Lecture Notes for Theory of Parallel Systems

Professor Charles Leiserson   Dr. Bradley Kuzmaul

Professor Michael Bender   Professor Hsu-Wen Jing

TA: Seth Gilbert

Fall, 2003

Contents

Lecture 1 – Introduction & Dynamic Multithreading **1-7**

1.1 Course Overview . . . 1-7
1.2 Introduction to Parallel Programs . . . 1-8
1.3 Modeling and Measuring Parallel Programs . . . 1-8
1.3.1 Instructions Streams as a DAG . . . 1-8
1.3.2 Performance Measures . . . 1-8
1.4 The Greedy Scheduler . . . 1-10
1.5 Anecdote . . . 1-12

Lecture 2 – Cilk, Matrix Multiplication, and Sorting **2-13**

2.1 Parallel Processing With Cilk . . . 2-13
2.1.1 Cilk . . . 2-13
2.1.2 Parallel-Execution Model . . . 2-14
2.1.3 Cactus Stack . . . 2-15
2.1.4 Race Conditions . . . 2-16
2.2 Matrix Multiplication and Merge Sort . . . 2-17
2.2.1 Matrix Multiplication . . . 2-17
2.2.2 Sorting . . . 2-19

Lecture 3 – High Performance Serial Programming and Caching **3-22**

3.1 High Performance Serial Programming . . . 3-22
3.1.1 Computer Hardware . . . 3-22
3.1.2 Compilers . . . 3-24
3.1.3 High-level optimizations . . . 3-24
3.1.4 Example: isort vs. sort5 [14] . . . 3-26
3.2 A Theoretic Machine Model . . . 3-26
3.2.1 Motivation of Model . . . 3-26
3.2.2 Memory Heirarchies . . . 3-27
3.2.3 Block Replacement Strategies . . . 3-28
3.2.4 Cache-Aware v. Cache-Oblivious . . . 3-28
3.3 Analysis of Several Algorithms . . . 3-29
3.3.1 Method of Asymptotic Analysis . . . 3-29
3.3.2 Example 1: Array-scanning . . . 3-29
3.3.3 Example 2: Binary Search . . . 3-29
3.3.4 Example 3: Recursive Matrix Multiplication . . . 3-30

Lecture 4 – Debugging Multithreaded Programs **4-33**
4.1 Determinacy Races . . . 4-33
4.1.1 A Cilk Program with a Determinacy Race . . . 4-33
4.1.2 Determinacy Races in Cilk . . . 4-34
4.1.3 Example: the N-Queens Puzzle . . . 4-35
4.2 Nondeterminator Background/Theory . . . 4-36
4.2.1 Definition and Performance . . . 4-37
4.2.2 Series-Parallel (SP) Dags . . . 4-37
4.2.3 Series-Parallel (SP) Parse Trees . . . 4-39
4.3 Execution of the Nondeterminator (SP-Bags Algorithm) . . . 4-40
4.3.1 Overview . . . 4-40
4.3.2 Properties of the Nondeterminator . . . 4-41
4.3.3 Disjoint-Set Data Structures . . . 4-42
4.3.4 SP-Bags Algorithm . . . 4-42
4.4 Nondeterminator Correctness . . . 4-43
4.4.1 Justification of Set Manipulation in SP-Bags Algorithm . . . 4-43
4.4.2 Some Lemmas . . . 4-44
4.4.3 Proof of SP-Bags Race Detection . . . 4-45

Lecture 5 – Consistency of the Memory Sub-System **5-48**
5.1 Mutual Exclusion . . . 5-48
5.2 Mutual Exclusion Algorithms . . . 5-50
5.2.1 Dekker's Algorithm . . . 5-50
5.2.2 Dijkstra's Algorithm . . . 5-50
5.3 Proof of Dijkstra's Algorithm . . . 5-52
5.4 Memory Consistency . . . 5-53
5.5 Memory Subsystem of Hardware Architectures . . . 5-54
5.5.1 Additional Memory Primitives . . . 5-54

Lecture 6 – Analyzing Space Bounds **6-56**
6.1 Bounds on space requirements . . . 6-56
6.2 Memory Allocators . . . 6-60
6.3 Project Ideas . . . 6-62

Lecture 7 – Memory Contention **7-63**
7.1 Introduction to Memory Contention . . . 7-63
7.1.1 Contention Delay . . . 7-63
7.2 The $(\mathbf{P}, \mathbf{M}, \mathbf{T})$ Recycling Game . . . 7-64
7.3 Bounds for the Expected Total Delay . . . 7-64

Lecture 8 – Cilk Scheduler **8-68**
8.1 "Ideal" Scheduling . . . 8-68
8.2 Cilk Computation . . . 8-69
8.3 Main Data Structure . . . 8-69
8.4 Cilk Scheduler . . . 8-70
8.5 Work Stealing . . . 8-72
8.6 Ready Deque Structural Lemma . . . 8-72

Lecture 9 – Analysis of Cilk Scheduler **9-75**
9.1 The *Cilk* Scheduler . . . 9-75
9.2 Location of shallowest thread . . . 9-76
9.3 Critical threads . . . 9-79
9.4 Execution Time Bounds For The *Cilk* Scheduler . . . 9-80
9.5 Performance Analysis of Scheduling Algorithm Using Delay Sequences . . . 9-82
9.5.1 Rounds . . . 9-82
9.5.2 Delay Sequence Argument . . . 9-83
9.6 Wrap-Up . . . 9-86
9.6.1 Death-Bed Formulae . . . 9-86

Lecture 10 – Cilk Implementation **10-88**
10.1 Compiling Cilk . . . 10-88
10.2 Deque Protocols . . . 10-92
10.2.1 TH Protocol . . . 10-93
10.2.2 THE Protocol . . . 10-95

Lecture 11 – Project Proposals I **11-97**

Lecture 12 – Project Proposals II **12-98**

Lecture 13 – Implementation of Memory Consistency **13-99**
13.1 Memory Consistency . . . 13-99
13.1.1 Sequential Consistency . . . 13-99
13.1.2 System Model . . . 13-99
13.1.3 Uncached Consistency Algorithm . . . 13-100
13.1.4 Proving Sequential Consistency . . . 13-100
13.2 More Complicated Memory . . . 13-101
13.2.1 Overlapping Operations . . . 13-102
13.2.2 Caching . . . 13-103
13.2.3 MESI . . . 13-103

Lecture 14 – Competitive Snoopy Caching **14-106**
14.1 Setup for snoopy caching algorithms . . . 14-106
14.1.1 Why do caches work ? . . . 14-107
14.2 Two strategies to ensure correctness while updating replicated blocks . . . 14-107
14.2.1 An introduction to the Pack Rat and Exclusive Write algorithms . . . 14-107
14.2.2 Good cases for pack rat and exclusive write . . . 14-108
14.2.3 A bad case for exclusive write . . . 14-108
14.2.4 A bad case for pack rat . . . 14-108
14.2.5 Goodman's algorithm . . . 14-109
14.3 Introduction to Competitive Algorithms . . . 14-109
14.3.1 An example of a competitive algorithm (Due to Larry Rudolph) . . . 14-109
14.4 Competitive Snoopy Caching . . . 14-109
14.4.1 Direct-mapped snoopy caching algorithm (*dsc*) . . . 14-110

Lecture 15 – Snoopy caching and the Spin-Block Problem **15-113**
15.1 Strategy for the proof (Snoopy caching) . . . 15-113
15.1.1 Strategy . . . 15-114
15.1.2 What is this potential function ? . . . 15-114
15.1.3 Proof by induction of the following statement . . . 15-114
15.2 Directory based caches . . . 15-117

15.3 Spin-Block Problem 15-118
15.4 Continuous ski rental problem 15-118
15.4.1 Spin 15-118

Lecture 16 – Interconnection Networks **16-120**
16.1 Basic Interconnection Networks 16-120
16.1.1 “Ideal” Parallel Computer and Definitions 16-120
16.2 Hypercubes 16-122
16.2.1 Hypercubes explained 16-122
16.2.2 Routing on a Hypercubes 16-122
16.2.3 Cube-Connected Cycles 16-123
16.3 Butterfly (FFT) Networks 16-123
16.3.1 Construction 16-123
16.3.2 Routing and Congestion 16-123

Lecture 17 – Butterfly Network Analysis and The Beneš Network **17-125**
17.1 Network with N Nodes 17-125
17.2 The Beneš Network 17-126
17.2.1 Worst Case Problems for the Butterfly Network 17-126
17.3 The Beneš Network 17-126
17.4 Routing on Butterfly Networks 17-128

Lecture 18 – Hypercube Networks-III **18-132**
18.1 Review of the previous lecture 18-132
18.2 Valiant's Algorithm 18-132
18.3 Ranade's Algorithm 18-133
18.4 Shuffle-exchange graph 18-133
18.5 DeBruijn Network 18-137
18.6 Deriving the Bisection Width for the Shuffle-Exchange Graph 18-138

Lecture 19 – Squish Routing and Disk-drive Hardware **19-141**
19.1 Squish Routing 19-141
19.2 Squish theorem 19-142
19.3 An application of squish routing 19-144
19.4 Implementing squish routing 19-146
19.5 Analysis of Scan and Routing 19-147
19.6 Disk Drives - Hardware 19-149

Lecture 20 – Permuting Data on Parallel Disks **20-150**
20.1 Parallel Disk Systems 20-150
20.2 Problem of permuting Disk Blocks 20-152
20.3 Conflict graph 20-153
20.4 **Hall's theorem -(General bi-partite graph)** 20-155

Lecture 21 – Sorting and Permuting on Sequential and Parallel Disks **21-157**
21.1 DAM Model 21-157
21.2 Mergesort Analysis 21-158
21.2.1 Normal Mergesort 21-158
21.2.2 Parallel Mergesort 21-159
21.3 Lower Bound for Sorting 21-161
21.4 Lower Bound for External Permutations 21-164
21.5 Cache-Oblivious Sorting 21-167

Lecture 22 – Pick a Winner **22-169**
22.1 The Pick-a-Winner game . . . 22-169
22.2 Strategies to maximize the profit . . . 22-170
22.3 Pick-a-Winner Game - with Variations . . . 22-172
22.4 Results in Summary . . . 22-173
22.5 Applications . . . 22-173
22.5.1 Scheduling a single job on NOWs . . . 22-173
22.5.2 Scheduling multiple Jobs on NOWs . . . 22-174

6.895 Theory of Parallel Systems	Lecture 1
Introduction & Dynamic Multithreading	
Lecturer: Charles Leiserson	*Scribe: Abhi Shelat and Ben Adida*

Lecture Summary

1. *Course Overview*
   We present an overview of the course topics.
2. *Introduction to Parallel Programs*
   We define basic terminology and present our first parallel program.
3. *Modeling and Measuring Parallel Programs*
   We define the graph model for parallel programming and introduce a methodology for measuring the performance of parallel progams.
4. *Greedy Scheduler*
   We prove that a simple greedy scheduler executes in no worse than twice the theoretically optimal schedule.
5. *An Anecode*
   We review the results of the ⋆Socrates to show how empirical performance analysis does not suffice in understanding parallel computation.

1.1 Course Overview

This course **6.895** will focus on theoretical underpinnings of parallel computation, though it is clear that theory doesn't explain everything. This year, the course will address topics in a top-down fashion so as to help students pick final projects early on:

1. High-level models
   (a) multithreading
   (b) scheduling and load-balancing
   (c) data-race detection
   (d) synchronization (locking and lock-free algorithms)
2. Network architecture
   (a) memory consistency
   (b) routing networks (cf. RAW project, configurable networks)
3. VLSI theory
   (a) graph layouts
   (b) area/volume
   (c) AT^2 complexity
   (d) arithmetic circuits
   (e) universality (routing networks and VLSI's version of the Universal Turing Machine)

1.2 Introduction to Parallel Programs

Consider the following program to compute the nth Fibonacci number. This program is particularly bad: it runs in exponential time where a straight-forward iterative method yields linear time and a clever approach based on repeated squarings of a particular matrix yields $O(\log(n))$ time [see Cormen, Leiserson, Rivest page 902]. Nonetheless, this program is a good example for introducing our model of parallel computation.

```
fib(n)
    if n < 2
    then return n

    x <- spawn fib(n-1)
    y <- spawn fib(n-2)

    sync
    return (x+y)
```

A `spawn` instruction allows the next instruction of the parent program to be executed at the same time as the child, which in Unix programming, is equivalent to a `fork`. A `sync` instruction requires that all locally spawned children finish their work before proceeding with the computation. In Unix programming, a `sync` is equivalent to a `join`.

Our syntax for the presented program describes ***logical parallelism***. We do not explicitly state which tasks are assigned to which processors. Instead, we focus on the computations which can be safely performed in parallel. It is the ***scheduler***'s job to determine how to map dynamically unfolding execution onto a set of processors.

1.3 Modeling and Measuring Parallel Programs

1.3.1 Instructions Streams as a DAG

We model the instruction stream of a program as a **computation directed acyclic graph (DAG)**. Figure 1 shows the ***computation DAG*** for `fib(4)`. Vertices are ***threads*** which is a maximal sequence of instructions not containing parallel control (`spawn`, `return` from a spawned procedure, or `sync`).

We have three threads:

- Thread A: The `if` statement up to the first `spawn`.
- Thread B: The computation of `n-2` before the second `spawn`.
- Thread C: The `x+y` operation before the `return`.

1.3.2 Performance Measures

We define several performance measures to evaluate various scheduler algorithms. We assume an ideal situation: no cache issues, no interprocessor communication costs. These results may not hold under different assumptions, but we are after the essence of understanding.

- T_P is the minimum running time on P processors
- T_1, also called ***work***, is the minimum running time on 1 processor
- T_∞ is the minimum running time on an infinite number of processors. This is equivalent to the longest path in the DAG and is referred to as the ***critical path length***.

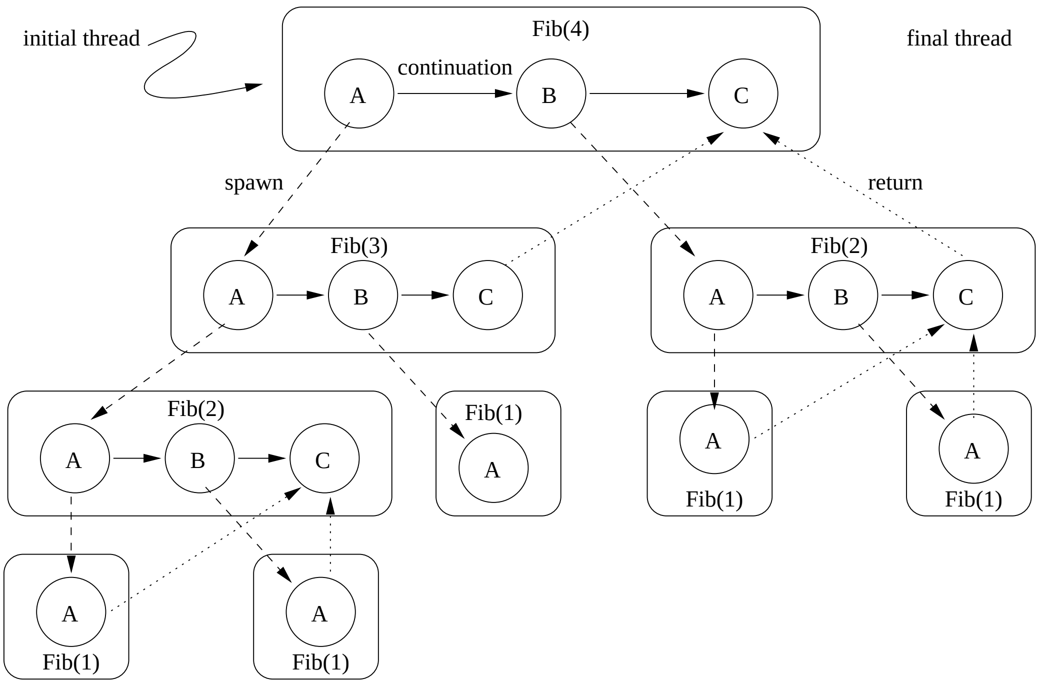


Figure 1.1: The instruction stream for `fib(4)`. Each circle represents a thread. Each solid arrow represents a `continuation`, each dashed arrow represents a `spawn` operation, and each dotted arrow represents a `return` from a spawned procedure.

In our example, if all threads run in unit time, we have $T_1 = 17$. In practice, the threads would be weighted by their individual runtimes. The ***critical path length*** is $T_\infty = 8$.

These measures immediately define lower bounds:

$$T_P \geq T_1/P \tag{1.1}$$

Proof Idea Assume by contradiction that $T_P < T_1/P$. In the best case, P processors can do at most P work in 1 step. By multiplying both sides by P, then $PT_P < T_1$, which implies that the total number of operations performed (left-hand side) is less than T_1, a contradiction. □

$$T_P \geq T_\infty \tag{1.2}$$

Proof Idea Once again, suppose not. If $T_P < T_\infty$, simply using P of the unlimited number of processors available reduces T_∞, which is a contradiction. □

$$T_1/T_P = \textbf{speedup} \tag{1.3}$$

Proof Idea With P processors, the maximum speedup is P. (Remember: the theory only applies to this simplified model.) □

A program's parallel execution plan can have:

- linear speedup: $T_1/T_P = \Theta(P)$
- superlinear speedup: $T_1/T_P = \omega(P)$ (not possible in this model, though it is possible in others).
- sublinear speedup: $T_1/T_P = o(p)$

The maximum possible speedup for a program given particular T_1 and T_∞ is T_1/T_∞, also called ***parallelism***. The ***parallelism*** is the average amount of work that can be done in parallel for every step along the critical path.

1.4 The Greedy Scheduler

A scheduler's job is to map a computation to particular processors. In reality, a scheduler typically must make decisions about what tasks or threads to perform on particular processors without the benefit of foresight, i.e, it does not know what it must schedule in the future. In general, these online schedulers are quite complex and will be discussed later in the term. Therefore, to begin our analysis, we focus on offline scheduling. Specifically, we consider the ***G***reedy Scheduler, a scheduler which attempts to do as much work as possible at every step.

In any schedule, there are two types of steps:

- **complete step**: There are at least P threads that are ready to run. The greedy scheduler selects any P of them and runs them.
- **incomplete step**: There are strictly less than P threads that are ready to run. The greedy scheduler runs them all.

Our first important theorem of the course is a beautiful argument, which extolls the virtue of greedy scheduling.

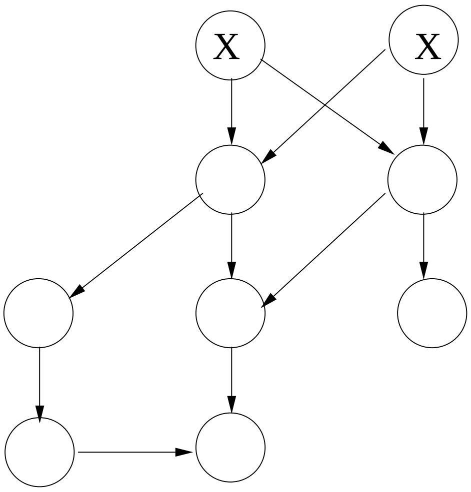


Figure 1.2: The leftover graph G' to execute during an incomplete step. Each nodes represents threads. The vertices represent continuations from one thread to another (due to parallel control such as `spawn` or `continue`). The nodes marked with an 'X' are threads already executed.

Theorem 1.1 *(Graham, Brent) Given P processors, a greedy scheduler executes any computation G with work T_1 and critical path length T_∞ in time:*

$$T_p \quad \leq \quad T_1/p + T_\infty \tag{1.4}$$

Proof In any schedule, there are only two types of steps: complete, and incomplete.

We first consider the complete steps. There can be no more than T_1/p complete steps; otherwise, work $> T_1$, which contradicts the definition of T_1.

We now consider the incomplete steps. Let G' be the subgraph of G that remains to be executed immediately prior to a given incomplete step, as shown in Figure 2. We note that all threads with in-degree 0 in G' are ready to be executed. Since we are dealing solely with incomplete steps, we also know that all of these threads can be immediately executed. The critical path of G' starts with a thread of in-degree 0. Thus, every incomplete step reduces the critical-path length by one unit. In other words, there can be no more than T_∞ incomplete steps.

Thus, the total time for executing incomplete and complete steps in a greedy scheduler is at most $T_1/P + T_\infty$. □

Corollary 1.2 *A greedy scheduler is always within a factor of 2 of optimal*

Proof Given the lower bounds 1.1 and 1.2, we can express a more concise lower bound as:

$$T_P \geq \max(T_1/P, T_\infty) \tag{1.5}$$

In addition, we can trivially express:

$$T_1/P \leq \max(T_1/P, T_\infty) \tag{1.6}$$

$$T_\infty \leq \max(T_1/P, T_\infty) \tag{1.7}$$

Given Theorem 1.1, we conclude

$$
\begin{aligned}
T_P &\leq T_1/P + T_\infty && (1.8)\\
&\leq \max(T_1/P, T_\infty) + \max(T_1/P, T_\infty) && (1.9)\\
&\leq 2\max(T_1/P, T_\infty) && (1.10)
\end{aligned}
$$

which shows that T_P is within twice lower bound 1.5. □

Every step either contributes to progress on the work (complete step), or progress on the critical path (incomplete step). A greedy scheduler is generally a good scheduler.

Corollary 1.3 *The greedy scheduler achieves linear speedup when* $P = O(T_1/T_\infty)$

Proof

$$
\begin{aligned}
T_p &\leq T_1/p + T_\infty && (1.11)\\
&= T_1/p + O(T_1/p) && (1.12)\\
&= \Theta(T_1/p) && (1.13)
\end{aligned}
$$

□

The idea is to operate in the range where T_1/P dominates T_∞. Given a particular problem with fixed T_1 and T_∞, adding more processors increases P and reduces T_1/P while T_∞ remains unaffected. Once T_∞ becomes significant in comparison to T_1/P, the critical path's dependencies reduce the ability to parallelize the computation to additional processors. As long as T_1/P dominates T_∞, all processors can be used efficiently.

1.5 Anecdote

$\star$Socrates, a chess program, had two versions, one including an optimization that significantly improved performance on 32 processors. The performance improvement reduced the total work, but increased the critical-path length:

- $T_{32} = 65sec$, $T_1 = 2048$, $T_\infty = 1$
- $T'_{32} = 40sec$, $T'_1 = 1024$, $T_\infty = 8$

On 512 processors, however, the optimization becomes a problem:

- $T_{512} = T_1/512 + T_\infty = 5$
- $T'_{512} = T'_1/512 + T'_\infty = 10$

The moral of the story: work and critical-path length are more important to understand thean simple empirical performance measurements.

6.895 Theory of Parallel Systems	Lecture 2
Cilk, Matrix Multiplication, and Sorting	
Lecturer: Charles Leiserson	*Scribe: Jason Hickey and Tyeler Quentmeyer*

Lecture Summary

1. *Parallel Processing With Cilk*
   This section provides a brief introduction to the Cilk language and how Cilk schedules and executes parallel processes.
2. *Parallel Matrix Multiplication*
   This section shows how to multiply matrices efficiently in parallel.
3. *Parallel Sorting*
   This section describes a parallel implementation of the merge sort algorithm.

2.1 Parallel Processing With Cilk

We need a systems background to implement and test our parallel systems theories. This section gives an introduction to the Cilk parallel-programming language. It then gives some background on how Cilk schedules parallel processes and examines the role of race conditions in Cilk.

2.1.1 Cilk

This section introduces Cilk. Cilk is a version of C that runs in a parallel-processing environment. It uses the same syntax as C with the addition of the keywords `spawn`, `sync`, and `cilk`. For example, the Fibonacci function written in Cilk looks like this:

```
cilk int fib(int n)
{
   if(n < 2) return n;
   else
    {
        int x, y;
        x = spawn fib(n-1);
        y = spawn fib(n-2);
        sync;
        return(x + y);
    }
}
```

Cilk is a ***faithful*** extension of C, in that if the Cilk keywords are elided from a Cilk program, the result is a C program which implements the Cilk semantics.

A function preceded by `cilk` is defined as a Cilk function. For example,

```
cilk int fib(int n);
```

defines a Cilk function called `fib`. Functions defined without the `cilk` keyword are typical C functions. A function call preceded by the `spawn` keyword tells the Cilk compiler that the function call can be made

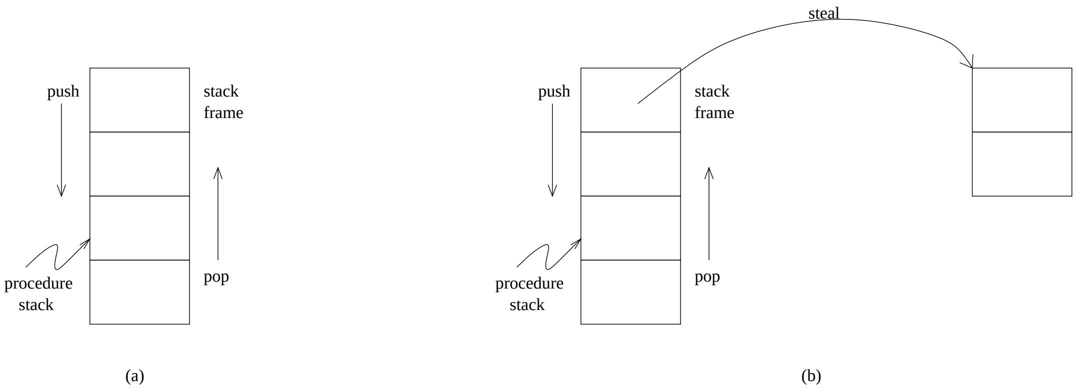


Figure 2.1: Call stack of an executing process. Boxes represent stack frames.

asynchronously in a concurrent thread. The `sync` keyword forces the current thread to wait for asynchronous function calls made from the current context to complete.

Cilk keywords introduce several idiosyncracies into the C syntax. A Cilk function cannot be called with normal C calling conventions – it must be called with `spawn` and waited for with `sync`. The `spawn` keyword can only be applied to a Cilk function. The `spawn` keyword cannot occur within the context of a C function. Refer to the Cilk manual for more details.

2.1.2 Parallel-Execution Model

This section examines how Cilk runs processes in parallel. It introduces the concepts of work-sharing and work-stealing and then outlines Cilk's implementation of the work-stealing algorithm.

Cilk processes are scheduled using an online greedy scheduler. The performance bounds of the online scheduler are close to the optimal offline scheduler. We will look at provable bounds on the performance of the online scheduler later in the term.

Cilk schedules processes using the principle of work-stealing rather than work-sharing. Work-sharing is where a thread is scheduled to run in parallel whenever the runtime makes an asynchronous function call. Work-stealing, in contrast, is where a processor looks around for work whenever it becomes idle.

To better explain how Cilk implements work-stealing, let us first examine the call stack of a vanilla C program running on a single processor. In figure 2.1, the stack grows downward. Each stack frame contains local variables for a function. When a function makes a function call, a new stack frame is pushed onto the stack (added to the bottom) and when a function returns, it's stack frame is popped from the stack. The call stack maintains synchronization between procedures and functions that are called.

In the work-sharing scheme, when a function is spawned, the scheduler runs the spawned thread in parallel with the current thread. This has the benefit of maximizing parallelism. Unfortunately, the cost of setting up new threads is high and should be avoided.

Work-stealing, on the other hand, only branches execution into parallel threads when a processor is idle. This has the benefit of executing with precisely the amount of parallelism that the hardware can take advantage of. It minimizes the number of new threads that must be setup. Work-stealing is the lazy way to put off work for parallel execution until parallelism actually occurs. It has the benefit of running with the same efficiency as a serial program in a uniprocessor environment.

