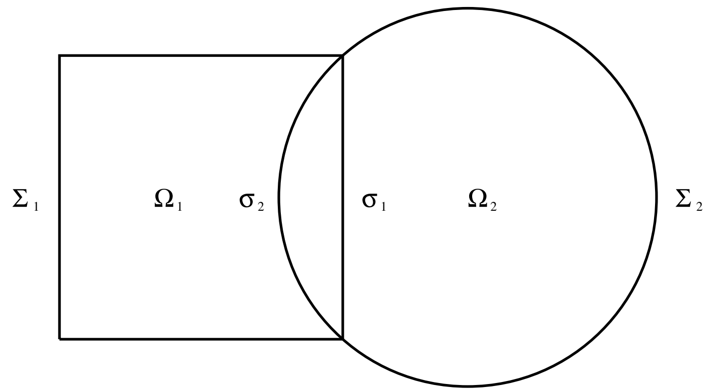


Figure 8.1: Example domain of circle and square with overlap

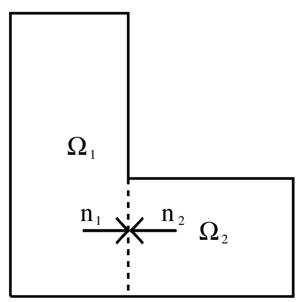


Figure 8.2: Domain divided into two subdomains without overlap

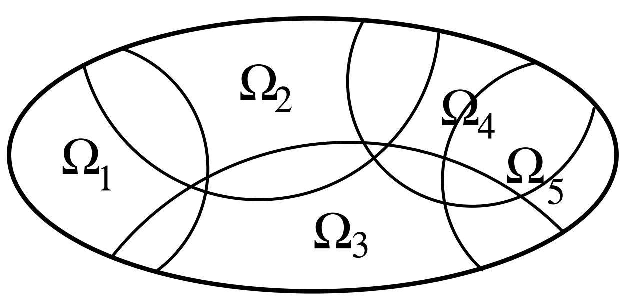


Figure 8.3: Example domain with many overlapping regions

Domain Decomposition Outline

1. Geometric Issues
   - Overlapping or non-overlapping regions
   - Geometric Discretization
     - Finite Difference or Finite Element
     - Matching or non-matching grids
2. Algorithmic Issues
   - Algebraic Discretization
     - Schwarz Approaches: Additive vs. Multiplicative
     - Substructuring Approaches
   - Accelerants
     - Domain Decomposition as a Preconditioner
     - Course (Hierarchical/Multilevel) Domains
3. Theoretical Considerations

8.1 Geometric Issues

The geometric issues in domain decomposition are 1) how are the domains decomposed into subregions, and 2) how is the region discretized using some form of grid or irregular mesh. We consider these issues in turn.

8.1.1 Overlapping vs. Non-overlapping regions

So as to emphasize the issue of overlap vs. non-overlap, we can simplify all the other issues by assuming that we are solving the continuous problem (no discretization) exactly on each domain (no choice of algorithm). The reader may be surprised to learn that domain decomposition methods divide neatly into either being overlapping or nonoverlapping methods. Though one can find much in common between these two methods, they are really rather different in flavor. When there is overlap, the methods are sometimes known as Schwarz methods, while when there is no overlap, the methods are sometimes known as substructuring. (Historically, the former name was used in the continuous case, and the latter in the discrete case, but this historical distinction has been, and even should be, blurred.)

We begin with the overlapping region illustrated in Figure 8.1. Schwarz in 1870 devised an obvious alternating procedure for solving Poisson's equation $\nabla^2 u = f$:

1. Start with any guess for u_2 on σ_1.
2. Solve $\nabla^2 u_1 = f$ on Ω_1 by taking $u_1 = u_2$ on σ_1. (i.e. solve in the square using boundary data from the interior of the circle)
3. Solve $\nabla^2 u_2 = f$ on Ω_2 by taking $u_2 = u_1$ on σ_2 (i.e. solve in the circle using boundary data from the interior of the square)
4. Goto 2 and repeat until convergence is reached

The procedure above is illustrated in Figure 8.4.

One of the characteristics of elliptic PDE's is that the solution at every point depends on global conditions. The information transfer between the regions clearly occurs in the overlap region.

If we "choke" the transfer of information by considering the limit as the overlap area tends to 0, we find ourselves in a situation typified by Figure 8.2. The basic Schwarz procedure no longer works. Do you see why? No matter what the choice of data on the interface, it would not be updated. The result would be that the solution would not be differentiable along the interface. An example is given in Figure 8.5.

One approach to solving the non-overlapped problem is to concentrate directly on the domain of intersection. Let g be a current guess for the solution on the interface. We can then solve $\nabla^2 u = f$ on Ω_1 and Ω_2 independently using the value of g as Dirichlet conditions on the interface. We can define the map

$$T : g \rightarrow \frac{\partial g}{\partial n_1} + \frac{\partial g}{\partial n_2}.$$

This is an affine map from functions on the interface to functions on the interface defined by taking a function to the jump in the derivative. The operator T is known as the Steklov-Poincaré operator.

Suppose we can find the exact solution to $Tg = 0$. We would then have successfully decoupled the problem so that it may be solved independently into the two domains Ω_1 and Ω_2. This is a "textbook" illustration of the divide and conquer method, in that solving $Tg = 0$ constitutes the "divide."

8.1.2 Geometric Discretization

In the previous section we contented ourselves with formulating the problem on a continuous domain, and asserted the existence of solutions either to the subdomain problems in the Schwarz case, or the Stekhlov-Poincaré operator in the continuous case.

Of course on a real computer, a discretization of the domain and a corresponding discretization of the equation is needed. The result is a linear system of equations.

Finite Differences or Finite Elements

Finite differences is actually a special case of finite elements, and all the ideas in domain decomposition work in the most general context of finite elements. In finite differences, one typically imagines a square mesh. The prototypical example is the five point stencil for the Laplacian in two dimensions. Using this stencil, the continuous equation $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y)$ is transformed to a linear system of equations of the form:

$$-\frac{1}{h^2}(-u_i{}^E + 2u_i - u_i{}^W) - \frac{1}{h^2}(-u_i{}^N + 2u_i - u_i{}^S) = 4f_i$$

where for each u_i, $u_i{}^E$ is the element to the right of u_i, $u_i{}^W$ is to the left of u_i, $u_i{}^N$ is above u_i, and $u_i{}^S$ is below u_i. An analog computer to solve this problem would consist of a grid of one ohm resistors. In finite elements, the protypical example is a triangulation of the region, and the appropriate formulation of the PDE on these elements.

Matching vs. Non-matching grids

When solving problems as in our square-circle example of Figure 8.1, it is necessary to discretize the interior of the regions with either a finite difference style grid or a finite element style mesh. The square may be nicely discretized by covering it with Cartesian graph-paper, while the circle may

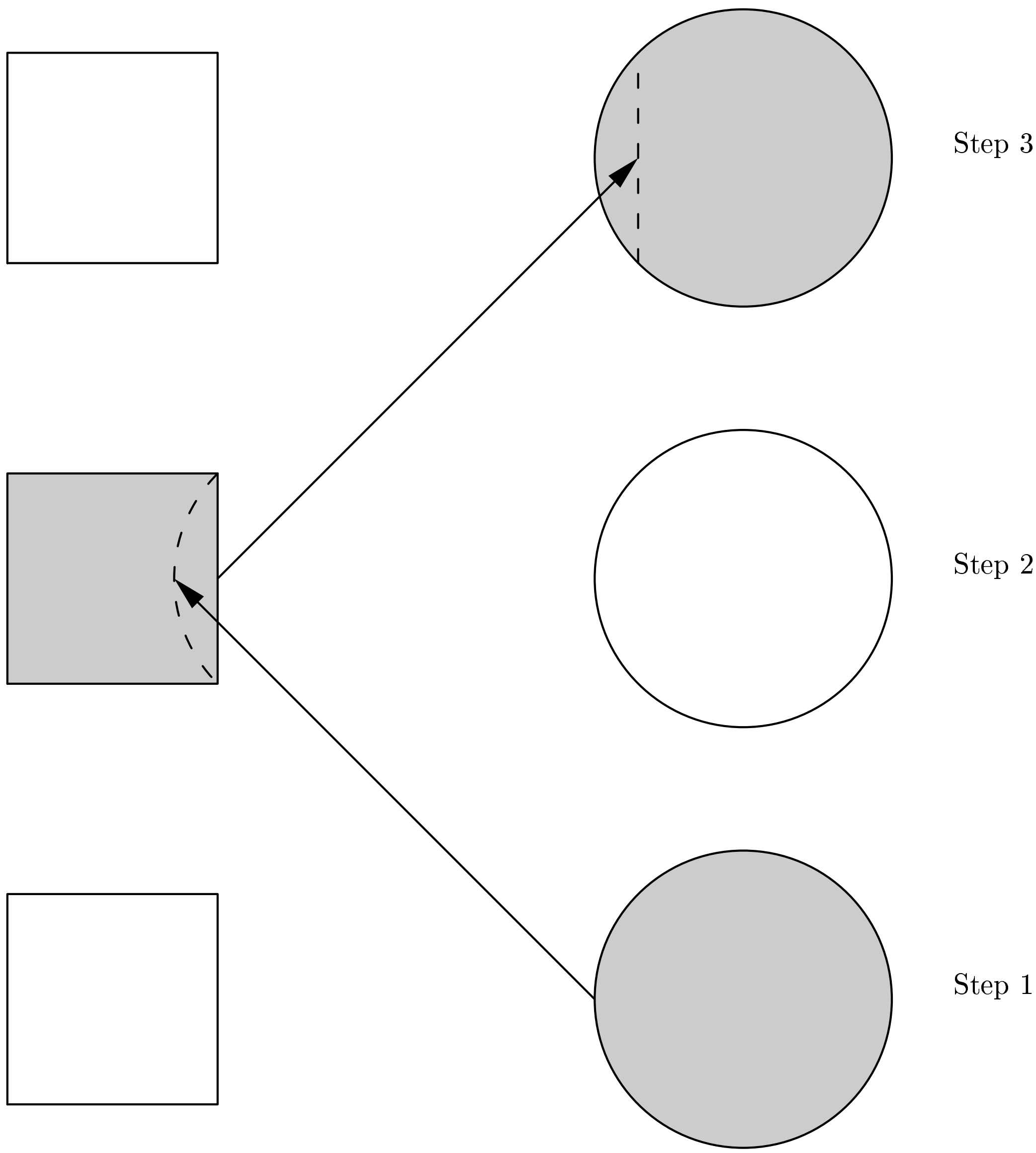


Figure 8.4: Schwarz' alternating procedure

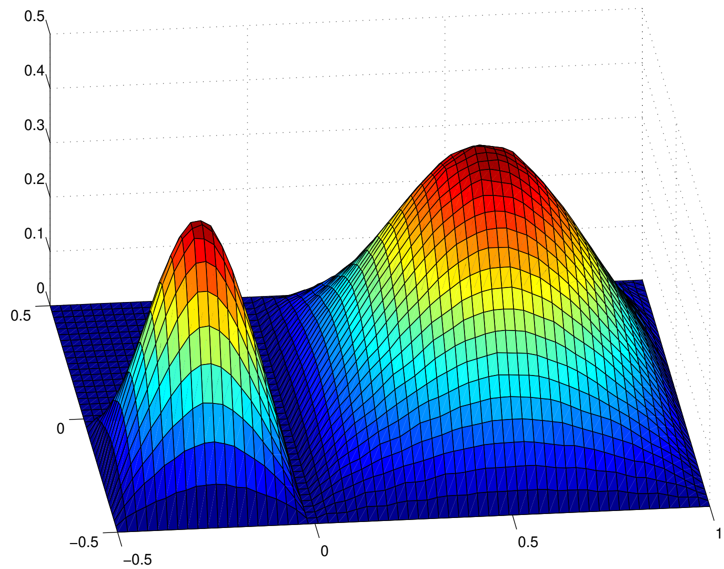


Figure 8.5: Incorrect solution for non-overlaped problem. The result is not differentiable along the boundary between the two regions.

be more conveniently discretized by covering it with polar graph paper. Under such a situation, the grids do not match, and it becomes necessary to transfer points interior to Ω_2 to the boundary of Ω_1 and vice versa. Figure 8.6 shows an example domain with non-mathing grids. Normally, grid values are interpolated for this kind of grid line up pattern.

8.2 Algorithmic Issues

Once the domain is discretized, numerical algorithms must be formulated. There is a definite line drawn between Schwarz (overlapping) and substructuring (non-overlapping) approaches.

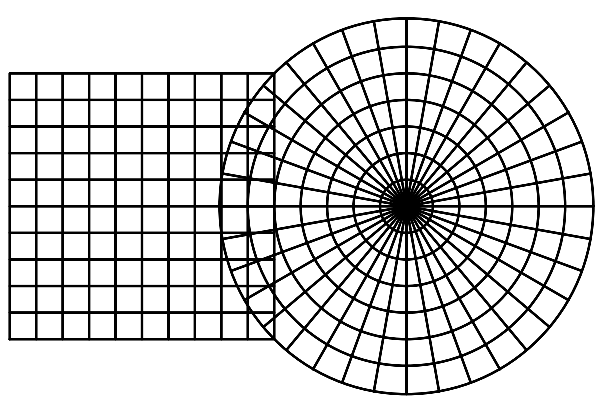


Figure 8.6: Example domain discretized into non-matching grids

8.2.1 Classical Iterations and their block equivalents

Let us review the basic classical methods for solving PDE's
on a discrete domain.

1. Jacobi - At step n, the neighboring values used are from step $n-1$
   Using Jacobi to solve the system Au=f requires using repeated applications of the iteration:

$$u_i{}^{(n+1)} = u_i{}^n + \frac{1}{a_i i}[f_i - \sum_{j \neq i} a_{ij} u_j{}^{(n)}] \ \forall \ i$$

2. Gauss-Seidel - Values at step n are used if available, otherwise the values are used from step $n-1$
   Gauss-Seidel uses applications the iteration:

$$u_i{}^{(n+1)} = u_i{}^n + \frac{1}{a_{ii}}[f_i - \sum_{j<i} a_{ij} u_j{}^{(n+1)} - \sum_{j>i} a_{ij} u_j{}^{(n)}] \ \forall \ i$$

3. Red Black Ordering - If the grid is a checkerboard, solve all red points in parallel using black values at $n-1$, then solve all black points in parallel using red values at step n For the checkerboard, this corresponds to the pair of iterations:

$$u_i{}^{(n+1)} = u_i{}^n + \frac{1}{a_i i}[f_i - \sum_{j \neq i} a_{ij} u_j{}^{(n)}] \ \forall \ i \ even$$

$$u_i{}^{(n+1)} = u_i{}^n + \frac{1}{a_i i}[f_i - \sum_{j \neq i} a_{ij} u_j{}^{(n+1)}] \ \forall \ i \ odd$$

Analogous *block* methods may be used on a domain that is decomposed into a number of multiple regions. Each region is thought of as an element used to solve the larger problem. This is known as block Jacobi, or block Gauss-Seidel.

1. Block Gauss-Seidel - Solve each region in series using the boundary values at n if available.

2. Block Jacobi - Solve each region on a separate processor in parallel and use boundary values at $n-1$. (Additive scheme)

3. Block coloring scheme - Color the regions so that like colors do not touch and solve all regions with the same color in parallel. (Multiplicative scheme)

The block Gauss-Seidel algorithm is called a multiplicative scheme for reasons to be explained shortly. In a corresponding manner, the block Jacobi scheme is called an additive scheme.

8.2.2 Schwarz approaches: additive vs. multiplicative

A procedure that alternates between solving an equation in Ω_1 and then Ω_2 does not seem to be parallel at the highest level because if processor 1 contains all of Ω_1 and processor 2 contains all of Ω_2 then each processor must wait for the solution of the other processor before it can execute. Figure 8.4 illustrates this procedure. Such approaches are known as multiplicative approaches because of the form of the operator applied to the error. Alternatively, approaches that allow for the solution of subproblems simultaneously are known as additive methods. The latter is illustrated in Figure 8.7. The difference is akin to the difference between Jacobi and Gauss-Seidel.

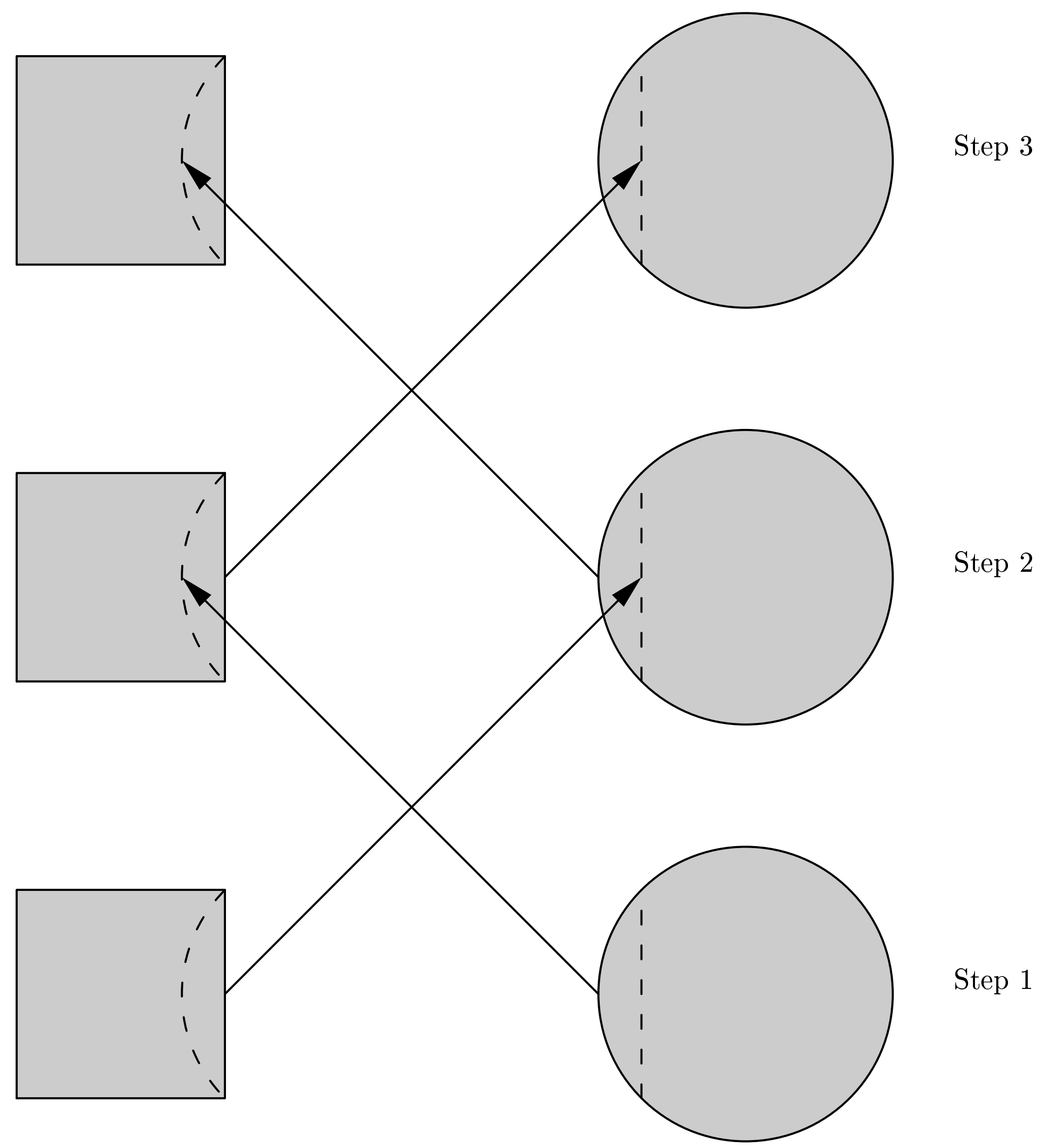


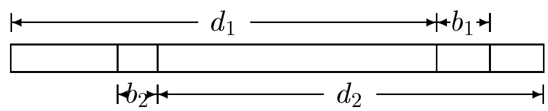
Figure 8.7: Schwarz' alternating procedure (additive)

Overlapping regions: A notational nightmare?

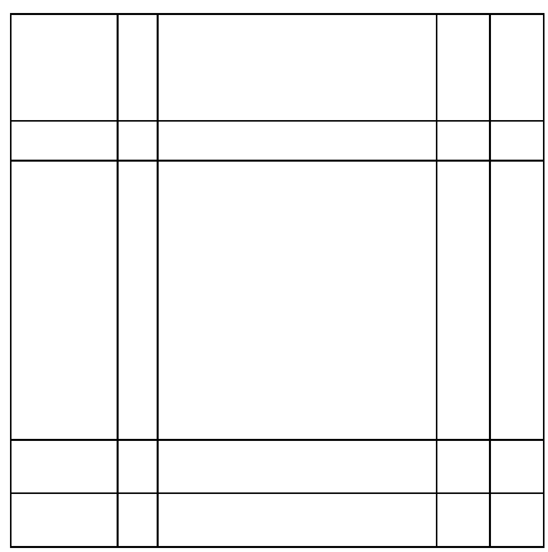
When the grids match it is somewhat more convenient to express the discretized PDE as a simple matrix equation on the gridpoints.

Unfortunately, we have a notational difficulty at this point. It is this difficulty that is probably the single most important reason that domain decomposition techniques are not used as extensively as they can be. Even in the two domain case, the difficulty is related to the fact that we have domains 1 and 2 that overlap each other and have internal and external boundaries. By setting the boundary to 0 we can eliminate any worry of external boundaries. I believe there is only one reasonable way to keep the notation manageable. We will use subscripts to denote subsets of indices. d_1 and d_2 will represent those nodes in domain 1 and domain 2 respectively. b_1 and b_2 will represent those notes in the boundary of 1 and 2 respectively that are not external to the entire domain.

Therefore u_{d_1} denotes the subvector of u consisting of those elements interior to domain 1, while A_{u_1,b_1} is the rectangular subarray of A that map the interior of domain 1 to the internal boundary of domain 1. If we were to write u^T as a row vector, the components might break up as follows (the overlap region is unusually large for emphasis:)



Correspondingly, the matrix A (which of course would never be written down) has the form



The reader should find A_{b_1,b_1} etc., on this picture. To further simplify notation, we write 1 and 2 for d_1 and d_2,1_b and 2_b for b_1 and b_2, and also use only a single index for a diagonal block of a matrix (i.e. $A_1 = A_{11}$).

Now that we have leisurely explained our notation, we may return to the algebra. Numerical analysts like to turn problems that may seem new into ones that they are already familiar with. By carefully writing down the equations for the procedure that we have described so far, it is possible to relate the classical domain decomposition method to an iteration known as *Richardson iteration.* Richardson iteration solves $Au = f$ by computing $u^{k+1} = u^k + M(f - Au^k)$, where M is a "good" approximation to A^{-1}. (Notice that if $M = A^{-1}$, the iteration converges in one step.)

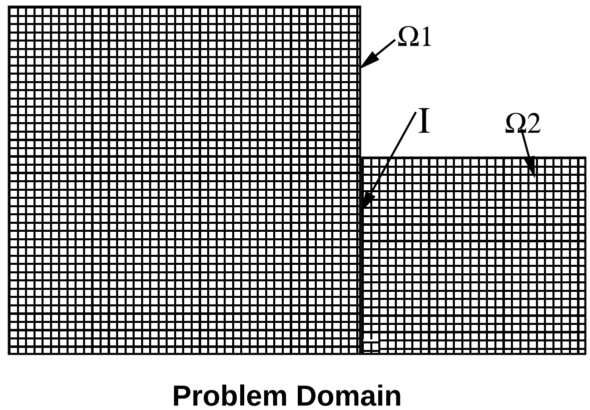


Figure 8.8: Problem Domain

The iteration that we described before may be written algebraically as

$$A_1 u_1^{k+1/2} + A_{1,1_b} u_{1_b}^k = f_1$$

$$A_2 u_2^{k+1} + A_{2,2_b} u_{2_b}^{k+1/2} = f_2$$

Notice that values of $u^{k+1/2}$ updated by the first equation, specifically the values on the boundary of the second region, are used in the second equation.

With a few algebraic manipulations, we have

$$u_1^{k+1/2} = u_1^{k-1} + A_1^{-1}(f - Au^{k-1})_1$$

$$u_2^{k+1} = u_2^{k+1/2} + A_2^{-1}(f - Au^{k+1/2})_2$$

This was already obviously a Gauss-Seidel like procedure, but those of you familiar with the algebraic form of Gauss-Seidel might be relieved to see the form here.

A roughly equivalent block Jacobi method has the form

$$u_1^{k+1/2} = u_1^{k-1} + A_1^{-1}(f - Au^{k-1})_1$$

$$u_2^k = u_2^{k+1/2} + A_2^{-1}(f - Au^k)_2$$

It is possible to eliminate $u^{k+1/2}$ and obtain

$$u^{k+1} = u^k + (A_1^{-1} + A_2^{-1})(f - Au^k),$$

where the operators are understood to apply to the appropriate part of the vectors. It is here that we see that the procedure we described is a Richardson iteration with operator $M = A_1^{-1} + A_2^{-1}$.

8.2.3 Substructuring Approaches

Figure 8.8 shows an example domain of a problem for a network of resistors or a discretized region in which we wish to solve the Poisson equation, $\bigtriangledown^2 v = g$. We will see that the discrete version of the Steklov-Poincaré operator has its algebraic equivalent in the form of the Schur complement.

In matrix notation, $Av = g$, where

$$A = \begin{pmatrix} A_1 & 0 & A_{1I} \\ 0 & A_2 & A_{2I} \\ A_{I1} & A_{I2} & A_I \end{pmatrix}$$

One of the direct methods to solve the above equation is to use LU or LDU factorization. We will do an analogous procedure with blocks. We can rewrite A as,

$$A = \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ A_{I1}A_1^{-1} & A_{I2}A_2^{-1} & I \end{pmatrix} \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & S \end{pmatrix} \begin{pmatrix} A_1 & 0 & A_{1I} \\ 0 & A_2 & A_{2I} \\ 0 & 0 & I \end{pmatrix}$$

where,

$$S = A_I - A_{I1}A_1^{-1}A_{1I} - A_{I2}A_2^{-1}A_{2I}$$

We really want A^{-1}

$$A^{-1} = \begin{pmatrix} A_1^{-1} & 0 & -A_1^{-1}A_{1I} \\ 0 & A_2^{-1} & -A_2^{-1}A_{2I} \\ 0 & 0 & I \end{pmatrix} \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & S^{-1} \end{pmatrix} \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ -A_{I1}A_1^{-1} & -A_{I2}A_2^{-1} & I \end{pmatrix} \tag{8.1}$$

Inverting S turns out to be the hardest part.

$$A^{-1} \begin{pmatrix} V_{\Omega 1} \\ V_{\Omega 2} \\ V_{Interface} \end{pmatrix} \begin{matrix} \rightarrow Voltages & in & region & \Omega_1 \\ \rightarrow Voltages & in & region & \Omega_2 \\ \rightarrow Voltages & at & interface & \end{matrix}$$

Let us examine Equation 8.1 in detail.

