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

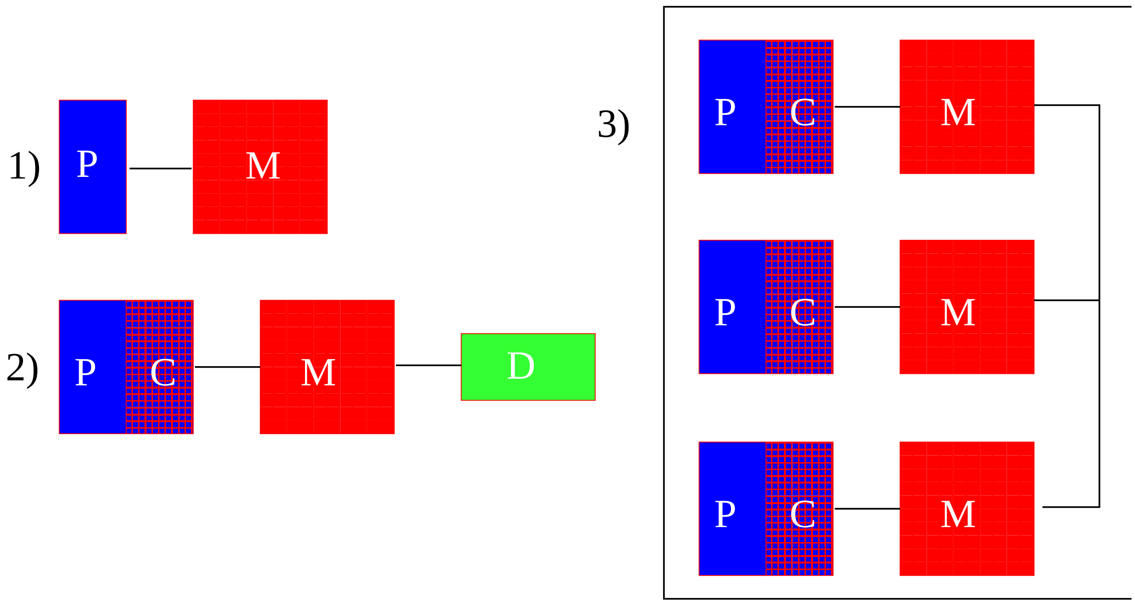

Figure 6.1: 1) A computer 2) A fancier computer 3)A multicomputer

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

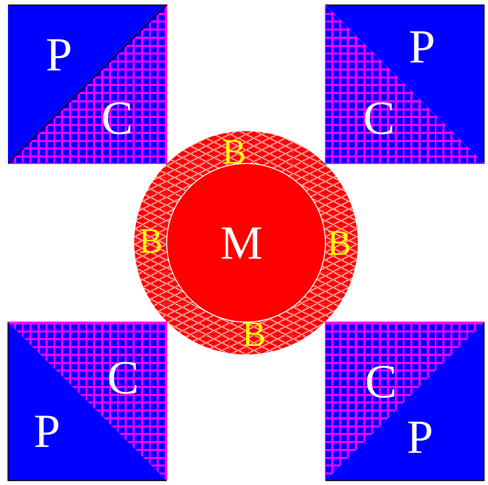

Figure 6.2: A four processor SMP (B denotes the bus between the central memory and the processor's cache

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.