Another way to view the distinction between work-stealing and work-sharing is in terms of how the scheduler walks the computation graph. Work-sharing branches as soon and as often as possible, walking the computation graph with a breadth-first search. Work-stealing only branches when necessary, walking the graph with a depth-first search.

Cilk's implementation of work-stealing avoids running threads that are likely to share variables by scheduling threads to run from the other end of the call stack. When a processor is idle, it chooses a random processor

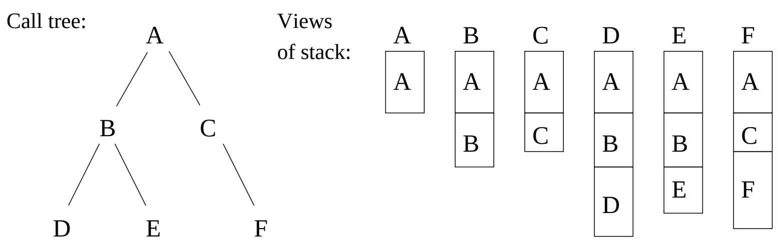


Figure 2.2: Example of a call stack shown as a cactus stack, and the views of the stack as seen by each procedure. Boxes represent stack frames.

and finds the sleeping stack frame that is closest to the base of that processor's stack and executes it. This way, Cilk always parallelizes code execution at the oldest possible code branch.

2.1.3 Cactus Stack

Cilk uses a ***cactus stack*** to implement C's rule for sharing of function-local variables. A cactus stack is a parallel stack implemented as a tree. A push maps to following a branch to a child and a pop maps to returning to the parent in a tree. For example, the cactus tree in figure 2.2 represents the call stack constructed by a call to A in the following code:

```
void A(void)
{
    B();
    C();
}

void B(void)
{
    D();
    E();
}

void C(void)
{
    F();
}

void D(void) {}

void E(void) {}

void F(void) {}
```

Cilk has the same rules for pointers as C. Pointers to local variables can be passed downwards in the call stack. Pointers can be passed upward only if they reference data stored on the heap (allocated with `malloc`). In other words, a stack frame can only see data stored in the current and in previous stack frames. Functions cannot return references to local variables.

The complete call tree is shown in figure 2.2. Each procedure sees a different view of the call stack based on how it is called. For example, B sees a call stack of A followed by B, D sees a call stack of A followed by B followed by D and so on. When procedures are run in parallel by Cilk, the running threads operate on their view of the call stack. The stack maintained by each process is a reference to the actual call stack, not a copy of it. Cilk maintains coherence among call stacks that contain the same frames using methods that we will discuss later.

2.1.4 Race Conditions

The single most prominent reason that parallel computing is not widely deployed today is because of race conditions. Identifying and debugging race conditions in parallel code is hard. Once a race condition has been found, no methodology currents exists to write a regression test to ensure that the bug is not reintroduced during future development. For these reasons, people do not write and deploy parallel code unless they absolutely must. This section examines an example race condition in Cilk.

Consider the following code:

```
cilk int foo(void)
{
    int x = 0;
    spawn bar(&x);
    spawn bar(&x);
    sync;
    return x;
}

cilk void bar(int *p)
{
    *p += 1;
}
```

If this were a serial code, we would expect that foo returns 2. What value is returned by foo in the parallel case? Assume the increment performed by bar is implemented with assembly that looks like this:

```
read x
add
write x
```

Then, the parallel execution looks like the following:

```
bar 1:
read x (1)
add
write x (2)

bar 2:
read x (3)
add
write x (4)
```

where bar 1 and bar 2 run concurrently. On a single processor, the steps are executed (1) (2) (3) (4) and foo returns 2 as expected. In the parallel case, however, the execution could occur in the order (1) (3) (2) (4), in which case foo would return 1. The simple code exhibits a race condition. Cilk has a tool called the Nondeterminator which can be used to help check for race conditions.

2.2 Matrix Multiplication and Merge Sort

In this section we explore multithreaded algorithms for matrix multiplication and array sorting. We also analyze the work and critical path-lengths. From these measures, we can compute the parallelism of the algorithms.

2.2.1 Matrix Multiplication

To multiply two $n \times n$ matrices in parallel, we use a recursive algorithm. This algorithm uses the following formulation, where matrix A multiplies matrix B to produce a matrix C:

$$\begin{aligned}
\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} &= \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \\
&= \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}.
\end{aligned}$$

This formulation expresses an $n \times n$ matrix multiplication as 8 multiplications and 4 additions of $(n/2) \times (n/2)$ submatrices. The multithreaded algorithm MULT performs the above computation when n is a power of 2. MULT uses the subroutine ADD to add two $n \times n$ matrices.

```
MULT(C, A, B, n)
  if n = 1
    then C[1,1] ← A[1,1] · B[1,1]
    else allocate a temporary matrix T[1..n, 1..n]
         partition A, B, C and T into (n/2) × (n/2) submatrices
         spawn MULT(C11, A11, B11, n/2)
         spawn MULT(C12, A11, B12, n/2)
         spawn MULT(C21, A21, B11, n/2)
         spawn MULT(C22, A21, B12, n/2)
         spawn MULT(T11, A12, B21, n/2)
         spawn MULT(T12, A12, B22, n/2)
         spawn MULT(T21, A22, B21, n/2)
         spawn MULT(T22, A22, B22, n/2)
         sync
         spawn ADD(C, T, n)
         sync
```

```
ADD(C, T, n)
  if n = 1
    then C[1,1] ← C[1,1] + T[1,1]
    else partition C and T into (n/2) × (n/2) submatrices
         spawn ADD(C11, T11, n/2)
         spawn ADD(C12, T12, n/2)
         spawn ADD(C21, T21, n/2)
         spawn ADD(C22, T22, n/2)
         sync
```

The analysis of the algorithms in this section requires the use of the Master Theorem. We state the Master Theorem here for convenience.

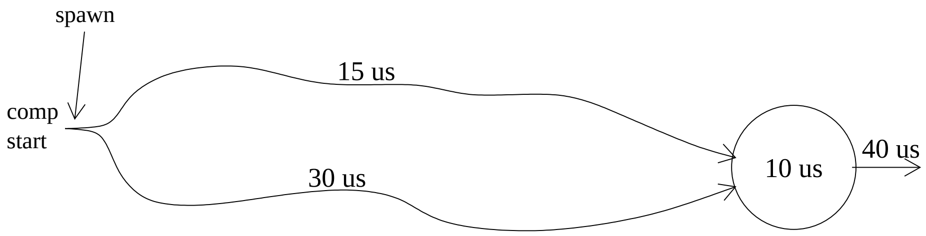


Figure 2.3: Critical path. The squiggles represent two different code paths. The circle is another code path.

Theorem 2.1 (Master Theorem) *Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence*

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows.

1. *If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{log_b a})$.*
2. *If $f(n) = \Theta(n^{log_b a} lg^k n)$ for some constant $k \geq 0$, then $T(n) = \Theta(n^{log_b a} lg^{k+1} n)$.*
3. *If $f(n) = \Omega(n^{log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.*

We begin by analyzing the work for MULT. The work is the running time of the algorithm on one processor, which we compute by solving the recurrence relation for the serial equivalent of the algorithm. We note that the matrix partitioning in MULT and ADD takes O(1) time, as it requires only a constant number of indexing operations. For the subroutine ADD, the work at the top level (denoted $A_1(n)$) then consists of the work of 4 problems of size $n/2$ plus a constant factor, which is expressed by the recurrence

$$\begin{aligned} A_1(n) &= 4A_1(n/2) + \Theta(1) & (2.1)\\ &= \Theta(n^2). & (2.2) \end{aligned}$$

We solve this recurrence by invoking case 1 of the Master Theorem. Similarly, the recurrence for the work of MULT (denoted $M_1(n)$):

$$\begin{aligned} M_1(n) &= 8M_1(n/2) + \Theta(n^2) & (2.3)\\ &= \Theta(n^3). & (2.4) \end{aligned}$$

We also solve this recurrence with case 1 of the Master Theorem. The work is the same as for the traditional triply-nested-loop serial algorithm.

The critical-path length is the maximum path length through a computation, as illustrated by figure 2.3. For ADD, all subproblems have the same critical-path length, and all are executed in parallel. The critical-path length (denoted $A_\infty(n)$) is a constant plus the critical-path length of one subproblem, and is represented by the recurrence (sovled by case 2 of the Master Theorem):

$$\begin{aligned} A_\infty &= A_\infty(n/2) + \Theta(1) & (2.5)\\ &= \Theta(\lg n). & (2.6) \end{aligned}$$

Using this result, the critical-path length for MULT (denoted $M_\infty(n)$) is

$$\begin{aligned} M_\infty &= M_\infty(n/2) + \Theta(\lg n) & (2.7)\\ &= \Theta(\lg^2 n), & (2.8) \end{aligned}$$

by case 2 of the Master Theorem. From the work and critical-path length, we compute the parallelism:

$$M_1(n)/M_\infty(n) = \Theta(n^3/\lg^2 n). \tag{2.9}$$

As an example, if $n = 1000$, the parallelism $\approx 10^7$. In practice, multiprocessor systems don't have more that $\approx$ 64,000 processors, so the algorithm has more than adequate parallelism.

In fact, it is possible to trade parallelism for an algorithm that runs faster in practice. MULT may run slower than an in-place algorithm because of the hierarchical structure of memory. We introduce a new algorithm, MULT-ADD, that trades parallelism in exchange for eliminating the need for the temporary matrix T.

```
MULT-ADD(C, A, B, n)
if n = 1
   then C[1,1] ← C[1,1] + A[1,1] · B[1,1]
   else partition A, B, and C into (n/2) × (n/2) submatrices
           spawn MULT(C11, A11, B11, n/2)
           spawn MULT(C12, A11, B12, n/2)
           spawn MULT(C21, A21, B11, n/2)
           spawn MULT(C22, A21, B12, n/2)
           sync
           spawn MULT(C11, A12, B21, n/2)
           spawn MULT(C12, A12, B22n/2)
           spawn MULT(C21, A22, B21, n/2)
           spawn MULT(C22, A22, B22, n/2)
           sync
```

The work for MULT-ADD (denoted $M_1'(n)$) is the same as the work for MULT, $M_1'(n) = \Theta(n^3)$. Since the algorithm now executes four recursive calls in parallel followed in series by another four recursive calls in parallel, the critical-path length (denoted $M_\infty'(n)$) is

$$M_\infty'(n) = 2M_\infty'(n/2) + \Theta(1) \tag{2.10}$$
$$= \Theta(n) \tag{2.11}$$

by case 1 of the Master Theorem. The parallelism is now

$$M_1'(n)/M_\infty' = \Theta(n^2). \tag{2.12}$$

When $n = 1000$, the parallelism $\approx 10^6$, which is still quite high.

The naive algorithm (M'') that computes n^2 dot-products in parallel yields the following theoretical results:

$$M_1''(n) = \Theta(n^3) \tag{2.13}$$
$$M_\infty'' = \Theta(\lg n) \tag{2.14}$$
$$=> Parallelism = \Theta(n^3/\lg n). \tag{2.15}$$

Although it does not use temporary storage, it is slower in practice due to less memory locality.

2.2.2 Sorting

In this section, we consider a parallel algorithm for sorting an array. We start by parallelizing the code for MERGE-SORT while using the traditional linear time algorithm MERGE to merge the two sorted subarrays.

```
MERGE-SORT(A, p, r)
if p < r
   then q ← ⌊(p + r)/2⌋
            spawn MERGE-SORT(A, p, q)
            spawn MERGE-SORT(A, q + 1, r)
            sync
            MERGE(A, p, q, r)
```

Since the running time of MERGE is $\Theta(n)$, the work (denoted $T_1(n)$) for MERGE-SORT is

$$T_1(n) = 2T_1(n/2) + \Theta(n) \tag{2.16}$$
$$= \Theta(n \lg n) \tag{2.17}$$

by case 2 of the Master Theorem. The critical-path length (denoted $T_\infty(n)$) is the critical-path length of one of the two recursive spawns plus that of MERGE:

$$T_\infty(n) = T_\infty(n/2) + \Theta(n) \tag{2.18}$$
$$= \Theta(n) \tag{2.19}$$

by case 3 of the Master Theorem. The parallelism, $T_1(n)/T_\infty = \Theta(\lg n)$, is not scalable. The bottleneck is the linear-time MERGE. We achieve better parallelism by designing a parallel version of MERGE.

```
P-MERGE(A[1..l], B[1..m], C[1..n])
if m > l
   then spawn P-MERGE(B[1 . . m], A[1 . . l], C[1 . . n])
elseif n = 1
   then C[1] ← A[1]
elseif l = 1
   then if A[1] ≤ B[1]
            then  C[1] ← A[1]; C[2] ← B[1]
            else   C[1] ← B[1]; C[2] ← A[1]
   else  find j such that B[j] ≤ A[l/2] ≤ B[j + 1] using binary search
         spawn P-MERGE(A[1..(l/2)], B[1..j], C[1..(l/2 + j)])
         spawn P-MERGE(A[(l/2 + 1)..l], B[(j + 1)..m], C[(l/2 + j + 1)..n])
   sync
```

P-MERGE puts the elements of arrays A and B into array C in sequential order, where $n = l + m$. The algorithm finds the median of the larger array and uses it to partition the smaller array. Then, it recursively merges the lower portions and the upper portions of the arrays. The operation of the algorithm is illustrated in figure 2.4.

We begin by analyzing the critical-path length of P-MERGE. The critical-path length is equal to the maximum critical-path length of the two spawned subproblems plus the work of the binary search. The binary search completes in $\Theta(\lg m)$ time, which is $\Theta(\lg n)$ in the worst case. For the subproblems, half of A is merged with all of B in the worst case. Since $l \geq n/2$, at least $n/4$ elements are merged in the smaller subproblem. That leaves at most $3n/4$ elements to be merged in the larger subproblem. Therefore, the critical-path is

$$\begin{aligned} T_\infty(n) &\leq T(3/4n) + O(\lg n) \\ &= O(\lg^2 n) \end{aligned}$$

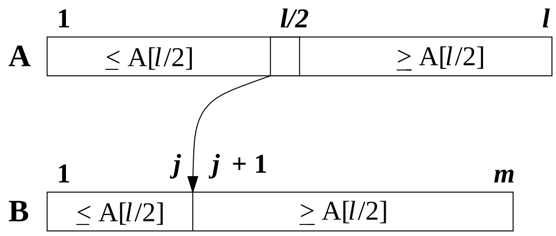


Figure 2.4: Find where middle element of A goes into B. The boxes represent arrays.

by case 2 of the Master Theorem. To analyze the work of P-MERGE, we set up a recurrence by using the observation that each subproblem operates on αn elements, where $1/4 \leq \alpha \leq 3/4$. Thus, the work satisfies the recurrence

$$T_1(n) \quad = T(\alpha n) + T((1-\alpha)n) + O(\lg n).$$

We shall show that $T_1(n) = \Theta(n)$ by using the substitution method. We take $T(n) \leq an - b\lg n$ as our inductive assumption, for constants $a, b > 0$. We have

$$\begin{aligned}
T_{(}n) &\leq a\alpha n - b\lg(\alpha n) + a(1-\alpha)n - b\lg((1-\alpha)n) + \Theta(\lg n) \\
&= an - b(\lg(\alpha n) + \lg((1-\alpha)n)) + \Theta(\lg n) \\
&= an - b(\lg\alpha + \lg n + \lg(1-\alpha) + \lg n) + \Theta(\lg n) \\
&= an - b\lg n - (b(\lg n + \lg(\alpha(1-\alpha))) - \Theta(\lg n)) \\
&\leq an - b\lg n,
\end{aligned}$$

since we can choose b large enough so that $b(\lg n + \lg(\alpha(1-\alpha)))$ dominates $\Theta(\lg n)$. We can also pick a large enough to satisfy the base conditions. Thus, $T_{(}n) = \Theta(n)$, which is the same as the work for the ordinary MERGE. Reanalyzing the MERGE-SORT algorithm, with P-MERGE replacing MERGE, we find that the work remains the same, but the critical-path length is now

$$T_\infty(n) = T_\infty(n/2) + \Theta(\lg^2 n) \tag{2.20}$$
$$= \Theta(\lg^3 n) \tag{2.21}$$

by case 2 of the Master Theorem. The parallelism is now $\Theta(n\lg n)/\Theta(\lg^3 n) = \Theta(n/\lg^2 n)$. By using a more clever algorithm, a parallelism of $\Omega(n/\lg n)$ can be achieved.

While it is important to analyze the theoritical bounds of algorithms, it is also necessary that the algorithms perform well in practice. One short-coming of MERGE-SORT is that it is not in-place. An in-place parallel version of QUICK-SORT exists, which performs better than MERGE-SORT in practice.

Additionally, while we desire a large parallelism, it is good practice to design algorithms that scale down as well as up. We want the performance of our parallel algorithms when running on one processor to compare well with the performance of the serial version of the algorithm. The best sorting algorithm to date requires only 20% more work than the serial equivalent. Coming up with a dynamic multithreaded algorithm which works well in practice is a good research project.

6.895 Theory of Parallel Systems	Lecture 3

High Performance Serial Programming and Caching

Lecturer: Bradley Kuszmaul and Michael Bender Scribe: Kenneth Barr and Zardosht Kasheff

Lecture Summary

1. *High Performance Serial Programming*
   Before writing fast parallel code, you must understand the hardware on which your program will run and how to write fast *serial* code on this hardware.

2. *A Theoretic Machine Model*
   In this section, we formalize a cache-dependent model for analyzing code on a particular computer system.

3. *Analysis of Several Algorithms*
   In this section, we analyze several algorithms using the model defined in the previous section

3.1 High Performance Serial Programming

3.1.1 Computer Hardware

High performance programming is impossible without knowledge of the hardware in the host computer. This section discusses one of the most important elements of the hardware: the memory hierarchy. Figure 3.1 sketches the memory structures in computer. Each level of the memory hierarchy is larger and slower than the one that preceded it. Closest to the processor is a small, fast register file. Data words (e.g., 64 bits) are brought to the register file from the cache. A cache contains a subset of the words in main memory and deals with ***lines*** (e.g., 128 bytes) rather than individual words. Table 3.1 shows the sizes and speeds of memory hierarchies for two recent computer systems.

	ygg: Origin 2000 w/R10000	Pentium 4
Processor speed	195 MHz	2500 MHz
Registers	64	8
L1 Cache size	32KB	8KB
L1 Cache access time (ns)	10	1
L1 Cache line size	128B	64B
L1 Associativity	2-way	4-way
L1 Cache lines	256	64
L2 Cache size	4MB	512KB
L2 Cache access time (ns) ns	62	8
L2 Cache line size	128B	128B
L2 Associativity	2	8-way
L2 Cache lines	32K	4K
Memory access time (ns)	484	150

Table 3.1: System parameters

Caches are small, fast subsets of main memory. Since they are smaller than main memory, words necessarily map to more than one location in the cache. Therefore, the address of the data (the tag) is part of the information that is stored alongside the data itself. To reduce miss-rates, associative caches, in which

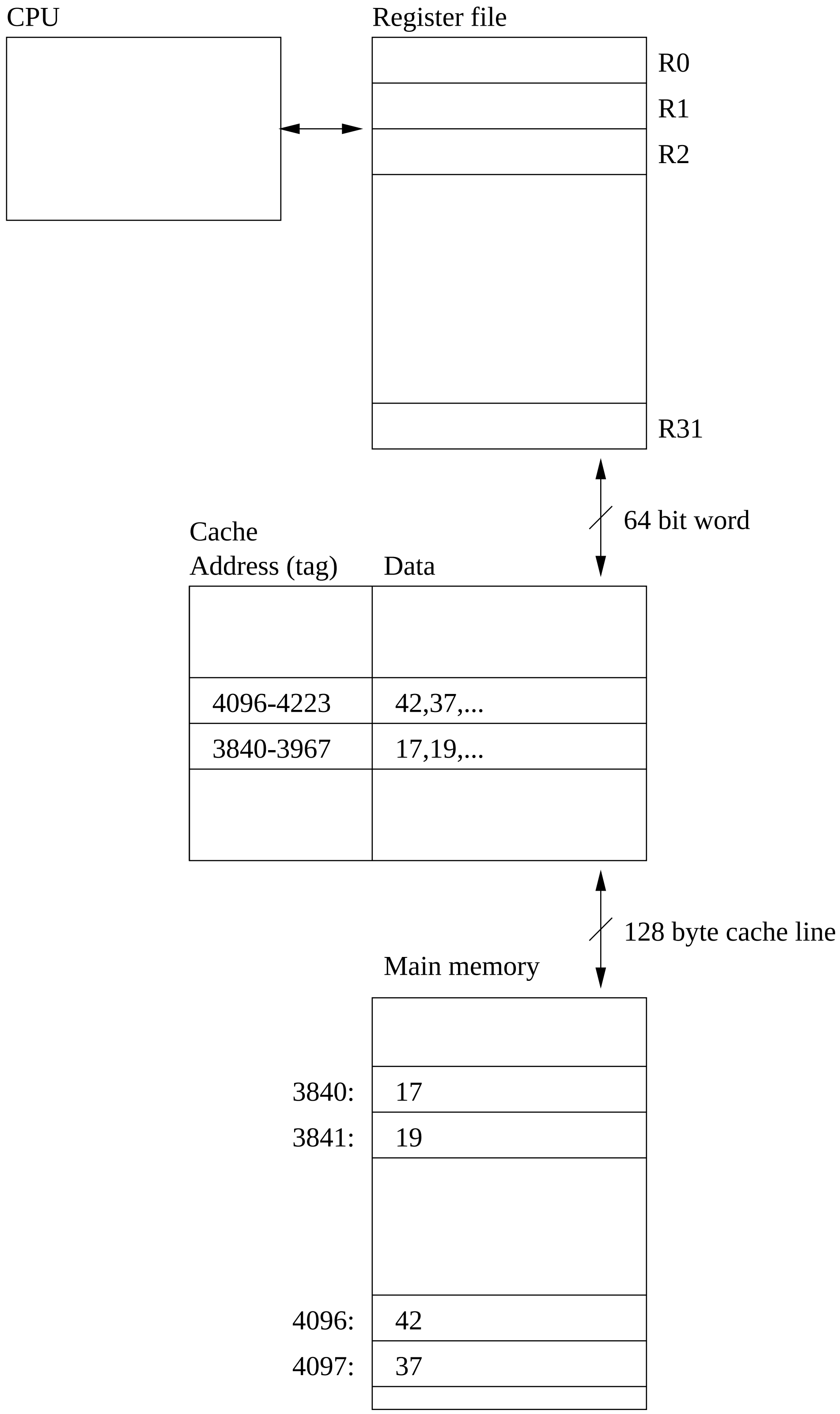


Figure 3.1: Memory hierarchy.

a word may be found in several locations, are more popular than direct-mapped caches (which we did not discuss). There are two types of associativity:

Fully associative Any word of memory can be stored in any cache line.

Set associative Words of memory can be placed in restricted set of places in the cache. A word is mapped onto a set and can be placed anywhere in that set. If there are n words in a set, the cache is n-way set associative.

For example, a 512 Kilobyte (KB), 4-way set associative cache with 128 byte lines would contain 1024 sets. Memory locations that are 128KB apart in memory are in the same set and compete for space in that set. If the replacement policy and data access pattern are compatible, the four ways permit four such addresses to coexist in the cache without evicting each other.

3.1.2 Compilers

Compilers translate high-level language to assembly language.[1] The handout [14] shows what this assembly language looks like. Note that instructions are simple (usually one or two operands, one result). To produce the best set of simple assembly instructions, you must use a "good compiler." Though SGI `cc` has a good reputation, `gcc` can outperform it occasionally. Here are some more explicit tricks and tools.

- Profile your code so that you know where to focus your optimization effort. Use the `-pg` option to gcc or the `-p` to `cc`. Then run `gprof` or `prof`, respectively. Profiling uses statistical sampling (e.g., every 1ms it asks: what function is currently executing?). This is the "real-life" way to do what we did in last lecture with theory (that is, attack the $O(n)$ merge).
- Debug your code. `-g` inserts debugging symbols for gdb. Caveat: we don't appear to have a working debugger on `ygg`.
- Enable compiler optimization with `-O2`; usually gets a dramatic speedup.
- `-S` produces assembly listings so you can examine the results of compilation textually.
- Become familiar with all of the other switches available to you by reading the manual:

```
ygg\% man cc
ygg\% info gcc
```

3.1.3 High-level optimizations

Ideally the compiler would perform all high-level optimizations for you, but in practice reorganizing/optimizing your code manually is necessary to help produce a fast implementation.

[1] Usually, the compiler will invoke an assembler to translate the assembly language to machine code

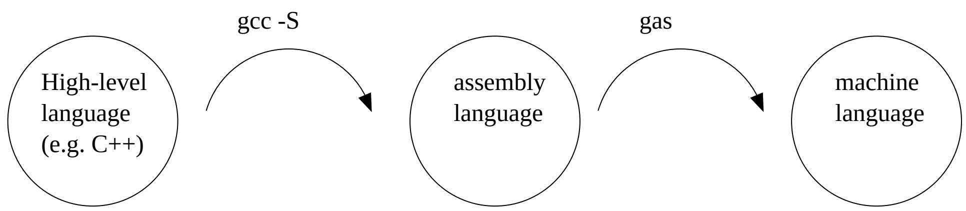


Figure 3.2: Compilation process

A good reference is AMD's Software Optimization Guide [3]. Though written with the AMD64 instruction set and microarchitecture in mind, many of the techniques can be applied broadly. Other vendors have similar guides which may have useful machine-specific tricks. The remainder of this section contains the high-level optimizations discussed in class.

Exploit registers

Registers are the fastest storage elements at our disposal, so structure your code to do as much work as possible with data in registers before that data is pushed further down the memory hierarchy. For example, a 2x2 matrix is small enough to allow all operands and results to fit in registers. Since each operand is used several times, it is best to leave them in registers for the duration of the computation.

Exploit fast memory

Carrying the above idea further, we would prefer to work on data in L1 cache to L2 cache, and we'd prefer L2 to main memory. Size your problems appropriately; use cache-conscious data layout; or use optimal, cache-oblivous algorithms (see Section 3.2).

Move computations out of loops

This loop initializes the array with $\sin x$ which is effectively a constant, but the compiler does not know this and will make 1000 calls to the sin library function. If you were doing `a[i]=x+y`, the compiler might be able to tell and remove the add from the loop, but it doesn't know that sin() is different (here) than an arbitrary function call.

```
for(i=0; i<1000; i++){
  a[i]=sin(x);
}
```

Strength reduction

Reduce complex operations to simpler forms (e.g., exponentiation into a multiply or multiply into an add). The examples from lecture:

```
for(i=0; i<1000; i++){
  a[i]=pow(S,i);
}
```

becomes

```
tmp=1;
for(i=0; i<1000; i++){
  a[i]=tmp;
  tmp*=S;
}
```

Unroll loops

To eliminate the overhead associated with looping, try unrolling your loops to some degree.

```
for(i=0; i<N; i++)
  f(i);
```

becomes

```
for(i=0; i+1<N; i+=2){
  f(i);
  f(i+1);
}
for(; i<N; i++)         //clean up last case
  f(i);
```

This effectively changes $r+ = a(i) * b(i)$ to $r+ = a(i) * b(i) + a(i+1) * b(i+1)$ Sometimes the compiler can do this for you, sometimes it cannot.

Remove control dependencies

In deeply pipelined machines, a mispredicted branch can require tens of cycles to repair, squashing hundreds of instructions. Sometimes you can replace a hard-to-predict branch with a predicated instruction such as MIPS's conditional move:

```
movz Ri, Rj, Rn       #if(Rn==0) Ri=Rj
```

Here a control dependence has become a data dependence....