In the third matrix,
A_1^{-1} - Poisson solve in Ω_1
A_{I1} - is putting the solution onto the interface
A_2^{-1} - Poisson solve in Ω_2
A_{I2} - is putting the solution onto the interface

In the second matrix,
Nothing happening in domain 1 and 2
Complicated stuff at the interface.

In the first matrix we have,
A_1^{-1} - Poisson solve in Ω_1
A_2^{-1} - Poisson solve in Ω_2
$A_1^{-1}A_{1I}$ and $A_2^{-1}A_{1I}$- Transferring solution to interfaces

In the above example we had a simple 2D region with neat squares but in reality we might have to solve on complicated 3D regions which have to be divided into tetrahedra with 2D regions at the interfaces. The above concepts still hold.

Getting to S^{-1},

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ c/a & 1 \end{pmatrix} \begin{pmatrix} a & b \\ 0 & d - bc/a \end{pmatrix}$$

where, $d - bc/a$ is the Schur complement of d.

In Block form

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ CA^{-1} & 1 \end{pmatrix} \begin{pmatrix} A & B \\ 0 & D - CA^{-1}B \end{pmatrix}$$

We have

$$S = A_I - A_{I1}A_1^{-1}A_{1I} - A_{I2}A_2^{-1}A_{2I}$$

Arbitrarily break A_I as

$$A_I = A_I^1 + A_I^2$$

Think of A as

$$\begin{pmatrix} A_1 & 0 & A_{1I} \\ 0 & 0 & 0 \\ A_{I1} & 0 & A_I^1 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 \\ 0 & A_2 & A_{2I} \\ 0 & A_{I2} & A_I^2 \end{pmatrix}$$

Schur Complements are

$$S^1 = A_I^1 - A_{I1}A_1^{-1}A_{1I}$$

$$S^2 = A_I^2 - A_{I2}A_2^{-1}A_{2I}$$

and

$$S = S^1 + S^2$$

$A_1^{-1} \rightarrow$ Poisson solve on Ω_1
$A_2^{-1} \rightarrow$ Poisson solve on Ω_2
A_{I1} $\Omega_1 \rightarrow I$
A_{21} $\Omega_2 \rightarrow I$
A_{1I} $I \rightarrow \Omega_1$
A_{2I} $I \rightarrow \Omega_2$

Sv - Multiplying by the Schur Complement involves 2 Poisson solves and some cheap transferring.

$S^{-1}v$ should be solved using Krylov methods. People have recommended the use of S_1^{-1} or S_2^{-1} or $(S_1^{-1} + S_2^{-1})$ as a preconditioner

8.2.4 Accellerants

Domain Decomposition as a Preconditioner

It seems wasteful to solve subproblems extremely accurately during the early stages of the algorithm when the boundary data is likely to be fairly inaccurate. Therefore it makes sense to run a few steps of an iterative solver as a preconditioner for the solution to the entire problem.

In a modern approach to the solution of the entire problem, a step or two of block Jacobi would be used as a preconditioner in a Krylov based scheme. It is important at this point not to lose track what operations may take place at each level. To solve the subdomain problems, one might use multigrid, FFT, or preconditioned conjugate gradient, but one may choose to do this approximately during the early iterations. The solution of the subdomain problems itself may serve as a preconditioner to the solution of the global problem which may be solved using some Krylov based scheme.

The modern approach is to use a step of block Jacobi or block Gauss-Seidel as a preconditioner for use in a Krylov space based subsolver. There is not too much point in solving the subproblems exactly on the smaller domains (since the boundary data is wrong) just an approximate solution suffices $\rightarrow$ **domain decomposition preconditioning**

Krylov Methods - Methods to solve linear systems : **A**u=g . Examples have names such as the Conjugate Gradient Method, GMRES (Generalized Minimum Residual), BCG (Bi Conjugate Gradient), QMR (Quasi Minimum Residual), CGS (Conjugate Gradient Squared). For this lecture, one can think of these methods in terms of a black-box. What is needed is a subroutine that given u computes Au. This is a matrix-vector multiply in the abstract sense, but of course it is not a dense matrix-vector product in the sense one practices in undergraduate linear algebra. The other needed ingredient is a subroutine to *approximately* solve the system. This is known as a preconditioner. To be useful this subroutine must roughly solve the problem quickly.

Course (Hierarchical/Multilevel) Techniques

These modern approaches are designed to greatly speed convergence by solving the problem on different sized grids with the goal of communicating information between subdomains more efficiently. Here the "domain" is a course grid. Mathematically, it is as easy to consider a contiguous domain consisting of neighboring points, as it is is to consider a course grid covering the whole region.

Up until now, we saw that subdividing a problem did not directly yield the final answer, rather it simplified or allowed us to change our approach in tackling the resulting subproblems with existing methods. It still required that individual subregions be composited at each level of refinement to establish valid conditions at the interface of shared boundaries.

Multilevel approaches solve the problem using a coarse grid over each sub-region, gradually accommodating higher resolution grids as results on shared boundaries become available. Ideally for a well balanced multi-level method, no more work is performed at each level of the hierarchy than is appropriate for the accuracy at hand.
In general a hierarchical or multi-level method is built from an understanding of the difference between the damping of low frequency and high components of the error. Roughly speaking one can kill of low frequency components of the error on the course grid, and higher frequency errors on the fine grid.

Perhaps this is akin to the Fast Multipole Method where p poles that are "well-separated" from a given point could be considered as clusters, and those nearby are evaluated more precisely on a finer grid.

8.3 Theoretical Issues

This section is not yet written. The rough content is the mathematical formulation that identifies subdomains with projection operators.

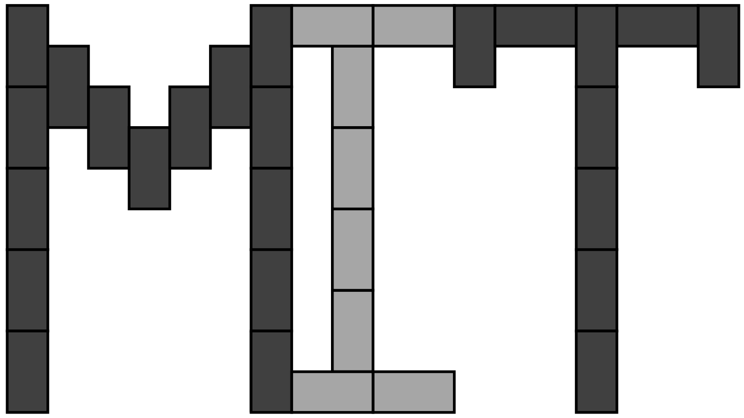


Figure 8.9: MIT domain

8.4 A Domain Decomposition Assignment: Decomposing MIT

Perhaps we have given you the impression that entirely new codes must be written for parallel computers, and furthermore that parallel algorithms only work well on regular grids. We now show you that this is not so.

You are about to solve Poisson's equation on our MIT domain:

Notice that the letters MIT have been decomposed into 32 rectangles – this is just the right number for solving

$$\frac{\partial^2 u}{dx^2} + \frac{\partial^2 u}{dy^2} = \rho(x, y)$$

on a 32 processor machine.

To solve the Poisson equation on the individual rectangles, we will use a FISHPACK library routine. (I know the French would cringe, but there really is a library called FISHPACK for solving the Poisson equation.) The code is old enough (from the 70's) but in fact it is too often used to really call it dusty.

As a side point, this exercise highlights the ease of grabbing kernel routines off the network these days. High quality numerical software is out there (bad stuff too). One good way to find it is via the World Wide Web, at http://www.netlib.org. The software you will need for this problem is found at http://www.netlib.org/netlib/fishpack/hwscrt.f.

All of the rectangles on the MIT picture have sides in the ratio 2 to 1; some are horizontal while others are vertical. We have arbitrarily numbered the rectangles accoding to scheme below, you might wish to write the numbers in the picture on the first page.

```
4                 10 21 21 22 22 23 24 24 25 26 26 27
4  5            9 10    20          23          25          27
3  5  6     8  9 11    20                      28
3     6  7  8    11    19                      28
2        7       12    19                      29
2                12    18                      29
1                13    18                      30
1                13    17                      30
0                14    17                      31
0                14 15 15 16 16                31
```

In our file neighbor.data which you can take from `~edelman/summer94/friday` we have encoded information about neighbors and connections. You will see numbers such as

```
 1  0  0  0  0  0
 4  0  0  0  0  0
 0  0  0  0
 0
```

This contains information about the 0th rectangle. The first line says that it has a neighbor 1. The 4 means that the neighbor meets the rectangle on top. (1 would be the bottom, 6 would be the lower right.) We starred out a few entries towards the bottom. Figure out what they should be.

In the actual code (solver.f), a few lines were question marked out for the message passing. Figure out how the code works and fill in the appropriate lines. The program may be compiled with the makefile.

Lecture 9

Particle Methods

9.1 Reduce and Broadcast: A function viewpoint

[This section is being rewritten with what we hope will be the world's clearest explanation of the fast multipole algorithm. Readers are welcome to take a quick look at this section, or pass to the next section which leads up to the multipole algorithm through the particle method viewpoint]

Imagine we have P processors, and P functions $f_1(z), f_2(z), \ldots, f_P(z)$, one per processor. Our goal is for every processor to know the sum of the functions $f(z) = f_1(z) + \ldots + f_P(z)$. Really this is no different from the reduce and broadcast situation given in the introduction.

As a practical question, how can functions be represented on the computer? Probably we should think of Taylor series or multipole expansion. If all the Taylor series or multipole expansions are centered at the same point, then the function reduction is easy. Simply reduce the corresponding coefficients. If the pairwise sum consists of functions represented using different centers, then a common center must be found and the functions must be transformed to that center before a common sum may be found.

> **Example: Reducing Polynomials** Imagine that processor i contains the polynomial $f_i(z) = (z-i)^3$. The coefficients may be expanded out as $f_i(z) = a_0 + a_1 z + a_2 z^2 + a_3 z^3$. Each processor i contains a vector (a_0, a_1, a_2, a_3). The sum of the vectors may be obtained by a usual reduce algorithm on vectors.
>
> An alternative that may seem like too much trouble at first is that every time we make a pairwise sum we shift to a common midpoint (see Figure 9.1).

There is another complication that occurs when we form pairwise sums of functions. If the expansions are multipole or Taylor expansions, we may shift to a new center that is outside the region of convergence. The coefficients may then be meaningless. Numerically, even if we shift towards the boundary of a region of convergence, we may well lose accuracy, especially since most computations choose to fix the number of terms in the expansion to keep.

Difficulties with shifting multipole or Taylor Expansions

The fast multipole algorithm accounts for these difficulties in a fairly simple manner. Instead of computing the sum of the functions all the way up the tree and then broadcasting back, it saves the intermediate partial sums summing them in only when appropriate. The figure below indicates when this is appropriate.

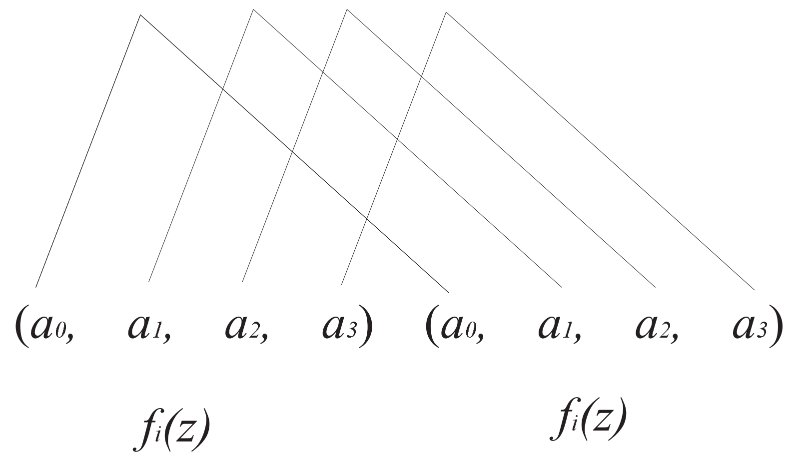


Figure 9.1: Pairwise Sum

9.2 Particle Methods: An Application

Imagine we want to model the basic mechanics of our solar system. We would probably start with the sun, somehow representing its mass, velocity, and position. We might then add each of the nine planets in turn, recording their own masses, velocities, and positions at a point in time. Let's say we add in a couple of hundred of the larger asteroids, and a few of our favorite comets. Now we set the system in motion. Perhaps we would like to know where Pluto will be in a hundred years, or whether a comet will hit us soon. To solve Newton's equations directly with more than even two bodies is intractably difficult. Instead we decide to model the system using discrete time intervals, and computing at each time interval the force that each body exerts on each other, and changing the velocities of the bodies accordingly. This is an example of an N-body problem. To solve the problem in a simple way requires $O(n^2)$ time for each time step. With some considerable effort, we can reduce this to $O(n)$ (using the fast multipole algorithm to be described below). A relatively simple algorithm the Barnes-Hut Algorithm, (to be described below) can compute movement in $O(n \log(n))$ time.

9.3 Outline

- Formulation and applications
- "The easiest part": the Euler method to move bodies.
- Direct methods for force computation.
- Hierarchical methods *(Barnes-Hut, Appel, Greengard and Rohklin)*

9.4 What is N-Body Simulation?

We take n bodies (or particles) with state describing the initial position $\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n \in \Re^k$ and initial velocities $\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n \in \Re^k$.

We want to simulate the evolution of such a system, i.e., to compute the trajectories of each body, under an interactive force: the force exerted on each body by the whole system at a given

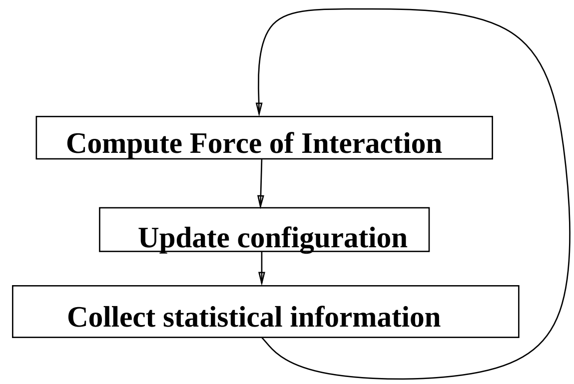


Figure 9.2: Basic Algorithm of N-body Simulation

point. For different applications we will have different interaction forces, such as gravitational or Coulombic forces. We could even use these methods to model spring systems, although the advanced methods, which assume forces decreasing with distance, do not work under these conditions.

9.5 Examples

- **Astrophysics**: The bodies are stars or galaxies, depending on the scale of the simulation. The interactive force is gravity.
- **Plasma Physics**: The basic particles are ions, electrons, etc; the force is Coulombic.
- **Molecular Dynamics**: Particles are atoms or clusters of atoms; the force is electrostatic.
- **Fluid Dynamics**: Vortex method where particle are fluid elements (fluid blobs).

Typically, we call this class of simulation methods, the **particle methods**. In such simulations, it is important that we choose both spatial and temporal scales carefully, in order to minimize running time and maximize accuracy. If we choose a time scale too large, we can lose accuracy in the simulation, and if we choose one too small, the simulations will take too long to run. A simulation of the planets of the solar system will need a much larger timescale than a model of charged ions. Similarly, spatial scale should be chosen to minimize running time and maximize accuracy. For example, in applications in fluid dynamics, molecular level simulations are simply too slow to get useful results in a reasonable period of time. Therefore, researchers use the vortex method where bodies represent large aggregates of smaller particles. Hockney and Eastwood's book **Computer Simulations Using Particles,**, McGraw Hill (1981), explores the applications of particle methods applications, although it is somewhat out of date.

9.6 The Basic Algorithm

Figure 9.2 illustrates the key steps in n-body simulation. The step of collecting statistical information is application dependent, and some of the information gathered at this step may be used during the next time interval.

We will use gravitational forces as an example to present N-body simulation algorithms. Assume there are n bodies with masses $m_1, m_2, \ldots, m_n$, respectively, initially located at $\vec{x_1}, ..., \vec{x_n} \in \Re^3$ with velocity $\vec{v_1}, ..., \vec{v_n}$. The gravitational force exert on the ith body by the jth body is given by

$$\vec{F_{ij}} = G\frac{m_i m_j}{r^2} = G\frac{m_i m_j}{|\vec{x_j} - \vec{x_i}|^3}(\vec{x_j} - \vec{x_i}),$$

where G is the gravitational constant. Thus the total force on the ith body is the vector sum of all these forces and is give by,

$$\vec{F_i} = \sum_{j \neq i} \vec{F_{ij}}.$$

Let $\vec{a_i} = d\vec{v_i}/dt$ be the acceleration of the body i, where where $\vec{v_i} = d\vec{x_i}/dt$. By Newton's second law of motion, we have $\vec{F_i} = m_i\vec{a_i} = m_i d\vec{v_i}/dt$.

In practice, we often find that using a potential function $V = \phi m$ will reduce the labor of the calculation. First, we need to compute the potential due to the N-body system position, i.e. $x_1, \ldots, x_n$, at positions $y_1, \ldots, y_n$.

The total potential is calculated as

$$V_i = \sum_{i,j=1; i \neq j}^{n} \phi(x_i - y_j) m_j \qquad 1 \leq i, j \leq n,$$

where ϕ is the potential due to gravity. This can also be written in the matrix form:

$$V = \begin{pmatrix} 0 & \ldots & (x_i - y_j) \\ \ldots & \ldots & \ldots \\ \phi(x_j - y_i) & \ldots & 0 \end{pmatrix}$$

In $\Re^3$,

$$\phi(x) = \frac{1}{\| x \|}$$

.

In $\Re^2$,

$$\phi(x) = log\| x \|$$

.

The update of particle velocities and positions are in three steps:

1. $F = \pi \cdot m$;

2. $V_{new} = V_{old} + \Delta t \cdot \frac{F}{m}$;

3. $x_{new} = x_{old} + \Delta t \cdot V_{new}$.

The first step is the most expensive part in terms of computational time.

9.6.1 Finite Difference and the Euler Method

In general, the force calculation is the most expensive step for N-body simulations. We will present several algorithms for this later on, but first assume we have already calculated the force $\vec{F_i}$ acting one each body. We can use a numerical method (such as the Euler method) to update the configuration.

To simulate the evolution of an N-body system, we decompose the time interval into discretized time steps: $t_0, t_1, t_2, t_3, ...$ For uniform discretizations, we choose a Δt and let $t_0 = 0$ and $t_k = k\Delta t$. The Euler method approximates the derivative by finite difference.

$$\begin{aligned} \vec{a_i}(t_k) &= \vec{F_i}/m_i = \frac{\vec{v_i}(t_k) - \vec{v_i}(t_k - \Delta t)}{\Delta t} \\ \vec{v_i}(t_k) &= \frac{\vec{x_i}(t_k + \Delta t) - \vec{x_i}(t_k)}{\Delta t}, \end{aligned}$$

where $1 \leq i \leq n$. Therefore,

$$\vec{v_i}(t_k) = \vec{v_i}(t_{k-1}) + \Delta t(\vec{F_i}/m_i) \tag{9.1}$$

$$\vec{x_i}(t_{k+1}) = \vec{x_i}(t_k) + \Delta t \vec{v_i}(t_k). \tag{9.2}$$

From the given initial configuration, we can derive the next time step configuration using the formulae by first finding the force, from which we can derive velocity, and then position, and then force at the next time step.

$$\vec{F_i} \to v_i(t_k) \to x_i(t_k + \Delta t) \to \vec{F_{i+1}}.$$

High order numerical methods can be used here to improve the simulation. In fact, the Euler method that uses uniform time step discretization performs poorly during the simulation when two bodies are very close. We may need to use non-uniform discretization or a sophisticated time scale that may vary for different regions of the N-body system.

In one region of our simulation, for instance, there might be an area where there are few bodies, and each is moving slowly. The positions and velocities of these bodies, then, do not need to be sampled as frequently as in other, higher activity areas, and can be determined by extrapolation. See figure 9.3 for illustration.[1]

How many floating point operations (flops) does each step of the Euler method take? The velocity update (step 1) takes $2n$ floating point multiplications and one addition and the position updating (step 2) takes 1 multiplication and one addition. Thus, each Euler step takes $5n$ floating point operations. In Big-O notation, this is an $O(n)$ time calculation with a constant factor 5.

Notice also, each Euler step can be parallelized without communication overhead. In data parallel style, we can express steps (1) and (2), respectively, as

$$\begin{aligned} V &= V + \Delta t(F/M) \\ X &= X + \Delta t V, \end{aligned}$$

where V is the velocity array; X is the position array; F is the force array; and M is the mass array. V, X, F, M are $3 \times n$ arrays with each column corresponding to a particle. The operator / is the elementwise division.

[1]In figure 9.3 we see an example where we have some close clusters of bodies, and several relatively disconnected bodies. For the purposes of the simulation, we can ignore the movement of relatively isolated bodies for short periods of time and calculate more frames of the proximous bodies. This saves computation time and grants the simulation more accuracy where it is most needed. In many ways these sampling techniques are a temporal analogue of the later discussed Barnes and Hut and Multipole methods.

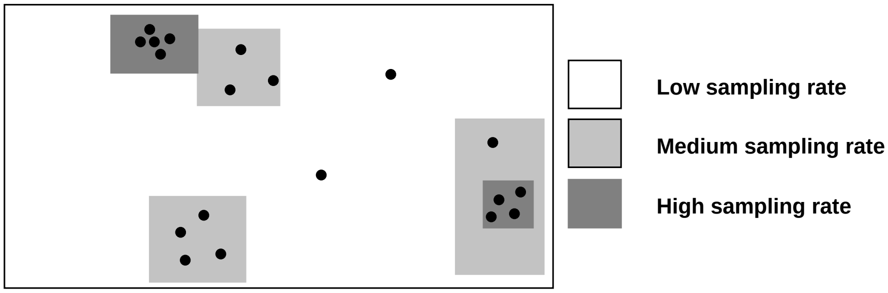


Figure 9.3: Adaptive Sampling Based on Proximity

9.7 Methods for Force Calculation

Computationally, the force calculation is the most time expensive step for N-body simulation. We now discuss some methods for computing forces.

9.7.1 Direct force calculation

The simplest way is to calculate the force directly from the definition.

$$\vec{F_{ij}} = G\frac{m_i m_j}{r^2} = G\frac{m_i m_j}{|\vec{x_j} - \vec{x_i}|^3}(\vec{x_j} - \vec{x_i}),$$

Note that the step for computing $\vec{F_{ij}}$ takes 9 flops. It takes n flops to add $\vec{F_{ij}}$ $(1 \leq j \leq n)$. Since $\vec{F_{ij}} = -\vec{F_{ji}}$, the total number of flops needed is roughly $5n^2$. In Big-O notation, this is an $O(n^2)$ time computation. For large scale simulation (e.g., $n = 100$ million), the direct method is impractical with today's level of computing power.

It is clear, then, that we need more efficient algorithms. The one fact that we have to take advantage of is that in a large system, the effects of individual distant particles on each other may be insignificant and we may be able to disregard them without significant loss of accuracy. Instead we will cluster these particles, and deal with them as though they were one mass. Thus, in order to gain efficiency, we will approximate in space as we did in time by discretizing.

9.7.2 Potential based calculation

For N-body simulations, sometimes it is easier to work with the (gravitational) potential rather than with the force directly. The force can then be calculated as the gradient of the potential.

In three dimensions, the gravitational potential at position $\vec{x}$ defined by n bodies with masses $m_1, ..., m_n$ at position $\vec{x_1}, ..., \vec{x_n}$, respectively is equal to

$$\Phi(\vec{x}) = \sum_{i=1}^{n} G\frac{m_i}{||\vec{x} - \vec{x_i}||}.$$

The force acting on a body with unit mass at position $\vec{x}$ is given by the *gradient* of Φ, i.e.,

$$F = -\nabla\Phi(x).$$

The potential function is a sum of local potential functions

$$\phi(\vec{x}) = \sum_{i=1}^{n} \phi_{\vec{x}_i}(\vec{x}) \tag{9.3}$$

where the local potential functions are given by

$$\phi_{\vec{x}_i}(\vec{x}) = \frac{G * m_i}{||\vec{x} - \vec{x}_i||} \qquad \text{in } \Re^3 \tag{9.4}$$

9.7.3 Poisson Methods

The earlier method from 70s is to use Poisson solver. We work with the gravitational potential field rather than the force field. The observation is that the potential field can be expressed as the solution of a Poisson equation and the force field is the gradient of the potential field.

The gravitational potential at position $\vec{x}$ defined by n bodies with masses $m_1, ..., m_n$ at position $\vec{x_1}, ..., \vec{x_n}$, respectively is equal to

$$\Phi(\vec{x}) = \sum_{i=1}^{n} G \frac{m_i}{|\vec{x} - \vec{x_i}|}.$$

The force acting on a body with unit mass at position $\vec{x}$ is given by the gradient of Φ:

$$\vec{F} = -\nabla\Phi(\vec{x}).$$

So, from Φ we can calculate the force field (by numerical approximation).

The potential field Φ satisfies a Poisson equation:

$$\nabla^2\Phi = \frac{\partial^2\Phi}{\partial x^2} + \frac{\partial^2\Phi}{\partial y^2} + \frac{\partial^2\Phi}{\partial z^2} = \rho(x, y, z),$$

where ρ measures the mass distribution can be determined by the configuration of the N-body system. (The function Φ is harmonic away from the bodies and near the bodies, $div\nabla\Phi = \nabla^2\Phi$ is determined by the mass distribution function. So $\rho = 0$ away from bodies).

We can use finite difference methods to solve this type of partial differential equations. In three dimensions, we discretize the domain by a structured grid.