3.1.4 Example: isort vs. sort5 [14]

Recall that despite its $O(n^3)$ running time, insertion sort needs few steps to complete for for small values of n. Despite a larger number of operations, the `sort5` code is measureably faster when sorting five elements. Why? The `isort` code is slowed down by unpredictable branches and more memory references. To demonstrate, Table 3.2 shows the time taken by each routine with varying compiler and machine combinations.

	gcc	cc	gcc (Pentium)
isort	599ns	580ns	88ns
sort5	275ns	112ns	34ns

Table 3.2: Sort timing

3.2 A Theoretic Machine Model

3.2.1 Motivation of Model

A model that takes caching into account is needed because data locality has a significant effect on the practical runtime of algorithms.

Poor Matrix Multiplication Algorithm. Below is the trivial algorithm for Matrix Multiplication for the n by n matrices $C = AB$. This algorithm works well on the RAM model, where all instructions complete in one timestep, but works poorly in practice.

Mult(C,A,B)
  for $i = 1$ **to** n **do**
    for $j = 1$ **to** n **do**
      $c_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$.

This algorithm performs poorly because it lacks data locality hence causing bad cache performance.

3.2.2 Memory Heirarchies

Multilevel Memory Heirarchy

A multilevel memory heirarchy has many parameters, making it complicated and difficult to analyze; see Figure 3.3.

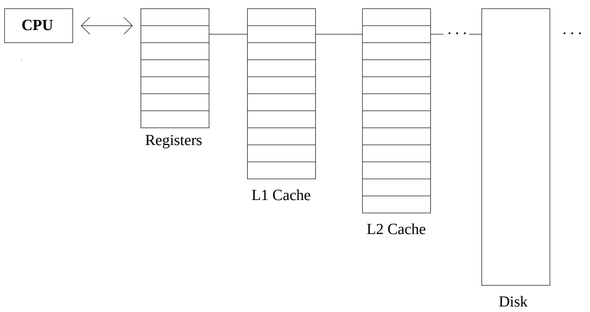


Figure 3.3: Multievel Memory Hierarchy. Each level has a different size, block size, and number of cache lines. The time for a memory transfer between levels differs.

Two-Level Memory Heirarchy

The following two-level memory model is a simplification that is easier to analyze; see Figure 3.4.

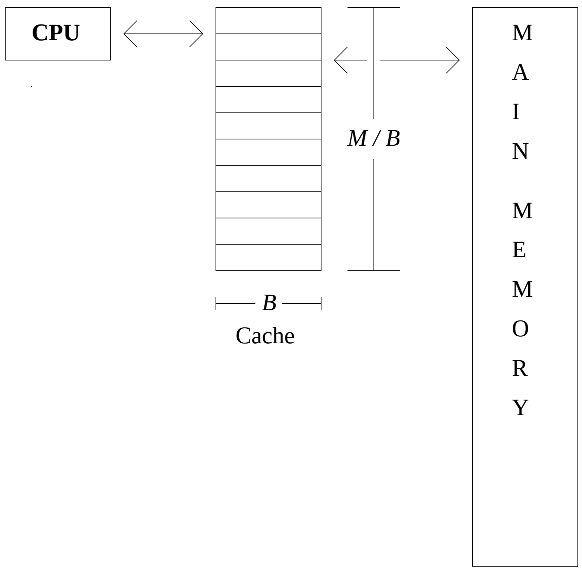


Figure 3.4: Two-Level Memory Hierarchy with block size B, cache size M, and M/B lines of cache.

Description. This two-level memory hierachy consists of a CPU, cache, and main memory. The cache is divided into cache lines, or ***blocks*** of size B. The cache has size M and is composed of M/B blocks. The act of moving a block between cache and main memory is called a ***memory transfer***. When a memory transfer brings a block from main memory into a "slot" in cache, the memory block that previously occupied that slot is ***evicted***.

Objective. We want to minimize the number of memory transfers in an algorithm, because they are the most expensive operations.

Although referred to as "cache" and "main memory", these two levels can represent any two contiguous levels in the memory heirarchy. For the remainder of these notes, we use the two-level memory model and not the multilevel memory model.

3.2.3 Block Replacement Strategies

One issue is how to select which block to evict during a memory transfer. Below we compare ideal solutions to realistic ones.

Ideal Cache Model vs. Other Replacement Strategies

Description. According to the ***ideal cache model*** [8], the best block to replace is the block that is accessed farthest in the future [Belady]. We call this model ***FIF***, which stands for "farthest in future".

Difficulty: Cannot Look into future. Although the optimal strategy, FIF is not implementable. One cannot look into the future to select which block to evict. However, there exists a simulation that avoids this difficulty.

Theorem 3.1 *LRU/FIFO with cache of size $M \le$ 2OPT with cache of size $2M$.*

For further details see [8].

Difficulty: Limited Assiciativity. The ideal cache model assumes full associativity while actual hardware has limited associativity. In our ideal cache model, we assume any block of memory can be stored anywhere in cache (***full associativity***). Realistically, caches do not have this property. Caches can be ***direct-mapped***, meaning block i can *only* be cached in slot $i \mod (M/B)$. Caches can also be ***2-way set associative***, meaning only two blocks can be stored in each slot ($M/2B$ slots total).

As described in Section 1, actual hardware has limited associativity and also has a simpler replacement policy.

Theorem 3.2 *[8] Auto-replacement, fully associative LRU/FIFO cache can be simulated using a manual replacement, direct mapped cache in $O(1)$ expected time*

For further details see [8].

3.2.4 Cache-Aware v. Cache-Oblivious

Sections 2.1 - 2.3 give a well defined computational model involving caching. We define two different types of algorithms:

Cache-Oblivious Algorithms. A cache-oblivious algorithm does not use the variables B and M, but its performance still depends on B and M. The cache-oblivious model enables us to reason about a simple two-level memory model but prove results about an unknown multilevel memory model. Cache-oblivious algorithms perform an *asymptotically optimal* number of memory transfers for *any* memory heirarchy at *all levels* of the hierarchy. The number of memory transfers between any two levels is within a constant factor of optimal. Any linear combination of the transfer counts is optimized.

Cache-Aware Algorithms. A cache-aware algorithm is an optimal algorithm that is aware of and uses the variables M and B.

3.3 Analysis of Several Algorithms

3.3.1 Method of Asymptotic Analysis

The goal is to use algorithms with a solid foundation to predict performance. Thus, we present in Section 2 a simple algorithmic model that gives more accurate running time than the standard RAM model.

Defining Cost. We define the ***cost*** of an algorithm as the number of memory transfers. Note we do not take into account the time to transfer memory from the cache to the CPU or operations on data in cache, because the time needed to perform these operations is much less than the time needed to perform a memory transfer between cache and main memory.

3.3.2 Example 1: Array-scanning

Consider array-scanning. The goal is to read in order the elements in an array $A[1...n]$.

```
for i = 1 to n
  sum += A[i].
```

Cost. We see:

$$\text{Cost} \leq \left\lceil \frac{N}{B} \right\rceil + 1.$$

An example is in Figure 3.5.

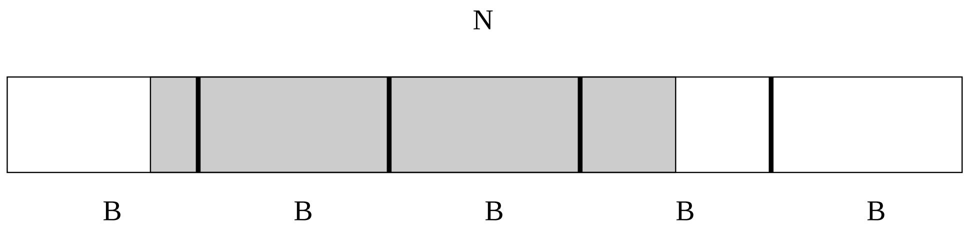


Figure 3.5: Array spread across blocks. The clear rectangles are blocks in cache. The shaded region is the location of the array in cache.

Analysis. At worst, this algorithm is optimal to within 1 memory transfer.

3.3.3 Example 2: Binary Search

Consider performing a binary search on an array using the traditional algorithm.

Cost. The total number of memory transfers is

$$\begin{aligned} \text{cost} &= \lg \frac{N}{B} && (3.1) \\ &= \lg N - \lg B. && (3.2) \end{aligned}$$

Analysis. This cost is *not* optimal. The optimal cost is $\log_B N$. We achieve this performance bound using B-trees (which are out of the scope of this lecture).

3.3.4 Example 3: Recursive Matrix Multiplication

Consider the recursive algorithm where $C = AB$. The recursive method, breaks each matrix into four parts as such:

$$\begin{aligned}\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} &= \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}\begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \\ &= \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}.\end{aligned}$$

Layout of Matrix in Memory. We first consider how to lay the matrix out in memory. For the bad serial algorithm presented above, A being stored as row-major and B being stored as column-major would be good.

For the recursive algorithm, however, a better layout exists in Figure 3.6.

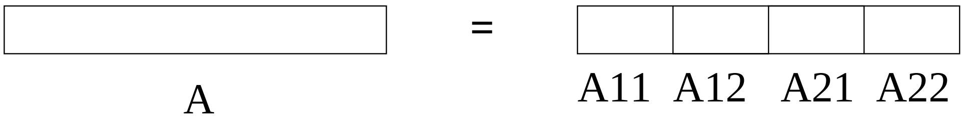


Figure 3.6: Representation of Matrix in memory.

The above representation has been referred to as ***Mortan Order***, and is similar to a *space-filling curve* or a *Hilbert curve.*

Cost.

Derivation of Recurrence Equation. We express this algorithm by the recurrence:

$$T(N) = 8T\left(\frac{N}{2}\right) + O\left(\left\lceil\frac{N^2}{B}\right\rceil\right).$$

The second term of the recurrence represents the number of memory transfers needed to perform the addition, as presented in Lecture 2. Adding N^2 elements in order causes $\lceil N^2/B \rceil$ memory transfers, thus it is $O\left(\lceil N^2/B \rceil\right)$.

Base Cases. We analyze the two base cases.

The first base case is,

$$T(\sqrt{B}) = O(1).$$

In this case, the entire matrix fits within one cache block.

The second base case is,

$$T(c\sqrt{M}) = O\left(\frac{M}{B}\right),$$

for a sufficiently small c such that all 3 matrices fit in cache. The result is $O(M/B)$ because the data is brought into memory. All other computation is performed in cache.

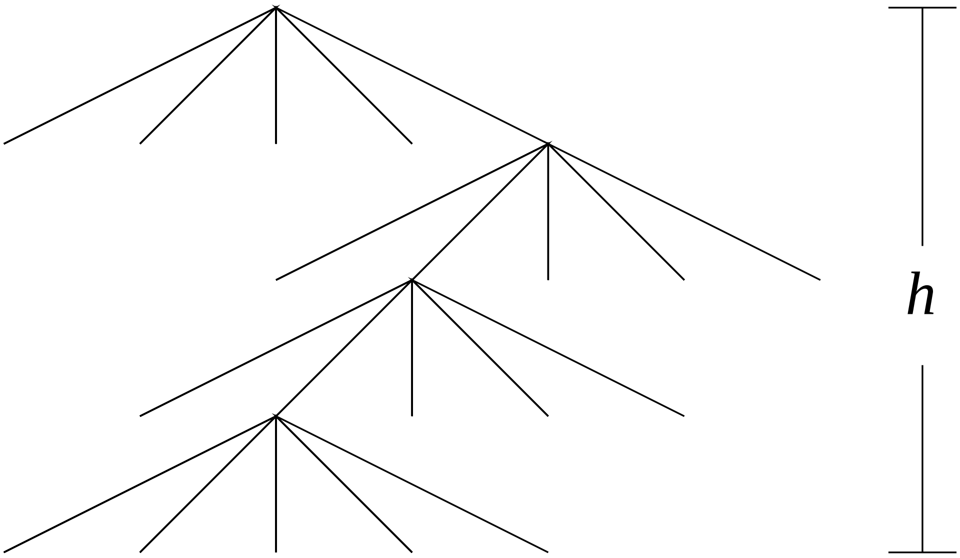


Figure 3.7: Recursion Tree Sketch. $h = \lg\left(\frac{N}{\sqrt{M}}\right)$

Solving the Recurrence Equation. We solve this recurrence using a tree instead of the Master Theorem. A sketch is in Figure 3.7.

The cost of the root is $O\left(\lceil N^2/B \rceil\right)$. We traverse the tree until we reach problems of size $O(\sqrt{M})$. The height of this tree is :

$$h = \lg\left(\frac{N}{O(\sqrt{M})}\right).$$

Thus, the number of leaves ℓ is:

$$\ell = 8^{\lg\left(\frac{N}{O(\sqrt{M})}\right)} \tag{3.3}$$

$$= O\left(\left(\frac{N}{\sqrt{M}}\right)^3\right). \tag{3.4}$$

The cost per leaf $= \Theta(M/B)$, thus the cost of all leaves is:

$$\text{cost of all leaves} = \Theta\left(\frac{M}{B}\frac{N^3}{M^{\frac{3}{2}}}\right) \tag{3.5}$$

$$= \Theta\left(\frac{N^3}{B\sqrt{M}}\right). \tag{3.6}$$

$$\tag{3.7}$$

Thus, the cost of the root is,

$$\text{cost of root} = \Theta\left(\left\lceil\frac{N^2}{B}\right\rceil\right). \tag{3.8}$$

Asymptotically, the leaves dominate the cost. Thus:

$$\text{Total cost} = \Theta\left(\frac{N^3}{B\sqrt{M}}\right).$$

Analysis. The cost of this cache-oblivious algorithm is optimal. It can be proved using pebble games, but the proof is out of the scope of the lecture.

6.895 Theory of Parallel Systems Lecture 4

Debugging Multithreaded Programs

Lecturer: Michael Bender *Scribe: Sid Sen and Jim Sukha*

Lecture Summary

1. *Determinacy Races*
   This section introduces the concept of a determinacy race in parallel programs, and provides two examples of Cilk programs where such races occur.

2. *Introduction to the Nondeterminator*
   This section introduces the Nondeterminator, discusses its performance, and describes how Cilk programs can be represented using series-parallel directed acyclic graphs and series-parallel parse trees.

3. *The SP-Bags Algorithm*
   This section presents the SP-Bags algorithm, the algorithm used by the Nondeterminator to detect determinacy races in Cilk programs.

4. *Nondeterminator Correctness*
   This section presents a proof of correctness for the SP-Bags algorithm for the Nondeterminator.

4.1 Determinacy Races

In a parallel program, two threads running concurrently may want to access the same memory location. If the order of the accesses is not well defined, this conflict is a determinacy race, which can lead to incorrect program behavior. This section defines a determinacy race and analyzes two Cilk programs that have determinacy races.

4.1.1 A Cilk Program with a Determinacy Race

In this section, we provide an example of a simple Cilk program with a determinacy race. We consider the following Cilk program:

```
int x;

cilk void foo(void) {
  x++;
  return;
}

cilk int main(void) {
  x = 0;
  spawn foo();
  spawn foo();
  sync;
  printf("x is %d\n", x);
  return 1;
}
```

a)

Thread 1	Thread 2
read x	
write x	
	read x
	write x

b)

Thread 1	Thread 2
read x	
	read x
write x	
	write x

Figure 4.1: Two possible execution interleavings of the spawned threads in our simple Cilk program: (a) At the end of this sequence, the value of x is 2. (b) At the end of this sequence, the value of x is 1.

If a thread updates a location while another thread is concurrently accessing the location, then we say that a ***determinacy race*** occurs. In the example program, the two threads that are spawned are both trying to modify the value of x.

Each increment of x involves both a read and a write to x's memory location. Figure 4.1 shows two possible ways the executions of the two threads can overlap. Depending on how these read and write operations are interleaved, at the end of the program, x may have a value of 1 or 2.

4.1.2 Determinacy Races in Cilk

In this section, we formally define a determinacy race in the context of Cilk programs.

Definition 4.1 (Determinacy Race) *A Cilk program contains a determinacy race if two logically parallel threads access the same shared location and one of the accesses is a write.*

We can have read/write races or write/write races, as illustrated in Figure 4.2.

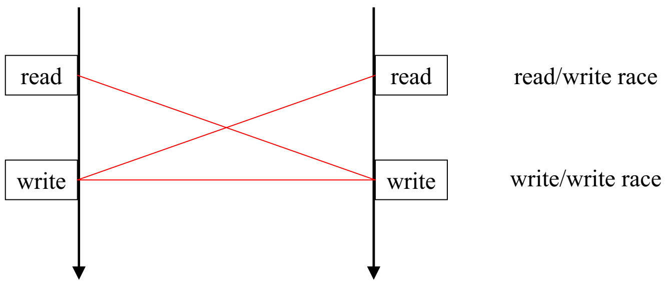


Figure 4.2: Two types of determinacy races.

In general, a determinacy race can cause a multithreaded parallel program to act nondeterministically. In most cases, programmers intend to write deterministic code, so a determinacy race is usually a bug.

However, it is still possible for a correct program to be nondeterministic. Suppose that the code in Section 4.1.1 has another procedure `bar` that decrements x, and assume that we are somehow guaranteed that both `foo` and `bar` execute atomically, so that operations from the two procedures are never interleaved. Then, if we spawn one thread executing `foo` and another executing `bar`, there is a determinacy race because we do not know which thread executes first. The resulting program is nondeterministic, but in this case we really do not care because the order of execution does not matter—that is, the resulting value in x is the same either way, making the output of the program deterministic.

The following definition summarizes the difference between these two situations:

Definition 4.2 (Internal and External Determinism) *A program is **internally deterministic** if the program has no determinacy races. A program is **externally deterministic** if the program has determinacy races but the output is deterministic.*

4.1.3 Example: the N-Queens Puzzle

The example in Section 4.1.1 showed a determinacy race that was fairly easy to spot. In more complicated programs, however, finding determinacy races can be much more difficult. In this section, we consider a subtle determinacy race in a program that solves the N-Queens puzzle.

Definition 4.3 (N-Queens Puzzle) *Find a configuration of N queens on an N by N chessboard such that no two queens attack each other.*

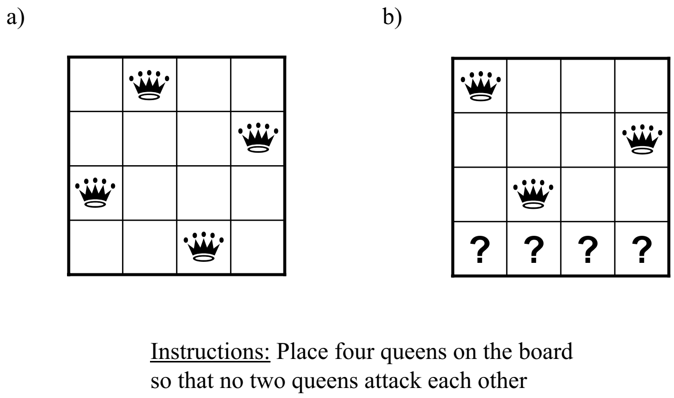


Figure 4.3: N-Queens problem for a 4 by 4 chessboard ($N = 4$): (a) A valid configuration. (b) An invalid configuration.

Figure 4.3 illustrates a valid and invalid solution to the N-Queens puzzle for the specific case when $N = 4$.

One common approach to solving the N-Queens puzzle makes use of a recursive backtracking algorithm, where queens are placed row by row (starting with the first row) and we backtrack to the previous row whenever a valid configuration of queens cannot be found for the current row.

An abbreviated version of this program is shown below (see Figure 4.4 for an explanation of the `board` and `newboard` parameters).

```
1:  cilk char *nqueens(char *board,         // current configuration of queens
2:                     int n,               // board size
3:                     int row) {           // row number where queen is
                                            // placed next

4:    char *new_board;
5:    ...
6:    new_board = malloc(row+1);            // malloc new space for child's board
7:    memcpy(new_board, board, row);        // copy existing board into new board
8:    for (j = 0; j < n; j++) {             // consider all places for new queen
9:      ...
10:     new_board[row] = j;                 // assign row queen to jth column
11:     spawn nqueens(new_board, n, row+1); // proceed to (row+1)th row
12:     ...
13:   }
14:   sync;
15:   ...
16: }
```

To highlight the data race, we include only the relevant sections of code (steps such as checking for a valid board configuration are omitted). Figure 4.4 illustrates the representation of the chessboard used in the program for the specific case where $N = 4$.

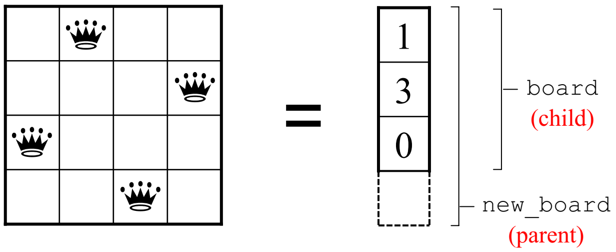


Figure 4.4: Representation of a 4 by 4 chessboard in the `nqueens` procedure.

When a parent procedure spawns a call to `nqueens` in line 11, it passes `new_board` to the child. However, a conflict may arise in the use of this board data when the child reads the `board` variable in line 7, since the parent writes to the board in the next iteration of the for loop (line 10). Thus, the old board may be updated by the parent before the child has a chance to copy it over into the new board, resulting in a determinacy race.

4.2 Nondeterminator Background/Theory

As the previous examples show, determinacy races in programs are usually bugs that a programmer should eliminate. Cilk has a tool called the Nondeterminator that helps programmers detect determinacy races. In this section, we introduce the Nondeterminator, discuss its performance, and describe how to represent Cilk programs using series-parallel directed acyclic graphs (or ***dag***s) and series-parallel parse trees. These abstractions will help us when we explain the execution of the Nondeterminator and prove its correctness in Sections 4.3 and 4.4.

4.2.1 Definition and Performance

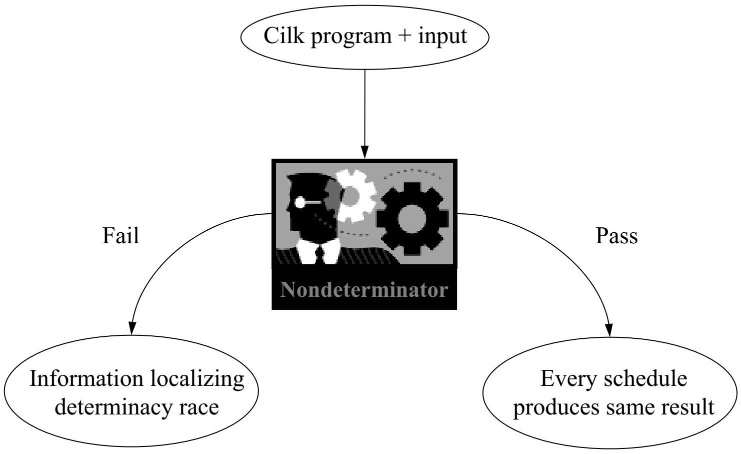


Figure 4.5: High-level behavior of the Nondeterminator.

Figure 4.5 illustrates the input-output behavior of the Nondeterminator. The Nondeterminator takes as input a Cilk program and a particular input data set to the program. The Nondeterminator passes the program if every scheduling of the computation produces the same result; otherwise, it fails the program and returns information for locating the determinacy race. It is important to note that the Nondeterminator is a debugging tool, not a program verifier. Although a Cilk program may pass on a particular data set, the same program may fail on other data sets.

The Nondeterminator has provable performance. The following theorem bounds its running time:

Theorem 4.4 (Nondeterminator Performance) *For a Cilk program that runs in T time serially and uses v shared-memory locations, the Nondeterminator runs in $O(T\alpha(v,v))$ time and uses $O(v)$ space, where α is Tarjan's functional inverse of Ackermann's function.*

Note that $\alpha(v,v)$ is much, much smaller than $\log^*(v)$, which is the number of times you need to take the log of v before you get a value below 1. For all practical purposes, $\alpha(v,v) \leq 4$.

We achieve this good performance for the Nondeterminator by exploiting the special structure of Cilk dags, as shown in the sections below.

4.2.2 Series-Parallel (SP) Dags

In this section, we show that Cilk programs can be represented as dags that are series-parallel. We say that a graph is ***series-parallel*** if it can be derived from a base graph using series and parallel compositions. The base graph and composition rules are summarized in the list below and illustrated in Figure 4.6.

- *Base Graph*: This graph is a single directed edge from a source node s to a sink node t.
- *Series Composition*: Given two series-parallel graphs, G_1 and G_2, with source and sink nodes (s_1, t_1) and (s_2, t_2), respectively, we compose the two graphs in series to form a new graph G with source node $s = s_1$ and sink node $t = t_2$. The series composition merges the sink node of G_1 with the source node of G_2, so that $t_1 = s_2$ in the resulting graph G.
- *Parallel Composition*: Given two series-parallel graphs, G_1 and G_2, with source and sink nodes (s_1, t_1) and (s_2, t_2), respectively, we compose the two graphs in parallel to form a new graph G with source node $s = s_1 = s_2$ and sink node $t = t_1 = t_2$.

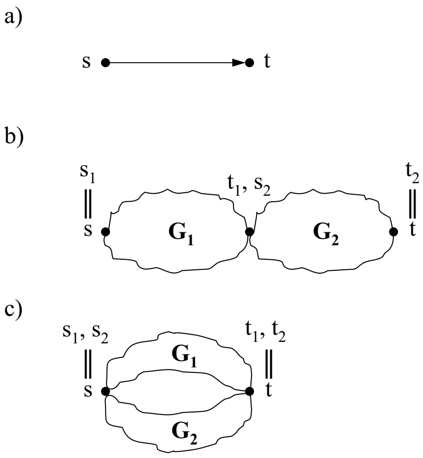


Figure 4.6: Series-Parallel Dags: (a) The base graph. (b) A series composition of graphs G_1 and G_2. (c) A parallel composition of graphs G_1 and G_2.

The execution of a Cilk program can be represented as an SP-dag, with threads corresponding to edges and spawn/sync steps corresponding to nodes. For example, consider a slightly modified version of the program in Section 4.1.1:

```
F:        cilk int main(void)
          {
    e0:      x = 0;
    F1:      spawn foo();
    e1:      // some additional serial code
    F2:      spawn foo();
    e2:      // some more serial code
             sync;
    e3:      printf("%d", x);
             return 1;
          }
```

Figure 4.7 illustrates the corresponding SP-dag for this program. The sections marked with the e_i identify the threads, while the F_i labels are function ID's for the Cilk procedures.

From our construction of the SP-dag above, it is not difficult to show the following theorem:

Theorem 4.5 (Cilk Dags) *A Cilk dag is a series-parallel dag with out-degree of at most 2 at each node.*

Proof Our proof is a simple extension of the proof for Theorem 3 in [13], which verifies the weaker statement that a Cilk dag is in fact a series-parallel dag. To show that each node in the resulting SP-dag has out-degree of at most two, we consider what happens when a `spawn` statement interrupts the control-flow of a given thread.

Every time a `spawn` statement is reached, we create a new node in our graph with two edges directed out. One edge corresponds to the spawned procedure, and the other corresponds to the continuation of the parent procedure (i.e. everything following the `spawn` statement). Both of these paths eventually converge at the node corresponding to the `sync` at the end of the parent procedure. Recall that the Cilk compiler inserts an implicit `sync` at the end of the parent procedure if none is included by the programmer.

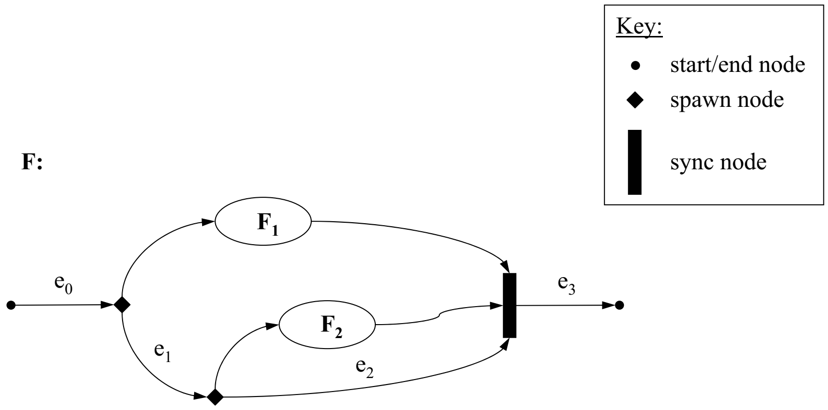


Figure 4.7: The SP-dag for procedure F.

Figure 4.7 illustrates the `spawn` and `sync` nodes clearly for procedure F above. A more complicated example (with multiple `sync` nodes) is shown in Figure 4.10. □

4.2.3 Series-Parallel (SP) Parse Trees

Another way of looking at the structure of Cilk procedures is in terms of series-parallel parse trees. In this section, we describe how to construct an SP parse tree for a given Cilk procedure.

In an SP parse tree, the leaves of the tree correspond to threads or (spawned) procedures, and the non-leaf nodes are either S-nodes or P-nodes. For S-nodes, everything in the left subtree is executed before everything in the right subtree. For P-nodes, everything in the left the left subtree can run logically in parallel with everything in the right subtree. Figure 4.8 shows an SP parse tree for procedure F in Section 4.2.2.

From [13], we recall that a sync block has the following form:

e_0; **spawn** F_0; e_1; **spawn** F_1; ... e_j; **sync** ;

To create an SP parse tree for a single sync block, first create a tree with root S and left child e_0. Set the right child of S to be a tree with root P, left child F_0, and right child equal to the subtree for the rest of the code up until the sync. Observe that the left child of an S-node in a sync block is always a thread, while the left child of a P-node is always a (spawned) procedure [13].

To create an SP parse tree for an entire Cilk procedure, string the parse trees for the sync blocks together using a ***spine*** of S-nodes (one S-node for each sync block). The left child of each S-node in the spine is the parse tree for the corresponding sync block, and the right child of the last S-node in the spine is the thread that immediately precedes the `return` statement of the procedure.

Figure 4.9 shows the SP parse tree for a generic Cilk procedure.

Following this construction of an SP parse tree, we see that a standard depth-first traversal of the parse tree visits the threads in the same execution order as if the program were run on a single processor (i.e., it follows a sequential execution of the procedure).

Note that since the S relationship is associative and the P relationship is commutative, it is possible to have multiple SP parse trees that are logically equivalent.

From the tree shown in Figure 4.9, we notice the following:

Observation 4.6 *The least common ancestor (LCA) of two threads determines whether the threads are logically in series or in parallel. We use the notation:*

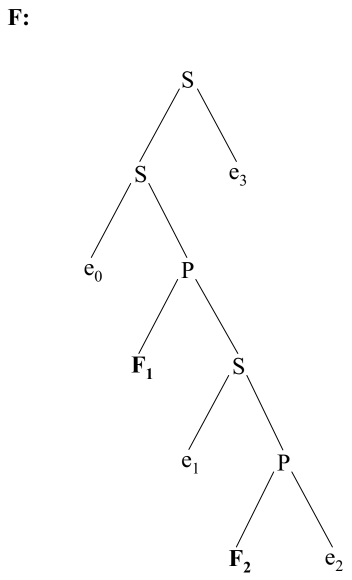


Figure 4.8: An SP parse tree for procedure F.

- *$e \prec e'$ if $LCA(e, e')$ is S and e is to the left of e'.*
- *$e \parallel e'$ if $LCA(e, e')$ is P.*

Every SP-dag has a corresponding SP parse tree. These representations of Cilk procedures help us explain the execution of the Nondeterminator and prove its correctness later on.

4.3 Execution of the Nondeterminator (SP-Bags Algorithm)

In this section, we describe the algorithm and data structures used by the Nondeterminator to detect determinacy races in Cilk programs.

4.3.1 Overview

The following is a high-level overview of the Nondeterminator's execution. It also introduces some of the concepts we discuss in this section.

- The `cilk2c` compiler instruments every load and store operation in the user program.
- While the user program executes, reader and writer ***shadow spaces*** are used to keep track of accesses to memory. We discuss this in more detail in Section 4.3.4.
- An SP-bags data structure based on Tarjan's least-common-ancestor algorithm maintains the series-parallel relationship between the threads. This data structure is based on the disjoint-set data structure discussed in Section 4.3.3.
- When a determinacy race is discovered, file names, line numbers, and the conflicting variables are reported.

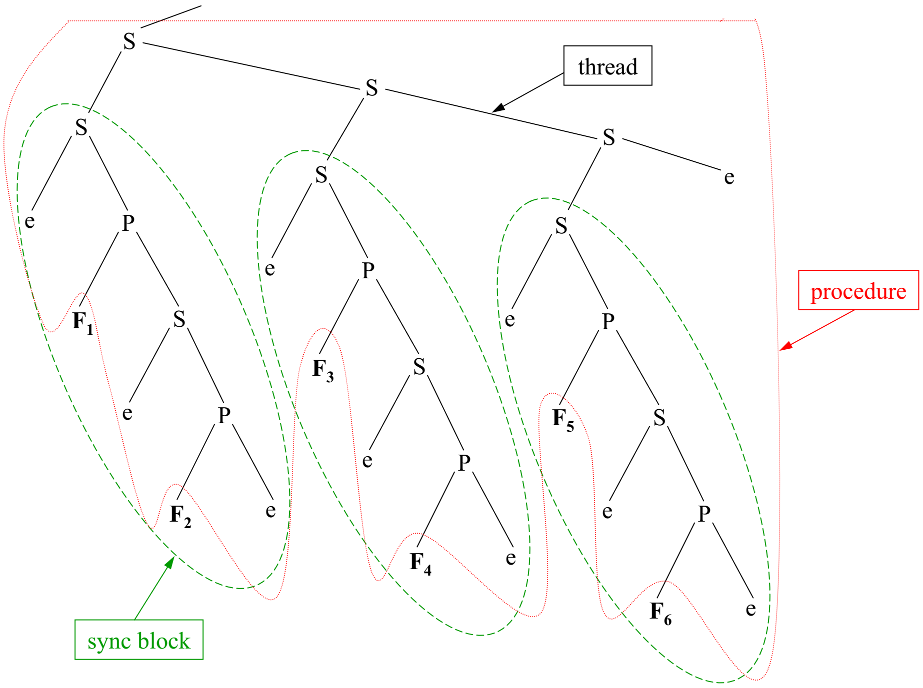


Figure 4.9: A detailed SP parse tree for a cilk procedure; threads are represented as edges, synchronization blocks are circled in green (dashed line), and the top-level procedure is circled in red (dotted line). Each F_i leaf represents an SP parse tree for another cilk procedure.

4.3.2 Properties of the Nondeterminator

Although the Nondeterminator is used to analyze parallel programs, the Nondeterminator itself is a sequential program: it executes tasks in the DFS order of the program's SP parse tree. At any given time in the Nondeterminator's execution, we can define the ***current thread*** to be the thread that is currently executing in the sequential DFS order.

The execution of the Nondeterminator maintains the following invariant: as the nondeterminator executes, it maintains two "bags" for each procedure on the call stack:

- S-Bag S_F – Contains the IDs of F's completed descendants (including F) that logically precede the current thread.

- P-Bag P_F – Contains the IDs of F's completed descendants that operate logically in parallel with the current thread.

Since the current thread changes over time, the contents of S_F and P_F also change.

Illustration of S-Bags and P-Bags

In this section, we analyze a sample Cilk procedure to better understand the notions of S-bags and P-bags.

Figure 4.10 shows the SP-dag for a Cilk procedure, which we call F_0 for ease of discussion. When the current thread being executed is e_1, the completed descendants of F_0 that logically precede e_1 are the procedures that were spawned prior to the most recent sync. Thus, $S_F = \{F_1, F_2, F_3\}$. Since there are no descendants of F_0 that have been spawned after the last sync (assuming the current thread is e_1), $P_F = \emptyset$.

When the current thread is e_2, the completed descendants of F_0 that logically precede e_2 are still the procedures that were spawned prior to the last sync. So $S_F = \{F_1, F_2, F_3\}$. However, the procedures that have been spawned after the last sync now operate logically in parallel with e_2 (since we have not reached the sync node), and thus $P_F = \{F_4, F_5, F_6\}$.

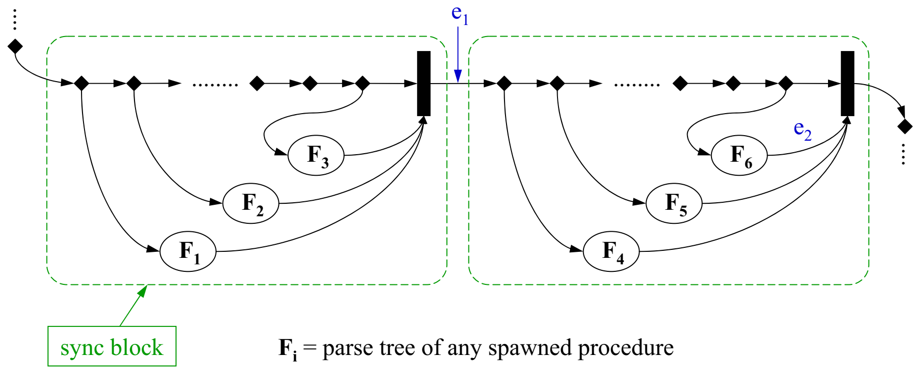


Figure 4.10: An SP-dag for a Cilk procedure.

4.3.3 Disjoint-Set Data Structures

The Nondeterminator uses a disjoint-set data structure to maintain the S and P-bags for all procedures in the call stack.

Definition 4.7 (Disjoint-Set Data Structure (UNION-FIND)) Union-Find *maintains a collection* Σ *of disjoint sets. So for two sets* X *and* Y, $X, Y \in \Sigma \Rightarrow X \cap Y = \emptyset$. *Each set* $X \in \Sigma$ *typically has a designated "leader" element* $x \in X$ *which is used to "name" the set. The data structure maintains the collection* Σ *subject to the following operations:*

- Make-Set(e)*:* $\Sigma \leftarrow \Sigma \cup \{\{e\}\}$. *This operation adds a new set* $\{e\}$ *to* Σ.
- Union(X, Y)*:* $\Sigma \leftarrow \Sigma - \{X, Y\} \cup \{X \cup Y\}$ *This removes the individual sets* X *and* Y *from* Σ *and replaces them with the union of* X *and* Y.
- Find-Set(e)*: Returns* $X \in \Sigma$ *such that* $e \in X$. *Since the sets in* Σ *are typically "named" by a distinguishing element,* Find-Set(e) *means "take me to your leader".*

The following theorem describes the performance of the above operations on the disjoint-set data structure:

Theorem 4.8 (Operations on Disjoint-Set Data Structure (Tarjan 1975)) *Any sequence of* m *operations on* n *sets can be performed in* $O(m\alpha(m, n))$ *time.*

4.3.4 SP-Bags Algorithm

The SP-Bags algorithm is the algorithm used by the Nondeterminator. It uses the disjoint-set data structure from the Section 4.3.3 to maintain the S and P-bags for all procedures in the call stack.

The SP-Bags algorithm has two types of operations. The first type updates the S and P-bags of the procedures to maintain the data structure invariant from Section 4.3.2. These operations are triggered during the DFS traversal of the Cilk procedure's SP parse tree; there is a separate operation for each type of node that is encountered in the corresponding SP-dag:

- `spawn` procedure F: $S_F \leftarrow \textsc{Make-Set}(F)$; $P_F \leftarrow \emptyset$
- `sync` in a procedure F: $S_F \leftarrow \textsc{Union}(S_F, P_F)$; $P_F \leftarrow \emptyset$
- `return` from F' to F: $P_F \leftarrow \textsc{Union}(P_F, S_{F'})$

The second type of operation uses the data structure to find determinacy races when the user program is accessing a memory location. As a Cilk program is executed, each shared memory location ℓ has two corresponding ***shadow locations*** that are updated by the S-P bags algorithm:

- $\textsc{writer}[\ell]$: the ID of the last procedure that wrote ℓ
- $\textsc{read}[\ell]$: the ID of a procedure that read ℓ (not necessarily the most recent reader)

The operations of the second type make use of these definitions:

- write location ℓ by procedure F:

  if ($\textsc{Find-Set}(\textsc{reader}[\ell])$ is a P-bag
  or $\textsc{Find-Set}(\textsc{writer}[\ell])$ is a P-bag)
  then determinacy race exists
  $\textsc{writer}[\ell] \leftarrow F$

- read location ℓ by procedure F:

  if $\textsc{Find-Set}(\textsc{writer}[\ell]$ is a P-bag)
  then determinacy race exists
  if $\textsc{Find-Set}(\textsc{reader}[\ell]$ is an S-bag)
  then $\textsc{reader}[\ell] \leftarrow F$

4.4 Nondeterminator Correctness

In this section, we prove the correctness of the Nondeterminator and the S-P bags algorithm. We begin by justifying the set manipulation operations described in Section 4.3.4 and proving some useful lemmas.

4.4.1 Justification of Set Manipulation in SP-Bags Algorithm

We show that the set manipulation operations described in Section 4.3.4 (the first type of operation in the SP-bags algorithm) maintain the data structure invariant from Section 4.3.2.

- `spawn` procedure F: $S_F \leftarrow \textsc{Make-Set}(F)$; $P_F \leftarrow \emptyset$

  Proof This operation is valid since the S-bag of F by definition contains itself, and F has no valid children yet. □

- `sync` in a procedure F: $S_F \leftarrow \textsc{Union}(S_F, P_F)$; $P_F \leftarrow \emptyset$

  Proof After a `sync` operation, we switch current threads, from some thread e before the `sync` to the single thread e' after the `sync`. Originally, P_F contains the procedures that were logically parallel to e, but when we switch to e', these procedures now logically precede the current thread (and any future procedures spawned by F). Therefore, we move the elements from P_F into S_F. □

- `return` from F' to F: $P_F \leftarrow \text{UNION}(P_F, S_{F'})$

  Proof Before a function returns, there is an implicit sync, so we know that $P_{F'}$ is empty, and $S_{F'}$ contains all the logical descendants of F'. But all logical descendants of F' are also descendants of F, and can now execute in parallel with any procedures that F might spawn in the future (before performing a sync). □

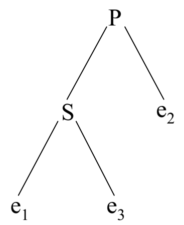
4.4.2 Some Lemmas

To help us prove the correctness of the second type of operation in the SP-bags algorithm (Section 4.3.4), we first prove some useful lemmas. We use the symbol $\prec$ to indicate an S relationship between two threads; that is, $e_1 \prec e_2 \Rightarrow LCA(e_1, e_2)$ is an S-node. Similarly, we use the $\|$ symbol to indicate a P relationship between two threads; that is, $e_1 \| e_2 \Rightarrow LCA(e_1, e_2)$ is a P-node.

Lemma 4.9 *Let threads e_1, e_2, and e_3 execute serially in order. If $e_1 \prec e_2$ and $e_1 \| e_3$, then $e_2 \| e_3$.*

Proof Suppose for sake of contradiction that $e_2 \prec e_3$. Then, by transitivity, $e_1 \prec e_3$, which is a contradiction. □

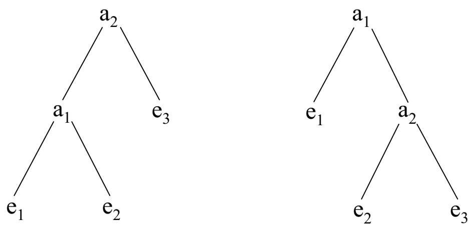
Is the $\|$ relation transitive? By considering a simple parse tree, we see that the answer is no:



In the tree above, $e_1 \| e_2$ and $e_2 \| e_3$, but $e_1 \not\| e_3$. However, the following lemma shows that the $\|$ relationship has a weaker property called ***pseudotransitivity***.

Lemma 4.10 (Pseudotransitivity of $\|$) *Let threads e_1, e_2, and e_3 execute serially in order. If $e_1 \| e_2$ and $e_2 \| e_3$, then $e_1 \| e_3$.*

Proof Since we do a depth-first traversal of the parse tree, the only possible options for the tree that enable us to execute the three threads in order are:



In both of the cases above, we see that $a_1 = LCA(e_1, e_2)$ and $a_2 = LCA(e_2, e_3)$. From these observations, we conclude that a_1 and a_2 must both be P-nodes. Since $LCA(e_1, e_3)$ is either a_1 or a_2, $e_1 \parallel e_3$. $\square$

Before stating the next lemma, we define the "procedurification" function h, taken from [13]. From our construction of an SP parse tree for a given Cilk dag (described in Section 4.2.3), we see that each procedure in the spawn tree is represented by an assembly of threads and internal nodes in the parse tree [13]. The procedurification function h maps threads or nodes in the parse tree to procedures in the spawn tree.

The next lemma makes use of the procedurification function to relate the S and P-nodes of an SP parse tree to the contents of the S and P-bags during the execution of the SP-bags algorithm.

Lemma 4.11 *Let e_1 be executed before e_2 serially, and let node $a = LCA(e_1, e_2)$ in the parse tree.*

- $e_1 \prec e_2 \Rightarrow h(e_1) \in$ S-bag$(h(a))$ when e_2 executes (i.e. is the current thread).
- $e_1 \parallel e_2 \Rightarrow h(e_1) \in$ P-bag$(h(a))$ when e_2 executes.

Sketch of Proof We present a sketch the proof here. For more details, please refer to Lemma 8 in [13].

Case 1: $e_1 \prec e_2 \Rightarrow a$ is an S-node.

1. If a belongs to the spine, then e_1 belongs to a's left subtree and e_2 belongs to a's right subtree. When e_2 is executed, which bag is $h(e_1)$ in? From our construction of the tree, either $h(e_1) = h(a)$ or $h(e_1)$ is a descendant of $h(a)$. Between the execution of e_1 and the execution of e_2, operations that move $h(e_1)$ are `sync` and `return`. This implies that we never move the procedure ID down the spawn tree; instead, $h(e_1)$'s ID moves up whenever any of its ancestors returns.

2. If a belongs to a sync block, then e_1 is in procedure F, and $h(e_1) = h(a) = F$. F is automatically placed in S_F when F is spawned.

Case 2: $e_1 \parallel e_2 \Rightarrow a$ is a P-node.

In this case, a must be in a sync block, since only S-nodes are on the spine. e_1 belongs to the left subtree of a. Note that the left child of a P-node is always a spawned procedure that is placed in P_F when F returns. At this point, a sync has not yet occurred, so $h(e_1)$ is in P_F when e_2 is executed.

$\square$

4.4.3 Proof of SP-Bags Race Detection

In this section, we use the lemmas presented in Section 4.4.2 to prove the correctness of the SP-bags algorithm used by the Nondeterminator.

Theorem 4.12 *The SP-bags algorithm detects a determinacy race in a Cilk program if and only if a determinacy race exists.*

Sketch of Proof We provide a sketch of the proof here. For more details, please refer to Theorem 10 in [13].

The $\Rightarrow$ case is straightforward.

To prove the other direction, let $e_1 \parallel e_2$ have a race on ℓ, and assume e_1 executes before e_2. If there are several races on ℓ, choose e_2 to be determinacy race whose second thread executes earliest in the serial execution order. There are three possible types of determinacy races:

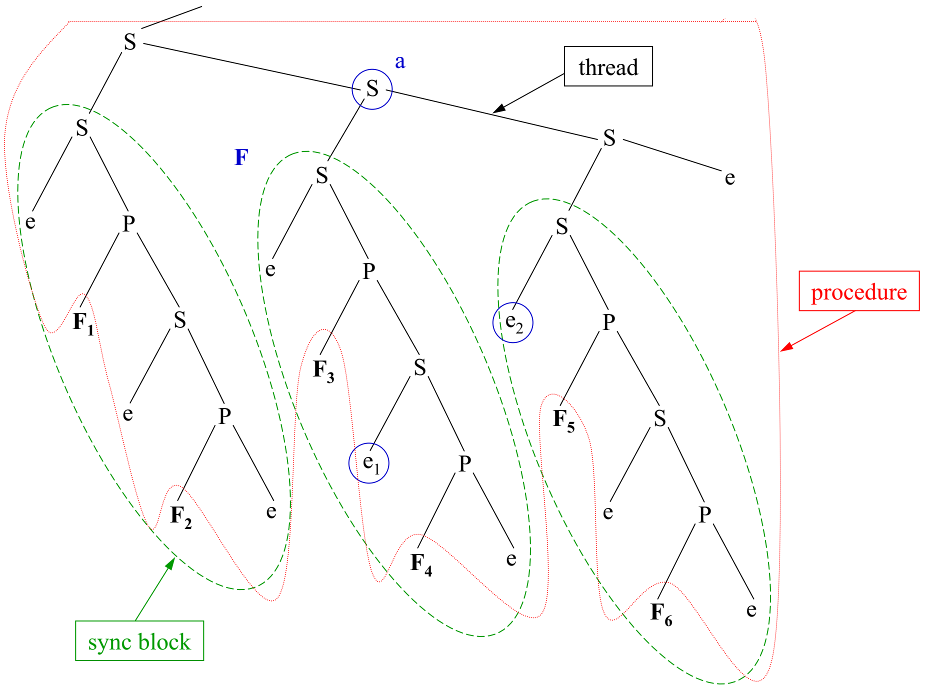


Figure 4.11: Case 1 in the proof of Lemma 4.11.

- Case 1: e_1 writes ℓ, e_2 reads ℓ

  Suppose when e_2 executes, $\text{WRITER}[\ell] = h(e)$ for some thread e. If $e = e_1$, we are done, since $h(e_1) \in$ P-bag$((LCA(e_1, e_2)))$. If $e \neq e_1$, then e is executed after e_1 and before e_2.

  1. $e_1 \prec e$: Then $e \parallel e_2$ by Lemma 4.9.
  2. $e_1 \parallel e$: Then a write/write race exists between e_1 and e, which contradicts the minimality of e.

- Case 2: e_1 writes ℓ, e_2 writes ℓ

  This case is similar to Case 1.

- Case 3: e_1 reads ℓ and e_2 writes ℓ

  This case is slightly more involved than the other two. Suppose when e_2 is executed, $h(e) = \text{READER}[\ell]$. The main idea is that if $e \parallel e_1$, then e_1 did not set $\text{READER}[\ell]$. However, this is okay since we catch $e \parallel e_2$ by pseudotransitivity (Lemma 4.10). If $e_1 \prec e$, then $e_1 \parallel e_2 \Rightarrow e \parallel e_2$.

$\square$

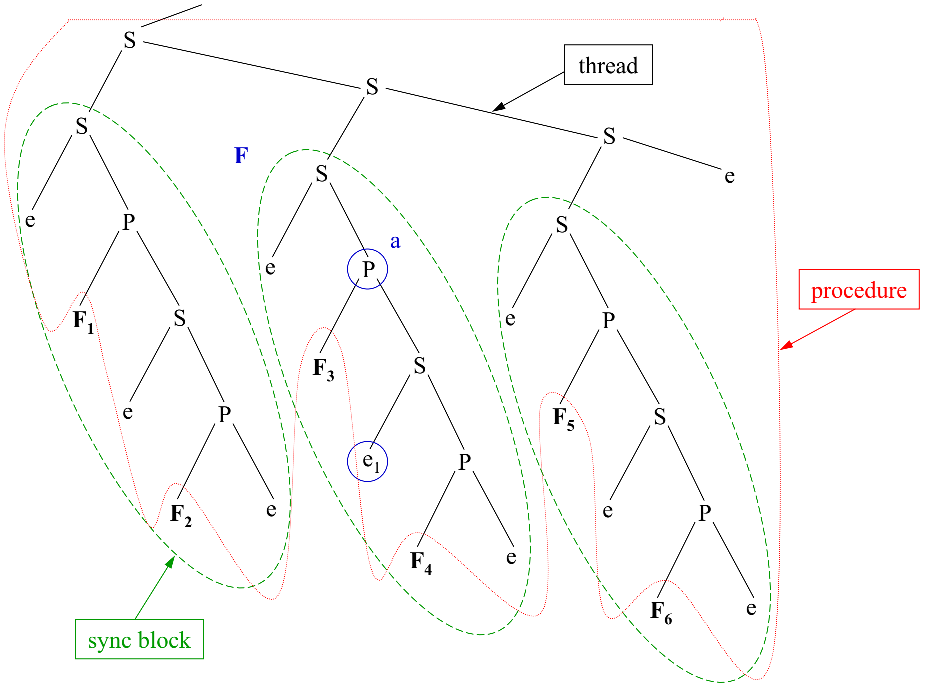


Figure 4.12: Case 2 in the proof of Lemma 4.11.

6.895 Theory of Parallel Systems	Lecture 5
Consistency of the Memory Sub-System	
Lecturer: Bradley C. Kuszmaul	*Scribe: Alexandru Caracas, Rui Fan*

Lecture Summary

1. *Mutual Exclusion*
   Introduction to the problems of shared-memory parallel systems and definition of mutual exclusion.
2. *Mutual Exclusion Algorithms*
   Description of the mutual exclusion algorithms of Dekker and Dijkstra.
3. *Memory Consistency*
   Definition of memory consistency in terms of atomic operations.
4. *Memory Subsystem of Hardware Architectures*
   Description of various memory operation primitives such as Load-Link-Store-Conditional and Transactional Memory.

5.1 Mutual Exclusion

The underlying problem of shared memory parallel systems is that we have more than one thread and shared memory locations, accessed by all concurrent executing threads. The question is how do the threads interact?

Example 1 One example Cilk code which contains a simple determinacy race, already discussed in the previous lectures, is the following:

```
cilk int foo (void) { x++; }
cilk int main (int argc, char *argv[]) {
   spawn foo();
   spawn foo();
   sync;
}
```

Once we detect the determinacy race, e.g. by using the Nondeterminator, our goal is to eliminate it. In order to avoid the determinacy race we have to make sure that only one thread at a time can perform the operation `x++`. As a start we could consider the following idea. Simply prevent context switching, for example by turning off the interrupts as in the following MIPS assembler code:

```
DI              ; turn off interrupts
LW R1, (0)R2
ADD R1,R1,1
SW R1, (0)R2
EI
```

However, there are a number of downfalls with using interrupts for mutual exclusion:

- It only works on a uniprocessor machine. On a multiprocessor machine the other processors are not affected by the `DI` instruction.

- It only works in kernel (not user) mode. On most architectures there is no user-level call that can disable interrupts.
- And it also prevents other threads from doing non-critical work, which is extremely dangerous, since the operating system time slice interrupt will never take place, and the operating system would fail gaining control.

Definition 5.1 *In a more general conceptual setting, mutual exclusion can be described in the following algorithmic setup:*

```
while (1){
   trying();
   critical_section()
   releasing();
   noncrit();
}
```

In an infinite loop each thread tries to execute `critical_section()`, that is the sequence of operations which need to be executed atomically. In order to be allowed to execute the critical section the thread first needs to get permission. In the above example the task of allocating the right to enter the critical section is given by the part `trying()`. Moreover the thread needs to be cooperative and release hold of its right to enter the critical section in `releasing()`. Consequently it performs the rest of its non-critical task `noncrit()`.

Now the problem is to write the functions `trying()` and `releasing()` so that at most one thread is in `critical_section` at a time. One extremely trivial solution would be:

Example 2 `trying(void) { halt(); }`

The trivial solution is not good since it simply stops the currently executing thread. This illustrates a problem with our specification of mutual exclusion. In addition to the algorithmic description of mutual exclusion from Definition 5.1 we now introduce some additional definitions.