We approximate the Laplace operator ∇^2 by finite difference and obtain from $\nabla^2\Phi = \rho(x, y, z)$ a system of linear equations. Let h denote the grid spacing. We have

$$\begin{aligned}
\Phi(x_i, y_j, z_k) &= \frac{1}{h^2}(\Phi(x_i + h, y_j, z_k) + \Phi(x_i - h, y_j, z_k) + \Phi(x_i, y_j + h, z_k) \\
&\qquad + \Phi(x_i, y_j - h, z_k) + \Phi(x_i, y_j, z_k + h) + \Phi(x_i, y_j, z_k - h) - 6\phi(x_i, y_j, z_k)) \\
&= \rho(x_i, y_j, z_k).
\end{aligned}$$

The resulting linear system is of size equal to the number of grid points chosen. This can be solved using methods such as FFT (fast Fourier transform), SOR (successive overrelaxation), multigrid methods or conjugate gradient. If n bodies give a relatively uniform distribution, then we can use a grid which has about n grid points. The solution can be fairly efficient, especially on parallel machines. For highly non-uniform set of bodies, hybrid methods such as finding the potential induced by bodies within near distance by direct method, and approximate the potential field induced by distant bodies by the solution of a much smaller Poisson equation discretization. More details of these methods can be found in Hockney and Eastwood's book.

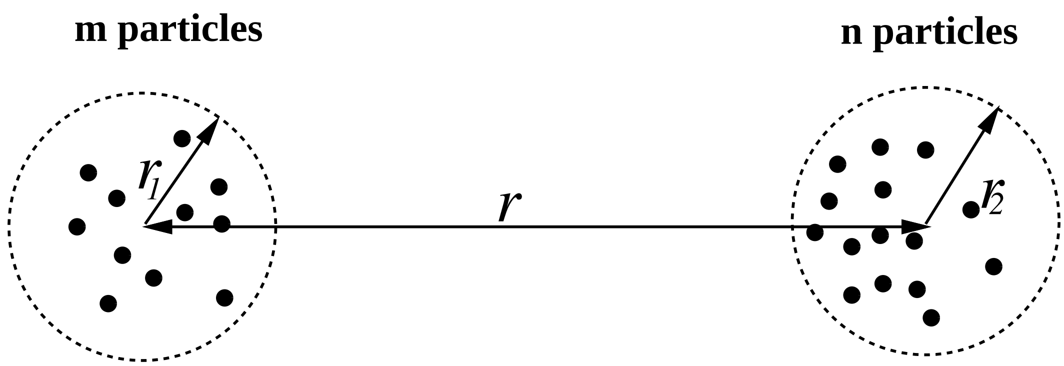


Figure 9.4: Well-separated Clusters

9.7.4 Hierarchical methods

We now discuss several methods which use a hierarchical structure to decompose bodies into clusters. Then the force field is approximated by computing the interaction between bodies and clusters and/or between clusters and clusters. We will refer this class of methods *hierarchical methods or tree-code methods.*

The crux of hierarchical N-body methods is to decompose the potential at a point x, $\phi(x)$, into the sum of two potentials: $\phi_N(x)$, the potential induced by "neighboring" or "near-field" particles; and $\phi_F(x)$, the potential due to "far-field" particles [5, 44]. In hierarchical methods, $\phi_N(x)$ is computed exactly, while $\phi_F(x)$ is computed approximately.

The approximation is based on a notion of well-separated clusters [5, 44]. Suppose we have two clusters A and B, one of m particles and one of n particles, the centers of which are separated by a distance r. See Figure 9.4.

Suppose we want to find the force acting on all bodies in A by those in B and vice versa. A direct force calculation requires $O(mn)$ operations, because for each body in A we need to compute the force induced by every body in B.

Notice that if r is much larger than both r_1 and r_2, then we can simplify the calculation tremendously by replacing B by a larger body at its center of mass and replacing A by a larger body at its center of mass. Let M_A and M_B be the total mass of A and B, respectively. The center of mass c_A and c_B is given by

$$\begin{aligned} c_A &= \frac{\sum_{i \in A} m_i x_i}{M_A} \\ c_B &= \frac{\sum_{j \in B} m_j x_j}{M_B}. \end{aligned}$$

We can approximate the force induced by bodies in B on a body of mass m_x located at position s by viewing B as a single mass M_B at location c_B. That is,

$$F(x) \approx \frac{G m_x M_B (x - c_B)}{||x - c_B||^3}.$$

Such approximation is second order: The relative error introduced by using center of mass is bounded by $(\max(r_1, r_2)/r)^2$. In other words, if $f(x)$ be the true force vector acting on a body at

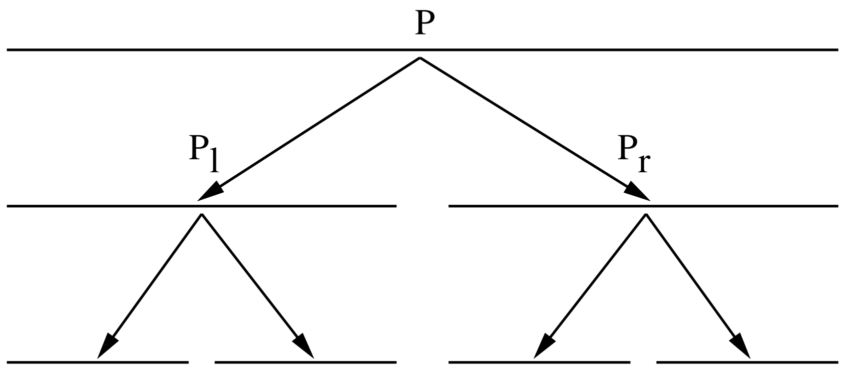


Figure 9.5: Binary Tree (subdivision of a straight line segment)

x, then

$$F(x) = f(x)\left(1 + O\left(\left(\frac{\max(r_1, r_2)}{r}\right)^2\right)\right).$$

This way, we can find all the interaction forces between A and B in $O(n+m)$ time. The force calculations between one m particle will computed separately using a recursive construction. This observation gives birth the idea of hierarchical methods.

We can also describe the method in terms of potentials. If r is much larger than both r_1 and r_2, i.e., A and B are "well-separated", then we can use the pth order multipole expansion (to be given later) to express the pth order approximation of potential due to all particles in B. Let $\Phi_B^p(x)$ denote such a multipole expansion. To (approximately) compute the potential at particles in A, we simply evaluate $\Phi_B^p()$ at each particle in A. Suppose $\Phi_B^p()$ has $g(p,d)$ terms. Using multipole expansion, we reduce the number of operations to $g(p,d)(|A|+|B|)$. The error of the multipole-expansion depends on p and the ratio $\max(r_1, r_2)/r$. We say A and B are *β-well-separated*, for a $\beta > 2$, if $\max(r_1, r_2)/r \leq 1/\beta$. As shown in [44], the error of the pth order multipole expansion is bounded by $(1/(\beta-1))^p$.

9.8 Quadtree (2D) and Octtree (3D) : Data Structures for Canonical Clustering

Hierarchical N-body methods use quadtree (for 2D) and octtree (for 3D) to generate a canonical set of boxes to define clusters. The number of boxes is typically linear in the number of particles, i.e., $O(n)$.

Quadtrees and octtrees provide a way of hierarchically decomposing two dimensional and three dimensional space. Consider first the one dimensional example of a straight line segment. One way to introduce clusters is to recursively divide the line as shown in Figure 9.5.

This results in a binary tree[2].

In two dimensions, a *box* decomposition is used to partition the space (Figure 9.6). Note that a box may be regarded as a "product" of two intervals. Each partition has at most one particle in it.

[2]A tree is a graph with a single *root* node and a number of subordinate nodes called *leaves* or *children*. In a binary tree, every node has at most two children.

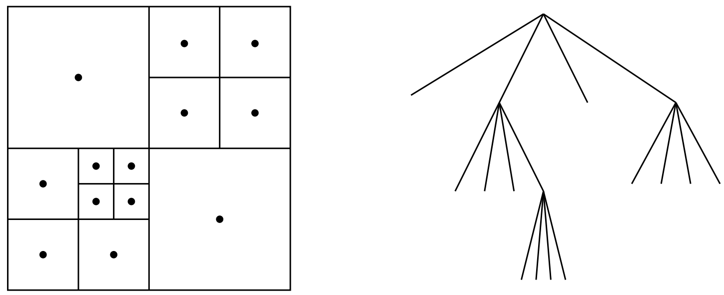


Figure 9.6: Quadtree

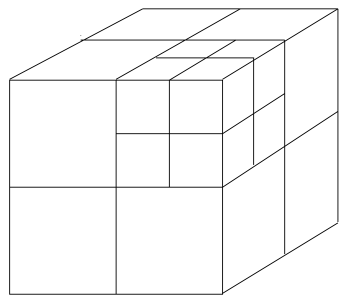


Figure 9.7: Octtree

A *quadtree* [88] is a recursive partition of a region of the plane into axis-aligned squares. One square, the *root*, covers the entire set of particles. It is often chosen to be the smallest (up to a constant factor) square that contains all particles. A square can be divided into four *child* squares, by splitting it with horizontal and vertical line segments through its center. The collection of squares then forms a tree, with smaller squares at lower levels of the tree. The recursive decomposition is often adaptive to the local geometry. The most commonly used termination condition is: the division stops when a box contains less than some constant (typically $m = 100$) number of particles (See Figure 9.6).

In 2D case, the height of the tree is usually $log_2\sqrt{N}$. This is in the order of . The complexity of the problem is $N \cdot O(log(N))$.

Octtree is the three-dimension version of quadtree. The root is a box covering the entire set of particles. Octtree are constructed by recursively and adaptively dividing a box into eight child-boxes, by splitting it with hyperplanes normal to each axes through its center (See Figure 9.7).

9.9 Barnes-Hut Method (1986)

The Barnes-Hut method uses these clustered data structures to represent the bodies in the simulation, and takes advantage of the distant-body simplification mentioned earlier to reduce computational complexity to $O(n \log(n))$.

The method of Barnes and Hut has two steps.

1. **Upward evaluation of center of mass**

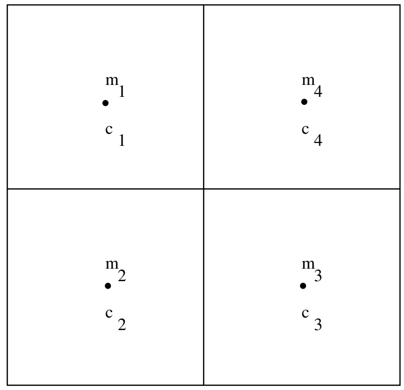


Figure 9.8: Computing the new Center of Mass

Refer to Figure 9.6 for the two dimensional case. Treating each box as a uniform cluster, the center of mass may be hierarchically computed. For example, consider the four boxes shown in Figure 9.8.

The total mass of the system is

$$m = m_1 + m_2 + m_3 + m_4 \tag{9.5}$$

and the center of mass is given by

$$\vec{c} = \frac{m_1\vec{c_1} + m_2\vec{c_2} + m_3\vec{c_3} + m_4\vec{c_4}}{m} \tag{9.6}$$

The total time required to compute the centers of mass at all layers of the quadtree is proportional to the number of nodes, or the number of bodies, whichever is greater, or in Big-O notation, $O(n + v)$, where v is for *vertex*

This result is readily extendible to the three dimensional case.

Using this approximation will lose some accuracy. For instance, in 1D case, consider three particles locate at $x = -1, 0, 1$ with strength $m = 1$. Consider these three particles as a cluster, the total potential is

$$V(x) = \frac{1}{x} + \frac{1}{x-1} + \frac{1}{x+1}.$$

Expand the above equation using Taylor's series,

$$V(x) = \frac{3}{x} + \frac{2}{x^3} + \frac{2}{x^5} + \ldots.$$

. It is seen that high order terms are neglected. This brings the accuracy down when x is close to the origin.

2. **Pushing the particle down the tree**

   Consider the case of the octtree i.e. the three dimensional case. In order to evaluate the potential at a point $\vec{x}_i$, start at the top of the tree and move downwards. At each node, check whether the corresponding box, b, is well separated with respect to $\vec{x}_i$ (Figure 9.9).

   Let the force at point $\vec{x}_i$ due to the cluster b be denoted by $\vec{F}(i, b)$. This force may be calculated using the following algorithm:

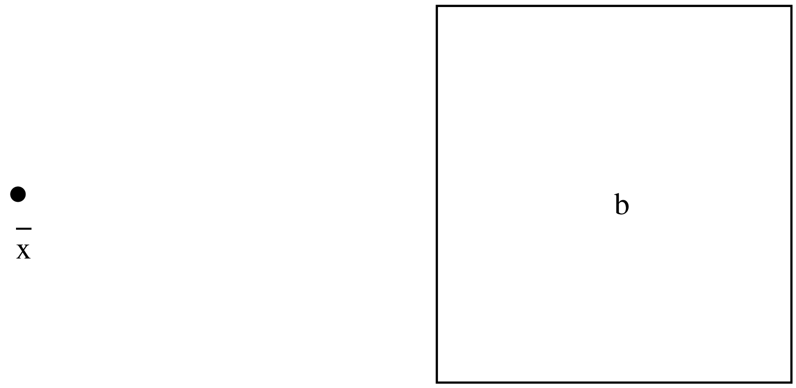


Figure 9.9: Pushing the particle down the tree

- *if b is "far" i.e. well separated from $\vec{x}_i$*, then

$$\vec{F}(\vec{x}_i) := \vec{F}(\vec{x}_i) + \frac{G m_x M_b (\vec{x} - \vec{c_b})}{||\vec{x}_i - \vec{c_b}||^3} \qquad \text{in } \Re^3 \tag{9.7}$$

- *else if b is "close" to $\vec{x}_i$*

$$\begin{aligned} & for\ k = 1\ to\ 8 \\ & \qquad \vec{F}(\vec{x}_i) = \vec{F}(\vec{x}_i) + \vec{F}(i, child(b,k)) \end{aligned} \tag{9.8}$$

$$\tag{9.9}$$

The computational complexity of pushing the particle down the tree has the upper bound $9hn$, where h is the height of the tree and n is the number of particles. (Typically, for more or less uniformly distributed particles, $h = \log_4 n$.)

9.9.1 Approximating potentials

We now rephrase Barnes and Hut scheme in term of potentials. Let

$$\begin{aligned} m_A &= \text{total mass of particles in } \mathbf{A} \\ m_B &= \text{total mass of particles in } \mathbf{B} \\ \vec{c_A} &= \text{center of mass of particles in } \mathbf{A} \\ \vec{c_B} &= \text{center of mass of particles in } \mathbf{B} \end{aligned}$$

The potential at a point $\vec{x}$ due to the cluster $\mathbf{B}$, for example, is given by the following second order approximation:

$$\phi(\vec{x}) \approx \frac{m_B}{||\vec{x} - \vec{c_B}||}(1 + \frac{1}{\delta^2}) \qquad \text{in } \Re^3 \tag{9.10}$$

In other words, each cluster may be regarded as an individual particle when the cluster is sufficiently far away from the evaluation point $\vec{x}$.

A more advanced idea is to keep track of a higher order (Taylor expansion) approximation of the potential function induced by a cluster. Such an idea provides better tradeoff between time required and numerical precision. The following sections provide the two dimensional version of the fast multipole method developed by Greengard and Rokhlin.

The Barnes-Hut method discussed above uses the *particle-cluster* interaction between two well-separated clusters. Greengard and Rokhlin showed that the *cluster-cluster* intersection among well-separated clusters can further improve the hierarchical method. Suppose we have k clusters B_1 ..., B_k that are well-separated from a cluster A. Let $\Phi_i^p()$ be the pth order multipole expansion of B_i. Using particle-cluster interaction to approximate the far-field potential at A, we need to perform $g(p,d)|A|(|B_1| + |B_2| + ... + |B_k|)$ operations. Greengard and Rokhlin [44] showed that from $\Phi_i^p()$ we can efficiently compute a Taylor expansion $\Psi_i^p()$ centered at the centroid of A that approximates $\Phi_i^p()$. Such an operation of transforming $\Phi_i^p()$ to $\Psi_i^p()$ is called a *FLIP*. The cluster-cluster interaction first flips $\Phi_i^p()$ to $\Psi_i^p()$; we then compute $\Psi_A^p() = \sum_{i=1}^k \Psi_i^p()$ and use $\Psi_A^p()$ to evaluate the potential at each particle in A. This reduces the number of operations to the order of

$$g(p,d)(|A| + |B_1| + |B_2| + ... + |B_k|).$$

9.10 Outline

- Introduction
- Multipole Algorithm: An Overview
- Multipole Expansion
- Taylor Expansion
- Operation No. 1 — SHIFT
- Operation No. 2 — FLIP
- Application on Quad Tree
- Expansion from 2-D to 3-D

9.11 Introduction

For N-body simulations, sometimes, it is easier to work with the (gravitational) potential rather than with the force directly. The force can then be calculated as the gradient of the potential.

In two dimensions, the potential function at z_j due to the other bodies is given by

$$\begin{aligned}\phi(z_j) &= \sum_{i=1, i\neq j}^{n} q_i \log(z_j - z_i) \\ &= \sum_{i=1, i\neq j}^{n} \phi_{z_i}(z_j)\end{aligned}$$

with

$$\phi_{z_i}(z) = q_i \log|z - z_i|$$

where $z_1, \ldots, z_n$ the position of particles, and $q_1, \ldots, q_n$ the strength of particles. The potential due to the bodies in the rest of the space is

$$\phi(z) = \sum_{i=1}^{n} q_i \log(z - z_i)$$

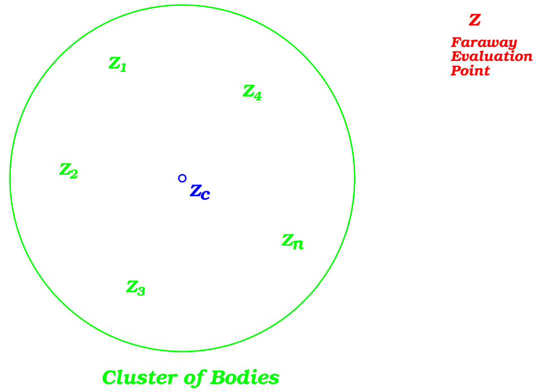


Figure 9.10: Potential of Faraway Particle due to Cluster

which is *singular* at each potential body. (Note: actually the potential is $Re\ \phi(z)$ but we take the complex version for simplicity.)

With the Barnes and Hut scheme in term of potentials, each cluster may be regarded as an individual particle when the cluster is sufficiently far away from the evaluation point. The following sections will provide the details of the fast multipole algorithm developed by Greengard and Rokhlin.

Many people are often mystified why the Green's function is a logarithm in two dimensions, while it is $1/r$ in three dimensions. Actually there is an intuitive explanation. In d dimensions the Green's function is the integral of the force which is proportional $1/r^{d-1}$. To understand the $1/r^{d-1}$ just think that the lines of force are divided "equally" on the sphere of radius r. One might wish to imagine an d dimensional ball with small holes on the boundary filled with d dimensional water. A hose placed at the center will force water to flow out radially at the boundary in a uniform manner. If you prefer, you can imagine 1 ohm resistors arranged in a polar coordinate manner, perhaps with higher density as you move out in the radial direction. Consider the flow of current out of the circle at radius r if there is one input current source at the center.

9.12 Multipole Algorithm: An Overview

There are three important concepts in the multipole algorithm:

- function representations (multipole expansions and Taylor series)
- operators to change representations (SHIFTs and FLIPs)
- the general tree structure of the computation

9.13 Multipole Expansion

The multipole algorithm flips between two point of views, or to be more precise, two representations for the potential function. One of them, which considers the cluster of bodies corresponding to many far away evaluation points, is treated in detail here. This part of the algorithm is often called the *Multipole Expansion*.

In elementary calculus, one learns about Taylor expansions for functions. This power series represents the function perfectly within the radius of convergence. A multipole expansion is also a perfectly valid representation of a function which typically converges *outside* a circle rather than inside. For example, it is easy to show that

$$\begin{aligned}\phi_{z_i}(z) &= q_i \log(z - z_i)\\ &= q_i \log(z - z_c) + \sum_{k=1}^{\infty} -\frac{q_i}{k}\left(\frac{z_i - z_c}{z - z_c}\right)^k\end{aligned}$$

where z_c is any complex number. This series converges in the region $|z - z_c| > |z - z_i|$, i.e., outside of the circle containing the singularity. The formula is particularly useful if $|z - z_c| \gg |z - z_i|$, i.e., if we are far away from the singularity.

Note that

$$\begin{aligned}\phi_{z_i}(z) &= q_i \log(z - z_i)\\ &= q_i \log[(z - z_c) - (z_i - z_c)]\\ &= q_i \left[\log(z - z_c) + \log(1 - \frac{z_i - z_c}{z - z_c})\right]\end{aligned}$$

The result follows from the Taylor series expansion for $\log(1 - x)$. The more terms in the Taylor series that are considered, the higher is the order of the approximation.

By substituting the single potential expansion back into the main equation, we obtain the multipole expansion as following

$$\begin{aligned}\phi(z) &= \sum_{i=1}^{n} \phi_{z_i}(z)\\ &= \sum_{i=1}^{n} q_i \log(z - z_c) + \sum_{i=1}^{n}\sum_{k=1}^{\infty} q_i \left(-\frac{1}{k}\left(\frac{z_i - z_c}{z - z_c}\right)^k\right)\\ &= Q \log(z - z_c) + \sum_{k=1}^{\infty} a_k \left(\frac{1}{z - z_c}\right)^k\end{aligned}$$

where

$$a_k = -\sum_{i=1}^{n} \frac{q_i (z_i - z_c)^k}{k}$$

When we truncate the expansion due to the consideration of computation cost, an error is introduced into the resulting potential. Consider a *p-term* expansion

$$\phi_p(z) = Q \log(z - z_c) + \sum_{k=1}^{p} a_k \frac{1}{(z - z_c)^k}$$

An error bound for this approximation is given by

$$||\phi(z) - \phi_p(z)|| \leq \frac{A}{(|\frac{z - z_c}{r}| - 1)} \left|\frac{r}{z - z_c}\right|^p$$

where r is the radius of the cluster and

$$A = \sum_{i=1}^{n} |q_i|$$

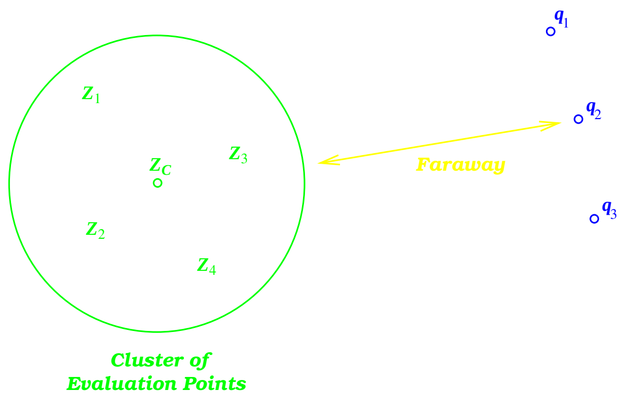


Figure 9.11: Potential of Particle Cluster

This result can be shown as the following

$$
\begin{aligned}
\text{Error} &= \left| \sum_{k=p+1}^{\infty} a_k \frac{1}{(z-z_c)^k} \right| \\
&= \left| - \sum_{k=p+1}^{\infty} \sum_{i=1}^{n} \frac{q_i}{k} \left(\frac{z_i - z_c}{z - z_c} \right)^k \right| \\
&\leq \sum_{k=p+1}^{\infty} \sum_{i=1}^{n} |q_i| \left| \frac{r}{z - z_c} \right|^k \\
&\leq A \sum_{k=p+1}^{\infty} \left| \frac{r}{z - z_c} \right|^k \\
&\leq A \frac{\left| \frac{r}{z-z_c} \right|^{p+1}}{1 - \left| \frac{r}{z-z_c} \right|} \\
&\leq \frac{A}{\left(\left| \frac{z-z_c}{r} \right| - 1 \right)} \left| \frac{r}{z - z_c} \right|^p
\end{aligned}
$$

At this moment, we are able to calculate the potential of each particle due to cluster of far away bodies, through multipole expansion.

9.14 Taylor Expansion

In this section, we will briefly discuss the other point of view for the multipole algorithm, which considers the cluster of evaluation points with respect to many far away bodies. It is called Taylor Expansion. For this expansion, each processor "ownes" the region of the space defined by the cluster of evaluation points, and compute the potential of the cluster through a Taylor series about the center of the cluster z_c.

Generally, the local Taylor expansion for cluster denoted by C (with center z_c) corresponding to some body z has the form

$$\phi_{C,z_c}(z) = \sum_{k=0}^{\infty} b_k (z - z_c)^k$$

Denote $z - z_i = (z - z_c) - (z_i - z_c) = -(z_i - z_c)(1 - \xi)$. Then for z such that $|z - z_c| < \min(z_c, C)$, we have $|\xi| < 1$ and the series $\phi_{C,z_c}(z)$ converge:

$$\begin{aligned}
\phi_{C,z_c}(z) &= \sum_C q_i \log(-(z_i - z_c)) + \sum_C q_i \log(1 - \xi) \\
&= \sum_C q_i \log(-(z_i - z_c)) + \sum_{k=1}^{\infty} \left(\sum_C q_i \right) k^{-1} (z_i - z_c)^{-k} (z - z_c)^k \\
&= b_0 + \sum_{k=1}^{\infty} b_k (z - z_c)^k
\end{aligned}$$

where formulæ for coefficients are

$$b_0 = \sum_C q_i \log(-(z_i - z_c)) \quad \text{and} \quad b_k = k^{-1} \sum_C q_i (z_i - z_c)^{-k} \quad k > 0.$$

Define the *p-order* truncation of local expansion ϕ^p_{C,z_c} as follows

$$\phi^p_{C,z_c}(z) = \sum_{k=0}^{p} b_k (z - z_c)^k.$$

We have error bound

$$\begin{aligned}
\left| \phi_{C,z_c}(z) - \phi^p_{C,z_c}(z) \right| &= \left| \sum_{k=p+1}^{\infty} k^{-1} \sum_C q_i \left(\frac{z - z_c}{z_i - z_c} \right)^k \right| \\
&\leq \frac{1}{p+1} \sum_C |q_i| \sum_{k=p+1}^{\infty} \left| \frac{z - z_c}{\min(z_c, C)} \right|^k = \frac{A}{(1+p)(1-c)} c^{p+1},
\end{aligned}$$

where $A = \sum_C |q_i|$ and $c = |z - z_c| / \min(z_c, C) < 1$.

By now, we can also compute the local potential of the cluster through the Taylor expansion. During the process of deriving the above expansions, it is easy to see that

- Both expansions are singular at the position of any body;
- Multipole expansion is valid outside the cluster under consideration;
- Taylor expansion converges within the space defined the cluster.

At this point, we have finished the basic concepts involved in the multipole algorithm. Next, we will begin to consider some of the operations that could be performed on and between the expansions.

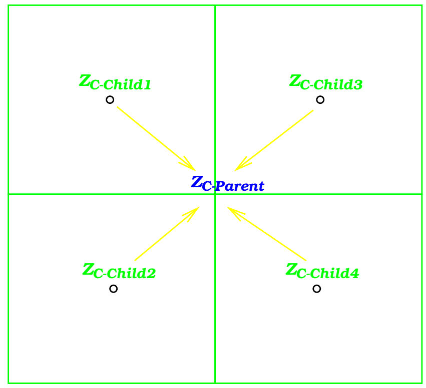


Figure 9.12: SHIFT the Center of Reference

9.15 Operation No.1 — SHIFT

Sometimes, we need to change the location of the center of the reference for the expansion series, either the multipole expansion or the local Taylor expansion. To accomplish this goal, we will perform the SHIFT operation on the expansion series.

For the multipole expansion, consider some far away particle with position z such that both series ϕ_{z_0} and ϕ_{z_1} , corresponding to different center of reference z_0 and z_1, converge: $|z - z_0| > \max(z_0, C)$ and $|z - z_1| > \max(z_1, C)$. Note that

$$z - z_0 = (z - z_1) - (z_0 - z_1) = (z - z_1)\left(1 - \frac{z_0 - z_1}{z - z_1}\right) = (z - z_1)(1 - \xi)$$

for appropriate ξ and if we also assume z sufficiently large to have $|\xi| < 1$, we get identity

$$(1 - \xi)^{-k} = \left(\sum_{l=0}^{\infty} \xi^l\right)^k = \sum_{l=0}^{\infty} \binom{k + l - 1}{l} \xi^l.$$

Now, we can express the SHIFT operation for multipole expansion as

$$\begin{aligned}
\phi_{z_1}(z) &= SHIFT\,(\phi_{z_0}(z), z_0 \Rightarrow z_1) \\
&= SHIFT\left(a_0 \log(z - z_0) + \sum_{k=1}^{\infty} a_k (z - z_0)^{-k}, z_0 \Rightarrow z_1\right) \\
&= a_0 \log(z - z_1) + a_0 \log(1 - \xi) + \sum_{k=1}^{\infty} a_k (1 - \xi)^{-k} (z - z_1)^{-k} \\
&= a_0 \log(z - z_1) - a_0 \sum_{k=1}^{\infty} k^{-1} \xi^k + \sum_{k=1}^{\infty} \sum_{l=1}^{\infty} a_k \binom{k + l - 1}{l} \xi^l (z - z_1)^{-k} \\
&= a_0 \log(z - z_1) + \sum_{l=1}^{\infty} \left(\sum_{k=1}^{l} a_k (z_0 - z_1)^{l-k} \binom{l - 1}{k - 1} - a_0 l^{-1} (z_0 - z_1)^l\right) (z - z_1)^{-l}
\end{aligned}$$

We can represent $\phi_{z_1}(z)$ as a sequence of its coefficients a'_k:

$$a'_0 = a_0 \quad \text{and} \quad a'_l = \sum_{k=1}^{l} a_k (z_0 - z_1)^{l-k} \binom{l - 1}{k - 1} - a_0 l^{-1} (z_0 - z_1)^l \quad l > 0.$$

Note that a'_l depends only on $a_0, a_1, \ldots, a_l$ and not on the higher coefficients. It shows that given $\phi^p_{z_0}$ we can compute $\phi^p_{z_1}$ exactly, that is without any further error! In other words, operators SHIFT and truncation commute on multipolar expansions:

$$SHIFT(\phi^p_{z_0}, z_0 \Rightarrow z_1) = \phi^p_{z_1}.$$

Similarly, we can obtain the SHIFT operation for the local Taylor expansion, by extending the operator on the domain of local expansion, so that $SHIFT(\phi_{C,z_0}, z_0 \Rightarrow z_1)$ produces ϕ_{C,z_1}. Both series converges for z such that $|z - z_0| < \min(z_0, C)$, $|z - z_1| < \min(z_1, C)$. Then

$$\begin{aligned}
\phi_{C,z_1}(z) &= SHIFT\,(\phi_{C,z_0}(z), z_0 \Rightarrow z_1) \\
&= \sum_{k=0}^{\infty} b_k((z - z_1) - (z_0 - z_1))^k \\
&= \sum_{k=0}^{\infty} b_k \sum_{l=0}^{\infty} (-1)^{k-l} \binom{k}{l} (z_0 - z_1)^{k-l} (z - z_1)^l \\
&= \sum_{l=0}^{\infty} \left(\sum_{k=l}^{\infty} b_k (-1)^{k-l} \binom{k}{l} (z_0 - z_1)^{k-l} \right) (z - z_1)^l
\end{aligned}$$

Therefore, formula for transformation of coefficients b_k of ϕ_{C,z_0} to b'_l of ϕ_{C,z_1} are

$$b'_l = \sum_{k=l}^{\infty} a_k (-1)^{k-l} \binom{k}{l} (z_0 - z_1)^{k-l}.$$

Notice that in this case, b'_l depends only on the higher coefficients, which means knowledge of the coefficients $b_0, b_1, \ldots, b_p$ from the truncated local expansion in z_0 does *not* suffice to recover the coefficients $b'_0, b'_1, \ldots b'_p$ at another point z_1. We do incur an error by the SHIFT operation applied to truncated local expansion:

$$\begin{aligned}
\left| SHIFT(\phi^p_{C,z_0}, z_0 \Rightarrow z_1) - \phi^p_{C,z_1} \right| &= \left| \sum_{l=0}^{\infty} \left(\sum_{k=p+1}^{\infty} b_k (-1)^{k-l} (z_0 - z_1)^{k-l} \right) (z - z_1)^l \right| \\
&\leq \left| \sum_{k=p+1}^{\infty} b_k (z_1 - z_0)^k \right| \left| \sum_{l=0}^{\infty} \left(\frac{z - z_1}{z_1 - z_0} \right)^l \right| \\
&= \left| \sum_{k=p+1}^{\infty} k^{-1} \sum_C q_i \left(\frac{z_1 - z_0}{z_i - z_0} \right)^k \right| \left| \sum_{l=0}^{\infty} \left(\frac{z - z_1}{z_0 - z_1} \right)^l \right| \\
&\leq \frac{A}{(p+1)(1-c)(1-D)} c^{p+1},
\end{aligned}$$

where $A = \sum_C |q_i|$. $c = |z_1 - z_0| / \min(z_0, C)$ and $D = |z - z_1| / |z_0 - z_1|$.

At this moment, we have obtained all the information needed to perform the SHIFT operation for both multipole expansion and local Taylor expansion. Next, we will consider the operation which can transform multipole expansion to local Taylor expansion.

9.16 Operation No.2 — FLIP

At this section, we will introduce the more powerful operation in multipole algorithm, namely the FLIP operation. For now, we will consider only the transformation in the direction from the

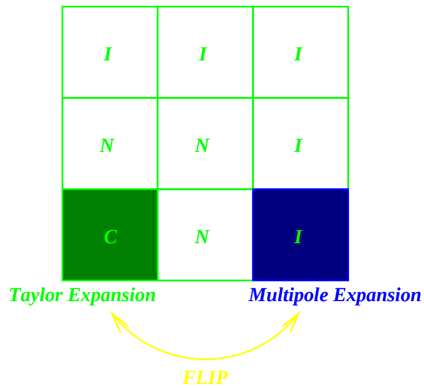


Figure 9.13: FLIP from Multipole to Taylor Expansion

multipole expansion $\phi_{z_0}(z)$ to the local Taylor expansion $\phi_{C,z_1}(z)$, denoted by

$$FLIP(\phi_{z_0}, z_0 \Rightarrow z_1) = \phi_{C,z_1}$$

For $|z - z_0| > \max(z_0, C)$ and $|z - z_1| < \min(z_1, C)$ both series converge. Note that

$$z - z_0 = -(z_0 - z_1)(1 - \frac{z - z_1}{z_0 - z_1}) = -(z_0 - z_1)(1 - \xi)$$

and assume also $|\xi| < 1$. Then,

$$\begin{aligned}
\phi_{z_0}(z) &= a_0 \log(z - z_0) + \sum_{k=1}^{\infty} a_k (z - z_0)^{-k} \\
&= a_0 \log(-(z_0 - z_1)) + a_0 \log(1 - \xi) + \sum_{k=1}^{\infty} a_k (-1)^k (z_0 - z_1)^{-k} (1 - \xi)^{-k} \\
&= a_0 \log(-(z_0 - z_1)) + \sum_{l=1}^{\infty} - a_0 l^{-1} \xi^l + \sum_{k=1}^{\infty} (-1)^k a_k (z_0 - z_1)^{-k} \sum_{l=0}^{\infty} \binom{k + l - 1}{l} \xi^l \\
&= \left(a_0 \log(-(z_0 - z_1)) + \sum_{k=1}^{\infty} (-1)^k a_k (z_0 - z_1)^{-k} \right) + \\
&\quad \sum_{l=1}^{\infty} \left(a_0 l^{-1} (z_0 - z_1)^{-l} + \sum_{k=1}^{\infty} (-1)^k a_k \binom{k + l - 1}{l} (z_0 - z_1)^{-(k+l)} \right) (z - z_1)^l.
\end{aligned}$$

Therefore coefficients a_k of ϕ_{z_0} transform to coefficients b_l of ϕ_{C,z_1} by the formula

$$\begin{aligned}
b_0 &= a_0 \log(-(z_0 - z_1)) + \sum_{k=1}^{\infty} (-1)^k a_k (z_0 - z_1)^{-k} \\
b_l &= a_0 l^{-1} (z_0 - z_1)^{-l} + \sum_{k=1}^{\infty} (-1)^k a_k \binom{k + l - 1}{l} (z_0 - z_1)^{-(k+l)} \quad l > 0
\end{aligned}$$

Note that FLIP does *not* commute with truncation since one has to know all coefficients $a_0, a_1, \ldots$ to compute $b_0, b_1, \ldots, b_p$ exactly. For more information on the error in case of truncation, see Greengard and Rokhlin (1987).

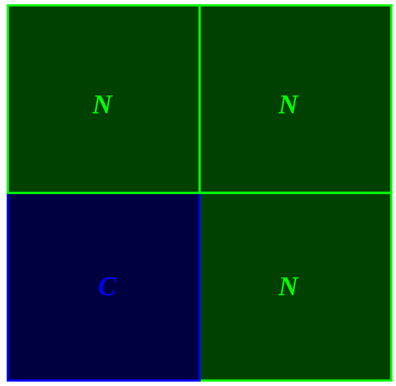


Figure 9.14: First Level of Quad Tree

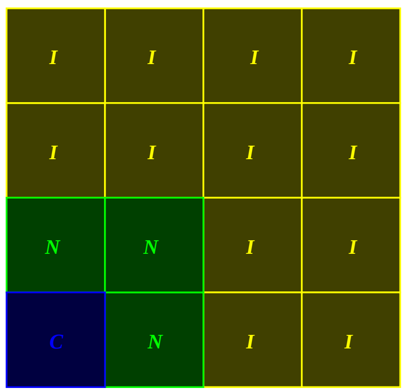


Figure 9.15: Second Level of Quad Tree

9.17 Application on Quad Tree

In this section, we will go through the application of multipole algorithm on quad tree in detail. During the process, we will also look into the two different operations $SHIFT$ and $FLIP$, and gain some experience on how to use them in real situations.

We will start at the lowest level h of the tree. For every node of the tree, it computes the multipole expansion coefficients for the bodies inside, with origin located at the center of the cell. Next, it will shift all of the four centers for the children cells into the center of the parent node, which is at the $h-1$ level, through the SHIFT operation for the multipole expansion. Adding up the coefficients from the four shifted expansion series, the multipole expansion of the whole parent node is obtained. And this SHIFT and ADD process will continue upward for every level of the tree, until the multipole expansion coefficients for each node of the entire tree are stored within that node. The computational complexity for this part is $O(N)$.

Before we go to the next step, some terms have to be defined first.

- NEIGHBOR — a neighbor N to a cell C is defined as any cell which shares either an edge or a corner with C
- INTERACTIVE — an interactive cell I to a cell C is defined as any cell whose parent is a neighbor to parent of C, excluding those which are neighbors to C

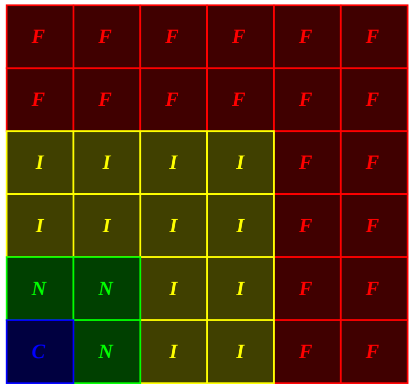


Figure 9.16: Third Level of Quad Tree

- FARAWAY — a faraway cell F to a cell C is defined as any cell which is neither a neighbor nor an interactive to C

Now, we start at the top level of the tree. For each cell C, FLIP the multipole expansion for the interactive cells and combine the resulting local Taylor expansions into one expansion series. After all of the FLIP and COMBINE operations are done, SHIFT the local Taylor expansion from the node in this level to its four children in the next lower level, so that the information is conserved from parent to child. Then go down to the next lower level where the children are. To all of the cells at this level, the faraway field is done (which is the interactive zone at the parent level). So we will concentrate on the interactive zone at this level. Repeat the FLIP operation to all of the interactive cells and add the flipped multipole expansion to the Taylor expansion shifted from parent node. Then repeat the COMBINE and SHIFT operations as before. This entire process will continue from the top level downward until the lowest level of the tree. In the end, add them together when the cells are *close* enough.

9.18 Expansion from 2-D to 3-D

For 2-D N-body simulation, the potential function is given as

$$\phi(z_j) = \sum_{i=1}^{n} q_i \log(z_j - z_i)$$

where $z_1, \ldots, z_n$ the position of particles, and $q_1, \ldots, q_n$ the strength of particles. The corresponding multipole expansion for the cluster centered at z_c is

$$\phi_{z_c}(z) = a_0 \log(z - z_c) + \sum_{k=1}^{\infty} a_k \frac{1}{(z - z_c)^k}$$

The corresponding local Taylor expansion looks like

$$\phi_{C,z_c}(z) = \sum_{k=0}^{\infty} b_k \frac{1}{(z - z_c)^k}$$

In three dimensions, the potential as well as the expansion series become much more complicated. The 3-D potential is given as

$$\Phi(x) = \sum_{i=1}^{n} q_i \frac{1}{||x - x_i||}$$

where $x = f(r, \theta, \phi)$. The corresponding multipole expansion and local Taylor expansion as following

$$\begin{aligned} \Phi_{multipole}(x) &= \sum_{n=0}^{\infty} \frac{1}{r^{n+1}} \sum_{m=-n}^{n} a_n^m Y_n^m(\theta, \phi) \\ \Phi_{Taylor}(x) &= \sum_{n=0}^{\infty} \sum_{m=-n}^{n} r^n b_n^m Y_n^m(\theta, \phi) \end{aligned}$$

where $Y_n^m(\theta, \phi)$ is the Spherical Harmonic function. For a more detailed treatment of 3-D expansions, see Nabors and White (1991).

9.19 Parallel Implementation

In Chapter 10.2, we will discussion issues on parallel N-body implementation.

Lecture 10

Partitioning and Load Balancing

Handling a large mesh or a linear system on a supercomputer or on a workstation cluster usually requires that the data for the problem be somehow partitioned and distributed among the processors. The quality of the partition affects the speed of solution: a good partition divides the work up evenly and requires as little communication as possible. Unstructured meshes may approximate irregular physical problems with fewer mesh elements, their use increases the difficulty of programming and requires new algorithms for partitioning and mapping data and computations onto parallel machines. Partitioning is also important for VLSI circuit layout and parallel circuit simulation.

10.1 Motivation from the Parallel Sparse Matrix Vector Multiplication

Multiplying a sparse matrix by a vector is a basic step of most commonly used iterative methods (conjugate gradient, for example). We want to find a way of efficiently parallelizing this operation.

Say that processor i holds the value of v_i. To update this value, processor need to compute a weighted sum of values at itself and all of its neighbors. This means it has to first receiving values from processors holding values of its neighbors and then computing the sum. Viewing this in graph terms, this corresponds to communicating with a node's nearest neighbors.

We therefore need to break up the vector (and implicitly matrix and graph) so that:

- We balance the computational load at each processor. This is directly related to the number of non-zero entries in its matrix block.
- We minimize the communication overhead. How many other values does a processor have to receive? This equals the number of these values that are held at other processors.

We must come up with a proper division to reduce overhead. This corresponds to dividing up the graph of the matrix among the processors so that there are very few crossing edges. First assume that we have 2 processors, and we wish to partition the graph for parallel processing. As an easy example, take a simplistic cut such as cutting the 2D regular grid of size n in half through the middle. Let's define the cut size as the number of edges whose endpoints are in different groups. A good cut is one with a small cut size. In our example, the cut size would be $\sqrt{n}$. Assuming that the cost of each communication is 10 times more than an local arithmetic operation. Then the total parallel cost of perform matrix-vector product on the grid is $(4n)/2 + 10\sqrt{n} = 2n + 10\sqrt{n}$. In general, for p processors, to need to partition the graph into p subgraphs.

Various methods are used to break up a graph into parts such that the number of crossing edges is minimized. Here we'll examine the case where p = 2 processors, and we want to partition the graph into 2 halves.

10.2 Separators

The following definitions will be of use in the discussion that follows. Let $G = (V, E)$ be an undirected graph.

- A **bisection** of G is a division of V into V_1 and V_2 such that $|V_1| = |V_2|$. (If $|V|$ is odd, then the cardinalities differ by at most 1). The **cost** of the bisection (V_1, V_2) is the number of edges connecting V_1 with V_2.
- If $\beta \in [1/2, 1)$, a β-**bisection** is a division of V such that $V_{1,2} \leq \beta|V|$.
- An **edge separator** of G is a set of edges that if removed, would break G into 2 pieces with no edges connecting them.
- A p-**way partition** is a division of V into p pieces V_1,V_2,...,V_p where the sizes of the various pieces differ by at most 1. The cost of this partition is the total number of edges crossing the pieces.
- A **vertex separator** is a set C of vertices that break G into 3 pieces A, B, and C where no edges connect A and B. We also add the requirement that A and B should be of roughly equal size.

Usually, we use edge partitioning for parallel sparse matrix-vector product and vertex partitioning for the ordering in direct sparse factorization.

10.3 Spectral Partitioning – One way to slice a problem in half

The three steps to illustrate the solution of that problem are:

A. Electrical Networks (for motiving the Laplace matrix of a graph)

B. Laplacian of a Graph

C. Partitioning (that uses the spectral information)

10.3.1 Electrical Networks

As an example we choose a certain configuration of resistors (1 ohm), which are combined as shown in fig. 10.1. A battery is connected to the nodes 3 and 4. It provides the voltage V_B. To obtain the current, we have to calculate the effective resistance of the configuration

$$I = \frac{V_B}{\text{eff. res.}}. \tag{10.1}$$

This so called *Graph* is by definition a *collection of nodes and edges.*

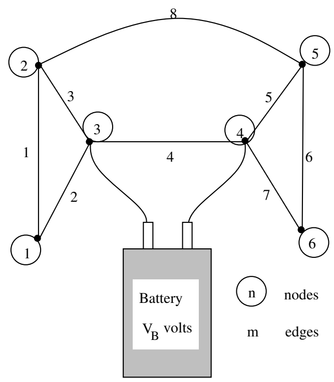


Figure 10.1: Resistor network with nodes and edges

10.3.2 Laplacian of a Graph

Kirchoff's Law tells us

$$\begin{pmatrix} 2 & -1 & -1 & & & \\ -1 & 3 & -1 & & -1 & \\ -1 & -1 & 3 & -1 & & \\ & & -1 & 3 & -1 & -1 \\ & -1 & & -1 & 3 & -1 \\ & & & -1 & -1 & 2 \end{pmatrix} \begin{pmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \\ V_5 \\ V_6 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ -1 \\ 0 \\ 0 \end{pmatrix} \tag{10.2}$$

The matrix contains the information of how the resistors are connected, it is called the *Laplacian of the Graph*

$$\nabla_G^2 = n \times n \quad \text{matrix,} \quad \begin{array}{l} \text{G...graph} \\ \text{n...nodes} \\ \text{m...edges} \end{array} \tag{10.3}$$

$$\left(\nabla_G^2\right)_{ii} = \text{degree of node } i \tag{10.4}$$

$$\left(\nabla_G^2\right)_{ij} = \begin{cases} 0 & \text{if } \nexists \text{ edge between node } i \text{ and node } j \\ -1 & \text{if } \exists \text{ edge between node } i \text{ and node } j \end{cases} \tag{10.5}$$

Yet there is another way to obtain the Laplacian. First we have to set up the so called *Node–edge Incidence matrix*, which is a $m \times n$ matrix and its elements are either 1, -1 or 0 depending on whether the edge (row of the matrix) connects the node (column of matrix) or not. We find

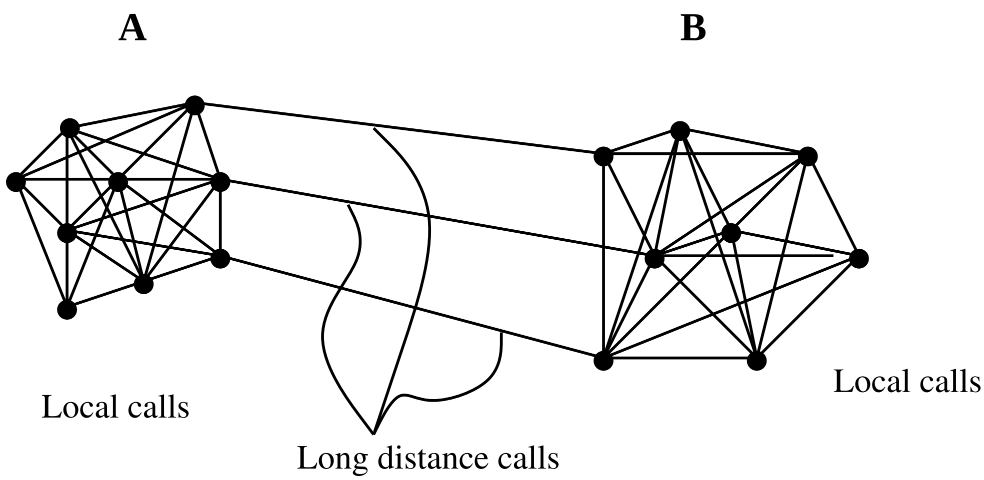


Figure 10.2: Partitioning of a telephone net as an example for a graph comparable in cost

$$\mathbf{M}_G = \begin{array}{cc} \text{nodes} & \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{array}{c} \text{edges} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array} & \begin{pmatrix} & & & & & \\ 1 & -1 & & & & \\ 1 & & -1 & & & \\ & 1 & -1 & & & \\ & & 1 & -1 & & \\ & & & 1 & -1 & \\ & & & & 1 & -1 \\ & & & 1 & & -1 \\ & 1 & & & -1 & \end{pmatrix} \end{array} \tag{10.6}$$

$$\mathbf{M}_G^T \mathbf{M}_G = \nabla_G^2 \tag{10.7}$$

10.3.3 Spectral Partitioning

Partitioning means that we would like to break the problem into 2 parts A and B, whereas the number of connections between both parts are to be as small as possible (because they are the most expensive ones), see fig. 10.2. Now we define a vector $\mathbf{x} \in \mathcal{R}^n$ having the values $\mathbf{x}_i = \pm 1$. $+1$ stands for *belongs to part A*, -1 means *belongs to part B*. With use of some vector calculus we can show the following identity

$$\sum_{i=1}^{n} (\mathbf{M}_G \mathbf{x})_i^2 = (\mathbf{M}_G \mathbf{x})^T (\mathbf{M}_G \mathbf{x}) = \mathbf{x}^T \mathbf{M}_G^T \mathbf{M}_G \mathbf{x} = \mathbf{x}^T \nabla_G^2 \mathbf{x} \tag{10.8}$$

$$\mathbf{x}^T \nabla_G^2 \mathbf{x} = 4 \times (\#\text{ edges between A and B}).$$

In order to find the vector $\mathbf{x}$ with the least connections between part A and B, we want to solve the minimization problem:

$$\left.\begin{array}{r} \text{Min. } \mathbf{x}^T \nabla_G^2 \mathbf{x} \\ x_i = \pm 1 \\ \sum x_i = 0 \end{array}\right\} \geq \left\{\begin{array}{r} \text{Min. } \mathbf{x}^T \nabla_G^2 \mathbf{x} \\ x_i \in \mathcal{R}^n \\ \sum x_i^2 = n \\ \sum x_i = 0 \end{array}\right. \tag{10.9}$$

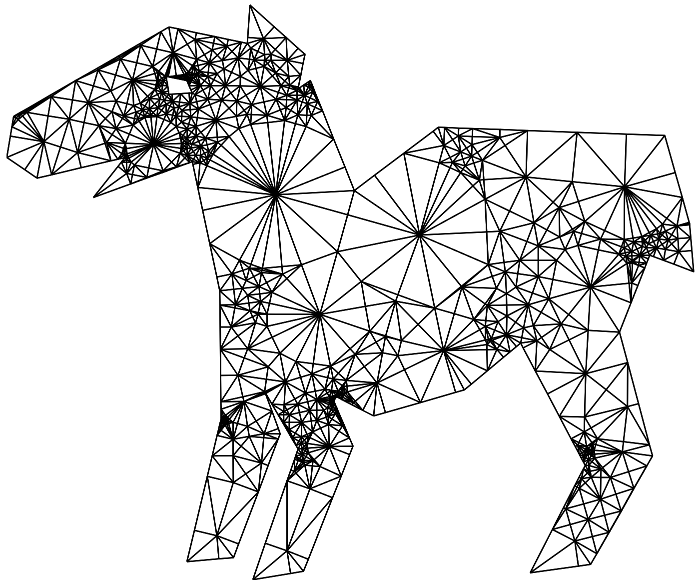


Figure 10.3: Tapir (Bern-Mitchell-Ruppert)

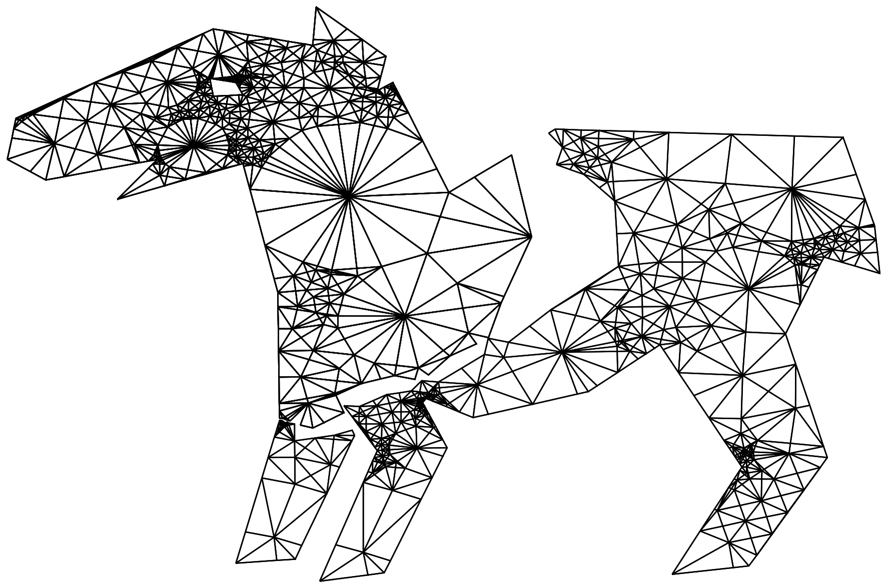
Finding an optimal ± 1 solution is intractable (NP-hard) in general. In practice, we relax the integer condition and allow the solution to be real. We do require the norm square of the solution, like the integer case, equal to n (see RHS of eqn. (10.9)). The relaxation enlarges the solution space, hence its optimal solution is no more than that of its integer counterpart. As an heuristic, we solve the relaxed version of the problem and "round" the solution to give an ± 1 solution.