Definition 5.2 *Mutual exclusion requires that there is* **Progress**. *If no thread is executing* `critical_section` *and there are some threads which are trying to enter* `critical_section` *(in our example by executing* `trying()`*) then some thread will do* `critical_section`.

Apart from the definition of mutual exclusion with guaranteed progress there are other aspects which are desirable, however not strictly required for *Mutual Exclusion*, such as:

Definition 5.3 No Starvation. *Every executing thread gets a fair chance of executing in the* `critical_section`.

Example 3 An alternate idea for the mutual exclusion problem in the case of two threads would be:

```
int turn=1;
int run(int i) {
   // i is the thread number for the currently executing thread,
   // 0 for one thread and 1 for the other
   while (turn!=i);
   crit();
   turn=1-i;  // it is the other thread's turn
   noncrit();
}
```

However, the problem with the previous approach is that it does not guarantee progress. If, for example, `noncrit()` is an infinite loop in thread 1, then thread 0 would get stuck.

5.2 Mutual Exclusion Algorithms

This section presents two algorithms that solve the mutual exclusion problem. The first one is due to Dekker and it solves the mutual exclusion problem for two threads. The second is Dijkstra's algorithm for mutual exclusion, which is a generalization applicable to any number of threads.

5.2.1 Dekker's Algorithm

A solution which works for two threads and uses turns was described by Dekker. The following C code presents his algorithm:

```
int turn;
int wants[2];

// i is the current thread, j=1-i is the other thread
while(1) {                 // trying
   wants[i] = TRUE;
   while (wants[j]) {
      if (turn==j) {
         wants[i] = FALSE;
         while (turn==j) ; // empty loop
         wants[i] = TRUE;
      }
  }
  critical_section();
  turn=j;                  // release
  wants[i] = FALSE;
  noncrit();
}
```

As a visualization of Dekker's algorithm consider the following railroad from Figure 5.1. Each thread has its own railroad system which intersects with the railroad of the other thread only on the `critical_section`. There are a number of switches which the threads activate as they traverse the railroad. Whenever a thread encounters a semaphore it can change its course according to the indication of the semaphore. The threads "move" in the direction shown by the arrows.

As an illustration, assume that thread 0 starts at the top of its railway system and wants to do `critical_section`. It will first set `turn=1`, and then `wants[0]=TRUE`. Next it will encounter a semaphore which tells it whether thread 1 wants to enter the critical section. Suppose that thread 1 also wants to enter, then thread 0 will switch tracks, and "move" downwards. On this new track it will set `wants[0]=FALSE` and go and check the next semaphore. Assume that now `turn=0`, then thread 0 will remain on this track, and set `wants[0]=TRUE`. Also assume that now thread 1 does not want to enter the `critical_section`. Then thread 0 can go and execute `critical_section`.

5.2.2 Dijkstra's Algorithm

Dijkstra devised a mutual exclusion algorithm for any number of threads. The following are the conditions for Dijkstra's mutual exclusion problem:

- Only 1 thread may execute `critical_section()` at a time.
- The threads execute in an infinite loop trying to enter `critical_section`.
- *Atomic read* and *atomic write* are available, but no other instructions (no locking instructions).

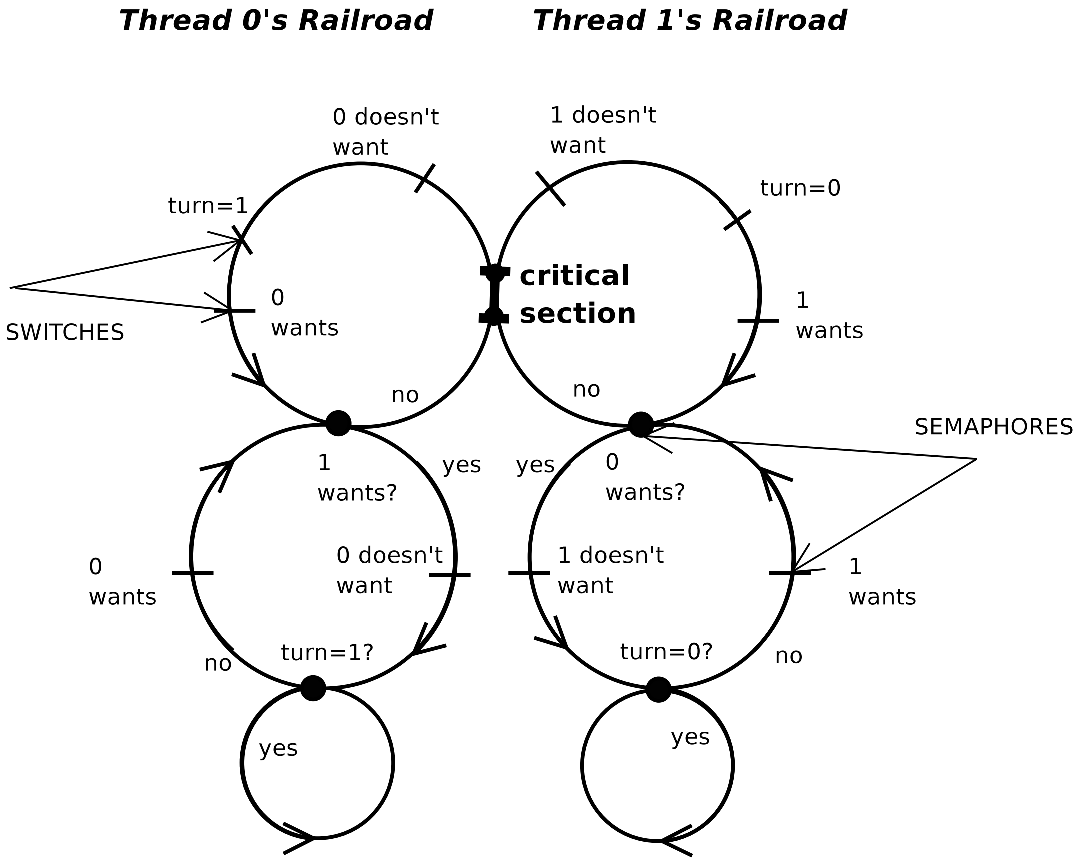


Figure 5.1: Railroad visualization of Dekker's algorithm for mutual exclusion. The threads "move" in the direction shown by the arrows.

- Progress condition is satisfied.
- No assumptions about relative speeds of threads.

The following is the C code for Dijkstra's Algorithm:

```
char b[N], c[N];  // b[i] and c[i] are only be written by thread i, initialize true
int  k;           // 0 <= k < N

while (1) {
   // i is my own thread number
   b[i] = FALSE;
L: if (k!=i) {
      c[i] = TRUE;
      if (b[k]) k = i;
      goto L;
   }
    else {
      c[i] = FALSE;
      for (j=0; j<N; j++) {
         if (j!=i && !c[j]) {
            goto L;
         }
      }
   }
   crit();
   c[i] = TRUE;
   b[i] = TRUE;
   noncrit();
}
```

5.3 Proof of Dijkstra's Algorithm

We first show that Dijkstra's algorithm satisfies mutual exclusion. Suppose for contradiction that there are two processes i and j in the critical section at the same time. Then $c[i] = c[j] = \text{FALSE}$. Suppose, without loss of generality, that i was the first to set $c[i] = \text{FALSE}$. Then, j must see $c[i] = \text{FALSE}$ in the for loop after assigning $c[j]$, hence j goes to L. Then, j does not enter the critical section, which is a contradiction. Thus, there is at most one process in the critical section at once.

To show that progress is always made, suppose several processes try to enter the critical section. Then each such process i sets $b[i] = \text{FALSE}$. We claim that at least one trying processes advances into the else clause. Consider all the trying processes for which the test "if $(b[k])$" succeeds. There is at least one such process. Consider the last process i among this set to set $k \leftarrow i$. Then i will enter the else clause.

Suppose only one process i enters the else clause. Then i finds $c[j] = \text{TRUE}$ for all $j \neq i$, and enters the critical section. Next, suppose a set P of more than one process enters the else clause. They may all find $c[j] = \text{FALSE}$ for some other $j \in P$, and goto L. But now, for every process $i \in P$, $b[i] = \text{FALSE}$, and so i cannot reset $k \leftarrow i$. On the other hand, for some $i \in P$, we have $k = i$. Then i will be the only process to go into the else clause[1], where it finds $c[j] = \text{TRUE}$ for all other j, and so can enter the critical section. Thus, Dijkstra's algorithm ensures progress.

[1] Actually, another (slow) process j can set $k \leftarrow j$ and enter the else clause with i. But then we can repeat the previous argument. Each time we repeat the previous argument, we eliminate at least one trying process from going into the else clause, while ensuring there is still at least one process which can go into the else clause. Since the number of trying processes is finite, we are eventually left with exactly one process which can go into the else clause.

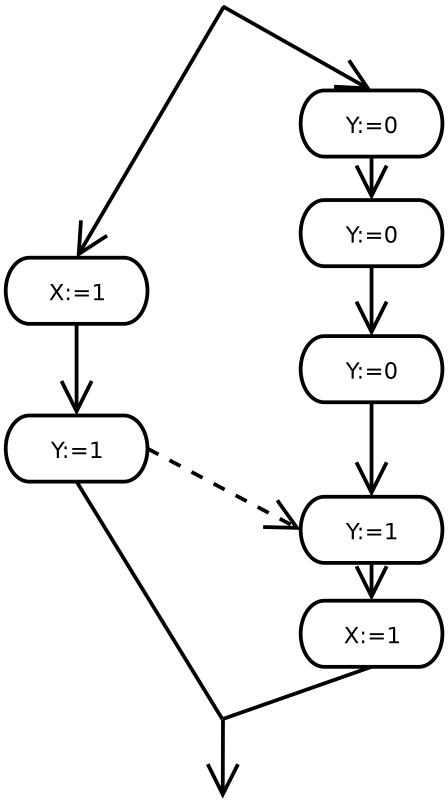


Figure 5.2: A DAG for the multithreaded program in Example 4. The solid arrows represent dependencies induced by the program order. The dashed arrow represents an implicit dependency of the execution.

5.4 Memory Consistency

Notice that Dijkstra's algorithm depends on a property of the memory system, which is that all the processes read the last value written to a memory location. This is called *atomic read* and *atomic write*. Here is an example to illustrate the idea:

Example 4

```
  main {
    X = 0; Y = 0;
    spawn thread0();
    spawn thread1();
    sync }

 thread0() {                thread1() {
    X = 1;                     while (Y == 0);
    Y = 1 }                     print(X) }
```

If the reads and writes are atomic, then the value that is printed for X must be 1. This is because to exit the while loop in thread 1, Y must be set to 1. But if writes are atomic, this means that X is already set to 1. We can draw each computation as a DAG, where the arrows point from earlier to later events. For example, one DAG for an execution of the above example is shown in the Figure 2.

As a side note, we define an *observer function* as a mapping that gives, at each point in the execution, for each memory location, which write operation wrote the value that a process sees if it reads that location. A *consistency model* is a set of allowable observer functions. An execution is consistent with the consistency model if it can be observed by an observer function of the model. A hardware implementation of memory is consistent with a consistency model if all its executions are consistent with the model. We now define a consistency model for memory called *sequential consistency.*

Definition 5.4 *Let G be a DAG of an execution α of a multithreaded program. Then α is* sequentially consistent *if and only if there is a topological sort S of G such that for each read, the most recent write to that memory location according to S wrote the value that the read received.*

Note that there can be more than one topological sort of G demonstrating consistency. All that Definition 5.4 requires is the existence of some sort consistent with the execution. Also, note that for sequential consistency, the allowable observer functions are the ones for which a topological sort exists on the DAG such that the value for a location seen is the one most recently written to that location.

5.5 Memory Subsystem of Hardware Architectures

The MIPS multiprocessors guarantee sequential consistency. However, the x86 multiprocessors do not. The reason is that CPU's pipeline instructions and can execute them out of order. Therefore, even though a read may occur later than a write in code, the read can actually be executed before the write. It is complicated to do out of order execution and guarantee strong consistency.

Instead of sequential consistency, Intel offers a *barrier* operation. Specifically, the instruction MFENCE forces all reads and writes before an MFENCE to finish before any reads and writes after the MFENCE can start. However, the Intel MFENCE instruction is very slow. On a 2.5Ghz Pentium 4, one MFENCE instruction takes 120ns. Interestingly, the barrier instruction on a 1.5Ghz AMD64 Opteron takes only 8ns.

5.5.1 Additional Memory Primitives

As we saw, mutual exclusion is very complicated to do using only atomic read and write memory. Thus, most machines provide additional memory instructions to make mutual exclusion easier. One common instruction is *test-and-set.* The following pseudocode demonstrates what test-and-set does:

Example 5

```
test_and_set(int *l) {
  atomically {
    int old_value = *l;
    *l = 1;
    return old_value;
  }
}
```

As the example shows, test-and-set atomically reads the old value from a memory location and writes a 1 to that location. To demonstrate its usefulness, consider the following simple algorithm for mutual exclusion using test-and-set.

Example 6

```
while (test_and_set(&lock));
critical_section();
lock = 0;
```

The first process which gets a 0 from test-and-set can enter the critical section. All the other processes have to wait till that process is done and resets lock.

Another popular memory instruction is *load-linked store-conditional (LLSC)*. This is provided, for example, on SGI. LLSC is illustrated in the following code:

Example 7

```
again:
  LL R1, R2       // R1 = *R2
  compute
  SC R1, R2       // *R2 = R1 but only if *R2 has not changed since LL
  bz again:       // the zero flag is set so we can do it again if it failed
```

As the example shows, LLSC lets a process read a memory location, do some computation, then write a value back to that memory location. But the write only succeeds if the location didn't change since the read. If the location did change, LLSC sets the zero flag to let the process know to try again. LLSC lets us increment a variable without a lock, as shown in the following example. Note that this code performs the increment directly, without attempting to get mutual exclusion on the variable. Also note that the algorithm may not make progress, if for example, two processes continually perform LLSC alternately.

Example 8

```
again:
  LL R1, R2
  ADD R1, R1, 1
  SC R1, R2
  bz again
```

To end this lecture, we mention a few other mutual-exclusion type memory operations offered by many architectures. The most important among these is *compare-and-swap (CAS)*. The operation CAS(x, y, v) does the following. If $x = y$, then the operation sets $x \leftarrow v$. Otherwise, it returns an error code saying the CAS failed. Some architectures also offer *double compare-and-swap (DCAS)*. Similar to CAS, DCAS$(x_1, x_2, y_1, y_2, v_1, v_2)$ compares x_1 to y_1, and x_2 to y_2. If both sets of comparisons return true, then DCAS sets $x_1 \leftarrow v_1$, and $x_2 \leftarrow v_2$. Otherwise, it returns an error code saying the DCAS failed.

Another approach to ensuring atomicity is to use *transactional memory*. This technique performs a sequence of memory operations atomically, all as part of a transaction. This allows a simple and uniform way of implementing many complex concurrent data structures.

6.895 Theory of Parallel Systems	Lecture 6
Analyzing Space Bounds	
Lecturer: Charles Leiserson	*Scribe: Jeremy Fineman and Siddhartha Sen*

Lecture Summary

1. *Bounds on space requirements*
   We look at upper and lower bounds on the space requirements of parallel Cilk programs.
2. *Memory Allocators*
   We compare different types of heap storage memory allocators, and we examine the Hoard memory allocator in detail.
3. *Project Ideas*
   We present several project ideas based on the topics in these notes.

6.1 Bounds on space requirements

In this section, we look at upper and lower bounds on space requirements for Cilk programs. We begin with a general upper bound and then consider a tighter bound for a specific matrix multiplication algorithm. Next, we look at a lower bound on general multithreaded computations. Finally, we mention the Blelloch-Narlikar algorithm, which provides a better upper bound on space for Cilk-like dags.

General upper bound

The Cilk scheduler is a busy-leaves scheduler. As a result, we can show an upper bound on space which is linear to P, the number of processors in the system.

We can view the execution of a parallel program as a cactus stack. Let S_1 be the amount of space required to execute the program on a single processor. Figure 6.1 shows that S_1 can never exceed the maximum height of the stack. When run on a P-processor system, the Cilk scheduler ensures that there are at most P leaves allocated. From Figure 6.1, we see that there are at most P stacks that have been allocated by the system, where each stack has a maximum height of S_1. Thus, we derive the general upper bound of $S_P \leq P \cdot S_1$.

Applying the general upper bound on space

We apply the general upper bound for space requirements to the divide-and-conquer matrix multiplication algorithm. Each iteration of this algorithm on an $n \times n$ matrix spawns 8 multiplications of submatrices of size $n/2 \times n/2$. For each of these multiplications, we need a temporary matrix to store the result.

For serial execution, we can describe the total space used with the following recurrence, illustrated by the recursion tree in Figure 6.2:

$$\begin{aligned} S_1(n) &= S_1\left(\frac{n}{2}\right) + n^2 \\ &= \Theta(n^2) \ . \end{aligned} \tag{6.1}$$

Therefore, a P-processor execution uses $O(P \cdot S_1) = O(Pn^2)$ space.

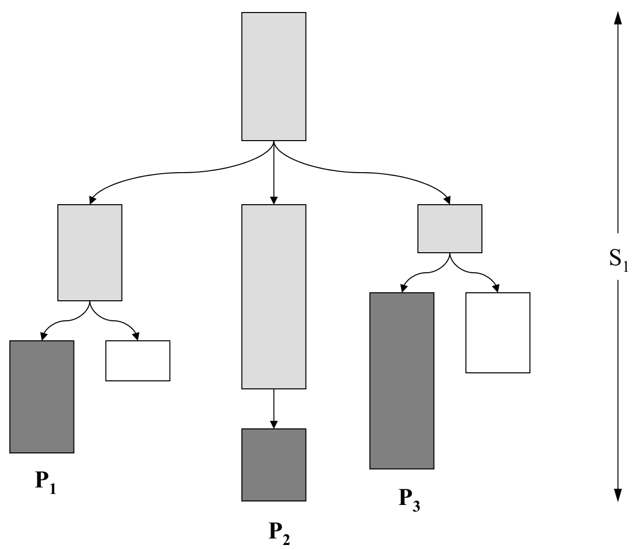


Figure 6.1: A cactus stack. Cilk uses a busy-leaves scheduler, which means that at most P leaves are scheduled at any given time. The shaded stack frames represent allocated memory, and the darker frames represent leaves currently being executed by the processors. In this example, $P = 3$.

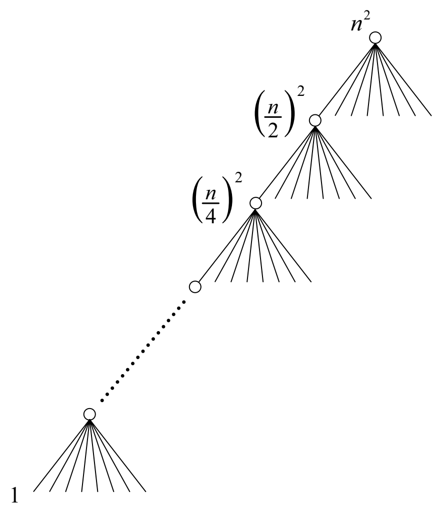


Figure 6.2: The recursion tree for the divide-and-conquer matrix multiplication algorithm on a single processor. The values shown indicate the space used at each level.

Stronger bound for divide-and-conquer matrix multiplication

We can derive a better bound on space for the divide-and-conquer algorithm. The general upper bound we derived earlier counts the maximum height (S_1) of the stack P times. As a result, many of the frames in the shallower levels are counted more than once; the top level frame, for example, is counted P times, but its space is only allocated once. Since shallower levels require more space than deeper ones, the problem of double-counting is worse towards the top of the tree. This poor accounting results in a much weaker bound than we can achieve.

Consider the worst case for the divide-and-conquer algorithm. Given that space requirements drop drastically at lower levels of the tree, the worst case is fully parallel at the top until there are P leaves, then serial thereafter. The recursion tree in /figrefmmultworstcasespace illustrates this transition. We can solve the recursion tree by considering the space used at each level. The top level requires n^2 space. The next level requires $8(n/2)^2$ space. If we continue this patten, the level with P leaves requires $8^{\log_8 P}(n/2^{\log_8 P})^2 = P(n/P^{1/3})^2$ space, since the depth of this level is $\log_8 P$. The serial execution following from each of the P leaves requires only a constant times $(n/P^{1/3})^2$ space. Summing the space for all the levels yields $O(n^2 P^{1/3})$ total space.

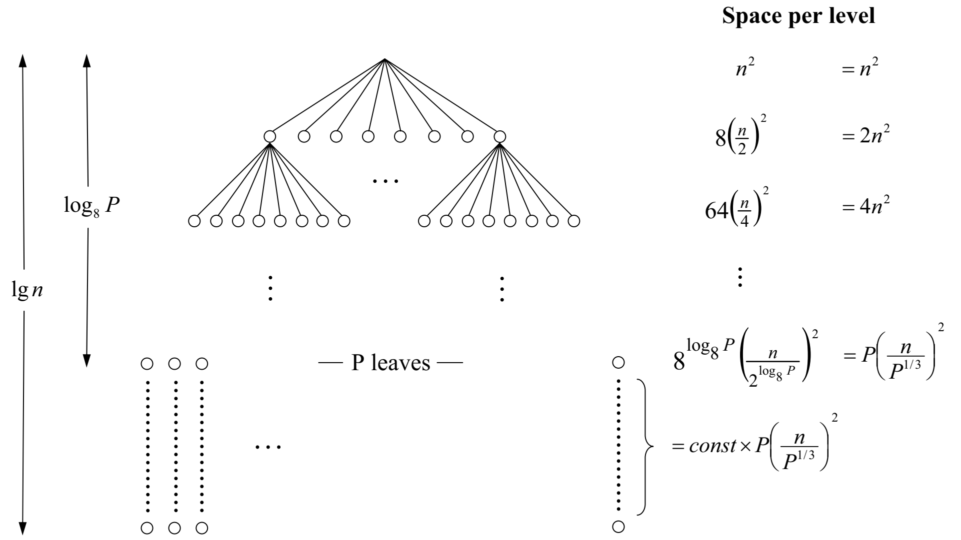


Figure 6.3: The worst case for the divide-and-conquer matrix multiplication algorithm running on P processors. The execution is fully parallel until we reach P leaves, after which the execution continues serially on each processor (since Cilk uses a busy-leaves scheduler).

Lower bound

We now consider a lower bound on space for computation dags. We know that Cilk produces series-parallel execution dags. Earlier, we showed that a busy-leaves scheduler results in an expansion of space that is linear to the number of processors—that is, $S_P \leq P \cdot S_1$. Both of these properties are necessary to get good space bounds; without these properties, we can find a multithreaded computation such that $T_P \leq T_1/2$ implies $S_P \approx \sqrt{T_1}$. In other words, space grows with work.

Definition 6.1 (Depth-first computation dag) *In a depth-first computation dag, a left to right depth-first traversal visits all threads on which a given thread depends before that thread. In other words, there exists a serial execution for the dag. The dag does not have to be series-parallel.*

Figure 6.4 shows an example of a depth-first computation dag. The edge from node K to D cannot exist in the graph because it violates the dependency condition above; namely, that a thread (node D) is only visited after all threads on which it depends (nodes K and C) are visited first. Clearly, a left to right depth-first traversal cannot visit both C and K before visiting D.

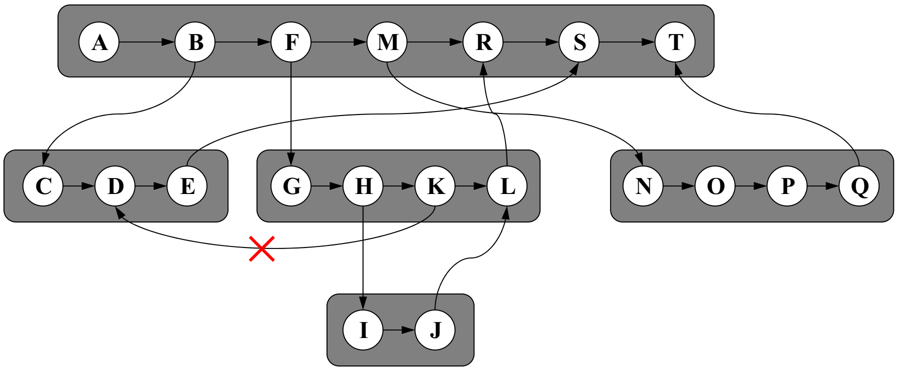


Figure 6.4: A depth-first computation dag. The nodes represent threads of execution, and the arrows represent the continuation, spawn, and return edges. An alphabetical listing of the nodes illustrates a left to right depth-first traversal of the threads.

Theorem 6.2 *For $S_1 \geq 4$ and $T_1 \geq 16S_1^2$, there exists a depth-first computation dag with work T_1, critical path length $T_\infty \leq 8\sqrt{T_1}$, and stack depth S_1 such that $\forall \rho$ and $1 \leq \rho \leq 1/8T_1/T_\infty$, if $T_P \leq T_1/\rho$, then $S_P \geq 1/4(\rho - 1)\sqrt{T_1} + S_1$.*

Proof Idea In the dag in Figure 6.5, there are dependencies from the last threads of each of the C_i to the first threads of C_{i+1}. To get speedup, one must continually spawn new procedures, but they stall immediately, consuming space in the form of activation records. For details of this proof, see [6]. □

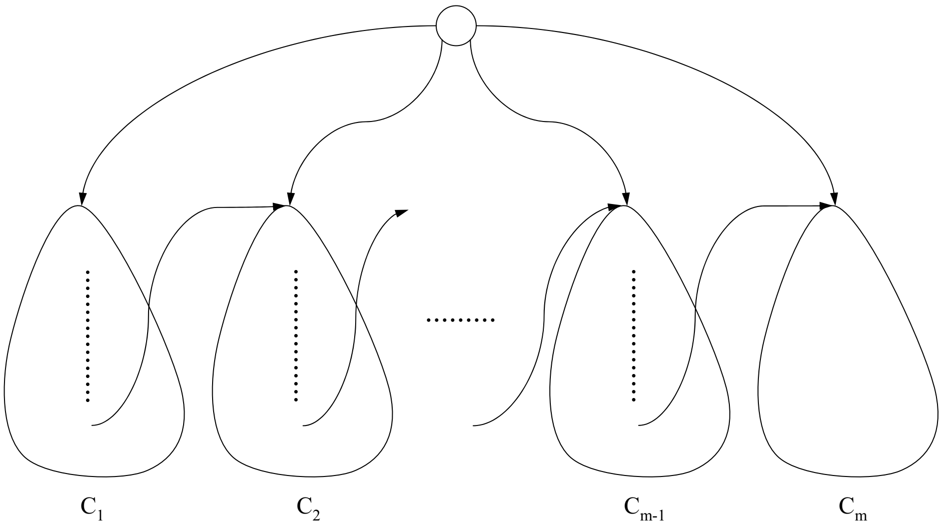


Figure 6.5: Depth-first computation dag for the proof of Theorem 6.2.

Blelloch-Narlikar algorithm

Blelloch and Narlikar prove a bound of $S_P \leq S_1 + O(P \cdot T_\infty)$ for Cilk-like dags. The main idea of their algorithm is to execute the P leftmost leaves of the computation dag in depth-first order.

Unfortunately, this approach has poor locality because if a new thread spawns on the left, then we have to rearrange processors to work on that thread. We can improve locality by executing any P of the nP leftmost leaves, where n is some tunable parameter.

6.2 Memory Allocators

In this section, we look at different types of heap storage memory allocators for parallel systems and examine the strengths and weaknesses of each. The main idea in designing a memory allocator is to ask the operating system for space as infrequently as possible. Once space is allocated by the OS, we can keep reusing it. In a serial allocator, memory is allocated to a process in chunks that are powers of 2 in size.

We use four measures to compare the different memory allocators:

- ***Overhead***, T_1/T_{serial}, is the increase in time that results from running a serial program in a parallel memory allocation environment.
- ***Scalability***, T_1/PT_P, reflects the additional time incurred by the parallel memory allocator (on P processors).
- ***False sharing*** occurs when two processors share the same cache line. If both processors write a different variable in the same line, there is no actual contention, but the memory system may behave as if a true conflict exists.
- ***Fragmentation*** is the ratio of memory allocated to memory used. When a thread performs a `malloc` followed by a `free`, we consider the memory to still be allocated. Our goal is to minimize the ratio A/U, where A is the maximum amount of space allocated so far, and U is the maximum amount of space used so far.

Global heap

In a global heap allocator, all processors share a single heap with a global lock. The overhead of allocating memory in this scheme is the global lock. Scalability is poor due to lock contention. Furthermore, false sharing is high for small objects as they are more likely to occur on the same cache lines. Fragmentation, on the other hand, is very low; it is comparable to that of a serial processor.

Local heap

A local heap allocator has a separate heap for each thread. Since there are no locks, local heaps do not introduce any overhead, and scalability is excellent. False sharing is minimal, but it can still occur when one thread passes memory to another. Fragmentation, on the other hand, can grow without bound.

In general, the way fragmentation grows is through ***blow-up*** or ***memory-drift***. Suppose one thread allocates a variable x_1, but a different thread frees it. If this behavior repeats, the second thread will eventually have a huge free list, while the first will continue to allocate memory from the OS.

Local heap with ownership

Similar to a local heap allocator, each thread in this scheme has a separate heap, but data is always returned to the thread that allocated it when it is freed. This scheme has additional overhead due to the synchronization between heaps. The scalability and false sharing are comparable to those of the local heap allocator. Fragmentation, however, approaches P rather than growing without bound.

The Hoard allocator

We now consider the Hoard allocator, designed by Berger et. al. [5]. The Hoard uses P local heaps and one global heap. Memory is allocated in page-sized ***superblocks***; we let S be the superblock size. Without loss of generality, we can treat all allocated objects as being the same size.

The Hoard `malloc` algorithm looks like the following:

`malloc`:
  if there exists a free object in local heap
    then return it
  else if there exists a superblock in global heap
    then move it to local heap
    return an object
  else
    allocate a superblock from the OS
    move superblock to local heap
    return an object .

When the fraction of free memory in the local heap is too big, the Hoard removes a mostly empty superblock from the local heap and returns it to the global heap. Let u_i be the space used in the i-th heap, and let a_i be the storage allocated to that heap. Let f and K be tunable parameters, usually set to about $1/4$ and 4 respectively. If $u_i < (1-f)a_i$ and $u_i < a_i - KS$, then we move an f-empty superblock to the global heap.

The Hoard allocator maintains the following invariant:

Invariant 6.3 $u_i \geq a_i - KS$ *or* $u_i \geq (1-f)a_i$.

The Hoard maintains Invariant 6.3 because moving a superblock reduces a_i by S and u_i by at most $(1-f)S$.

The Hoard always allocates from the fullest superblock. It maintains a constant overhead by using a binning strategy. In other words, it keeps a table that maps each superblock to the number of objects in that superblock. Whenever a local heap releases a superblock, it can lookup the emptiest superblock from the table in constant time.

Theorem 6.4 *Recall that A is the maximum memory allocated up to now, and U is the maximum memory in use up to now. Then $A = O(U + P)$.*

Proof Consider a time t when the last allocation from the OS occurred. At that time, the global heap must have been empty, or we would have passed a superblock from the global heap rather than allocating it from the OS. Thus, we have $A = \sum_{i=1}^{P} a_i(t)$ and $a_0(t) = 0$, where a_0 is the storage allocated in the global heap.

All heaps must satisfy either $u_i \geq a_i - KS$ or $u_i \geq (1-f)a_i$, from Invariant 6.3. We can rearrange these inequalities to read $u_i + KS \geq a_i$ and $u_i/(1-f) \geq a_i$ respectively. Without loss of generality, we assume that the heaps that satisfy the first term of the invariant are numbered $1 \ldots R$, while those that satisfy the second term are numbered $R+1 \ldots P$. If any heap satisfies both terms of the invariant, we place it in either group. Therefore, we have the following:

$$
\begin{aligned}
A &= \sum_{i=1}^{P} a_i(t) \\
&= \sum_{i=1}^{R} a_i(t) + \sum_{i=R+1}^{P} a_i(t) \\
&\leq \sum_{i=1}^{R} (u_i(t) + KS) + \sum_{i=R+1}^{P} \left(\frac{u_i(t)}{1-f}\right) \\
&\leq PKS + \frac{U}{1-f} \\
&= O(U+P)\ . \qquad (6.2)
\end{aligned}
$$

□

From Equation (6.2), the fragmentation or blow-up is at most $A/U = O(1 + P/U)$. This bound reduces to $O(1)$ if $P = O(U)$, which is usually a fair assumption.

6.3 Project Ideas

The following is a list of project ideas based on some of the concepts covered in this lecture:

- We can introduce some *slop* to the Blelloch-Narlikar algorithm to increase locality. For example, we can look at the $2P$ leftmost leaves rather than just the P leftmost leaves. This topic is open for further exploration.
- The Hoard allocator has only one global heap. Perhaps we can get a better bound by introducing more heaps or a tree of heaps.

6.895 Theory of Parallel Systems	Lecture 7

Memory Contention

Lecturer: Charles Leiserson *Scribe: C. Scott Ananian, Barbara Mack, and Sean Lie*

Lecture Summary

1. *Introduction to Memory Contention*
   This section introduces the memory contention model that we will use for the remainder of this lecture.

2. *The (P, M, T) Recycling Game*
   This section presents a game played by an adversary which models concurrent random accesses to memory.

3. *Bounds on the Expected Total Delay*
   This section proves that the expected total delay is bounded.

7.1 Introduction to Memory Contention

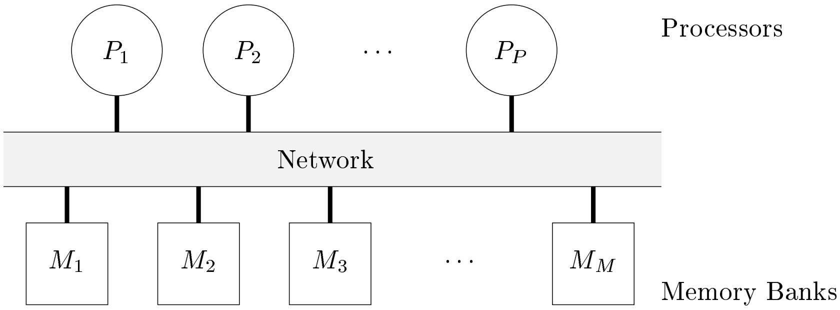


Figure 7.1: A parallel computer with P processors and M memory banks connected by a network.

Our model, illustrated in Figure 7.1, consists of P processors and M memories distributed along an interconnect. P processors concurrently access M memory banks. All P processors access the memory banks at once, but they do not necessarily all access the same memory bank. In our model the network can carry an unlimited number of requests and replies at once.

A memory bank can only serve one request per time step. If more than one request if made of a particular bank, we have contention. In that case, we must choose which request to serve. All other requests are queued and are served in some future time step.

7.1.1 Contention Delay

We are interesting in how much delay is introduced by contention. The length of the delay depends on the pattern of the accesses to the memory banks: if all P processors access the same memory bank (say, M_1), the delay is larger than if the accesses are distributed (even randomly) across the banks. As a trivial example, no access is ever delayed if P_i always accesses M_i.

Since the pattern matters, we will analyze a random pattern. This approach is similar to the one we will use for Cilk work-stealing in the next lecture. Our results with this assumption are useful enough that there have even been proposals to hash memory addresses to make the location access more uniformly random.

7.2 The $(\mathbf{P}, \mathbf{M}, \mathbf{T})$ Recycling Game

To find an upper bound on contention delay, we assume that an adversary picks which request is served when there is contention. The adversary cannot choose to ignore every request. That is, if 1 or more requests are queued, it much choose one. The adversary chooses the ***worst possible*** request to service in banks where there is a choice.

To model such a situation, we use a game. The (P, M, T) recycling game [Blumofe 1995] is a "balls and bins" game with P balls (representing processors), M bins (representing memory banks), and T tosses (representing memory accesses).

The game is played by an adversary, with all the balls resting initially in a reservoir. On each turn of the game:

(a) The adversary chooses a subset of the balls in the reservoir and tosses each ball into a random bin (uniform and independent). Each ball tossed counts towards T. A "toss" is *not* a turn.

(b) For each bin that contains 1 or more balls, the adversary chooses which ball to return to the reservoir. The rest remain in the bin.

The combination of (a) and (b) is a "turn."

The adversary's goal is to make the delay as large as possible. (We will define "delay" in a moment.) Intuitively, the adversary's best strategy is always to toss *all* the balls in the reservoir each turn. This will result in the largest delay since it has the highest probability of contention. The adversary does not know the outcome of future random events, and therefore cannot exploit any foreknowledge of the future.

The adversary can leave a ball in a bin indefinitely, as long as there is at least one other ball in that bin. Therefore, the adversary can make a given processor wait "forever" for its access to M_i as long there is always at least one other request contending for M_i. The chance of this scenario occurring depends on the sizes of P and M. For example, if $M = 1$ and $P \geq 2$ than the adversary can always starve a given processor.

We now define ***delay***. Let b_j be the number of balls in bins immediately after turn j. Then the ***total delay***, D, is given by

$$D = \sum_j b_j \ . \tag{7.1}$$

Figure 7.2 illustrates the actions in a 3-turn sample game where $P = 5$, $M = 4$, and $T = 8$. The total delay D for this game is 3, which is the sum of $b_1 = 1$, $b_2 = 2$, and $b_3 = 0$. The adversary's strategy for this game is not optimal: balls are left in the reservoir turns 1 and 3. Tossing these balls may have allowed the adversary to increase the total delay.

7.3 Bounds for the Expected Total Delay

We can bound the expected total delay in terms of the number of balls (processors) P, bins (memory banks) M, and tosses (total accesses) T.

Theorem 7.1 *The expected total delay* $\mathrm{E}[D]$ *is bounded by*

$$\mathrm{E}[D] \leq \frac{PT}{M} \ .$$

In particular, if $P = M$ *then* $\mathrm{E}[D] \leq T$.

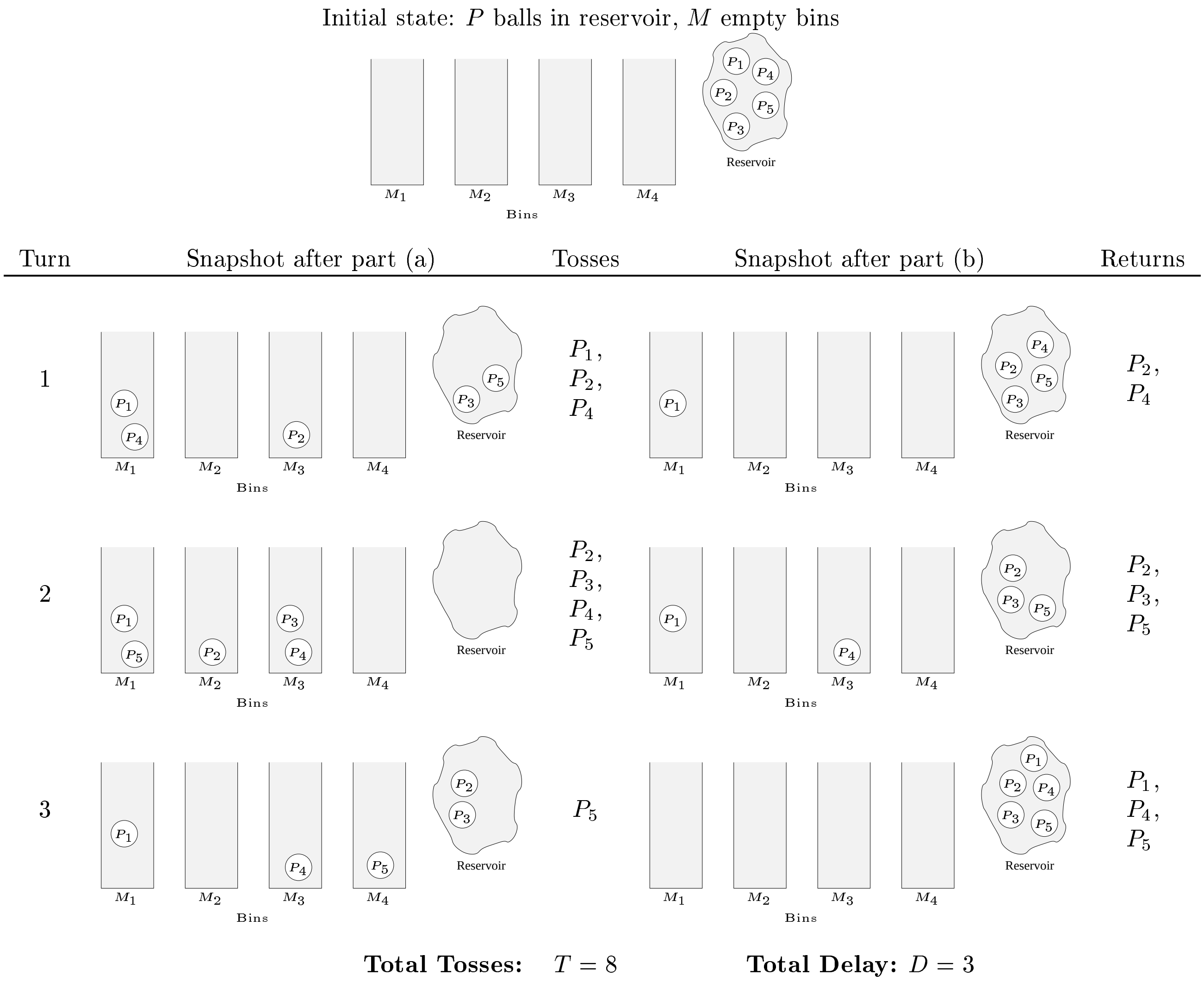


Figure 7.2: Three turns of the (P, M, T) recycling game with $P = 5$ balls, $M = 4$ bins, and $T = 8$ tosses. Bins are on the left and the reservoir is on the right. Circles represent balls. Each snapshot represents the state of the game after part (a) of a turn (on the left) or part (b) of a turn (on the right). Time advances toward the bottom of the page.

Proof Without loss of generality, the adversary's strategy in part (b) of the game is FIFO: the returned ball is always the one which has been in the bin the longest. In terms of total delay, it doesn't matter which ball is chosen in part (b), because the number of balls in a bin is the same regardless of the identity of the ball which is removed.

Let δ_r be the number of turns where ball r finishes in a bin (not a reservoir). Then, the total delay is

$$D = \sum_{r=1}^{P} \delta_r \ , \tag{7.2}$$

which is just another way of looking at the total-delay summation from Equation (7.1).

Let the ***ith cycle*** of a ball r be those turns in which ball r remains in a bin from its ith toss to its removal. Then we say that the ***ith delay*** of a ball is the number of turns in the ith cycle. This quantity may be 0 if the ball went into a bin and was immediately removed.

All balls are symmetric, so let us focus on the delay of the *first* ball δ_1. Let t_1 be the number of times ball 1 is tossed. Note that the total number of tosses is $T = \sum_{r=1}^{P} t_r$. Let $d_i \geq 0$ denote the ith delay of ball 1; i.e. how long this ball stays in a bin during the ith cycle. Therefore, we have

$$\delta_1 = \sum_{i=1}^{t_1} d_i \ , \tag{7.3}$$

which is to say that the sum of all the cycle delays of ball 1 is the total delay of ball 1.

The expected ith delay of ball one, $\mathrm{E}[d_i]$, is simply the expected number of balls in the bin that ball one lands in after the ith toss. However, this is the same as the expected number of balls in any bin at any time. Since there are P balls and M bins, $\mathrm{E}[\text{number of balls in bin}] \leq P/M$. Therefore, $\mathrm{E}[d_i] \leq P/M$.

From Equation (7.3), we get

$$\mathrm{E}[\delta_1] \leq t_1 \frac{P}{M} \ ,$$

By symmetry, if t_r is the number of tosses of ball r, then $\mathrm{E}[\delta_r] \leq t_r P/M$. Then, using Equation (7.2), we have

$$\begin{aligned} \mathrm{E}[D] &= \sum_{r=1}^{P} \mathrm{E}[\delta_r] \\ &\leq \sum_{r=1}^{P} t_r \frac{P}{M} \\ &= \frac{PT}{M} \end{aligned} \tag{7.4}$$

□

From this result we see that, in general, the total delay is not that high. This is expected since memory contention is not the biggest problem that we face in parallel processing. It is more important to worry about load-balancing, which we will cover later this semester.

Alternative Proof There is an alternative way of proving this theorem. The technique in the alternative proof is very useful in high probability analysis. Since we were only concerned with expected value, we did not use this technique. However, it is useful to see it.

We say the ith cycle of ball 1 is ***delayed*** by ball r if ball 1 and ball r are in the same bin when ball r is removed. Recall that we're assuming FIFO order, so balls are delayed *once* or *not at all.*

This binary choice means we can use ***indicator random variables*** (IRV), which are random variables that only take on two values, typically 0 and 1. Indicator random variables have many nice properties that make them easy to deal with [CLRS, 5.2]. We'll be seeing more IRVs throughout the semester.

We define an indicator random variable x_{ir} as follows:

$$x_{ir} = \begin{cases} 1 \text{ if } i\text{th cycle of ball 1 is delayed by } r, \\ 0 \text{ otherwise.} \end{cases}$$

Then, we have

$$d_i = \sum_{r=2}^{P} x_{ir} \ ,$$

and

$$\delta = \sum_{i=1}^{t_1} \sum_{r=2}^{P} x_{ir} \ . \tag{7.5}$$

By the definition of expectation,

$$\begin{aligned} \mathrm{E}\left[x_{ir}\right] &= 1 \cdot \Pr\left[x_{ir} = 1\right] + 0 \cdot \Pr\left[x_{ir} = 0\right] \\ &= \Pr\left[x_{ir} = 1\right] \\ &= \Pr\left[i\text{th cycle of ball 1 is delayed by ball } r\right] \ . \end{aligned}$$

The probability that the ith cycle of ball 1 is delayed by ball r is either $1/M$ or 0: if r is in a bin when ball 1 is tossed, the probability that we get tossed in that same bin is $1/M$; and if r is in the reservoir instead, the probability of landing in a bin with it is 0. Although a tighter bound on $\mathrm{E}\left[x_{ir}\right]$ is possible, the following bound is adequate for our purposes.

$$\mathrm{E}\left[x_{ir}\right] \leq \frac{1}{M}$$

Using this bound and Equation (7.5), we obtain

$$\begin{aligned} \mathrm{E}\left[\delta\right] &= \sum_{i=1}^{t_1} \sum_{r=2}^{P} \mathrm{E}\left[x_{ir}\right] \\ &\leq \sum_{i=1}^{t_1} \sum_{r=2}^{P} \frac{1}{M} \\ &= t_1 \frac{(P-1)}{M} \\ &\leq t_1 \frac{P}{M} \end{aligned}$$

Then Equation (7.4) follows directly.

6.895 Theory of Parallel Systems	Lecture 8
Cilk Scheduler	
Lecturer: Michael Bender	*Scribe: Kevin Matulef, Barbara Mack, Paul Youn, and Sean Lie*

Lecture Summary

1. *"Ideal" Scheduling*
   This section describes the desirable properties of an "ideal" global scheduler.

2. *Cilk Computation*
   This section gives a brief overview of the Cilk computation model.

3. *Main Data Structure*
   This section describes the main data structure used in the Cilk scheduler, the ready deque.

4. *Cilk Scheduler*
   This section describes the Cilk scheduling algorithm.

5. *Work Stealing*
   This section describes the Cilk scheduler's work stealing algorithm.

6. *Ready Deque Structural Lemma*
   This section proves a lemma that shows some very important properties of the ready deque.

8.1 "Ideal" Scheduling

The goal is to develop a distributed scheduler that approximates the performance of an "ideal" global scheduler. A global scheduler knows the entire system state (the state of all processors), while a distributed scheduler knows only the local state.

To design a good scheduler, we need to understand what properties we would want in an "ideal" global scheduler. Three of these properties are outlined here:

1. **The scheduler should be *greedy*.**

   Recall that a greedy scheduler has a very good bound on total run time [Graham, Brent]:

$$T_p \leq \frac{T_1}{P} + T_\infty \tag{8.1}$$

   Moral: In most practical instances, T_∞ is much less than $\frac{T_1}{P}$. Thus, if we have a choice, we would like for any overhead incurred by a distributed scheduler to be a multiple of T_∞, rather than $\frac{T_1}{P}$. So, ideally, we would like any additional overhead to take away from the critical path of the DAG.

2. **The scheduler should have the *Busy Leaves* property.**

   As shown in a previous lecture, a busy leaves scheduler has a good bound on S_p, the space for p processors:

$$S_p \leq P \cdot S_1 \tag{8.2}$$

This property is desirable because we do not want the scheduler to take up too much memory in overhead. If our scheduler can have the busy leaves property, each processor on average is using at most the amount of memory to schedule tasks as is required by the serial execution.

3. **The scheduler should have a small number of steals/migrations.**

   When a processor does not have any work to do, it must get (steal) work from another processor. In general this is very efficient because every processor is being used as much as possible, but at the same time there is also a cost associated with the steal itself. It is important to keep the number of steals low since this overhead can be quite high. Later we will show that our scheduler can, with high probability, bound the number of steals:

$$\text{number of steals} = O(PT_\infty) \tag{8.3}$$

   If we can develop a greedy scheduler that satisfies that bound, then the cost of steals, averaged over the number of processors, is $O(T_\infty)$. Therefore, in accordance with the moral stated above, the total cost of steals is related to the critical path instead of $\frac{T_1}{P}$ term.

8.2 Cilk Computation

We will develop a scheduler for the Cilk computation model. For clarity, we define the terms ***thread***, ***procedure***, ***child thread***, and ***child procedure*** in terms of a Cilk DAG.

A ***thread*** is node in the Cilk DAG. In Figure 8.1, the circle nodes $(t_{11}, t_{12}, \ldots)$ are threads. Every node in the Cilk DAG is a thread. We assume that each thread executes in the same amount of time. If this is not the case, longer threads can be modeled as several smaller threads executing in series.

A ***procedure*** is a group of threads that must execute in series. In Figure 8.1, procedures are represented by the gray large oval $(\alpha_1, \alpha_2, \ldots)$ around some group of thread nodes. For the purpose of the scheduler, we will often refer to the execution of a Cilk program in terms of these procedures as opposed to threads.

A ***child thread*** of thread i is one of thread i's successor nodes in the DAG. Each thread can have 0, 1, or 2 child threads. For example, in Figure 8.1, threads t_{13} and t_{21} are both child threads of t_{12}. A child thread is either a thread that follows i within the same procedure, or is a spawned parallel thread. Thus, we can have serial or parallel child threads. Note that a spawned parallel thread will be the first thread in a seperate procedure to be executed in parallel. Lastly, the only thread with no child threads is the final thread executed in the program.

A ***child procedure*** of a procedure α is a procedure that was spawned from a thread in α. While every thread can have up to 2 child threads, a procedure can have any number of parallel child procedures. Each procedure can consist of any number of threads, and each of these threads can spawn off a parallel procedure. For example, in Figure 8.1, procedures α_2 and α_4 are both child procedures of α_1.

8.3 Main Data Structure

In this section, we describe the main data structure that we will use in our scheduler. Each processor maintains a data structure called a ***ready deque***. A ready deque is a double ended queue.

Every level of the ready deque contains a procedure that is ready to execute. Specifically, every level contains the thread that is next ready to execute in a given procedure, but it is easier to think of each level as just containing the portion of a procedure that has not yet executed. For the remainder of this lecture, levels of the deque will be treated as containing procedures.

Procedures are inserted at the bottom of the deque, but they may be deleted from either end. As shown in Figure 8.2, the deque resembles a queue at the top, and a stack at the bottom. Soon we will see that deletion from the top of the deque corresponds to that procedure being stolen, and adding a procedure to

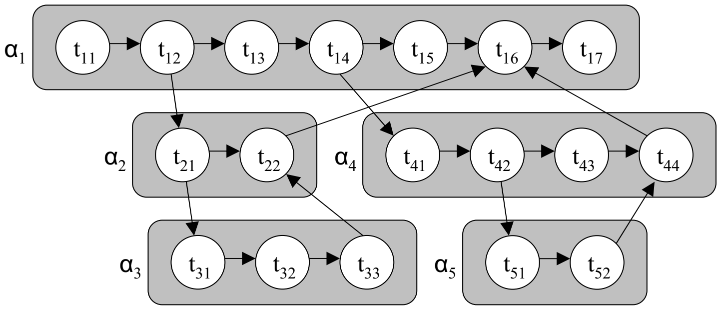


Figure 8.1: An example Cilk DAG. Nodes (white circles) are marked t_{ij} where i is the procedure number. Procedures (gray ovals) are marked α_i.

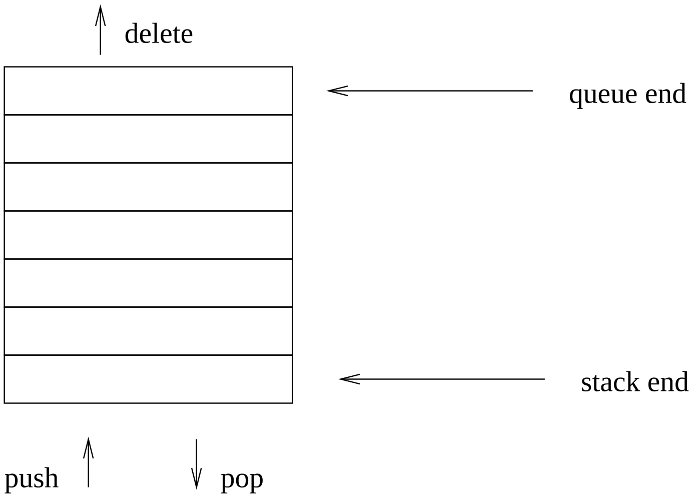


Figure 8.2: The ready deque. Procedures can be pushed into the deque from the bottom. Procedures can be popped (deleted) from the deque from both the bottom and the top.

the bottom represents to a procedure being spawned. Likewise, a procedure being deleted from the bottom of the deque represents the processor beginning/resuming execution on that procedure.

As shown in Figure 8.3, when used in serial execution, the deque operates similar to the stack of a normal C program. Note that a processor's ready deque represents procedures that are ready to execute but not already executing. In particular, this means that the procedure currently being executed by a processor is *not* in the deque. A processor can be busy and still have an empty deque.

8.4 Cilk Scheduler

This section describes the Cilk scheduler. The Cilk scheduler uses a ready deque as its main data structure. It also satisfies the properties that we originally set.