The solution of the problem on the RHS of eqn. (10.9) gives us the second smallest eigenvalue of the Laplacian ∇_G^2 and the corresponding eigenvector, which is in fact the vector $\mathbf{x}$ with the least connections in the relaxation sense. To obtain a partition of the graph, we can divide its node set according to the vector x. We can even control the ratio of the partition. For example to obtain a bisection, i.e., a partition of each size, we can find the median of x and put those nodes whose x values is smaller than the median on one side and the remaining on the other side. We can also use the similar idea to divide the graph into two parts in which one is twice as big as the another.

Figure 10.3 shows a mesh together with its spectral partition. For those of you as ignorant as your professors, a tapir is defined by Webster as any of several large inoffensive chiefly nocturnal ungulates (family Tap iridae) of tropical America, Malaya, and Sumatra related to the horses and rhinoceroses. The tapir mesh is generated by a Matlab program written by Scott Mitchell based on a non-obtuse triangulation algorithm of Bern-Mitchell-Ruppert. The partition in the figure is generated by John Gilbert's spectral program in Matlab.

One has to notice that the above method is merely a heuristic. The algorithm does not come with any performance guarantee. In the worst case, the partition may be far from the optimal, partially due to the round-off effect and partially due the requirement of equal-sized partition. In fact the partition for the tapir graph is not quite optimal. Nevertheless, the spectral method, with the help of various improvements, works very well in practice. We will cover the partitioning problem more systematically later in the semester.

The most commonly used algorithm for finding the second eigenvalue and its eigenvector is the Lanczos algorithm though it makes far better sense to compute singular values rather than eigenvalues. Lanczos is an iterative algorithm that can converge quite quickly.



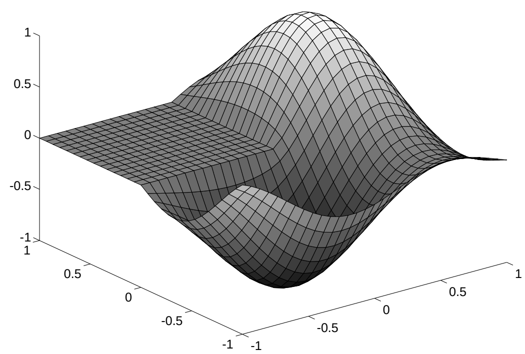


Figure 10.5: The 2nd eigenfunction of MathWork's logo

In general, we may need to divide a graph into more than one piece. The most commonly used approach is to recursively apply the partitioner that divides the graph into two pieces of roughly equal size. More systematic treatment of the partitioning problem will be covered in future lectures.

The success of the spectral method in practice has a physical interpretation. Suppose now we have a continuous domain in place of a graph. The Laplacian of the domain is the continuous counterpart of the Laplacian of a graph. The kth eigenfunction gives the kth mode of vibration and the second eigenfunction induce a cut (break) of the domain along the weakest part of the domain. (See figure10.5)

10.4 Geometric Methods

The spectral partitioning method only deals with the topology of the graph; the vertex locations are not considered. For many practical problems, the graph is derived from a mesh. The geometric method developed by Miller, Teng, Thurston, and Vavasis makes use of the vertex locations, and is guaranteed to partition well-shaped [1] d-dimensional unstructured mesh with n vertices using a

[1] For example, no grid angles can be too small/too large.

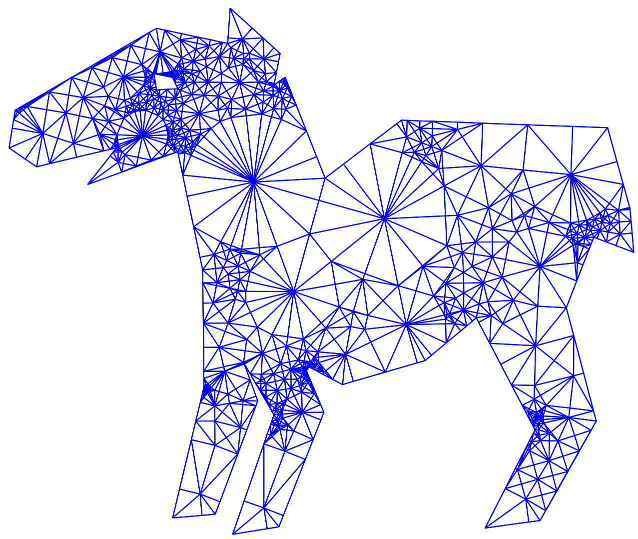


Figure 10.6: The input is a mesh with specified coordinates. Every triangle must be "well-shaped", which means that no angle can be too small. **Remarks:** The mesh is a subgraph of the intersection graph of a set of disks, one centered at each mesh point. Because the triangles are well-shaped, only a bounded number of disks can overlap any point, and the mesh is an "alpha-overlap graph". This implies that it has a good separator, which we proceed to find.

cut size $O(n^{1-1/d})$, where the cut size is the number of elements in the vertex separator. Note that this bound on the cut size is of the same order as for a regular grid of the same size. Such a bound does not hold for spectral method in general.

The geometric partitioning method has a great deal of theory behind it, but the implementation is relatively simple. For this reason, we will begin with an illustrative example before discussing the method and theoretical background in more depth. To motivate the software development aspect of this approach, we use the following figures (Figures 10.6 – 10.13) generated by a Matlab implementation (written by Gilbert and Teng) to outline the steps for dividing a well-shaped mesh into two pieces. The algorithm works on meshes in any dimension, but we'll stick to two dimensions for the purpose of visualization.

To recap, the geometric mesh partitioning algorithm can be summed up as follows (a more precise algorithm follows):

- Perform a stereographic projection to map the points in a d-dimensional plane to the surface of a $d+1$ dimensional sphere
- (Approximately) compute the centerpoint of the points on the sphere
- Perform a conformal map to move the centerpoint to the center of the sphere
- Find a plane through the center of the sphere that approximately divides the nodes equally; translate this plane to obtain a more even division
- Undo the conformal mapping and the stereographic mapping. This leaves a circle in the plane.

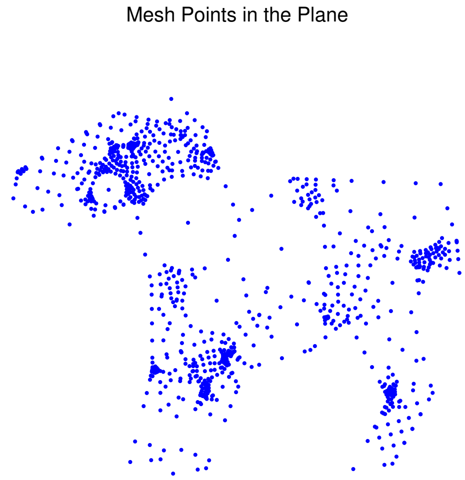


Figure 10.7: Let's redraw the mesh, omitting the edges for clarity.

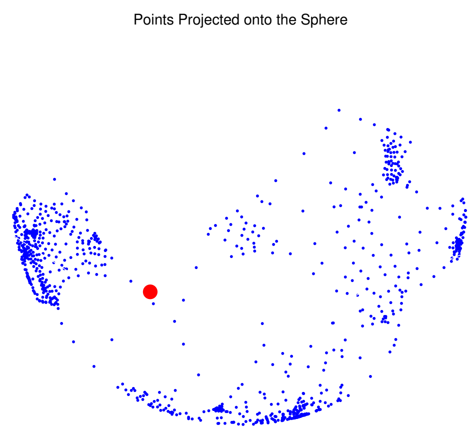


Figure 10.8: First we project the points stereographically from the plane onto the surface of a sphere (of one higher dimension than the mesh) tangent to the plane of the mesh. A stereographic projection is done by drawing a line between a point A on the plane and the north pole of the sphere, and mapping the point A to the intersection of the surface of the sphere and the line. Now we compute a "centerpoint" for the projected points in 3-space. A centerpoint is defined such that every plane through the centerpoint separates the input points into two roughly equal subsets. (Actually it's too expensive to compute a real centerpoint, so we use a fast, randomized heuristic to find a pretty good approximation.

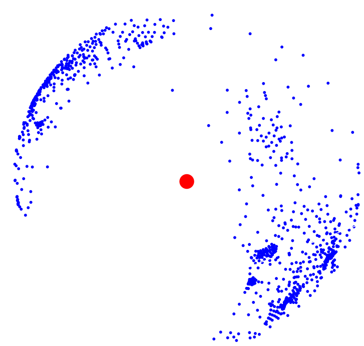


Figure 10.9: Next, we conformally map the points so that the centerpoint maps to the center of the sphere. This takes two steps: First we rotate the sphere about the origin (in 3-space) so that the centerpoint is on the z axis, and then we scale the points in the plane to move the centerpoint along the z axis to the origin (this can be thought of as mapping the sphere surface back to the plane, stretching the plane, then re-mapping the plane back to the surface of the sphere). The figures show the final result on the sphere and in the plane.

10.4.1 Geometric Graphs

This method applies to meshes in both two and three dimensions. It is based on the following important observation: graphs from large-scale problems in scientific computing are often defined geometrically. They are meshes of elements in a fixed dimension (typically two and three dimensions), that are *well shaped* in some sense, such as having elements of bounded aspect ratio or having elements with angles that are not too small. In other words, they are graphs embedded in two or three dimensions that come with natural geometric coordinates and with structures.

We now consider the types of graphs we expect to run these algorithms on. We don't expect to get a truly random graph. In fact, Erdös, Graham, and Szemerédi proved in the 1960s that with probability $= 1$, a random graph with cn edges does not have a two-way division with $o(n)$ crossing edges.

Structured graphs usually have divisions with $\sqrt{n}$ crossing edges. The following classes of graphs usually arise in applications such as finite element and finite difference methods (see Chapter **??**):

- **Regular Grids**: These arise, for example, from finite difference methods.
- **'Quad"-tree graphs and "Meshes"**: These arise, for example, from finite difference methods and hierarchical N-body simulation.
- **k-nearest neighbor graphs in d dimensions** Consider a set $P = \{p_1, p_2, \ldots, p_n\}$ of n points in $\mathbb{R}^d$. The vertex set for the graph is $\{1, 2, \ldots, n\}$ and the edge set is $\{(i, j) : p_j$is one of the k-nearest neighbors of p_i or vice-versa$\}$. This is an important class of graphs for image processing.

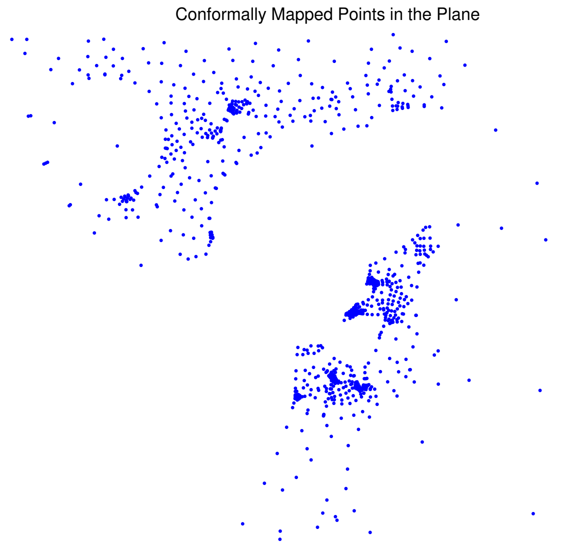


Figure 10.10: Because the approximate centerpoint is now at the origin, any plane through the origin should divide the points roughly evenly. Also, most planes only cut a small number of mesh edges (O($\sqrt{(n)}$), to be precise). Thus we find a separator by choosing a plane through the origin, which induces a great circle on the sphere. In practice, several potential planes are considered, and the best one accepted. Because we only estimated the location of the centerpoint, we must shift the circle slightly (in the normal direction) to make the split exactly even. The second circle is the shifted version.

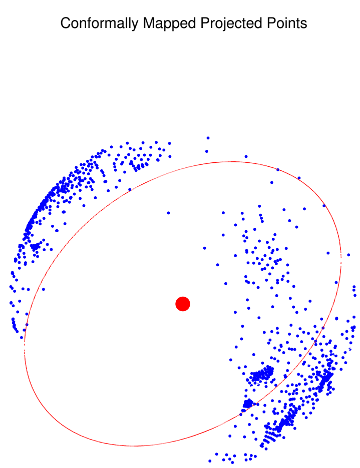


Figure 10.11: We now begin "undoing" the previous steps, to return to the plane. We first undo the conformal mapping, giving a (non-great) circle on the original sphere ...

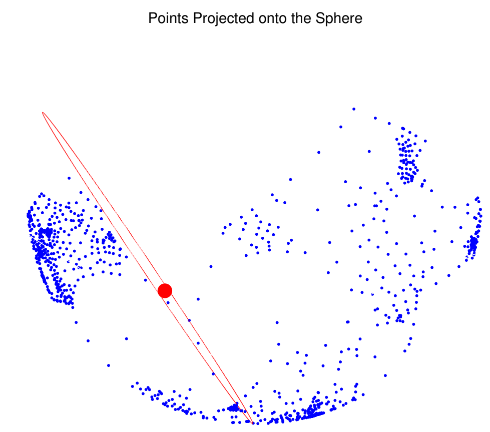


Figure 10.12: ... and then undo the stereographic projection, giving a circle in the original plane.

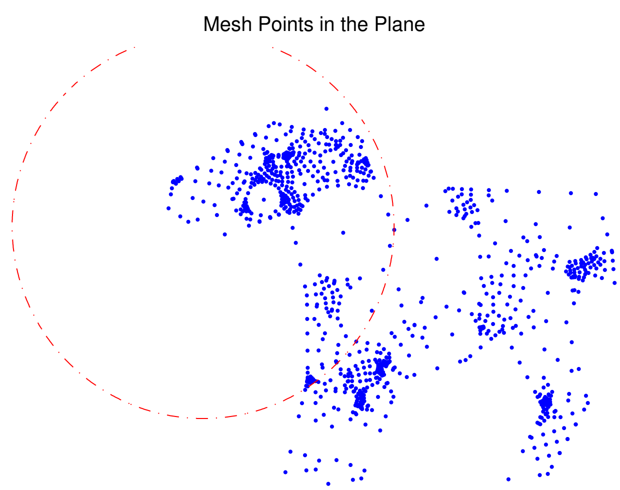


Figure 10.13: This partitions the mesh into two pieces with about n/2 points each, connected by at most $O(\sqrt{(}n))$ edges. These connecting edges are called an "edge separator". This algorithm can be used recursively if more divisions (ideally a power of 2) are desired.

- **Disk packing graphs**: If a set of non-overlapping disks is laid out in a plane, we can tell which disks touch. The nodes of a disk packing graph are the centers of the disks, and edges connect two nodes if their respective disks touch.
- **Planar graphs**: These are graphs that can be drawn in a plane without crossing edges. Note that disk packing graphs are planar, and in fact every planar graph is isomorphic to some disk-packing graph (Andreev and Thurston).

10.4.2 Geometric Partitioning: Algorithm and Geometric Modeling

The main ingredient of the geometric approach is a novel geometrical characterization of graphs embedded in a fixed dimension that have a small *separator*, which is a relatively small subset of vertices whose removal divides the rest of the graph into two pieces of approximately equal size. By taking advantage of the underlying geometric structure, partitioning can be performed efficiently.

Computational meshes are often composed of elements that are *well-shaped* in some sense, such as having bounded aspect ratio or having angles that are not too small or too large. Miller et al. define a class of so-called *overlap graphs* to model this kind of geometric constraint.

An overlap graph starts with a *neighborhood system*, which is a set of closed disks in d-dimensional Euclidean space and a parameter k that restricts how deeply they can intersect.

Definition 10.4.1 *A* k-ply neighborhood system in d dimensions *is a set* $\{D_1, \ldots, D_n\}$ *of closed disks in* $\mathbb{R}^d$*, such that no point in* $\mathbb{R}^d$ *is strictly interior to more than* k *of the disks.*

A neighborhood system and another parameter α define an overlap graph. There is a vertex for each disk. For $\alpha = 1$, an edge joins two vertices whose disks intersect. For $\alpha > 1$, an edge joins two vertices if expanding the smaller of their two disks by a factor of α would make them intersect.

Definition 10.4.2 *Let* $\alpha \geq 1$*, and let* $\{D_1, \ldots, D_n\}$ *be a* k*-ply neighborhood system. The* (α, k)-overlap graph *for the neighborhood system is the graph with vertex set* $\{1, \ldots, n\}$ *and edge set*

$$\{(i,j) | (D_i \cap (\alpha \cdot D_j) \neq \emptyset) \text{ and } ((\alpha \cdot D_i) \cap D_j \neq \emptyset)\}.$$

We make an overlap graph into a mesh in d-space by locating each vertex at the center of its disk.

Overlap graphs are good models of computational meshes because every mesh of bounded-aspect-ratio elements in two or three dimensions is contained in some overlap graph (for suitable choices of the parameters α and k). Also, every planar graph is an overlap graph. Therefore, any theorem about partitioning overlap graphs implies a theorem about partitioning meshes of bounded aspect ratio and planar graphs.

We now describe the geometric partitioning algorithm.

We start with two preliminary concepts. We let Π denote the *stereographic projection* mapping from $\mathbb{R}^d$ to S^d, where S^d is the unit d-sphere embedded in $\mathbb{R}^{d+1}$. Geometrically, this map may be defined as follows. Given $\boldsymbol{x} \in \mathbb{R}^d$, append '0' as the final coordinate yielding $\boldsymbol{x}' \in \mathbb{R}^{d+1}$. Then compute the intersection of S^d with the line in $\mathbb{R}^{d+1}$ passing through $\boldsymbol{x}'$ and $(0, 0, \ldots, 0, 1)^T$. This intersection point is $\Pi(\boldsymbol{x})$.

Algebraically, the mapping is defined as

$$\Pi(\boldsymbol{x}) = \begin{pmatrix} 2\boldsymbol{x}/\chi \\ 1 - 2/\chi \end{pmatrix}$$

where $\chi = \boldsymbol{x}^T\boldsymbol{x} + 1$. It is also simple to write down a formula for the inverse of Π. Let $\boldsymbol{u}$ be a point on S^d. Then

$$\Pi^{-1}(\boldsymbol{u}) = \frac{\bar{\boldsymbol{u}}}{1 - u_{d+1}}$$

where $\bar{\boldsymbol{u}}$ denotes the first d entries of $\boldsymbol{u}$ and u_{d+1} is the last entry. The stereographic mapping, besides being easy to compute, has a number of important properties proved below.

A second crucial concept for our algorithm is the notion of a *center point*. Given a finite subset $P \subset \mathbb{R}^d$ such that $|P| = n$, a *center point* of P is defined to be a point $\boldsymbol{x} \in \mathbb{R}^d$ such that if H is any open halfspace whose boundary contains $\boldsymbol{x}$, then

$$|P \cap H| \leq dn/(d+1). \tag{10.10}$$

It can be shown from Helly's theorem [25] that a center point always exists. Note that center points are quite different from centroids. For example, a center point (which, in the $d = 1$ case, is the same as a median) is largely insensitive to "outliers" in P. On the hand, a single distant outliers can cause the centroid of P to be displaced by an arbitrarily large distance.

Geometric Partitioning Algorithm

Let $P = \{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n\}$ be the input points in $\mathbb{R}^d$ that define the overlap graph.

1. Given $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n$, compute $P' = \{\Pi(\boldsymbol{p}_1), \ldots, \Pi(\boldsymbol{p}_n)\}$ so that $P' \subset S^d$.

2. Compute a center point $\boldsymbol{z}$ of P'.

3. Compute an orthogonal $(d+1) \times (d+1)$ matrix Q such that $Q\boldsymbol{z} = \boldsymbol{z}'$ where

$$\boldsymbol{z}' = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \theta \end{pmatrix}$$

   such that θ is a scalar.

4. Define $P'' = QP'$ (i.e., apply Q to each point in P'). Note that $P'' \subset S^d$, and the center point of P'' is $\boldsymbol{z}'$.

5. Let D be the matrix $[(1-\theta)/(1+\theta)]^{1/2}I$, where I is the $d \times d$ identity matrix. Let $P''' = \Pi(D\Pi^{-1}(P''))$. Below we show that the origin is a center point of P'''.

6. Choose a random great circle S_0 on S^d.

7. Transform S_0 back to a sphere $S \subset \mathbb{R}^d$ by reversing all the transformations above, i.e., $S = \Pi^{-1}(Q^{-1}\Pi(D^{-1}\Pi^{-1}(S_0)))$.

8. From S compute a set of vertices of G that split the graph as in Theorem **??**. In particular, define C to be vertices embedded "near" S, define A be vertices of $G - C$ embedded outside S, and define B to be vertices of $G - C$ embedded inside S.

We can immediately make the following observation: because the origin is a center point of P''', and the points are split by choosing a plane through the origin, then we know that $|A| \leq (d+1)n/(d+2)$ and $|B| \leq (d+1)n/(d+2)$ regardless of the details of how C is chosen. (Notice that the constant factor is $(d+1)/(d+2)$ rather than $d/(d+1)$ because the point set P' lies in

$\mathbb{R}^{d+1}$ rather than $\mathbb{R}^d$.) Thus, one of the claims made in Theorem **??** will follow as soon as we have shown that the origin is indeed a center point of P''' at the end of this section.

We now provide additional details about the steps of the algorithm, and also its complexity analysis. We have already defined stereographic projection used Step 1. Step 1 requires $O(nd)$ operations.

Computing a true center point in Step 2 appears to a very expensive operation (involving a linear programming problem with n^d constraints) but by using random (geometric) sampling, an approximate center point can be found in random constant time (independent of n but exponential in d) [100, 48]. An *approximate* center point satisfies 10.10 except with $(d+1+\epsilon)n/(d+2)$ on the right-hand side, where $\epsilon > 0$ may be arbitrarily small. Alternatively, a deterministic linear-time sampling algorithm can be used in place of random sampling [65, 96], but one must again compute a center of the sample using linear programming in time exponential in d [67, 41].

In Step 3, the necessary orthogonal matrix may be represented as a single Householder reflection—see [43] for an explanation of how to pick an orthogonal matrix to zero out all but one entry in a vector. The number of floating point operations involved is $O(d)$ independent of n.

In Step 4 we do not actually need to compute P''; the set P'' is defined only for the purpose of analysis. Thus, Step 4 does not involve computation. Note that the $\boldsymbol{z}'$ is the center point of P'' after this transformation, because when a set of points is transformed by any orthogonal transformation, a center point moves according to the same transformation (more generally, center points are similarly moved under any affine transformation). This is proved below.

In Step 6 we choose a random great circle, which requires time $O(d)$. This is equivalent to choosing plane through the origin with a randomly selected orientation. (This step of the algorithm can be made deterministic; see [**?**].) Step 7 is also seen to require time $O(d)$.

Finally, there are two possible alternatives for carrying out Step 8. One alternative is that we are provided with the neighborhood system of the points (i.e., a list of n balls in $\mathbb{R}^d$) as part of the input. In this case Step 8 requires $O(nd)$ operations, and the test to determine which points belong in A, B or C is a simple geometric test involving S. Another possibility is that we are provided with the nodes of the graph and a list of edges. In this case we determine which nodes belong in A, B, or C based on scanning the adjacency list of each node, which requires time linear in the size of the graph.

Theorem 10.4.1 *If M is an unstructured mesh with bounded aspect ratio, then the graph of M is a subgraph of a bounded overlap graph of the neighborhood system where we have one ball for each vertex of M of radius equal to half of the distance to its nearest vertices. Clearly, this neighborhood system has ply equal to 1.*

Theorem 10.4.1 (Geometric Separators [67]) *Let G be an n-vertex (α, k)-overlap graph in d dimensions. Then the vertices of G can be partitioned into three sets A, B, and C, such that*

- *no edge joins A and B,*
- *A and B each have at most $(d+1)/(d+2)$ vertices,*
- *C has only $O(\alpha k^{1/d} n^{(d-1)/d})$ vertices.*

Such a partitioning can be computed by the geometric-partitioning-algorithm in randomized linear time sequentially, and in $O(n/p)$ parallel time when we use a parallel machine of p processors.