In the Cilk scheduler, a processor works on procedure α until:

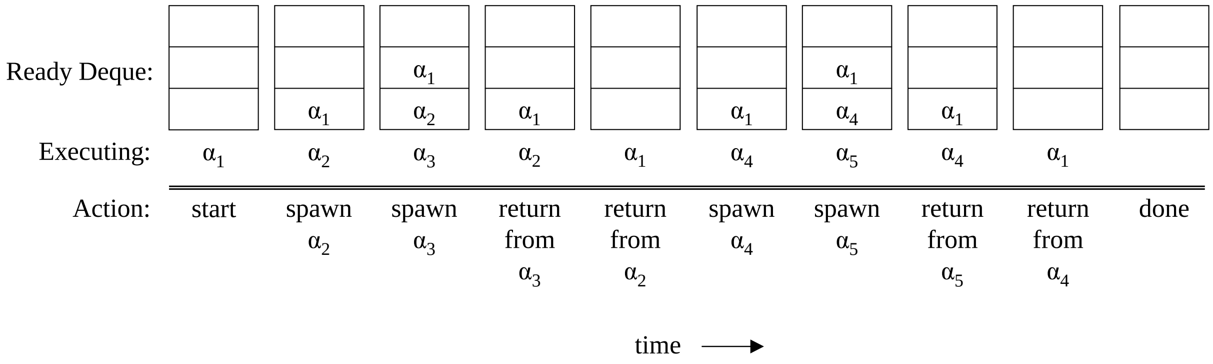


Figure 8.3: The ready deque as we execute the DAG from Figure 8.1 on one processor. When we spawn a procedure, we push the previously running procedure onto the deque and start executing the new procedure. When we return from a procedure, we pop the previous procedure from the deque and continue executing it. This resembles the stack of a normal C program.

1. α *spawns* procedure β

   - In this case, the processor first pushes procedure α onto the bottom of the ready deque. Putting α onto the deque means that α has not fully executed yet but that it is not being worked on currently. Then the processor starts to execute the procedure β.

2. We reach a `sync` in α

   - If there exists outstanding children of α that have not completed, the computation cannot proceed. In this case, the processor steals work from another processor. Work stealing is described in detail in the next section. Note that the deque is empty in this case. That means that α has not been completely executed and it is *not* on any deque.
   - If there are no outstanding children of α (there are no children or all children have completed), the processor just continues executing α past the `sync`.

3. α *returns*

   - If the processor's deque is nonempty, the processor pops the procedure at the bottom of its deque and begins working on it.
   - If the deque is empty, the processor tries to execute α's parent (the procedure that spawned α). If α's parent is currently running, the processor steals work from another processor.

Intuitively, the Cilk scheduler enables each processor to do as much work as possible. When a processor runs out of work, it steals some work from another processor.

However, of all the steps in the algorithm described above, the last step is the most odd. Why do we need to try to run α's parent when we return? The answer becomes clear when we look at what the scheduler does when we reach a `sync` statement in, say, procedure β. If β has outstanding children, we just drop β and steal work from another processor. At that point, procedure β is not finished but it is also not on any deque. This may seem strange but it is fine since we rely on β's last outstanding child to execute β when it returns. Note that all of the β's children will have empty deques when they return. Therefore, when a procedure α returns and its deque is empty, it must check if it is the last outstanding child. If so, it needs to continue execution of its parent. The way we check if it is the last outstanding child is by simply trying to execute the parent. If we are not the last outstanding child, we will simply fail and work steal. If we are the last outstanding child, we will continue to execute the parent.

Another point that should be mentioned is that single processor execution is the same as a normal C program. As discussed in Section 8.3, the deque acts just like the stack in a normal C program when we are running on 1 processor. If we spawn, we simply suspend the parent and work on the child. Once the child is fully executed, execution on the parent resumes. This corresponds to the depth-first execution of the DAG as shown in Figure 8.3.

8.5 Work Stealing

Here we discuss the work stealing algorithm in detail. When a processor p begins work stealing, it operates as follows:

- Processor p chooses a victim processor uniformly at random from the set of all processors. Let us call the victim processor v.
- If processor v's deque is empty, processor p repeats the first step.
- Otherwise if processor v's deque is not empty, processor p steals the top procedure from v's deque. This is the reason the deque data structure must support pops from the top. Processor p then begins to work on this stolen procedure.

The reason we steal work at random is because a distributed scheduler has no global knowledge of which processors have available work. Ideally, we would just steal from a processor that has a lot of work. However, since this is not possible in a distributed scheduler, we steal at random.

8.6 Ready Deque Structural Lemma

We proceed by proving a structural lemma about the ready deque. This will help us understand the data structure. It will also begin to explain why the scheduling, and ready deque data structure, as defined above allow the Cilk scheduler to work steal in a manner that steals the optimal procedure "with high probability".

The idea is that the procedure at the top of the deque is always the oldest (and closest to the root of the DAG). Therefore, if we always steal the top procedure, we will likely be taking away from the critical path. Therefore, the steal overhead can be used towards the critical path as we originally set out to do. We will prove this next lecture.

In this section we will prove a structural lemma about the ready deque that will set up the proof for next class. Consider any processor p at any time t. Let α_0 be the procedure that processor p is working on. Let $\alpha_1, \ldots, \alpha_k$ be the procedures in p's deque from bottom to top, as shown in Figure 8.4.

.

Lemma 8.1 *At time t:*

1. - α_i is a child of α_{i+1}, for $i = 0, \ldots, k-1$
   - α_i is the *only* active child of α_{i+1}, for $i = 0, \ldots, k-2$. An active procedure is a procedure that is on a deque or currently executing.
   - α_{k-1} may be one of many children of α_k
2. The number of actives leaves is at most P, the number of processors. A leaf is defined as a procedure with no current outstanding children.

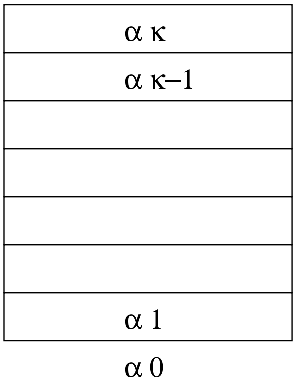


Figure 8.4: The ready deque of a processor. Procedure α_0 is being executed while procedures $\alpha_1 \ldots \alpha_k$ are in the deque.

We notice that in order for the scheduler to maintain the Busy Leaves property as discussed above, the second property in the lemma hold.

Proof To prove this lemma, we do an induction on actions. That is, we examine what happens during each type of action (spawn, return, steal, sync) and show that all properties in the lemma are preserved. Note that the following proof is not complete, but should be sufficient to convince the reader that the structural lemma is true.

- **Spawn:** We consider what happens when α_0 spawns a child procedure.

  Let us label the child of α_0 as α_{child}. Then we push α_0 onto the deque and start executing α_{child}.

  Consider the last property in the lemma. While α_0 was a leaf prior to the spawn, following the spawn, α_{child} becomes a leaf and α_0 ceases to be a leaf. Thus, the number of leaves remains the same and by the inductive hypothesis, the property is met.

  Also, α_{child} is the only child of α_0. This implies the first two properties of the lemma by inductive hypothesis directly.

- **Sync:** We consider what happens when we hit a `sync` while executing α_0.

  In the base case, α_0 has no outstanding children, and execution on α_0 continues past the sync. In this case, if the structural lemma held before the sync was reached, it holds afterwards since no changes have been made to any deque.

  In the second case, α_0 has outstanding children so we give up on α_0. Then we attempt to steal work from another processor. If there is no work to be stolen, we sit idle. In this case, the lemma holds as this deque is now empty and the other deques have not changed. If there is work to be stolen, then this is a trivial case of the steal argument described below.

- **Steal:** We consider what happens in the case of a steal.

  When another processor steals from this deque, right after a steal *all* the remaining processes α_{k-1} through α_0 left in the deque must be only-children. This is true since we assumed, in the inductive hypothesis, that α_k is the only process with multiple children.

  The stealing processor now has only one process α_k in its deque. This also satisfies the first three properties of the lemma. Moreover, the number of active leaves is clearly preserved because the stolen procedure is not a leaf.

- **Return:** We consider what happens in the case of a return.

  If the deque is not empty when α_0 returns, we simply begin working on α_1. This preserves the properties of the lemma trivially since we only took the bottom procedure off the deque.

  If the deque is empty when α_0 returns and we start working on α_0's parent, the parent must have become a leaf. The parent is a leaf since it has no more outstanding children. Therefore, the number of active leaves is maintained. Also, the other deques do not change in this case and the current deque is empty. Thus, the first three properties of the lemma have also been preserved.

  If the deque is empty when α_0 returns and we cannot execute α_0's parent, we work steal. We already considered the case of steals and the lemma is preserved.

In the next lecture, we will see how this scheduler can ensure that the steals take away from the critical path "with high probability". We find that we would like to steal the procedure that contains the shallowest thread in the DAG on that processor. We claim that this procedure is "always" at the top of the deque. The remainder of this idea is presented in the following lecture.

$\square$

6.895 Theory of Parallel Systems	Lecture 9
Analysis of Cilk Scheduler	
Lecturer: Michael A. Bender	*Scribe: Alexandru Caracaş, C. Scott Ananian*

Lecture Summary

1. *The Cilk Scheduler*
   We review the Cilk scheduler.

2. *Location of Shallowest Thread*
   We define the depth of a thread and the shallowest thread. Next, We prove that the shallowest thread on a processor is either at the top of a deque or being executed.

3. *Critical Threads*
   We construct a computation graph G' similar to the computation graph G, such that when a thread has no incomplete predecessors in G', then it is at the top of a deque.

4. *Execution Time Bounds For The* Cilk *Scheduler*
   We present the execution time bounds of the *Cilk* scheduler. We introduce an accounting argument to help in proving the bound.

5. *Performance Analysis of Scheduling Algorithm Using Delay Sequences*
   We analyze the performance of the *Cilk* work-stealing scheduling algorithm. We define delay sequences and use them to prove the execution time bounds of the *Cilk* scheduler.

9.1 The *Cilk* Scheduler

In this section we review the *work-stealing* algorithm of the *Cilk* scheduler. Recall that in *Cilk* a processor works on a procedure α until one of three events occur:

1. Procedure α *Spawns procedure* β. In this case the processor pushes α on (the bottom of) the ready deque, and starts work on procedure β.

2. Procedure α *Returns.* There are three cases:

   (a) If the deque is nonempty, the processor pops a procedure (from the bottom) and begins working on it.

   (b) If the deque is empty, the processor tries to execute α's parent (which is not in a deque).

   (c) If the deque is empty and the processor is unable to execute α's parent (because the parent is busy), then the processor work steals.

3. Procedure α *Syncs.* If there are no outstanding children, then continue: we're properly synced. Otherwise (when there are no outstanding children), work steal. Note that deque is empty in this case.

When a processor begins work stealing it operates as follows:

1. Processor chooses a victim *uniformly at random.*

2. If the victim's deque is empty, the processor repeats the first step.

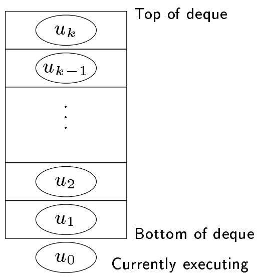


Figure 9.1: Deque of a processor.

3. Otherwise, the processor steals the top (oldest) thread from the victim's deque and begins to work on it.

Note: A thread can be in exactly one of three places:

1. in the deque,
2. on a processor being executed, or
3. waiting for predecessors to complete.

In the following we use the notion of ***procedure*** as composed of multiple threads.

9.2 Location of shallowest thread

In this section we define the depth of a thread and the shallowest thread. Next we prove that the shallowest thread on a processor is either at the top of a deque or being executed.

Definition 9.1 (Depth of thread) *The* ***depth*** $d(u_i)$ *of a thread* u_i *is the longest distance in the DAG from the root thread to* u_i*.*

Definition 9.2 (Shallowest Thread) *The* ***shallowest thread*** *is the thread with minimal depth.*

We prove some structural properties of the deque.

Lemma 9.3 (Structural lemma) *Consider any processor* p *at time* t*. Let* u_0 *be the current root thread (of procedure* α_0*) executing on processor* p*. Let* $u_1, u_2, \ldots, u_k$ *be the threads (of procedures* $\alpha_1, \alpha_2, \ldots, \alpha_k$*) in* p*'s deque ordered from bottom to top. Let* $d(u_i)$ *be the depth of thread* u_i *in DAG* G*. Then,*

$$d(u_0) \geq d(u_1) > \cdots > d(u_{k-1}) > d(u_k).$$

Observation 9.4 *A procedure is made up of multiple threads, but* at a given time t, *there is only one thread in the procedure which is being executed.*

Corollary 9.5 *The shallowest thread in a processor is either at the top of a deque or being executed.*

Proof Proof is by *induction on actions.* The **base case** (deque is empty; u_0 executing) trivially satisfies the inequality. Thereafter, only four possible actions can alter the currently-executing thread and/or the deque. We prove that each of these actions maintains the desired depth-ordering property.

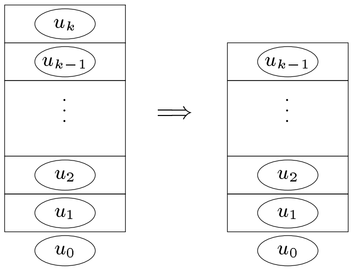


Figure 9.2: Change in the deque of a processor when a procedure is stolen.

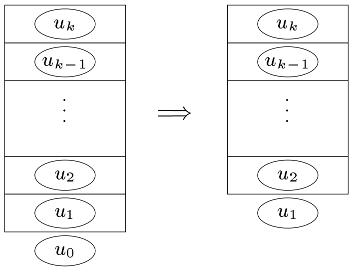


Figure 9.3: Change in the deque of a processor when the processor returns from a procedure.

- **Case 1: Steal.** A steal removes the top entry from the deque (see Figure 9.2). The processor performing the steal begins executing u_k with an empty deque, trivially satisfying the inequality as in the base case. The processor from whom the thread was stolen also maintains the invariant, because if before the steal

  $$d(u_0) \geq d(u_1) > \cdots > d(u_{k-1}) > d(u_k),$$

  after the steal

  $$d(u_0) \geq d(u_1) > \cdots > d(u_{k-1}).$$

  Removing threads from anywhere in the deque cannot invalidate the ordering.

- **Case 2: Return.** A return removes the bottom entry from the deque (see Figure 9.3). As in the case of the steal, removing terms from the inequality cannot invalidate it. If before the return

  $$d(u_0) \geq d(u_1) > \cdots > d(u_{k-1}) > d(u_k),$$

  afterwards we have

  $$d(u_1) > \cdots > d(u_{k-1}) > d(u_k).$$

  As before, removing threads from anywhere in the deque cannot invalidate the ordering.

- **Case 3: Continue or sync.** A sync or continue replaces the bottom entry of the deque (see Figure 9.4) with a thread with depth increased by one. A sync with one or more children outstanding becomes a steal, and is handled by Case 1. If all children are complete, then a sync and a continue are identical: we replace the currently-executing thread u_0 with its successor in the DAG, u_a. Figure 9.5 shows the relevant piece of the computation DAG.

  If the longest path to u_0 has length $d(u_0)$, then clearly there is a path to u_a through u_0 which has length $d(u_0) + 1$. Therefore,

  $$d(u_a) \geq d(u_0) + 1,$$

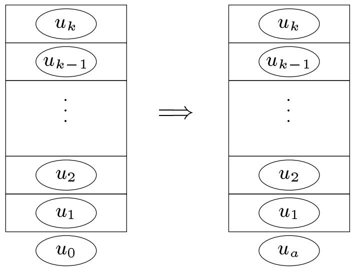


Figure 9.4: Change in the deque of a processor when the processor reaches a sync point or continues a procedure.

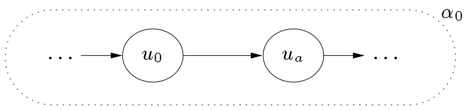


Figure 9.5: A piece of a computation DAG. Thread u_0 is followed by thread u_a in procedure α_0.

and since $d(u_0) \geq d(u_1)$, then

$$d(u_a) > d(u_1) > \cdots > d(u_{k-1}) > d(u_k).$$

- **Case 4: Spawn.** A spawn replaces the bottom entry in the deque and also pushes a new procedure (see Figure 9.6). Figure 9.7 shows the portion of the computation DAG involved in a spawn. Note that, since the only edges to u_a and u_s come from u_0, that u_a and u_s have the same depth in the DAG, namely $d(u_0) + 1$. Therefore since $d(u_0) \geq d(u_1)$, then $d(u_s) = d(u_a) > d(u_1)$ and, as required

  $$d(u_s) \geq d(u_a) > d(u_1) > \cdots > d(u_{k-1}) > d(u_k).$$

  Note: This is the only case where the depth of the currently-executing thread may become equal to the depth of the thread on the bottom of the deque; hence the only case where a shallowest thread may be the currently-executing thread instead of the thread at the top of the deque.

Since the base case satisfies the property, and every action maintains the property, then at any time t in the execution for all deques

$$d(u_0) \geq d(u_1) > \cdots > d(u_{k-1}) > d(u_k).$$

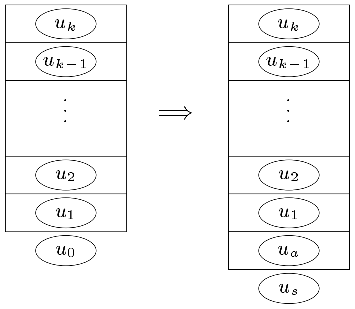


Figure 9.6: Change in the deque of a processor when the processor spwans a procedure.

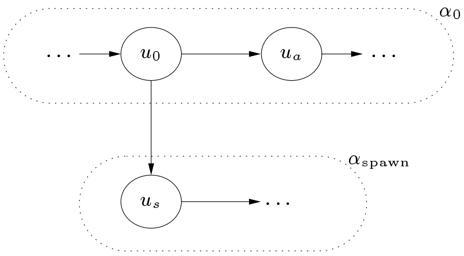


Figure 9.7: A piece of a computation DAG. Thread u_0 is followed by thread u_a in procedure α_0, and spawns thread u_s in procedure α_{spawn}.

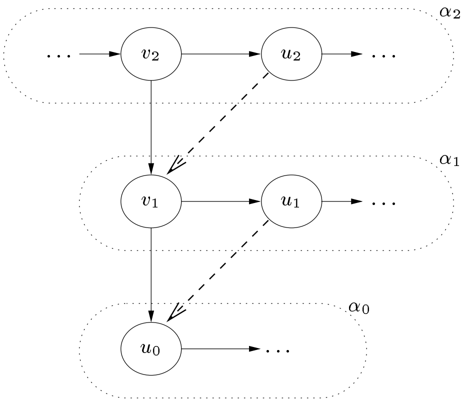


Figure 9.8: An augmented DAG corresponding to the execution DAG of Figure 9.9. The dashed edges have been added from continuation threads to spawn threads.

□

9.3 Critical threads

In this section we construct an augmented computation DAG G' similar to the computation DAG G, such that when a thread has no incomplete predecessors in G', then it is at the top of a deque. Next, we introduce **critical threads**.

Our goal is to show that any thread on the top of a deque cannot stay there long because it will be stolen.

Definition 9.6 (Construction of $G\prime$) *We create an augmented DAG $G\prime$ by making a (back) edge to every spawn thread from the parent's continuation thread. Note that if a sync is immediately followed the spawn, we need to insert an extra continuation thread before the sync to prevent cycles in the DAG. Figure 9.8 shows the execution graph of Figure 9.9 augmented with these edges.*

We show that **work-stealing** makes progress on the **critical path**. Roughly speaking, $\Theta(P)$ steals decrease the critical path by a constant. We require a stronger property for the computation DAG.

Proposition 9.7 (Necessary property) *Whenever a thread has no incomplete predecessors in the augmented DAG $G\prime$, it is being executed or is on the top of some deque.*

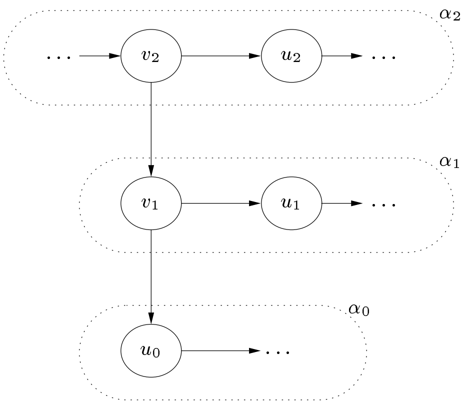
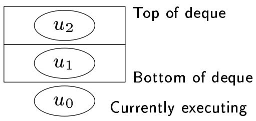


Figure 9.9: An execution DAG with threads v_2 and u_2 in procedure α_2, threads v_1 and u_1 in procedure α_1, and thread u_0 in procedure α_0.

Note: Proposition 9.7 does not hold for the execution DAG G, that is why we create G'.

Example 9 To see why, consider the execution DAG in Figure 9.9. After threads v_2 and v_1 have been executed, we begin to execute thread u_0. We have the following deque:



The thread u_1 has no unexecuted precedessors left in the DAG, but it is neither being executed nor at the top of the deque.

Observation 9.8 (Depth of Augmented DAG G') *The critical path of the augmented DAS G' is $2T_\infty$. We can now get to a spawned thread via our back edge from the continuation edge, adding a distance of one to the longest path to the spawned thread. But we can only add one such extra edge to the path per spawn edge on the critical path of the original graph G. If we define the* ***depth of a graph*** *$D(G)$ to be the depth of the deepest node, then,*

$$D(G) \leq D(G') \leq 2D(G),$$

where $D(G) = T_\infty$.

Proposition 9.7 guarantees that whenever you execute a thread, you will be executing one of the **shallowest** threads in the augmented DAG G'. Hence, when you steal work (i.e. not otherwise making progress) you make progress on the critical path by executing the threads which are shallowest.

Definition 9.9 (Critical thread) *A thread is* **critical** *if it has no un-executed predecessors in the augmented DAG G'.*

Note: The extra edges in the augmented DAG G' are not execution edges, they simply induce extra ordering that we require for our performance analysis of the *Cilk* scheduler.

9.4 Execution Time Bounds For The *Cilk* Scheduler

The *Cilk* work-stealing scheduling algorithm has provably good execution time bounds. We introduce an accounting argument based on **two buckets** to help in proving the bound.

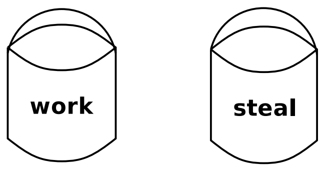


Figure 9.10: The **work** and **steal** buckets used in the accounting argument.

Theorem 9.10 (Cilk Scheduler execution time bound) *Consider the execution of any fully strict multithreaded computation with work T_1 and critical path T_∞ by* Cilk*'s work-stealing algorithm on a parallel computer with P processors. For any $\varepsilon > 0$, with probability at least $1 - \varepsilon$, the execution time on P processors is*

$$T_P \le \frac{T_1}{P} + O\left(T_\infty + \lg\left(\frac{1}{\varepsilon}\right)\right). \tag{9.1}$$

Proof Idea The proof is based on an **accounting argument**. We have 2 buckets, one for **work** and one for **steal**. On each time-step, a processor puts one dollar into one of the two buckets, depending on the action that it performs in that step. The possible two actions and their respective results are:

- execute instruction – one dollar into **work** bucket,
- steal attempt – one dollar into **steal** bucket.

We assume that all threads are of equal size and that all processors have the same time-step. In the following we overload the notation of steal attempt with steal. □

Lemma 9.11 (Work Bucket) *At the end of computation, the number of dollars in the* **work** *bucket is T_1.*

Proof The total work is exactly T_1 so at the end of the computation exactly T_1 work has been executed. Thus the number of dollars in the **work** bucket is precisely T_1. □

We know bound the steals with the following lemma.

Lemma 9.12 (Steal Bucket) *At the end of a computation the number of dollars in the* **steal** *bucket is $O(PT_\infty)$, with* **high probability**.

We give an exact definition of **high probability** during the course of the proof. In the rest of the lecture, we prove the previous lemma. We introduce several concepts to prove the bound.

Observation 9.13 (Dollars) *Per time-step P dollars enter the buckets.*

Thus, the running time on P processors is the amount of time it takes to fill out all the buckets.

Claim 9.14 *The running time on P processors is less than the total number of dollars (actions) divided by the number of processors:*

$$\begin{aligned} T_P &\le \frac{\#\text{dollars in work bucket} + \#\text{dollars in steal bucket}}{P} \\ &\le \frac{T_1}{P} + O(T_\infty). \end{aligned} \tag{9.2}$$

We know how many dollars will be in the work bucket. We need a bound for the number of dollars in the steal bucket.

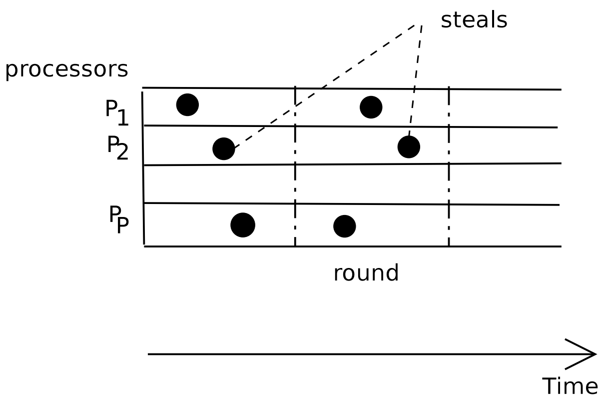


Figure 9.11: Dividing computation into steal rounds.

9.5 Performance Analysis of Scheduling Algorithm Using Delay Sequences

In this section, we analyze the performance of the *Cilk* work-stealing scheduling algorithm. We define ***delay sequences*** and use them to prove the execution time bounds of the *Cilk* scheduler.

We prove equation (9.2) in the rest of the lecture.

Observation 9.15 *There is no constant in front of the term T_1/P in the previous bound. All scheduling overhead cost is amortized against the critical path T_∞. Also, one important observation is that the term T_1/P is much bigger than the term T_∞.*

9.5.1 Rounds

As long as no processor is stealing, the computation is progressing efficiently. What divide the computation into **rounds**, which (more or less) contain the same number of steals (see Figure 9.11).

Definition 9.16 (Round) *A **round** of work stealing attempts is a set of at least P and at most $2P-1$ steal attempts. We assume that steal attempts complete in one unit of time.*

Round 1 begins at time 0 and ends when there have been at least P steals. In general, round i begins when round $i-1$ ends, and round i ends when there have been at least P steals.

Recall by Definition 9.9 that a **critical thread** is at the top of some deque or is being executed.

Proof Idea The main idea is that, if a thread is on the top of a deque for CP steal attempts, then the probability that it will not be executed for C rounds is very small, namely:

$$\left(1 - \frac{1}{P}\right)^{CP} \leq e^{-C}.$$

□

The idea is that a thread is unlikely to be critical for many rounds. Also, a thread may be ready for many rounds but not critical.

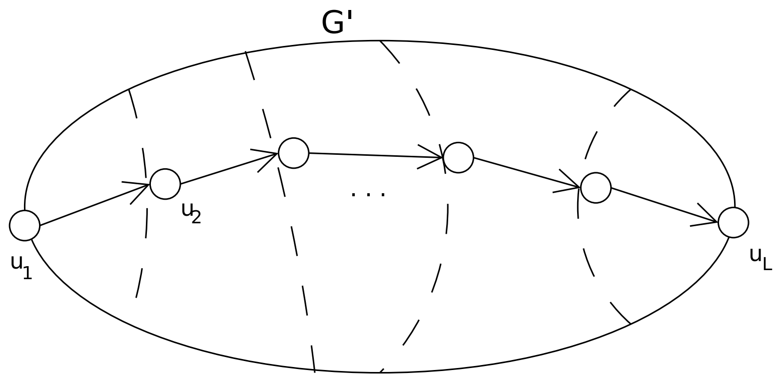


Figure 9.12: A directed path $U = (u_1, u_2, \ldots, u_L)$ in the augmented DAG $G\prime$.