10.4.3 Other Graphs with small separators

The following classes of graphs all have small separators:

- Lines have edge-separators of size 1. Removing the middle edge is enough.
- Trees have a 1-vertex separator with $\beta = 2/3$ - the so-called centroid of the tree.
- Planar Graphs. A result of Lipton and Tarjan shows that a planar graph of bounded degree has a $\sqrt{8n}$ vertex separator with $\beta = 2/3$.
- d dimensional regular grids (those are used for basic finite difference method). As a folklore, they have a separator of size $n^{1-1/d}$ with beta $\beta = 1/2$.

10.4.4 Other Geometric Methods

Recursive coordinate bisection

The simplest form of geometric partitioning is recursive coordinate bisection (RCB) [91, 101]. In the RCB algorithm, the vertices of the graph are projected onto one of the coordinate axes, and the vertex set is partitioned around a hyperplane through the median of the projected coordinates. Each resulting subgraph is then partitioned along a different coordinate axis until the desired number of subgraphs is obtained.

Because of the simplicity of the algorithm, RCB is very quick and cheap, but the quality of the resultant separators can vary dramatically, depending on the embedding of the graph in $\mathbb{R}^d$. For example, consider a graph that is "+"-shaped. Clearly, the best (smallest) separator consists of the vertices lying along a diagonal cut through the center of the graph. RCB, however, will find the largest possible separators, in this case, planar cuts through the centers of the horizontal and vertical components of the graph.

Inertia-based slicing

Williams [101] noted that RCB had poor worst case performance, and suggested that it could be improved by slicing orthogonal to the principal axes of inertia, rather than orthogonal to the coordinate axes. Farhat and Lesoinne implemented and evaluated this heuristic for partitioning [33].

In three dimensions, let $v = (v_x, v_y, v_z)^t$ be the coordinates of vertex v in $\mathbb{R}^3$. Then the inertia matrix I of the vertices of a graph with respect to the origin is given by

$$I = \begin{pmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{pmatrix}$$

where,

$$I_{xx} = \sum_{v \in V} v_y^2 + v_z^2, \quad I_{yy} = \sum_{v \in V} v_x^2 + v_z^2, \quad I_{zz} = \sum_{v \in V} v_x^2 + v_y^2$$

and, for $i, j \in \{x, y, z\}, i \neq j$,

$$I_{ij} = I_{ji} = -\sum_{v \in V} v_i v_j$$

The eigenvectors of the inertia matrix are the principal axes of the vertex distribution. The eigenvalues of the inertia matrix are the principal moments of inertia. Together, the principal axes and principal moments of inertia define the inertia ellipse; the axes of the ellipse are the principal axes of inertia, and the axis lengths are the square roots of the corresponding principal moments. Physically, the size of the principal moments reflect how the mass of the system is distributed with respect to the corresponding axis - the larger the principal moment, the more mass is concentrated at a distance from the axis.

Let I_1, I_2, and I_3 denote the principal axes of inertia corresponding to the principal moments $\alpha_1 \leq \alpha_2 \leq \alpha_3$. Farhat and Lesoinne projected the vertex coordinates onto I_1, the axis about which the mass of the system is most tightly clustered, and partitioned using a planar cut through the median. This method typically yielded a good initial separator, but did not perform as well recursively on their test mesh - a regularly structured "T"-shape.

Farhat and Lesoinne did not present any results on the theoretical properties of the inertia slicing method. In fact, there are pathological cases in which the inertia method can be shown to yield a very poor separator. Consider, for example, a "+"-shape in which the horizontal bar is very wide and sparse, while the vertical bar is relatively narrow but dense. I_1 will be parallel to the horizontal axis, but a cut perpendicular to this axis through the median will yield a very large separator. A diagonal cut will yield the smallest separator, but will not be generated with this method.

Gremban, Miller, and Teng show how to use moment of inertia to improve the geometric partitioning algorithm.

10.4.5 Partitioning Software

- **Chaco** written by Bruce Hendrickson and Rob Leland. To get code send email to bahendr@cs.sandia.gov (Bruce Hendrickson).

- Matlab Mesh Partitioning Toolbox: written by Gilbert and Teng. It includes both edge and vertex separators, recursive bipartition, nested dissection ordering, visualizations and demos, and some sample meshes. The complete toolbox is available by anonymous ftp from machine `ftp.parc.xerox.com` as file `/pub/gilbert/meshpart.uu`.

- **Spectral code**: Pothen, Simon, and Liou.

10.5 Load-Balancing N-body Simulation for Non-uniform Particles

The discussion of Chapter **??** was focused on particles that are more or less uniformly distributed. However, in practical simulations, particles are usually not uniformly distributed. Particles may be highly clustered in some regions and relatively scattered in some other regions. Thus, the hierarchical tree is adaptively generated, with smaller box for regions of clustered particles. The computation and communication pattern of a hierarchical method becomes more complex and often is not known explicitly in advance.

10.5.1 Hierarchical Methods of Non-uniformly Distributed Particles

In this chapter, we use the following notion of non-uniformity: We say a point set $P = \{\boldsymbol{p}_1, ..., \boldsymbol{p}_n\}$ in d dimensions is *μ-non-uniform* if the hierarchical tree generated for P has height $\log_{2^d}(n/m) + \mu$.

In other words, the ratio of the size of smallest leaf-box to the root-box is $1/2^{\log_{2^d}(n/m)+\mu}$. In practice, μ is less than 100.

The Barnes-Hut algorithm, as an algorithm, can be easily generalized to the non-uniform case. We describe a version of FMM for non-uniformly distributed particles. The method uses the box-box interaction. FMM tries to maximize the number of FLIPs among large boxes and also tries to FLIP between roughly equal sized boxes, a philosophy which can be described as: let parents do as much work as possible and then do the left-over work as much as possible before passing to the next generation. Let $\boldsymbol{c}_1, ..., \boldsymbol{c}_{2^d}$ be the set of child-boxes of the root-box of the hierarchical tree. FMM generates the set of all *interaction-pairs* of boxes by taking the union of Interaction-pair$(\boldsymbol{c}_i, \boldsymbol{c}_j)$ for all $1 \leq i < j \leq 2^d$, using the Interaction-Pair procedure defined below.

Procedure Interaction-Pair ($\boldsymbol{b}_1$, $\boldsymbol{b}_2$)

- If $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are β-well-separated, then $(\boldsymbol{b}_1, \boldsymbol{b}_2)$ is an interaction-pair.
- Else, if both $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are leaf-boxes, then particles in $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are near-field particles.
- Else, if both $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are not leaf-boxes, without loss of generality, assuming that $\boldsymbol{b}_2$ is at least as large as $\boldsymbol{b}_1$ and letting $\boldsymbol{c}_1, ..., \boldsymbol{c}_{2^d}$ be the child-boxes of $\boldsymbol{b}_2$, then recursively decide interaction pair by calling: Interaction-Pair$(\boldsymbol{b}_1,\boldsymbol{c}_i)$ for all $1 \leq i \leq 2^d$.
- Else, if one of $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ is a leaf-box, without loss of generality, assuming that $\boldsymbol{b}_1$ is a leaf-box and letting $\boldsymbol{c}_1, ..., \boldsymbol{c}_{2^d}$ be the child-boxes of $\boldsymbol{b}_2$, then recursively decide interaction pairs by calling: Interaction-Pair$(\boldsymbol{b}_1,\boldsymbol{c}_i)$ for all $1 \leq i \leq 2^d$.

FMM for far-field calculation can then be defined as: for each interaction pair $(\boldsymbol{b}_1, \boldsymbol{b}_2)$, letting $\Phi_i^p()$ $(i = 1, 2)$ be the multipole-expansion of $\boldsymbol{b}_i$, flip $\Phi_1^p()$ to $\boldsymbol{b}_2$ and add to $\boldsymbol{b}_2$'s potential Taylor-expansion. Similarly, flip $\Phi_2^p()$ to $\boldsymbol{b}_1$ and add to $\boldsymbol{b}_1$'s potential Taylor-expansion. Then traverse down the hierarchical tree in a preordering, shift and add the potential Taylor-expansion of the parent box of a box to its own Taylor-expansion.

Note that FMM for uniformly distributed particles has a more direct description (see Chapter **??**).

10.5.2 The Communication Graph for N-Body Simulations

In order to efficiently implement an N-body method on a parallel machine, we need to understand its communication pattern, which can be described by a graph that characterizes the pattern of information exchange during the execution of the method. The communication graph is defined on basic computational elements of the method. The basic elements of hierarchical N-body methods are boxes and points, where points give the locations of particles and boxes are generated by the hierarchical method. Formally, the communication graph is an edge-weighted directed graph, where the edges describe the pattern of communication and the weight on an edge specifies the communication requirement along the edge.

A Refined FMM for Non-Uniform Distributions

For parallel implementation, it is desirable to have a communication graph that uses small edge-weights and has small in- and out-degrees. However, some boxes in the set of interaction-pairs defined in the last section may have large degree!

FMM described in the last subsection has a drawback which can be illustrated by the following 2D example. Suppose the root-box is divided into four child-boxes A, B, C, and D. Assume further

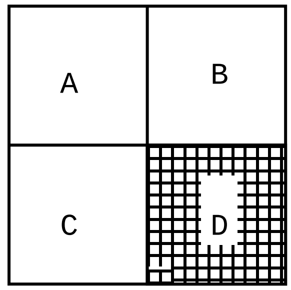


Figure 10.14: A non-uniform example

that boxes A, B and C contains less than m (< 100) particles, and most particles, say n of them, are uniformly distributed in D, see Figure 10.14. In FMM, we further recursively divide D by $\log_4(n/m)$ levels. Notice that A, B, and C are not well-separated from any box in D. Hence the FMM described in the previous subsection will declare all particles of D as near-field particles of A, B, and C (and vice versa). The drawback is two-folds: (1) From the computation viewpoint, we cannot take advantage of the hierarchical tree of D to evaluate potentials in A, B, and C. (2) From the communication viewpoint, boxes A, B, and C have a large in-degree in the sense that each particle in these boxes need to receive information from all n particles in D, making partitioning and load balancing harder. Notice that in BH most boxes of D are well-separated from particles in A, B, and C. Hence the well-separation condition is different in BH: *because BH uses the particle-box interaction, the well-separation condition is measured with respect to the size of the boxes in D. Thus most boxes are well-separated from particles in A, B, and C. In contrast, because FMM applies the FLIP operation, the well-separation condition must measure up against the size of the larger box. Hence no box in D is well-separated from A, B, and C.*

Our refined FMM circumvents this problem by incorporating the well-separation condition of BH into the Interaction-Pair procedure: if $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are not well-separated, and $\boldsymbol{b}_1$, the larger of the two, is a leaf-box, then we use a well-separation condition with respect to $\boldsymbol{b}_2$, instead of to $\boldsymbol{b}_1$, and apply the FLIP operation directly onto particles in the leaf-box $\boldsymbol{b}_1$ rather than $\boldsymbol{b}_1$ itself.

We will define this new well-separation condition shortly. First, we make the following observation about the Interaction-Pair procedure defined in the last subsection. We can prove, by a simple induction, the following fact: if $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are an interaction-pair and both $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are not leaf-boxes, then $1/2 \leq size(\boldsymbol{b}_1)/size(\boldsymbol{b}_2) \leq 2$. This is precisely the condition that FMM would like to maintain. For uniformly distributed particles, such condition is always true between any interaction-pair (even if one of them is a leaf-box). However, for non-uniformly distributed particles, if $\boldsymbol{b}_1$, the larger box, is a leaf-box, then $\boldsymbol{b}_1$ could be much larger than $\boldsymbol{b}_2$.

The new *β-well-separation* condition, when $\boldsymbol{b}_1$ is a leaf-box, is then defined as: $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are *β-well-separated* if $\boldsymbol{b}_2$ is well-separated from all particles of $\boldsymbol{b}_1$ (as in BH). Notice, however, with the new condition, we can no longer FLIP the multipole expansion of $\boldsymbol{b}_1$ to a Taylor-expansion for $\boldsymbol{b}_2$. Because $\boldsymbol{b}_1$ has only a constant number of particles, we can directly evaluate the potential induced by these particles for $\boldsymbol{b}_2$. This new condition makes the FLIP operation of this special class of interaction-pairs *uni-directional*: We only FLIP $\boldsymbol{b}_2$ to $\boldsymbol{b}_1$.

We can describe the refined Interaction-Pair procedure using modified well-separation condition when one box is a leaf-box.

Procedure Refined Interaction-Pair ($\boldsymbol{b}_1$, $\boldsymbol{b}_2$)

- If $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are β-well-separated and $1/2 \leq size(\boldsymbol{b}_1)/size(\boldsymbol{b}_2) \leq 2$, then $(\boldsymbol{b}_1, \boldsymbol{b}_2)$ is a bi-directional interaction-pair.

- Else, if the larger box, without loss of generality, $\boldsymbol{b}_1$, is a leaf-box, then the well-separation condition becomes: $\boldsymbol{b}_2$ is well-separated from all particles of $\boldsymbol{b}_1$. If this condition is true, then $(\boldsymbol{b}_1, \boldsymbol{b}_2)$ is a uni-directional interaction-pair from $\boldsymbol{b}_2$ to $\boldsymbol{b}_1$.

- Else, if both $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are leaf-boxes, then particles in $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are near-field particles.

- Else, if both $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are not leaf-boxes, without loss of generality, assuming that $\boldsymbol{b}_2$ is at least as large as $\boldsymbol{b}_1$ and letting $\boldsymbol{c}_1, ..., \boldsymbol{c}_{2^d}$ be the child-boxes of $\boldsymbol{b}_2$, then recursively decide interaction-pairs by calling: Interaction-Pair($\boldsymbol{b}_1$,$\boldsymbol{c}_i$) for all $1 \leq i \leq 2^d$.

- Else, if one of $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ is a leaf-box, without loss of generality, assuming that $\boldsymbol{b}_1$ is a leaf box and letting $\boldsymbol{c}_1, ..., \boldsymbol{c}_{2^d}$ be the child-boxes of $\boldsymbol{b}_2$, then recursively decide interaction pairs by calling: Interaction-Pair($\boldsymbol{b}_1$,$\boldsymbol{c}_i$) for all $1 \leq i \leq 2^d$.

Let $\boldsymbol{c}_1, ..., \boldsymbol{c}_{2^d}$ be the set of child-boxes of the root-box of the hierarchical tree. Then the set of all interaction-pair can be generated as the union of Refined-Interaction-Pair($\boldsymbol{c}_i, \boldsymbol{c}_j$) for all $1 \leq i < j \leq 2^d$.

The refined FMM for far-field calculation can then be defined as: for each bi-directional interaction pair $(\boldsymbol{b}_1, \boldsymbol{b}_2)$, letting $\Phi_i^p()$ $(i = 1, 2)$ be the multipole expansion of $\boldsymbol{b}_i$, flip $\Phi_1^p()$ to $\boldsymbol{b}_2$ and add to $\boldsymbol{b}_2$'s potential Taylor-expansion. Similarly, flip $\Phi_2^p()$ to $\boldsymbol{b}_1$ and add to $\boldsymbol{b}_1$'s potential Taylor-expansion. Then traverse down the hierarchical tree in a preordering, shift and add the potential Taylor-expansion of the parent box of a box to its own Taylor-expansion. For each uni-directional interaction pair $(\boldsymbol{b}_1, \boldsymbol{b}_2)$ from $\boldsymbol{b}_2$ to $\boldsymbol{b}_1$, letting $\Phi_2^p()$ be the multipole-expansion of $\boldsymbol{b}_2$, evaluate $\Phi_2^p()$ directly for each particle in $\boldsymbol{b}_2$ and add its potential.

Hierarchical Neighboring Graphs

Hierarchical methods (BH and FMM) explicitly use two graphs: the *hierarchical tree* which connects each box to its parent box and each particle to its leaf-box, and the *near-field graph* which connects each box with its near-field boxes. The hierarchical tree is generated and used in the first step to compute the multipole expansion induced by each box. We can use a bottom-up procedure to compute these multipole expansions: First compute the multipole expansions at leaf-boxes and then *SHIFT* the expansion to the parent boxes and then up the hierarchical tree until multipole-expansions for all boxes in the hierarchical tree are computed.

The near-field graph can also be generated by the Refined-Interaction-Pair procedure. In Section 10.5.3, we will formally define the near-field graph.

Fast-Multipole Graphs (FM)

The Fast-Multipole graph, FM^β, models the communication pattern of the refined FMM. It is a graph defined on the set of boxes and particles in the hierarchical tree. Two boxes $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are connected in FM^β iff (1) $\boldsymbol{b}_1$ is the parent box of $\boldsymbol{b}_2$, or vice versa, in the hierarchical tree; or (2) $(\boldsymbol{b}_1, \boldsymbol{b}_2)$ is an interaction-pair generated by Refined-Interaction-Pair defined in Section 10.5.2. The edge is bi-directional for a bi-directional interaction-pair and uni-directional for a uni-directional interaction-pair. Furthermore, each particle is connected with the box that contains the particle.

The following Lemma that will be useful in the next section.

Lemma 10.5.1 *The refined FMM flips the multipole expansion of $\boldsymbol{b}_2$ to $\boldsymbol{b}_1$ if and only if (1) $\boldsymbol{b}_2$ is well-separated from $\boldsymbol{b}_1$ and (2) neither the parent of $\boldsymbol{b}_2$ is well-separated from $\boldsymbol{b}_1$ nor $\boldsymbol{b}_2$ is well-separated from the parent of $\boldsymbol{b}_1$.*

It can be shown that both in- and out-degrees of FM^{β} are small.

Barnes-Hut Graphs (BH)

BH defines two classes of communication graphs: BH_S^{β} and BH_P^{β}. BH_S^{β} models the sequential communication pattern and BH_P^{β} is more suitable for parallel implementation. The letters S and P, in BH_S^{β} and BH_P^{β}, respectively, stand for "Sequential" and "Parallel".

We first define BH_S^{β} and show why parallel computing requires a different communication graph BH_P^{β} to reduce total communication cost.

The graph BH_S^{β} of a set of particles P contains two sets of vertices: P, the particles, and B, the set of boxes in the hierarchical tree. The edge set of the graph BH_S^{β} is defined by the communication pattern of the sequential BH. A particle $\boldsymbol{p}$ is connected with a box $\boldsymbol{b}$ if in BH, we need to evaluate $\boldsymbol{p}$ against $\boldsymbol{b}$ to compute the force or potential exerted on $\boldsymbol{p}$. So the edge is directed from $\boldsymbol{b}$ to $\boldsymbol{p}$. Notice that if $\boldsymbol{p}$ is connected with $\boldsymbol{b}$, then $\boldsymbol{b}$ must be well-separated from $\boldsymbol{p}$. Moreover, the parent of $\boldsymbol{b}$ is not well-separated from $\boldsymbol{p}$. Therefore, if $\boldsymbol{p}$ is connected with $\boldsymbol{b}$ in BH_S^{β}, then $\boldsymbol{p}$ is not connected to any box in the subtree of $\boldsymbol{b}$ nor to any ancestor of $\boldsymbol{b}$.

In addition, each box is connected directly with its parent box in the hierarchical tree and each point $\boldsymbol{p}$ is connected its leaf-box. Both types of edges are bi-directional.

Lemma 10.5.2 *Each particle is connected to at most $O(\log n + \mu)$ number of boxes. So the in-degree of BH_S^{β} is bounded by $O(\log n + \mu)$.*

Notice, however, BH_S^{β} is not suitable for parallel implementation. It has a large out-degree. This major drawback can be illustrated by the example of n uniformly distributed particles in two dimensions. Assume we have four processors. Then the "best" way to partition the problem is to divide the root-box into four boxes and map each box onto a processor. Notice that in the direct parallel implementation of BH, as modeled by BH_S^{β}, each particle needs to access the information of at least one boxes in each of the other processors. Because each processor has $n/4$ particles, the total communication overhead is $\Omega(n)$, which is very expensive.

The main problem with BH_S^{β} is that many particles from a processor need to access the information of the same box in some other processors (which contributes to the large out-degree). We show that a combination technique can be used to reduce the out-degree. The idea is to *combine* the "same" information from a box and send the information as one unit to another box on a processor that needs the information. We will show that this combination technique reduces the total communication cost to $O(\sqrt{n \log n})$ for the four processor example, and to $O(\sqrt{pn \log n})$ for p processors. Similarly, in three dimensions, the combination technique reduces the volume of messages from $\Omega(n \log n)$ to $O(p^{1/3} n^{2/3} (\log n)^{1/3})$.

We can define a graph BH_P^{β} to model the communication and computation pattern that uses this combination technique. Our definition of BH_P^{β} is inspired by the communication pattern of the refined FMM. It can be shown that the communication pattern of the refined FMM can be used to guide the message combination for the parallel implementation of the Barnes-Hut method!

The combination technique is based on the following observation: Suppose $\boldsymbol{p}$ is well-separated from $\boldsymbol{b}_1$ but not from the parent of $\boldsymbol{b}_1$. Let $\boldsymbol{b}$ be the largest box that contains $\boldsymbol{p}$ such that $\boldsymbol{b}$ is well-separated from $\boldsymbol{b}_1$, using the well-separation definition in Section 10.5.2. If $\boldsymbol{b}$ is not a leaf-box, then $(\boldsymbol{b}, \boldsymbol{b}_1)$ is a bi-directional interaction-pair in the refined FMM. If $\boldsymbol{b}$ is a leaf-box, then $(\boldsymbol{b}, \boldsymbol{b}_1)$ is a uni-directional interaction-pair from $\boldsymbol{b}_1$ to $\boldsymbol{b}$. Hence $(\boldsymbol{b}, \boldsymbol{b}_1)$ is an edge of FM^{β}. Then, any other particle $\boldsymbol{q}$ contained in $\boldsymbol{b}$ is well-separated from $\boldsymbol{b}_1$ as well. Hence we can combine the information from $\boldsymbol{b}_1$ to $\boldsymbol{p}$ and $\boldsymbol{q}$ and all other particles in $\boldsymbol{b}$ as follows: $\boldsymbol{b}_1$ sends its information (just one copy) to $\boldsymbol{b}$ and $\boldsymbol{b}$ forwards the information down the hierarchical tree, to both $\boldsymbol{p}$ and $\boldsymbol{q}$ and all other particles in $\boldsymbol{b}$. This combination-based-communication scheme defines a new communication graph BH_P^{β} for parallel BH: The nodes of the graph are the union of particles and boxes, i.e., $P \cup B(P)$. Each particle is connected to the leaf-box it belongs to. Two boxes are connected iff they are connected in the Fast-Multipole graph. However, to model the communication cost, we must introduce a weight on each edge along the hierarchical tree embedded in BH_P^{β}, to be equal to the number of data units needed to be sent along that edge.

Lemma 10.5.3 *The weight on each edge in BH_P^{β} is at most $O(\log n + \mu)$.*

It is worthwhile to point out the difference between the comparison and communication patterns in BH. In the sequential version of BH, if $\boldsymbol{p}$ is connected with $\boldsymbol{b}$, then we have to compare $\boldsymbol{p}$ against all ancestors of $\boldsymbol{b}$ in the computation. The procedure is to first compare $\boldsymbol{p}$ with the root of the hierarchical tree, and then recursively move the comparison down the tree: if the current box compared is not well-separated from $\boldsymbol{p}$, then we will compare $\boldsymbol{p}$ against all its child-boxes. However, in terms of force and potential calculation, we only evaluate a particle against the first box down a path that is well-separated from the particle. The graphs BH_S^{β} and BH_P^{β} capture the communication pattern, rather than the comparison pattern. The communication is more essential to force or potential calculation. The construction of the communication graph has been one of the bottlenecks in load balancing BH and FMM on a parallel machine.

10.5.3 Near-Field Graphs

The near-field graph is defined over all leaf-boxes. A leaf-box $\boldsymbol{b}_1$ is a *near-field neighbor* of a leaf-box $\boldsymbol{b}$ if $\boldsymbol{b}_1$ is not well-separated from some particles of $\boldsymbol{b}$. Thus, FMM and BH directly compute the potential at particles in $\boldsymbol{b}$ induced by particles of $\boldsymbol{b}_1$.

There are two basic cases: (1) if $size(\boldsymbol{b}_1) \leq size(\boldsymbol{b})$, then we call $\boldsymbol{b}_1$ a *geometric near-field neighbor* of $\boldsymbol{b}$. (2) if $size(\boldsymbol{b}_1) > size(\boldsymbol{b})$, then we call $\boldsymbol{b}_1$ a *hierarchical near-field neighbor* of $\boldsymbol{b}$. In the example of Section 10.5.2, A, B, C are hierarchical near-field neighbors of all leaf-boxes in D; while A, B, and C have some geometric near-field neighbors in D.

We introduce some notations. The *geometric in-degree* of a box $\boldsymbol{b}$ is the number of its geometric near-field neighbors. The *geometric out-degree* of a box $\boldsymbol{b}$ is the number of boxes to which b is the geometric near-field neighbors. The *hierarchical in-degree* of a box $\boldsymbol{b}$ is the number of its hierarchical near-field neighbors. We will define the *hierarchical out-degree* of a box shortly.

It can be shown that the geometric in-degree, geometric out-degree, and hierarchical in-degree are small. However, in the example of Section 10.5.2, A, B, and C are hierarchical near-field neighbors for all leaf-boxes in D. Hence the number of leaf-boxes to which a box is a hierarchical near-field neighbor could be very large. So the near-field graph defined above can have a very large out-degree.

We can use the combination technique to reduce the degree when a box $\boldsymbol{b}$ is a hierarchical near-field neighbor of a box $\boldsymbol{b}_1$. Let $\boldsymbol{b}_2$ be the ancestor of $\boldsymbol{b}_1$ of the same size as $\boldsymbol{b}$. Instead of $\boldsymbol{b}$ sending its information directly to $\boldsymbol{b}_1$, $\boldsymbol{b}$ sends it to $\boldsymbol{b}_2$ and $\boldsymbol{b}_2$ then forwards the information down the hierarchical tree. Notice that $\boldsymbol{b}$ and $\boldsymbol{b}_2$ are not well-separated. We will refer to this modified near-field graph as the near-field graph, denoted by NF^β. We also define the *hierarchical out-degree* of a box $\boldsymbol{b}$ to be the number of edges from $\boldsymbol{b}$ to the set of non-leaf-boxes constructed above. We can show that the *hierarchical out-degree* is also small.

To model the near-field communication, similar to our approach for BH, we introduce a weight on the edges of the hierarchical tree.

Lemma 10.5.4 *The weight on each edge in NF^β is at most $O(\log n + \mu)$.*

10.5.4 N-body Communication Graphs

By abusing notations, let $FM^\beta = FM^\beta \cup NF^\beta$ and $BH_P^\beta = BH_P^\beta \cup NF^\beta$. So the communication graph we defined simultaneously supports near-field and far-field communication, as well as communication up and down the hierarchical tree. Hence by partitioning and load balancing FM^β and BH_P^β , we automatically partition and balance the hierarchical tree, the near-field graph, and the far-field graph.

10.5.5 Geometric Modeling of N-body Graphs

Similar to well-shaped meshes, there is a geometric characterization of N-body communication graphs. Instead of using neighborhood systems, we use box-systems. A *box-system* in $\mathbb{R}^d$ is a set $B = \{B_1, \ldots, B_n\}$ of boxes. Let $P = \{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n\}$ be the centers of the boxes, respectively. For each integer k, the set B is a *k-ply box-system* if no point $\boldsymbol{p} \in \mathbb{R}^d$ is contained in more than k of $\mathrm{int}(B_1), \ldots, \mathrm{int}(B_n)$.

For example, the set of all leaf-boxes of a hierarchical tree forms a 1-ply box-system. The box-system is a variant of neighborhood system of Miller, Teng, Thurston, and Vavasis [67], where a neighborhood system is a collection of Euclidean balls in $\mathbb{R}^d$. We can show that box-systems can be used to model the communication graphs for parallel adaptive N-body simulation.

Given a box-system, it is possible to define the *overlap* graph associated with the system:

Definition 10.5.1 *Let $\alpha \geq 1$ be given, and let $\{B_1, \ldots, B_n\}$ be a k-ply box-system. The α-***overlap graph** *for this box-system is the undirected graph with vertices $V = \{1, \ldots, n\}$ and edges*

$$E = \{(i,j) : B_i \cap (\alpha \cdot B_j) \neq \emptyset \text{ and } (\alpha \cdot B_i) \cap B_j \neq \emptyset\}.$$

The edge condition is equivalent to: $(i,j) \in E$ iff the α dilation of the smaller box touches the larger box.

As shown in [96], the partitioning algorithm and theorem of Miller *et al* can be extended to overlap graphs on box-systems.

Theorem 10.5.1 *Let G be an α-overlap graph over a k-ply box-system in $\mathbb{R}^d$, then G can be partitioned into two equal sized subgraphs by removing at most $O(\alpha k^{1/d} n^{1-1/d})$ vertices. Moreover, such a partitioning can be computed in linear time sequentially and in parallel $O(n/p)$ time with p processors.*

The key observation is the following theorem.

Theorem 10.5.2 *Let $P = \{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n\}$ be a point set in $\mathbb{R}^d$ that is μ-non-uniform. Then the set of boxes $B(P)$ of hierarchical tree of P is a $(\log_{2^d} n + \mu)$-ply box-system and $FM^\beta(P)$ and $BH_P^\beta(P)$ are subgraphs of the 3β-overlap graph of $B(P)$.*

Therefore,

Theorem 10.5.3 *Let G be an N-body communication graph (either for BH or FMM) of a set of particles located at $P = \{\boldsymbol{p}_1, ..., \boldsymbol{p}_n\}$ in $\mathbb{R}^d$ ($d = 2$ or 3). If P is μ-non-uniform, then G can be partitioned into two equal sized subgraphs by removing at most $O(n^{1-1/d}(\log n + \mu)^{1/d})$ nodes. Moreover, such a partitioning can be computed in linear time sequentially and in parallel $O(n/p)$ time with p processors.*

Lecture 11

Mesh Generation

An essential step in scientific computing is to find a proper discretization of a continuous domain. This is the problem of *mesh generation*. Once we have a discretization or sometimes we just say a "mesh", differential equations for flow, waves, and heat distribution are then approximated by finite difference or finite element formulations. However, not all meshes are equally good numerically. Discretization errors depend on the geometric shape and size of the elements while the computational complexity for finding the numerical solution depends on the number of elements in the mesh and often the overall geometric quality of the mesh as well.

The most general and versatile mesh is an unstructured triangular mesh. Such a mesh is simply a triangulation of the input domain (e.g., a polygon), along with some extra vertices, called *Steiner points*. A triangulation is a decomposition of a space into a collection of interior disjoint simplices so that two simplices can only intersect at a lower dimensional simplex. We all know that in two dimensions, a simplex is a triangle and in three dimensions a simplex is a tetrahedron. A triangulation can be obtained by triangulating a point set, that form the vertices of the triangulation.

Even among triangulations some are better than others. Numerical errors depend on the *quality* of the triangulation, meaning the shapes and sizes of triangles.

In order for a mesh to be useful in approximating partial differential equations, it is necessary that discrete functions generated from the mesh (such as the piecewise linear functions) be capable of approximating the solutions sought. Classical finite element theory [18] shows that a sufficient condition for optimal approximation results to hold is that the minimum angle or the aspect ratio of each simplex in the triangulation be bounded independently of the mesh used; however, Babuska [4] shows that while this is sufficient, it is not a necessary condition. See Figure **??** for a triangulation whose minimum degree is at least 20 degree.

In summary, properties of a good mesh are:

- Fills the space
- Non-overlapping ($\sum$ areas = total area)
- Conforming Mesh (every edge shared by exactly 2 triangles)
- High Quality Elements (approximately equilateral triangles)

Automatic mesh generation is a relatively new field. No mathematically sound procedure for obtaining the 'best' mesh distribution is available. The criteria are usually heuristic.

The input description of physical domain has two components: the geometric definition of the domain and the numerical requirements within the domain. The geometric definition provides the

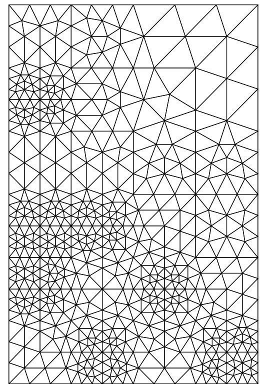


Figure 11.1: A well-shaped triangulation

boundary of the domain either in the form of a continuous model or of a discretized boundary model. Numerical requirements within the domain are typically obtained from an initial numerical simulation on a preliminary set of points. The numerical requirements obtained from the initial point set define an additional local spacing function restricting the final point set.

An automatic mesh generator try to generate an additional points to the internally and boundary of the domain to smooth out the mesh generation and concentrate mesh density where necessary - to optimize the total number of mesh points.

11.1 How to Describe a Domain?

The most intuitive and obvious structure of a domain (for modeling a scientific problem) is its geometry.

One way to describe the geometry is to use *constructive solid geometry* formula. In this approach, we have a set of basic geometric primitive shapes, such as boxes, spheres, half-spaces, triangles, tetrahedra, ellipsoids, polygons, etc. We then define (or approximate) a domain as finite unions, intersections, differences, and complementation of primitive shapes, i.e., by a well-structured formula of a finite length of primitive shapes with operators that include union, intersection, difference, and complementation.

An alternative way is to discretize the boundary of the domain, and describes the domain as a polygonal (polyhedral) object (perhaps with holes). Often we convert the constructive solid geometry formula into the discretized boundary description for mesh generation.

For many computational applications, often, some other information of a domain and the problem are equally important for quality mesh generation.

The numerical spacing functions, typically denoted by $h(\boldsymbol{x})$, is usually defined at a point $\boldsymbol{x}$ by the eigenvalues of the Hessian matrix of the solution u to the governing partial differential equations (PDEs) [4, 95, 69]. Locally, u behaves like a quadratic function

$$u(\boldsymbol{x} + d\boldsymbol{x}) = \frac{1}{2}(\boldsymbol{x} H \boldsymbol{x}^T) + \boldsymbol{x} \nabla u(\boldsymbol{x}) + u(\boldsymbol{x}),$$

where H is the *Hessian matrix* of u, the matrix of second partial derivatives. The spacing of mesh points, required by the accuracy of the discretization at a point $\boldsymbol{x}$, is denoted by $h(\boldsymbol{x})$ and should depend on the reciprocal of the square root of the largest eigenvalues of H at $\boldsymbol{x}$.

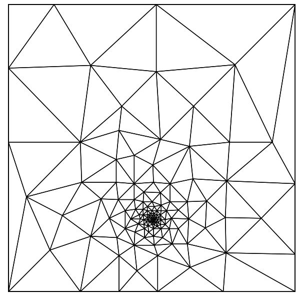


Figure 11.2: Triangulation of well-spaced point set around a singularity

When solving a PDE numerically, we estimate the eigenvalues of Hessian at a certain set of points in the domain based on the numerical approximation of the previous iteration [4, 95]. We then expand the spacing requirement induced by Hessian at these points over the entire domain.

For a problem with a smooth change in solution, we can use a (more-or-less) uniform mesh where all elements are of roughly equal size. On the other hand, for problem with rapid change in solution, such as earthquake, wave, shock modeling, we may use much dense grinding in the area of with high intensity. See Fig 11.2.So, the information about the solution structure can be of a great value to quality mesh generation.

Other type of information may come in the process of solving a simulation problem. For example, in adaptive methods, we may start with a much coarse and uniform grid. We then estimate the error of the previous step. Based on the error bound, we then adaptively refine the mesh, e.g., make the area with larger error much more dense for the next step calculation. As we shall argue later, unstructured mesh generation is more about finding the proper distribution of mesh point then the discretization itself (this is a very personal opinion).

11.2 Types of Meshes

- **Structured grids** divide the domain into regular grid squares. For examples finite difference gridding on a square grid. Matrices based on structured grids are very easy and fast to assemble. Structured grids are easy to generate; numerical formulation and solution based on structured grids are also relatively simple.

- **Unstructured grids** decompose the domain into simple mesh elements such as simplices based on a density function that is defined by the input geometry or the numerical requirements (e.g., from error estimation). But the associated matrices are harder and slower to assemble compared to the previous method; the resulting linear systems are also relatively hard to solve. Most of finite element meshes used in practice are of the unstructured type.

- **Hybrid grids** are generated by first decomposing the domain into non-regular domain and then decomposing each such domain by a regular grid. Hybrid grids are often used in domain decomposition.

Structured grids are much easy to generate and manipulate. The numerical theory of this discretization is better understood. However, its applications is limited to problems with simple domain and smooth changes in solution. For problems with complex geometry whose solution changes rapidly, we need to use an *unstructured mesh* to reduce the problem size. For example,

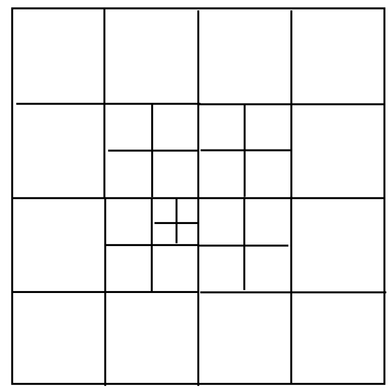


Figure 11.3: A quadtree

when modeling earthquake we want a dense discretization near the quake center and a sparse discretization in the regions with low activities. It would be waste to give regions with low activities as fine a discretization as the regions with high activities. Unstructured meshes are especially important for three dimensional problems.

The adaptability of unstructured meshes comes with new challenges, especially for 3D problems. However, the numerical theory becomes more difficult – this is an outstanding direction for future research; the algorithmic design becomes much harder.

11.3 Refinement Methods

A mesh generator usually does two things: (1) it generates a set of points that satisfies both geometric and numerical conditions imposed on the physical domain. (2) it builds a robust and well-shaped meshes over this point set, e.g., a triangulation of the point set. Most mesh generation algorithms merge the two functions, and generate the point set implicitly as part of the mesh generation phase. A most useful technique is to generate point set and its discretization by an iterative refinement. We now discuss *hierarchical refinement* and *Delaunay refinement*, two of the most commonly used refinement methods.

11.3.1 Hierarchical Refinement

The hierarchical refinement uses quadtrees in two dimensions and octtrees in three dimensions. The basic philosophy of using quad- and oct-trees in meshes refinements and hierarchical N-body simulation is the same: adaptively refining the domain by selectively and recursively divide boxes enable us to achieve numerical accuracy with close to an optimal discretization. The definition of quad- and oct-tree can be found in Chapter **??**. Figure 11.3 shows a quad-tree.

In quadtree refinement of an input domain, we start with a square box encompassing the domain and then adaptively splitting a box into four boxes recursively until each box small enough with respect to the geometric and numerical requirement. This step is very similar to quad-tree decomposition for N-body simulation. However, in mesh generation, we need to ensure that the mesh is well-shaped. This requirement makes mesh generation different from hierarchical N-body approximation. In mesh generate, we need to generate a set of smooth points. In the context of quad-tree refinement, it means that we need to make the quad-tree *balanced* in the sense that no leaf-box is adjacent to a leaf-box more than twice its side length.

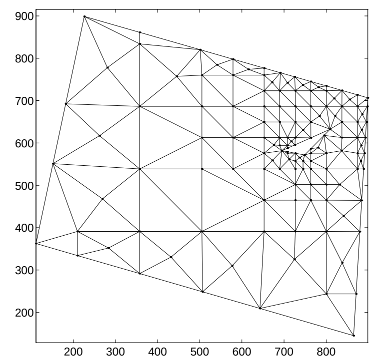


Figure 11.4: A well-shaped mesh generated by quad-tree refinement

With adaptive hierarchical trees, we can "optimally" approximate any geometric and numerical spacing function. The proof of the optimality can be found in the papers of Bern, Eppstein, and Gilbert for 2D and Mitchell and Vavasis for 3D. Formal discuss of the numerical and geometric spacing function can be found in the point generation paper of Miller, Talmor and Teng.

The following procedure describes the basic steps of hierarchical refinement.

1. Construct the hierarchical tree for the domain so that the leaf boxes approximate the numerical and geometric spacing functions.

2. Balance the hierarchical tree.

3. Warping and triangulation: If a point is too close to a boundary of its leaf box then one of the corners collapses to that point.

11.3.2 Delaunay Triangulation

Suppose $P = \{p_1, \ldots, p_n\}$ is a point set in d dimensions. The convex hull of $d+1$ affinely independent points from P forms a *Delaunay simplex* if the circumscribed ball of the simplex contains no point from P in its interior. The union of all Delaunay simplices forms the *Delaunay diagram*, $DT(P)$. If the set P is not degenerate then the $DT(P)$ is a simplex decomposition of the convex hull of P. We will sometimes refer to the Delaunay simplex as a triangle or a tetrahedron.

Associated with $DT(P)$ is a collection of balls, called *Delaunay balls*, one for each cell in $DT(P)$. The Delaunay ball circumscribes its cell. When points in P are in general position, each Delaunay simplex define a Delaunay ball, its circumscribed ball. By definition, there is no point from P lies in the interior of a Delaunay ball. We denote the set of all Delaunay balls of P by $DB(P)$.

The geometric dual of Delaunay Diagram is the *Voronoi Diagram*, which of consists a set of polyhedra $V_1, \ldots, V_n$, one for each point in P, called the *Voronoi Polyhedra*. Geometrically, V_i is the set of points $p \in \mathbb{R}^d$ whose Euclidean distance to p_i is less than or equal to that of any other point in P. We call p_i the *center* of polyhedra V_i. For more discussion, see [78, 31].

The DT has some very desired properties for mesh generation. For example, among all triangulations of a point set in 2D, the DT maximizes the smallest angle, it contains the nearest-neighbors graph, and the minimal spanning tree. Thus Delaunay triangulation is very useful for computer graphics and mesh generation in two dimensions. Moreover, discrete maximum principles will only exist for Delaunay triangulations. Chew [17] and Ruppert [84] have developed Delaunay refinement algorithms that generate provably good meshes for 2D domains.

Notice that an internal diagonal belongs to the Delaunay triangulation of four points if the sum of the two opposing angles is less than π.

A 2D Delaunay Triangulation can be found by the following simple algorithm: *FLIP algorithm*

- Find any triangulation (can be done in $O(n \lg n)$ time using divide and conquer.)
- For each edge pq, let the two faces the edge is in be prq and psq. Then pq is not an local Delaunay edge if the interior the circumscribed circle of prq contains s. Interestingly, this condition also mean that the interior of the circumscribed circle of psq contains r and the sum of the angles prq and psq is greater than π. We call the condition that the sum of the angles of prq and psq is no more than π *the angle condition*. Then, if pq does not satisfy the angle property, we just flip it: remove edge pq from T, and put in edge rs. Repeat this until all edges satisfy the angle property.

It is not too hard to show that if FLIP terminates, it will output a Delaunay Triangulation. A little addition geometric effort can show that the FLIP procedure above, fortunately, always terminate after at most $O(n^2)$ flips.

The following is an interesting observation of Guibas, Knuth and Sharir. If we choose a random ordering π of from $\{1, ..., n\}$ to $\{1, ..., n\}$ and permute the points based on π: $p_{\pi(1)} \ldots p_{\pi(n)}$. We then incrementally insert the points into the the current triangulation and perform flip if needed. Notice that the initial triangulation is a triangle formed by the first three points. It can be shown that the expected number of flips of the about algorithm is $O(n \log n)$. This gives a randomized $O(n \log n)$ time DT algorithm.

11.3.3 Delaunay Refinement

Even though the Delaunay triangulation maximize the smallest angles. The Delaunay triangulation of most point sets are bad in the sense that one of the triangles is too 'skinny'. In this case, Chew and Ruppert observed that we can refine the Delaunay triangulation to improve its quality. This idea is first proposed by Paul Chew. In 1992, Jim Ruppert gave a quality guaranteed procedure.

- Put a point at the circumcenter of the skinny triangle
- if the circum-center encroaches upon an edge of an input segment, split an edge adding its middle point; otherwise add the circumcenter.
- Update the Delaunay triangulation by FLIPPING.

A point *encroaches* on an edge if the point is contained in the interior of the circle of which the edge is a diameter. We can now define two operations, Split-Triangle and Split-Segment

Split-Triangle(T)

- Add circumcenter C of Triangle T
- Update the Delaunay Triangulation $(P \cup C)$

Split-Segment(S)

- Add midpoint m
- Update the Delaunay Triangulation $(P \cup M)$

The *Delaunay Refinement* algorithm then becomes

- Initialize
  - Perform a Delaunay Triangulation on P
  - IF some segment, l, is not in the Delaunay Triangle of P, THEN Split-Segment (l)
- Repeat the following until $\alpha > 25°$
  - IF T is a skinny triangle, try to Split(T)
  - IF C of T is 'close' to segment S then Split-Seg(S)
  - ELSE Split Triangle (T)

The following theory was then stated, without proof. The proof is first given by Jim Ruppert in his Ph.D thesis from UC. Berkeley.

Theorem 11.3.1 Not only does the Delaunay Refinement produce all triangles so that $MIN\ \alpha > 25°$, the size of the mesh it produces is no more than $C * Optimal(size)$.

11.4 Working With Meshes

- Smoothing: Move individual points to improve the mesh.
- Laplacian Smoothing: Move every element towards the center of mass of neighbors
- Refinement: Have a mesh, want smaller elements in a certain region. One option is to put a circumscribed triangles inside the triangles you are refining. Another approach is to split the longest edge on the triangles. However, you have to be careful of hanging nodes while doing this.

11.5 Unsolved Problems

There still is no good algorithm to generate meshes of shapes. There has been a fair amount of research and this is an important topic, but at the current time the algorithms listed above are merely best efforts and require a lot of work and/or customization to produce good meshes. A good algorithm would fill the shape with quadrilaterals and run without any user input, beyond providing the geometry to be meshed. The holy grail of meshing remains filling a 3-D shape with 3-D blocks.

Lecture 12

Support Vector Machines and Singular Value Decomposition

Andrew Hogue April 29, 2004

12.1 Support Vector Machines

Support Vector Machines, or SVMs, have emerged as an increasingly popular algorithm for effectively learning a method to categorize objects based on a set of properties. SVMs were originally introduced by Vapnik [97], although his work was in Russian and was not translated to English for several years. Recently, more attention has been paid to them, and there are several excellent books and tutorials [14, 51, 98, 99].

We first develop the notion of *learning* a categorization function, then describe the SVM algorithm in more detail, and finally provide several examples of its use.

12.1.1 Learning Models

In traditional programming, a function is given a value x as input, and computes a value $y = h(x)$ as its output. In this case, the programmer has an intimate knowledge of the function $h(x)$, which he explicitly defines in his code.

An alternative approach involves *learning* the function $h(x)$. There are two methods to generate this function. In *unsupervised learning*, the program is given a set of objects $\mathbf{x}$ and is asked to privide a function $h(\mathbf{x}$ to categorize them without any *a priori* knowledge about the objects.

Support Vector Machines, on the other hand, represent one approach to *supervised learning*. In supervised learning, the program is given a *training set* of pairs $(\mathbf{x}, \mathbf{y})$, where $\mathbf{x}$ are the objects being classified and $\mathbf{y}$ are a matching set of *categories*, one for each x_i. For example, an SVM attempting to categorize handwriting samples might be given a training set which maps several samples to the correct character. From these samples, the supervised learner induces the function $y = h(x)$ which attempts to return the correct category for each sample. In the future, new samples may be categorized by feeding them into the pre-learned $h(x)$.

In learning problems, it is important to differentiate between the types of possible output for the function $y = h(x)$. There are three main types:

Binary Classification $h(x) = y \in \pm 1$ (The learner classifies objects into one of two groups.)

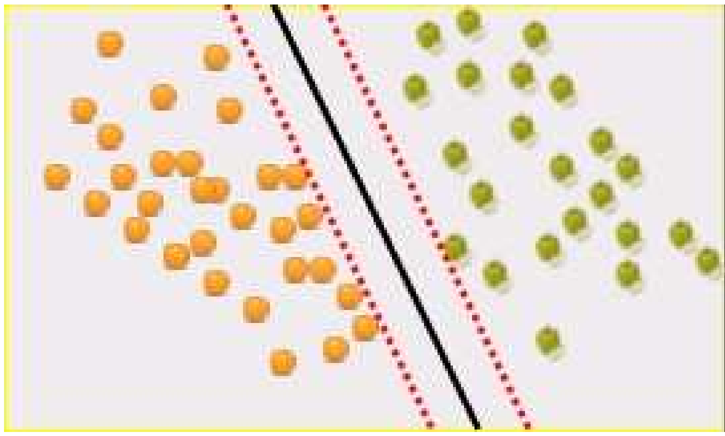


Figure 12.1: An example of a set of linearly separable objects.

Discrete Classification $h(x) = y \in \{1, 2, 3, ..., n\}$ (The learner classifies objects into one of n groups.)

Continuous Classification $h(x) = y \in \Re^n$ (The learner classifies objects in the space of n-dimensional real numbers.)

As an example, to learn a linear regression, x could be a collection of data points. In this case, the $y = h(x)$ that is learned is the best fit line through the data points.

All learning methods must also be aware of *overspecifying* the learned function. Given a set of training examples, it is often inappropriate to learn a function that fits these examples *too* well, as the examples often include noise. An overspecified function will often do a poor job of classifying new data.

12.1.2 Developing SVMs

A Support Vector Machine attempts to find a linear classification of a set of data. For dimension $d > 2$ this classification is represented by a *separating hyperplane*. An example of a separating hyperplane is shown in Figure 12.1. Not that the points in Figure 12.1 are *linearly separable*, that is, there exists a hyperplane in $\Re^2$ such that all oranges are on one side of the hyperplane and all apples are on the other.

In the case of a linearly separable training set, the *perceptron* model [80] is useful for finding a linear classifier. In this case, we wish to solve the equation

$$h(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$$

subject to the constraints that $h(\mathbf{x_1}) \leq -1$ and $h(\mathbf{x_2}) \geq +1$, where $\mathbf{x_1}$ are the objects in category 1 and $\mathbf{x_2}$ are the objects in category 2. For example, in Figure 12.1, we would wish to satisfy:

$$\begin{aligned} h(orange) &\leq -1 \\ h(apple) &\geq +1 \end{aligned}$$

The SVM method for finding the *best* separating hyperplane is to solve the following linear program:

$$\min_{w,b} \frac{\|\mathbf{w}\|^2}{2} \qquad \text{subject to} \tag{12.1}$$

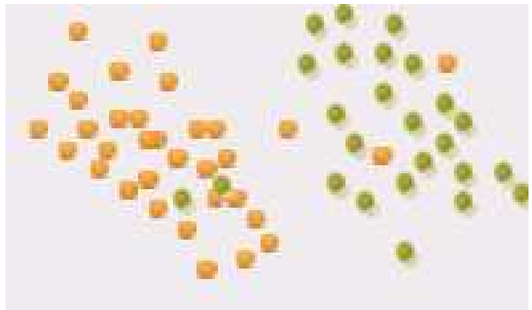


Figure 12.2: An example of a set of objects which are not linearly separable.

$$y_i(w_i^T x_i + b) \geq 1, i = 1, ..., n \tag{12.2}$$

This method works well for training sets which are linearly separable. However, there are also many cases where the training set is *not* linearly separable. An example is shown in Figure 12.2. In this case, it is impossible to find a separating hyperplane between the apples and oranges.

There are several methods for dealing with this case. One method is to add *slack* variables, ε_i to the linear program:

$$\min_{w,b,\varepsilon} \frac{\|\mathbf{w}\|^2}{2} + c\sum_i \varepsilon_i \qquad \text{subject to}$$
$$y_i(w_i^T x_i + b) \geq 1 - \varepsilon_i, i = 1, ..., n$$

Slack variables allow some of the x_i to "move" in space to "slip" onto the correct side of the separating hyperplane. For instance, in Figure 12.2, the apples on the left side of the figure could have associated ε_i which allow them to move to the right and into the correct category.

Another approach is to non-linearly distort space around the training set using a function $\Phi(x_i)$:

$$\min_{w,b} \frac{\|\mathbf{w}\|^2}{2} \qquad \text{subject to}$$
$$y_i(w_i^T \Phi(x_i) + b) \geq 1, i = 1, ..., n$$

In many cases, this distortion moves the objects into a configuration that is more easily separated by a hyperplane. As mentioned above, one must be careful not to overspecify $\Phi(x_i)$, as it could create a function that is unable to cope easily with new data.

Another way to approach this problem is through the dual of the linear program shown in Equations (12.1) and (12.2) above. If we consider those equations to be the primal, the dual is:

$$\max_{\alpha} \frac{\alpha^{\mathbf{T}}\mathbf{1} - \alpha^{\mathbf{T}}\mathbf{H}\alpha}{2} \qquad \text{subject to}$$
$$\mathbf{y}^{\mathbf{T}}\alpha = 0$$
$$\alpha \geq 0$$

Note that we have introduced *Lagrange Multipliers* α_i for the dual problem. At optimality, we have

$$w = \sum_i y_i \alpha_i x_i$$

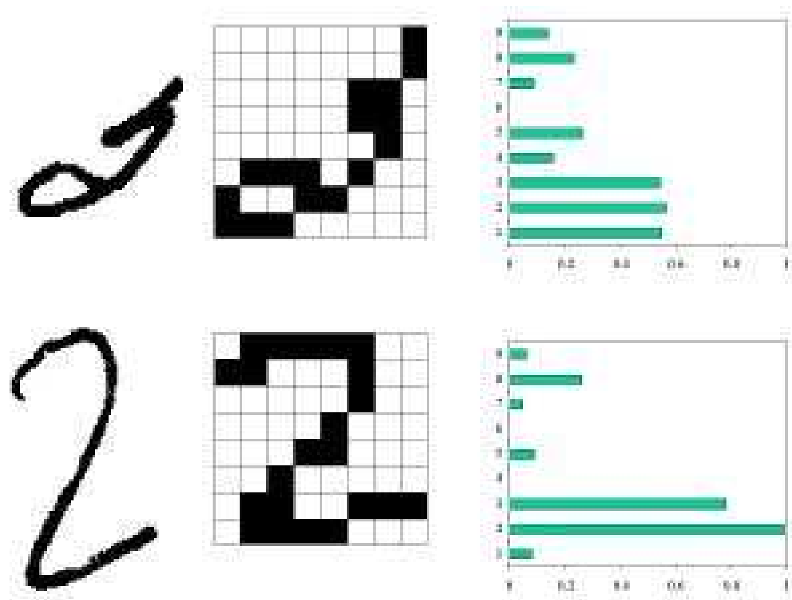


Figure 12.3: An example of classifying handwritten numerals. The graphs on the right show the probabilities of each sample being classified as each number, 0-9.

This implies that we may find a separating hyperplane using

$$H_{ij} = \mathbf{y_i}(\mathbf{x_i^T}\mathbf{x_j})\mathbf{y_j}$$

This dual problem also applies to the slack variable version using the constraints:

$$\mathbf{y^T}\alpha = 0$$
$$\mathbf{c} \geq \alpha \geq 0$$

as well as the distorted space version:

$$H_{ij} = \mathbf{y_i}(\Phi(\mathbf{x_i})^T\Phi(\mathbf{x_j}))\mathbf{y_j}$$

12.1.3 Applications

The type of classification provided by Support Vector Machines is useful in many applications. For example, the post office must sort hundreds of thousands of hand-written envelopes every day. To aid in this process, they make extensive use of handwriting recognition software which uses SVMs to automatically decipher handwritten numerals. An example of this classification is shown in Figure 12.3.

Military uses for SVMs also abound. The ability to quickly and accurately classify objects in a noisy visual field is essential to many military operations. For instance, SVMs have been used to identify humans or artillery against the backdrop of a crowded forest of trees.

12.2 Singular Value Decomposition

Many methods have been given to decompose a matrix into more useful elements. Recently, one of the most popular has been the *singular value decomposition*, or SVD. This decomposition has been

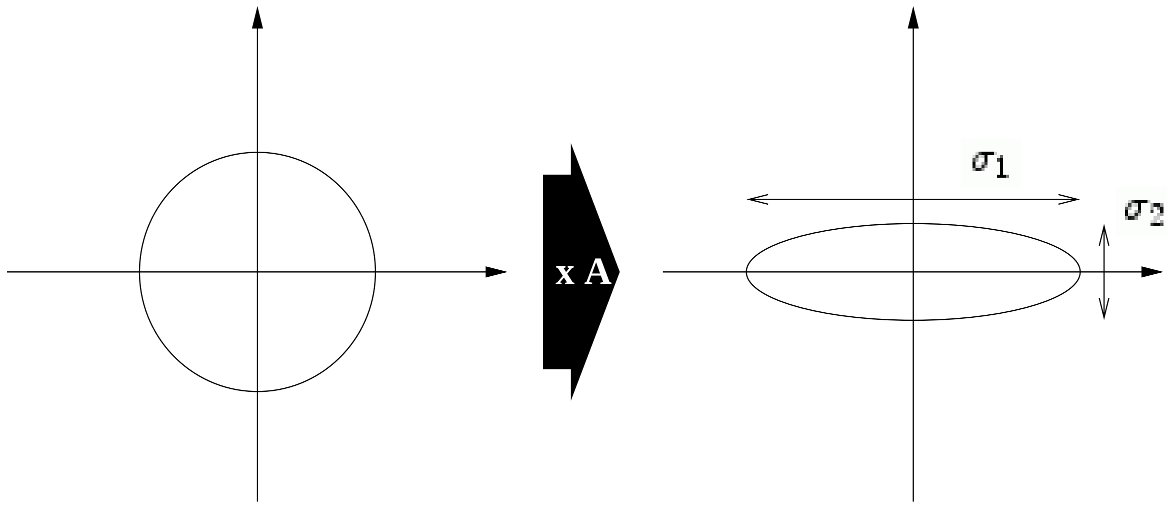


Figure 12.4: The deformation of a circle using a matrix **A**.

known since the early 19th century [94].

The SVD has become popular for several reasons. First, it is stable (that is, small perturbations in the input **A** result in small perturbations in the singular matrix $\mathbf{\Sigma}$, and vice versa). Second, the singular values σ_i provide an easy way to approximate **A**. Finally, there exist fast, stable algorithms to compute the SVD [42].

The SVD is defined by

$$\mathbf{A} = \mathbf{U\Sigma V^T}$$

Both **U** and **V** are orthagonal matrices, that is, $\mathbf{U^TU} = \mathbf{I}$ and $\mathbf{V^TV} = \mathbf{I}$. $\mathbf{\Sigma}$ is the *singular matrix*. It is non-zero except for the diagonals, which are labeled with σ_i:

$$\mathbf{\Sigma} = \begin{pmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \cdots & \\ & & & \sigma_n \end{pmatrix}$$

There are several interesting facts associated with the SVD of a matrix. First, the SVD is well-defined for any matrix **A** of size m x n, even for $m \neq n$. In physical space, if a matrix **A** is applied to a unit hypercircle in n dimensions, it deforms it into a hyperellipse. The diameters of the new hyperellipse are the singular values σ_i. An example is shown in Figure 12.4.

The singular values σ_i also have a close relation to its eigenvalues λ_i. The following table enumerates some of these relations:

Matrix Type	**Relationship**
Symmetric Positive Definite	$\sigma_i = \lambda_i$
Symmetric	$\sigma_i = \lvert\lambda_i\rvert$
General Case	$\sigma_i^2 = i^{th}$ eigenvalue of $\mathbf{A^TA}$

These relationships often make it much more useful as well as more efficient to utilize the singular value decomposition of a matrix rather than computing $\mathbf{A^TA}$, which is an intensive operation.

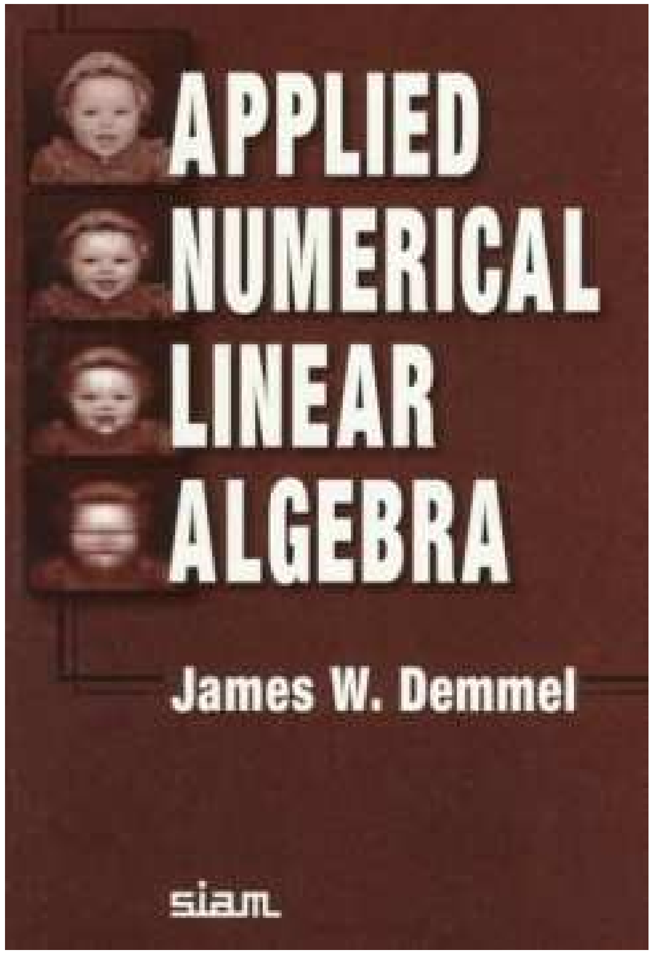


Figure 12.5: The approximation of an image using its SVD.

The SVD may also be used to *approximate* a matrix $\mathbf{A}$ with n singular values:

$$\begin{aligned} \mathbf{A} &= \sum_{i=1}^{n} \sigma_i u_i v_i^T \\ &\approx \sum_{i=1}^{p} \sigma_i u_i v_i^T, p < n \end{aligned}$$

where u_i is the i^{th} row of $\mathbf{U}$ and v_i^T is the i^{th} column of $\mathbf{V}$. This is is also known as the "rank-p approximation of $\mathbf{A}$ in the 2-norm or F-norm."

This approximation has an interesting application for image compression. By taking an image as a matrix of pixel values, we may find its SVD. The rank-p approximation of the image is a compression of the image. For example, James Demmel approximated an image of his daugher for the cover of his book *Applied Numerical Linear Algebra*, shown in Figure 12.5. Note that successive approximations create horizontal and vertical "streaks" in the image.

The following MATLAB code will load an image of a clown and display its rank-p approximation:

```
>> load clown;
```

```
>> image(X);
>> colormap(map);
>> [U,S,V] = svd(X);
>> p=1; image(U(:,1:p)*S(1:p,1:p)*V(:,1:p)');
```

The SVD may also be used to perform *latent semantic indexing*, or clustering of documents based on the words they contain. We build a matrix $\mathbf{A}$ which indexes the documents along one axis and the words along the other. $A_{ij} = 1$ if word j appears in document i, and 0 otherwise. By taking the SVD of $\mathbf{A}$, we can use the singular vectors to represent the "best" subset of documents for each cluster.

Finally, the SVD has an interesting application when using the FFT matrix for parallel computations. Taking the SVD of one-half of the FFT matrix results in singular values that are approximately one-half zeros. Similarly, taking the SVD of one-quarter of the FFT matrix results in singular values that are approximately one-quarter zeros. One can see this phenomenon with the following MATLAB code:

```
>> f = fft(eye(100));
>> g = f(1:50,51:100);
>> plot(svd(g),'*');
```

These near-zero values provide an opportunity for compression when communicating parts of the FFT matrix across processors [30].

Bibliography

[1] N. Alon, P. Seymour, and R. Thomas. A separator theorem for non-planar graphs. In *Proceedings of the 22th Annual ACM Symposium on Theory of Computing*, Maryland, May 1990. ACM.

[2] C. R. Anderson. An implementation of the fast multipole method without multipoles. *SIAM J. Sci. Stat. Comp.*, 13(4):932–947, July 1992.

[3] A. W. Appel. An efficient program for many-body simulation. *SIAM J. Sci. Stat. Comput.*, 6(1):85–103, 1985.

[4] I. Babuška and A.K. Aziz. On the angle condition in the finite element method. *SIAM J. Numer. Anal.*, 13(2):214–226, 1976.

[5] J. Barnes and P. Hut. A hierarchical $O(n \log n)$ force calculation algorithm *Nature*, 324 (1986) pp446-449.

[6] M. Bern, D. Eppstein, and J. R. Gilbert. Provably good mesh generation. *J. Comp. Sys. Sci.* 48 (1994) 384–409.

[7] M. Bern and D. Eppstein. Mesh generation and optimal triangulation. In *Computing in Euclidean Geometry*, D.-Z. Du and F.K. Hwang, eds. World Scientific (1992) 23–90.

[8] M. Bern, D. Eppstein, and S.-H. Teng. Parallel construction of quadtrees and quality triangulations. In *Workshop on Algorithms and Data Structures, Springer LNCS 709*, pages 188–199, 1993.

[9] G. Birkhoff and A. George. Elimination by nested dissection. *Complexity of Sequential and Parallel Numerical Algorithms*, J. F. Traub, Academic Press, 1973.

[10] P. E. Bjørstad and O. B. Widlund. Iterative methods for the solution of elliptic problems on regions partitioned into substructures. *SIAM J. Numer. Anal.*, 23:1097-1120, 1986.

[11] G. E. Blelloch. *Vector Models for Data-Parallel Computing.* MIT-Press, Cambridge MA, 1990.

[12] J. A. Board, Z. S. Hakura, W. D. Elliott, and W. T. Ranklin. Saclable variants of multipole-based algorithms for molecular dynamic applications. In *Parallel Processing for Scientific Computing*, pages 295–300. SIAM, 1995.

[13] R. Bryant, *Bit-level analysis of an SRT circuit,* preprint, CMU (See `http://www.cs.cmu.edu:8001/afs/cs.cmu.edu/user/bryant/www/home.html`)

[14] C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167, 1998.

[15] V. Carpenter, compiler, `http://vinny.csd.my.edu/pentium.html`.

[16] T. F. Chan and D. C. Resasco. A framework for the analysis and construction of domain decomposition preconditioners. UCLA-CAM-87-09, 1987.

[17] L. P. Chew. *Guaranteed-quality triangular meshes.* TR-89-983, Cornell, 1989.

[18] P. G. Ciarlet. *The Finite Element Method for Elliptic Problems.* North–Holland, 1978.

[19] K. Clarkson, D. Eppstein, G. L. Miller, C. Sturtivant, and S.-H. Teng. Approximating center points with and without linear programming. In *Proceedings of 9th ACM Symposium on Computational Geometry*, pages 91–98, 1993.

[20] T. Coe, Inside the Pentium FDIV bug, *Dr. Dobb's Journal 20* (April, 1995), pp 129–135.

[21] T. Coe, T. Mathisen, C. Moler, and V. Pratt, Computational aspects of the Pentium affair, *IEEE Computational Science and Engineering 2* (Spring 1995), pp 18–31.

[22] T. Coe and P. T. P. Tang, *It takes six ones to reach a flaw,* preprint.

[23] J. Conroy, S. Kratzer, and R. Lucas, Data parallel sparse LU factorization, in *Parallel Processing for Scientific Computing*, SIAM, Philadelphia, 1994.

[24] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1992.

[25] L. Danzer, J. Fonlupt, and V. Klee. Helly's theorem and its relatives. *Proceedings of Symposia in Pure Mathematics, American Mathematical Society*, 7:101–180, 1963.

[26] J. Dongarra, R van de Geijn, and D. Walker, A look at scalable dense linear algebra libraries, in *Scalable High Performance Computer Conference*, Williamsburg, VA, 1992.

[27] I. S. Duff, R. G. Grimes, and J. G. Lewis, Sparse matrix test problems, *ACM TOMS*, 15 (1989), pp. 1-14.

[28] A. L. Dulmage and N. S. Mendelsohn. Coverings of bipartite graphs. *Canadian J. Math.* 10, pp 517-534, 1958.

[29] I. S. Duff. Parallel implementation of multifrontal schemes. *Parallel Computing*, 3, 193–204, 1986.

[30] A. Edelman, P. McCorquodale, and S. Toledo. The future fast fourier transform. *SIAM Journal on Scientific Computing*, 20(3):1094–1114, 1999.

[31] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical CS.* Springer-Verlag, 1987.

[32] D. Eppstein, G. L. Miller, and S.-H. Teng. A deterministic linear time algorithm for geometric separators and its applications. In *Proceedings of 9th ACM Symposium on Computational Geometry*, pages 99–108, 1993.

[33] C. Farhat and M. Lesoinne. Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics. Int. J. Num. Meth. Eng. 36:745-764 (1993).

[34] J. Fixx, *Games for the Superintelligent.*

[35] I. Fried. Condition of finite element matrices generated from nonuniform meshes. *AIAA J.* 10, pp 219–221, 1972.

[36] M. Garey and M. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[37] J. A. George. Nested dissection of a regular finite element mesh. *SIAM J. Numerical Analysis*, 10: 345–363, 1973.

[38] J. A. George and J. W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, 1981.

[39] A. George, J. W. H. Liu, and E. Ng, Communication results for parallel sparse Cholesky factorization on a hypercube, *Parallel Comput.* 10 (1989), pp. 287–298.

[40] A. George, M. T. Heath, J. Liu, E. Ng. Sparse Cholesky factorization on a local-memory multiprocessor. *SIAM J. on Scientific and Statistical Computing*, 9, 327–340, 1988

[41] J. R. Gilbert, G. L. Miller, and S.-H. Teng. Geometric mesh partitioning: Implementation and experiments. In *SIAM J. Sci. Comp.*, to appear 1995.

[42] G. Golub and W. Kahan. Calculating the singular values and pseudoinverse of a matrix. *SIAM Journal on Numerical Analysis*, 2:205–224, 1965.

[43] G. H. Golub and C. F. Van Loan. *Matrix Computations, 2nd Edition.* Johns Hopkins University Press, 1989.

[44] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comp. Phys.* 73 (1987) pp325-348.

[45] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7:381–413, 1992.

[46] R. W. Hackney and J. W. Eastwood. *Computer Simulation Using Particles.* McGraw Hill, 1981.

[47] G. Hardy, J. E. Littlewood and G. Pólya. *Inequalities.* Second edition, Cambridge University Press, 1952.

[48] D. Haussler and E. Welzl. ϵ-net and simplex range queries. *Discrete and Computational Geometry*, 2: 127–151, 1987.

[49] N.J. Higham, The Accuracy of Floating Point Summation *SIAM J. Scient. Comput.* , 14:783–799, 1993.

[50] Y. Hu and S. L. Johnsson. A data parallel implementation of hierarchical N-body methods. Technical Report TR-26-94, Harvard University, 1994.

[51] T. Joachims. Text categorization with support vector machines: learning with many relevant features. In Claire Nédellec and Céline Rouveirol, editors, *Proceedings of ECML-98, 10th European Conference on Machine Learning*, number 1398, pages 137–142, Chemnitz, DE, 1998. Springer Verlag, Heidelberg, DE.

[52] M. T. Jones and P. E. Plassman. Parallel algorithms for the adaptive refinement and partitioning of unstructured meshes. *Proc. Scalable High-Performance Computing Conf.* (1994) 478–485.

[53] W. Kahan, *A Test for SRT Division,* preprint.

[54] F. T. Leighton. *Complexity Issues in VLSI.* Foundations of Computing. MIT Press, Cambridge, MA, 1983.

[55] F. T. Leighton and S. Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In *29th Annual Symposium on Foundations of Computer Science*, pp 422-431, 1988.

[56] C. E. Leiserson. *Area Efficient VLSI Computation.* Foundations of Computing. MIT Press, Cambridge, MA, 1983.

[57] C. E. Leiserson and J. G. Lewis. Orderings for parallel sparse symmetric factorization. in *3rd SIAM Conference on Parallel Processing for Scientific Computing*, 1987.

[58] G. Y. Li and T. F. Coleman, A parallel triangular solver for a distributed memory multiproces SOR, *SIAM J. Scient. Stat. Comput.* 9 (1988), pp. 485–502.

[59] R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM J. on Numerical Analysis*, 16:346–358, 1979.

[60] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM J. of Appl. Math.*, 36:177–189, April 1979.

[61] J. W. H. Liu. The solution of mesh equations on a parallel computer. in 2nd Langley Conference on Scientific Computing, 1974.

[62] P.-F. Liu. *The parallel implementation of N-body algorithms.* PhD thesis, Yale University, 1994.

[63] R. Lohner, J. Camberos, and M. Merriam. Parallel unstructured grid generation. *Computer Methods in Applied Mechanics and Engineering* 95 (1992) 343–357.

[64] J. Makino and M. Taiji, T. Ebisuzaki, and D. Sugimoto. Grape-4: a special-purpose computer for gravitational N-body problems. In *Parallel Processing for Scientific Computing*, pages 355–360. SIAM, 1995.

[65] J. Matoušek. Approximations and optimal geometric divide-and-conquer. In *23rd ACM Symp. Theory of Computing*, pages 512–522. ACM, 1991.

[66] G. L. Miller. Finding small simple cycle separators for 2-connected planar graphs. *Journal of Computer and System Sciences*, 32(3):265–279, June 1986.

[67] G. L. Miller, S.-H. Teng, W. Thurston, and S. A. Vavasis. Automatic mesh partitioning. In A. George, J. Gilbert, and J. Liu, editors, *Sparse Matrix Computations: Graph Theory Issues and Algorithms*, IMA Volumes in Mathematics and its Applications. Springer-Verlag, pp57–84, 1993.

[68] G. L. Miller, S.-H. Teng, W. Thurston, and S. A. Vavasis. Finite element meshes and geometric separators. *SIAM J. Scientific Computing*, to appear, 1995.

[69] G. L. Miller, D. Talmor, S.-H. Teng, and N. Walkington. A Delaunay Based Numerical Method for Three Dimensions: generation, formulation, partition. In *the proceedings of the twenty-sixth annual ACM symposium on the theory of computing*, to appear, 1995.

[70] S. A. Mitchell and S. A. Vavasis. Quality mesh generation in three dimensions. *Proc. 8th ACM Symp. Comput. Geom.* (1992) 212–221.

[71] K. Nabors and J. White. A multipole accelerated 3-D capacitance extraction program. *IEEE Trans. Comp. Des.* 10 (1991) v11.

[72] D. P. O'Leary and G. W. Stewart, Data-flow algorithms for parallel matrix computations, *CACM*, 28 (1985), pp. 840–853.

[73] L.S. Ostrouchov, M.T. Heath, and C.H. Romine, *Modeling speedup in parallel sparse matrix factorization,* Tech Report ORNL/TM-11786, Mathematical Sciences Section, Oak Ridge National Lab., December, 1990.

[74] V. Pan and J. Reif. Efficient parallel solution of linear systems. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing*, pages 143–152, Providence, RI, May 1985. ACM.

[75] A. Pothen, H. D. Simon, K.-P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Anal. Appl.* 11 (3), pp 430–452, July, 1990.

[76] Vaughan Pratt, *personal communication*, June, 1995.

[77] V. Pratt, Anatomy of the Pentium Bug, *TAPSOFT'95, LNCS 915*, Springer-Verlag, Aarhus, Denmark, (1995), 97–107. `ftp://boole.stanford.edu/pub/FDIV/anapent.ps.gz`.

[78] F. P. Preparata and M. I. Shamos. *Computational Geometry An Introduction.* Texts and Monographs in Computer Science. Springer-Verlag, 1985.

[79] A. A. G. Requicha. Representations of rigid solids: theory, methods, and systems. In *ACM Computing Survey*, 12, 437–464, 1980.

[80] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–408, 1958.

[81] E. Rothberg and A. Gupta, *The performance impact of data reuse in parallel dense Cholesky factorization,* Stanford Comp. Sci. Dept. Report STAN-CS-92-1401.

[82] E. Rothberg and A. Gupta, An efficient block-oriented approach to parallel sparse Cholesky factorization, *Supercomputing* '93, pp. 503-512, November, 1993.

[83] E. Rothberg and R. Schreiber, Improved load distribution in parallel sparse Cholesky factorization, *Supercomputing* '94, November, 1994.

[84] J. Ruppert. A new and simple algorithm for quality 2-dimensional mesh generation. *Proc. 4th ACM-SIAM Symp. Discrete Algorithms* (1993) 83–92.

[85] Y. Saad and M.H. Schultz, Data communication in parallel architectures, *Parallel Comput.* 11 (1989), pp. 131–150.

[86] J. K. Salmon. *Parallel Hierarchical N-body Methods.* PhD thesis, California Institute of Technology, 1990. CRPR-90-14.

[87] J. K. Salmon, M. S. Warren, and G. S. Winckelmans. Fast parallel tree codes fro gravitational and fluid dynamical N-body problems. *Int. J. Supercomputer Applications*, 8(2):129–142, 1994.

[88] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, pages 188–260, 1984.

[89] K. E. Schmidt and M. A. Lee. Implementing the fast multipole method in three dimensions. *J. Stat. Phy.*, page 63, 1991.

[90] H.P. Sharangpani and M.L. Barton, *Statistical analysis of floating point flaw in the Pentium* ^{TM} *Processor* (1994). `http://www.intel.com/product/pentium/white11.ps`

[91] H. D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering* 2:(2/3), pp135-148.

[92] H. D. Simon and S.-H. Teng. How good is recursive bisection? *SIAM J. Scientific Computing*, to appear, 1995.

[93] J. P. Singh, C. Holt, T. Ttsuka, A. Gupta, and J. L. Hennessey. Load balancing and data locality in hierarchical N-body methods. Technical Report CSL-TR-92-505, Stanford, 1992.

[94] G. W. Stewart. On the early history of the singular value decomposition. Technical Report CS-TR-2855, 1992.

[95] G. Strang and G. J. Fix. *An Analysis of the Finite Element Method.* Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

[96] S.-H. Teng. *Points, Spheres, and Separators: a unified geometric approach to graph partitioning.* PhD thesis, Carnegie-Mellon University, School of Computer Science, 1991. CMU-CS-91-184.

[97] V. Vapnik. Estimation of dependencies based on empirical data [in Russian]. 1979.

[98] V. Vapnik. *The Nature of Statistical Learning Theory.* Springer, New York, 1995.

[99] V. Vapnik. *Statistical Learning Theory.* Wiley, 1998.

[100] V. N. Vapnik and A. Ya. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory Probab. Appl.*, 16: 264-280, 1971.

[101] R. D. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. Concurrency, 3 (1991) 457

[102] F. Zhao. An O(n) algorithm for three-dimensional n-body simulation. Technical Report TR AI Memo 995, MIT, AI Lab., October 1987.

[103] F. Zhao and S. L. Johnsson. The parallel multipole method on the Connection Machines. *SIAM J. Stat. Sci. Comp.*, 12:1420–1437, 1991.