Theorem 9.17 *The probability that a critical thread is un-stolen for C rounds is:*

$$\left(1 - \frac{1}{P}\right)^{CP} \leq e^{-C}. \tag{9.3}$$

Proof For C rounds, there are at least CP steals. The steal process is done at random and there are P places that a processor could steal from. Hence, the probability that a critical thread would be stolen during a round is $1/P$, and the probability that the critical thread will not be stolen is $1 - 1/P$. Hence, for C rounds we obtain the probability that a critical thread is not stolen:

$$\left(1 - \frac{1}{P}\right)^{CP}.$$

For a simplified bound of the above equation we use Euler's inequality:

$$\left(1 + \frac{1}{x}\right)^{x} \leq e.$$

We obtain

$$\left(1 - \frac{1}{P}\right)^{CP} \leq e^{-C}.$$

□

For a small number of rounds a thread might not be executed. However, for a large number of rounds a thread will be executed with very high probability, because the term e^{-C} in the previous bound becomes exponentially small. If a thread is at the top of a deck then it has no un-executed predecessors in the augmented DAG $G\prime$. Intuitively we would expect that in every time-step we are able to peal some threads from the augmented DAG such that in time $O(PT_\infty)$ there would be no augment DAG left (see Figure 9.12). This idea is expressed in the following lemma and explanation.

9.5.2 Delay Sequence Argument

Lemma 9.18 (Structural Lemma) *There exists a directed path $U = (u_1, u_2, \ldots, u_L)$ in G' in which, during every time-step of the computation, some thread in U is critical.*

Proof The proof is by induction backwards in time.

Base case: During last time-step t, last thread in G' is u_L and it is critical and executed (see Figure 9.12).

Induction step: Suppose during time-step t, thread u_j is critical. During time-step $t-1$, either u_j is also critical, or if not it is because it has un-executed predecessors in $G\prime$. Some threads $u_i, u_{i\prime}, \ldots \prec_{G\prime} u_j$ are being executed during t-1. One of them is minimal and is critical.

Main idea: We prove that since the probability that any thread remains critical for a long period of time is small, there cannot be too many steal attempts. □

Observation 9.19 *An executing thread is not necessarily critical. The very last thread in the computation u_L when it is being executed it is critical. The extra edges in the augmented DAG introduce additional ordering (see Figure 9.9).*

In order to complete the proof for the execution time bound of the *Cilk* scheduler, we introduce some definitions and lemmas.

Definition 9.20 (Delay Sequence) *A **delay sequence** is an event described by the triple (U, R, Π) where:*

- *$U = (u_1, u_2, \ldots, u_L)$ is a directed path in G' from* **source** *of DAG to* **sink***,*
- *$R > 0$ is an integer,*
- *$\Pi = (\pi_1, \pi_2, \ldots, \pi_L)$ is a partition of R, namely $\sum_i \pi_i = R$*

*A **delay sequence** occurs if $\forall i$ at least π_i rounds occur while u_i is critical and un-executed.*

A ***delay sequence*** is a combinatorial object. A ***delay sequence*** occurs if there is a directed path in $G\prime$ in which one of the threads in the path is critical at each time-step.

Lemma 9.21 *If at least $2P(2T_\infty + R)$ steal attempts occur, then some (U, R, Π) delay sequence occurs.*

Proof Consider a path $U = (u_1, u_2, \ldots, u_L)$ for which during each time-step t some thread is critical. There are at most $2T_\infty$ rounds during which a thread in U executes, since $L \leq 2T_\infty$. For all the other R rounds, some thread is critical and not executed.
Note: The converse need not be true. □

The idea is to sum up over all possible events the probability that the event occurs. Show that the sum is small. We would then be able to show that with *high probability* there at most $2P(2T_\infty + R)$ steal attempts.

A ***delay sequence*** is an **event** that explains why there are at least $2P(2T_\infty + R)$ steal attempts. The probability that at least $2P(2T_\infty + R)$ steal attempts occur is at most the probability that there exists a delay sequence (U, R, Π) that occurs.

Observation 9.22 (Number of Steal Attempts) *We make the following observation about the probability that at least $2P(2T_\infty + R)$ steal attempts occur.*

$$
\begin{aligned}
Pr\left[\begin{array}{c}\text{at least } 2P(2T_\infty + R) \\ \text{steal attempts occur}\end{array}\right] &\leq Pr\left[\begin{array}{c}\text{exists delay sequence} \\ (U, R, \Pi) \text{ that occurs}\end{array}\right] \\
&\leq \sum_{\substack{\text{delay sequences} \\ (U, R, \Pi)}} Pr\left[(U, R, \Pi) \text{ occurs}\right] \\
&\leq \left(\begin{array}{c}\text{Number of} \\ \text{Delay Sequences}\end{array}\right)\left(\begin{array}{c}\text{Maximum Probability} \\ \text{that any delay} \\ \text{sequence occurs}\end{array}\right).
\end{aligned}
$$

Lemma 9.23 (Probability of Delay Sequence) *The probability that a particular delay sequence (U, R, Π) occurs is e^{-R}, which is exponentially small in the number of rounds, R.*

Proof

$$\begin{aligned}\Pr[(U,R,\Pi)\,\text{occurs}] &= \prod_{1\le i\le L} \Pr\left[\begin{array}{c}\pi_i\,\text{rounds occur}\\ \text{while } u_i \text{ is critical}\end{array}\right] \\ &\le \prod_{1\le i\le L} e^{-\pi_i} \\ &= e^{-R}.\end{aligned}$$

□

Lemma 9.24 (Number of Delay Sequences) *The number of delay sequences is at most*

$$2^{2T_\infty}\binom{2T_\infty+R}{2T_\infty}.$$

Proof We have

$$\#\text{paths} \le 2^{2T_\infty},$$

because no path is longer that $2T_\infty$ and the out degree of any thread (vertex) is at most two. Thus, there are two choices (either a continuation edge or a spawn edge) starting from the root node. Also

$$\begin{array}{c}\#\text{partitions of a path}\\ \text{in augmented DAG } G'\end{array} = \binom{2T_\infty+R}{2T_\infty}.$$

We are partitioning R into L pieces. However, for large R, this is smaller than partitioning R into $2T_\infty$ pieces. We simply count up all the possible ways of partitioning the path. Thus, by multiplying the number of paths with the number of ways in which a path can be partitioned we obtain the number of delay sequences, which is at most

$$2^{2T_\infty}\binom{2T_\infty+R}{2T_\infty}.$$

□

Our goal is to show that the probability that any delay sequence occurs is very small. Since we know the probability of a given delay sequence we use the *Boole's inequality* to compute the probability of any delay sequence. Recall that:

$$\Pr[A \vee B \vee C \vee \cdots] \le \Pr[A] + \Pr[B] + \Pr[C] + \cdots.$$

Following from Lemma 9.23 and Lemma 9.24 we have that:

$$\begin{aligned}\Pr[\text{any } (U,R,\Pi)\,\text{occurs}] &\le \left(\begin{array}{c}\text{Number of}\\ \text{Delay Sequences}\end{array}\right)\left(\begin{array}{c}\text{Maximum Probability}\\ \text{that any delay}\\ \text{sequence occurs}\end{array}\right) \\ &\le 2^{2T_\infty}\binom{2T_\infty+R}{2T_\infty}e^{-R}\end{aligned}$$

Next, we use the following "death-bed" formula to simplify the above inequality.

$$\left(\frac{y}{x}\right)^x \le \binom{y}{x} \le \left(\frac{ey}{x}\right)^x.$$

We have:

$$\begin{aligned}\Pr\left[\text{any } (U,R,\Pi)\,\text{occurs}\right] &\le 2^{2T_\infty}\left(\frac{e(2T_\infty+R)}{2T_\infty}\right)^{2T_\infty} e^{-R}\\ &\le \left(\frac{2e(2T_\infty+R)}{2T_\infty}\right)^{2T_\infty} e^{-R}\end{aligned}$$

In the right part of the inequality, the first term is considerably large, while the second term is exponentially small in R. We want to choose a value for R such that the product of the two terms is small. We let $R = 2CT_\infty$ where C is a constant. Replacing in the previous equation we have:

$$\Pr\left[\text{any } (U,R,\Pi)\,\text{occurs}\right] \le \left(\frac{[2e(C+1)]^{1/C}}{e}\right)^{R}. \tag{9.4}$$

Definition 9.25 (High Probability) *An event E occurs with **high probability** if the probability that the event occurs that is $Pr[E] \ge 1 - P(n)$, where $P(n)$ is a polynomial. That is the probability that the event does not occur is polynomially small.*

Observation 9.26 *When $C \ge 4$, probability decreases exponentially in R.*

Example 10 If we take $C = 4$ then the probability is less or equal to 0.84^R.

Let:

$$\varepsilon = \left(\frac{[2e(1+C)]^{1/C}}{e}\right)^{R}.$$

Then $\Pr\left[\begin{array}{c}\ge 2P(2T_\infty+R)\\ \text{steal attempts occur}\end{array}\right] \le \varepsilon$, when $R = \Omega(\lg\left(\frac{1}{\varepsilon}\right))$.

9.6 Wrap-Up

We have shown that with **high probability** the number of steal attempts is small. We have come up with an event which explains why there are steal attempts. This event is a **delay sequence**. Next we counted the number of events. We computed the probability that such an event occurs. By summing up all the possible events we have shown that with **high probability** there are few steal attempts. With probability at least $1-\varepsilon$ the number of steal attempts is $O(P\left(T_\infty + \lg\left(\frac{1}{\varepsilon}\right)\right))$.

We have started our analysis using an **accounting argument**, based on a **work** bucket and a **steal** bucket.We have shown that at the end of the computation there are T_1 dollars in the **work** bucket and with probability at least $1-\varepsilon$ there are $O(P\left(T_\infty + \lg\left(\frac{1}{\varepsilon}\right)\right))$ dollars in the **steal** bucket. Thus, with probability at least $1-\varepsilon$:

$$T_P \le \frac{T_1}{P} + O\left(T_\infty + \lg\left(\frac{1}{\varepsilon}\right)\right).$$

9.6.1 Death-Bed Formulae

Figure 9.13 shows a list of useful ("death-bed") formulae that were used in the analysis of the *Cilk* scheduler.

Euler's inequality:

$$\left(1+\frac{1}{x}\right)^{x} \leq e \leq \left(1+\frac{1}{x}\right)^{x+1}, \tag{9.5}$$

$$\left(1-\frac{1}{x}\right)^{x} \leq \frac{1}{e} \leq \left(1-\frac{1}{x}\right)^{x-1}. \tag{9.6}$$

Boole's inequality (Union bound):

$$\Pr\left[A \vee B \vee C \vee \ldots\right] \leq \Pr\left[A\right] + \Pr\left[B\right] + \Pr\left[C\right] + \ldots. \tag{9.7}$$

Combination of y choose x bound:

$$\left(\frac{y}{x}\right)^{x} \leq \binom{y}{x} \leq \left(\frac{ey}{x}\right)^{x}. \tag{9.8}$$

Figure 9.13: Death-bed formulae.

6.895 Theory of Parallel Systems	Lecture 10
Cilk Implementation	
Lecturer: Charles Leiserson	*Scribe: Kunal Agrawal and Xie Yong*

Lecture Summary

1. *Compiling Cilk*
   This section gives some of the implementation details and clever tricks used in compilation of Cilk.

2. *Deque Protocols*
   Description of the TH and THE protocols Cilk uses for managing the deques.

10.1 Compiling Cilk

In this section, we will see some implementation details of conversion of Cilk source to C post-source. We will subsequently refer to this conversion as compiling the Cilk program. We will also see the reasons behind choosing to implement the conversion in this particular way.

The Cilk compilation path

The Cilk compilation path is shown in Figure 10.1. The Cilk source is run through the ***cilk2c*** translator which converts the Cilk source to C post-source. This C post-source is compiled using `gcc` and the object code is linked with the ***Cilk run-time system*** to produce the native binary code.

The software components that needed to be built to implement the Cilk system were `cilk2c` and `Cilk run-time system`. The `cilk2c` translator translates the Cilk commands like `spawn` and `sync` to C instructions that call the run-time system routines and do the parallel bookkeeping. The translation process is simple, because it only needs to transform Cilk keywords, leaving the C instructions as they are. In this section, we shall discuss some implementation details of the `cilk2c` translator. Some of the mechanisms used in the `Cilk run-time system` will be discussed in the next section.

Work-First Principle

Cilk's greedy scheduler operates on the "work-first" principle: it is better to amortize overheads against the critical path than against work. Recall that for a job with T_1 work and T_∞ critical path length, the Graham Brent greedy scheduler achieves the following bound on time taken to complete the job on P processors:

$$T_P \leq \frac{T_1}{P} + T_\infty \ .$$

Recall also that this bound is within a factor of 2 of optimal. Thus, a greedy scheduler with comparable performance should exhibit a running time something like :

$$T_P \leq C_1 \frac{T_1}{P} + C_\infty T_\infty \ . \tag{10.1}$$

Where C_1 and C_∞ are constants due to the overhead in the system. For parallel programs for which we can expect linear speedups, we have $P \ll T_1/T_\infty$. Thus, the first term in Inequality (10.1) is much larger than the second term. Consequently, in order to optimize the performance of the scheduler, we should try to move the overhead from C_1 to C_∞.

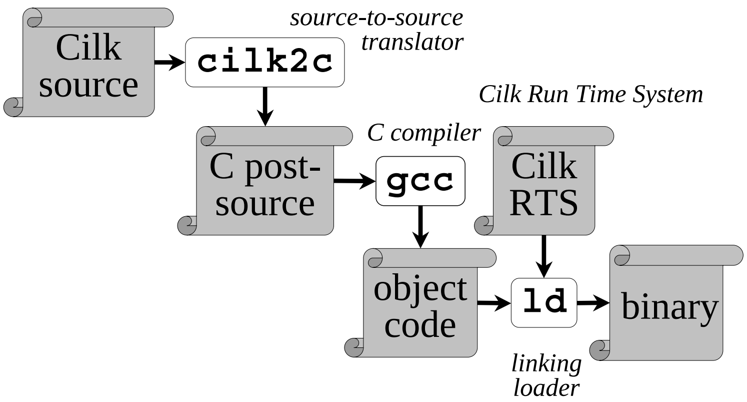


Figure 10.1: The compilation path for Cilk programs.

Shadow Stack

Cilk maintains a ***shadow stack*** made up of ***frames*** each of which holds all the live local variables for a single procedure. Since `cilk2c` has no control over the C stack, the code to store the local variables on the stack may duplicate some work done by the C run-time system. The overhead incurred due to this additional storage is small. In addition to storing the local variables, each frame has two extra locations ***entry*** and ***join***. The ***entry*** location holds the point at which the procedure should start executing when it is resumed in case it gets stolen and the ***join*** location holds the number of currently spawned children of the procedure.

Cilk's Compiler Strategy

The `cilk2c` converter generates two clones of each function: a ***fast clone*** and a ***slow clone***. The ***fast clone*** is the serial common case code and it is almost the same as what a C function looks like. The ***slow clone*** is the clone with the parallel book-keeping.[1]

Whenever a Cilk procedure is spawned, the fast clone is invoked after saving local live variables on the shadow stack. If a thread is stolen however, the slow clone is resumed at the other processor. Thus, a slow clone of a procedure is only executed if and when this procedure is stolen from one processor to another. which means that most of the overhead of executing the slow clone goes in the C_∞ term in Inequality (10.1).

As can be seen in Figure 10.2 all the procedures on a processor's deque are fast clones, except possibly the procedure on the top of the deque. Moreover, a procedure can have more than one outstanding child only if it has been stolen from some other processor in which case, it would be a slow clone. Thus, fast clones can have at most one outstanding child at any time, and this child is below them on the processor's deque. Thus, when a fast clone is being executed, it has no outstanding children.

Fibonacci Example

We can see how `cilk2c` compiles programs by considering the following Cilk program for computing the Fibonacci numbers:

[1]The slow clone is only 20-30 % slower than the fast clone. Its called the slow clone just to contrast it with the fast clone.

SLOW
FAST
FAST
FAST
FAST
FAST

Figure 10.2: Shadow Stack view of any processor

```
1: cilk int proc{fib}(int n) {
2:   if (n<2) return (n);
3:   else {
4:     int x,y;
5:     x = spawn fib(n-1);
6:     y = spawn fib(n-2);
7:     sync;
8:     return(x+y);
9:   }
10:}
```

This procedure is subsequently referred to as `fib` program:

Compiling `spawn` in the fast clone

Figure 10.3 shows the compilation of `spawn` in line 5 of the `fib` program. In the fast clone, whenever a procedure spawns a child, all the live local variables of the parent are stored on a shadow stack. The procedure also saves the appropriate `entry` location, in this case entry number 1 on the shadow stack and increments the `join` counter.

After the `return` from the spawned function, a check is made to see if the parent is still on this processor's deque. If it is, then the parent executes just as it would in C. If the parent has been stolen, the returned value is updated in the shadow stack and the `join` counter is decremented by 1. If the value in the `join` counter is now 0 and the parent is stalled on a `sync`, then this child was the last of it's siblings to complete in which case, this processor begins executing the parent. If this child has executing siblings or the parent is not stalled, then this processor's control goes to the scheduler, and it starts work-stealing.

Compiling `sync` in the fast clone

As shown in Figure 10.4, in the fast clone, `sync` statement compiles to a `no-op`. This is because a fast clone has no outstanding children when it is executing.

Compiling `spawn` in a slow clone

Figure 10.5 shows the compilation of the `spawn` statement on line 5 in the slow clone. The slow clone of a procedure is executed whenever it is stolen from a processor and resumed on another processor. Thus, in a slow clone, we need entry points to resume the thread after each `spawn`. At the beginning of the procedure, the slow clone reads the `entry` location and a `switch` statement dispatches control to the appropriate entry point as shown in Figure 10.5. The live variables are then restored from the shadow stack and the execution of the program continues.

As shown in Figure 10.5, the segment of the program that restores the local variables is enclosed in a false conditional. This construct prevents the program from restoring the variables when it reaches this

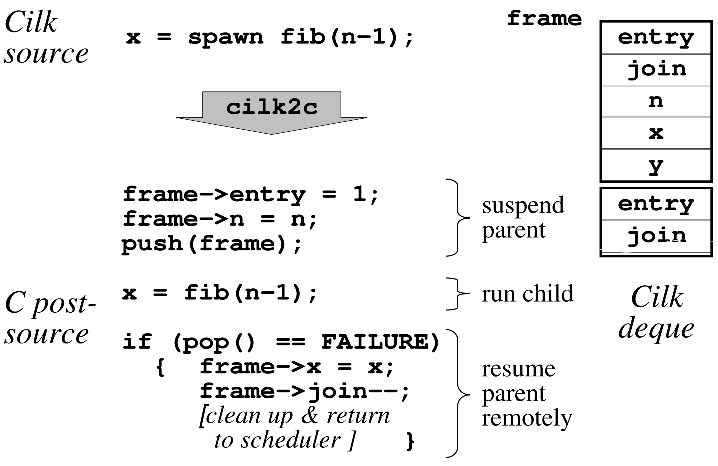


Figure 10.3: The compilation of `spawn` in the fast clone.

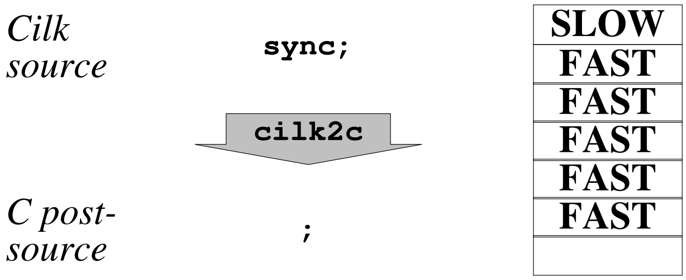


Figure 10.4: The compilation of `sync` in the fast clone.

point during the normal sequential execution, that is when it is not being resumed after a steal. Thus, this overhead of restoring variables is only incurred when this procedure has been stolen and can be amortized against the T_∞ term in Inequality (10.1).

Compiling `sync` in the slow clone

When the slow clone of a procedure reaches a `sync` point, it checks whether it has any outstanding children by examining the `join` counter in its shadow stack. If this counter is 0, then there are no outstanding children and the procedure continues executing. If there are outstanding children, then the processor starts work-stealing. Figure 10.6 shows the example for the `fib` program.

Breakdown of Work Overhead

The efficiency of Cilk's compiling strategy can be tested by using the Fibonacci number calculation benchmark. The Figure 10.7 shows the performance on various single-processors machines. The breakdown of time spent on doing the various tasks is also shown in the figure.

According to these benchmarks, the Cilk program is about 6 times worse than the equivalent C program in the worst case. This overhead might sound large, but the `fib` program has very little computation. Every

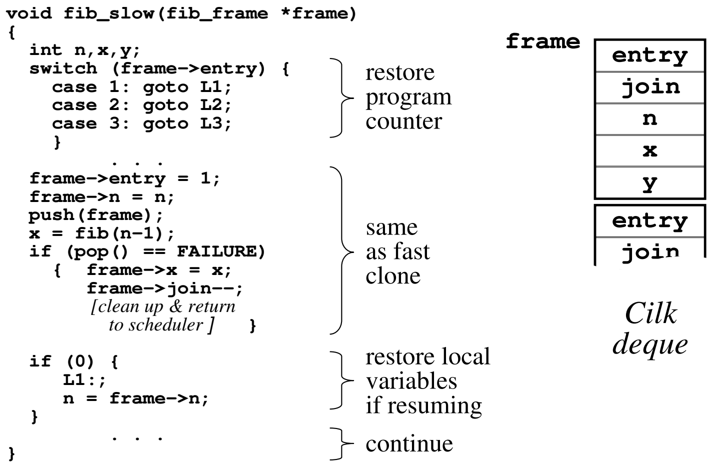


Figure 10.5: The compilation of `spawn` in the slow clone.

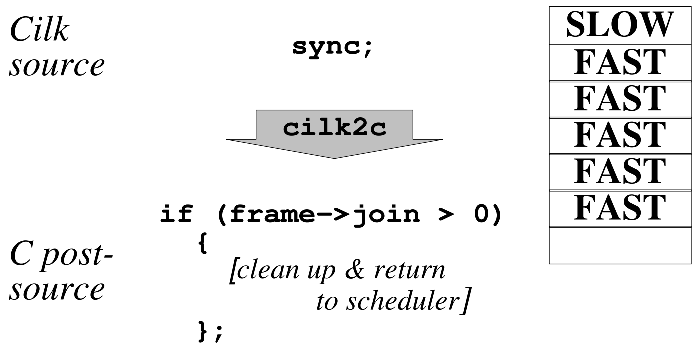


Figure 10.6: The compilation of `sync` in the slow clone.

procedure just does two decrements and one addition. On the other hand it has many spawns. In general, most programs are likely to have much more computation than `fib` does and in that case the spawn overhead will be amortized against the computation, that is, for most programs, we expect that

$$Number\ of\ spawns \ll Number\ of\ C\ function\ calls \ll Number\ of\ total\ instructions\ executed.$$

The fact that *fib* is only 6 times slower in Cilk than in C is actually encouraging. Most parallel programs are actually as fast in Cilk as they are in C. In practice, for most reasonable programs, the constant C_1 in Inequality (10.1) is close to 1.

10.2 Deque Protocols

In Cilk, each processor, called a ***worker***, maintains a ***ready deque*** (doubly-ended queue) of ready Cilk procedures. Each deque has two ends, a ***head*** and a ***tail***, from which procedures can be added or removed. The worker treats the deque as a stack, pushing and popping procedures from the ***tail*** of the deque. If a worker runs out of work, it becomes a ***thief*** and tries to steal work at the head of the ready deque of a

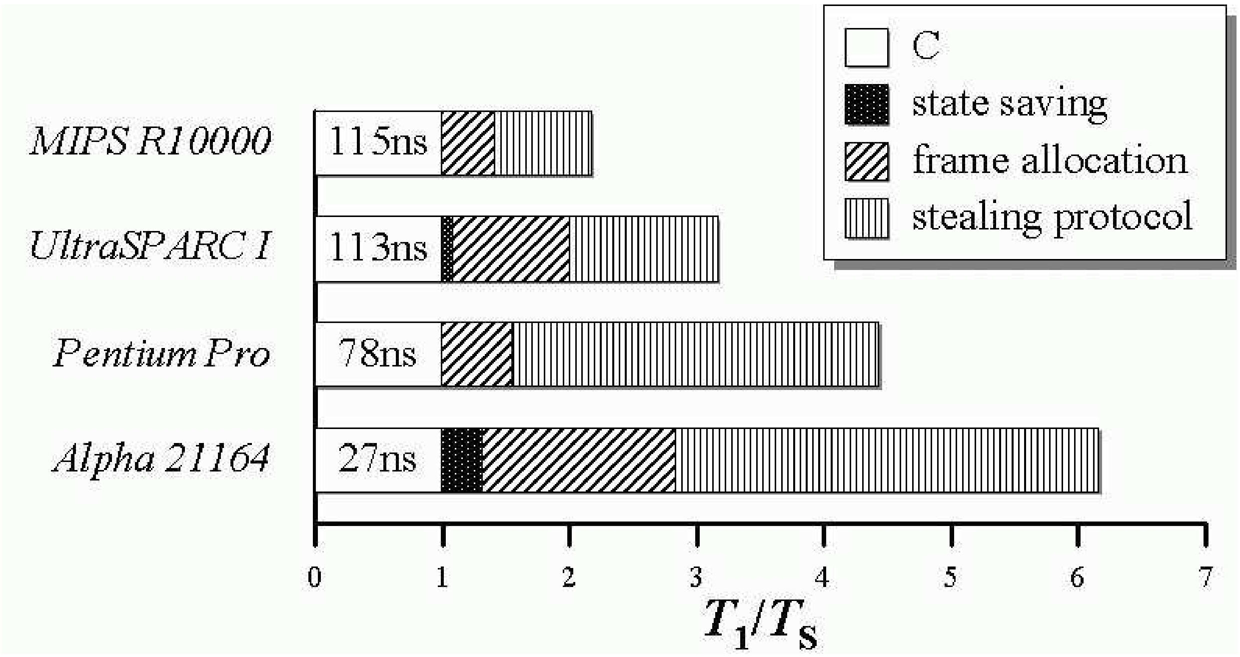


Figure 10.7: The breakdown of work overhead, benchmarked using *fib* on one processor.

victim processor. Work-stealing is implemented through shared-memory, and the thief and victim operate directly on the victim's ready deque. The main issue is how to resolve the potential race condition that may arise because of the concurrent accesses while at the same time, achieving good efficiency.

The protocol that Cilk adopts for managing deque make uses of three atomic shared variables T, H, and E, and it is called the THE protocol. We first present a simplified protocol that uses only two shared variables T and H designating the tail and the head of the deque, respectively. Later, we extend the protocol with a third variable E that allows exceptions to be signaled to a worker. The exception mechanism is used to implement Cilk's `abort` statement.

10.2.1 TH Protocol

The TH protocol is an almost non-blocking protocol for managing deque, and the pseudocode is shown in Figure 10.8. The main idea is to minimize work overhead by moving the costs from the worker to the thief. To arbitrate among different thieves attempting to steal from the same victim, a hardware lock is used, and the overhead caused by the lock can be amortized against the critical path.

The code assumes that the deque is implemented as an array of frames. The head and tail of the deque are determined by two indices T and H. The index T points to the first unused element in the array, and H points to the first frame on the deque. Indices grow from the head towards the tail, and most of the time, T $\geq$ H. The worker pushes and pops frames by altering T, while the thief only increments H and does not alter T. The lock L ensures that only one thief can steal from the deque at a time.

The push operation is always safe, because it does not involve any interaction between a thief and its victim. For the pop operation, there are three cases, shown in Figure 10.9. When there are many thieves, only one of them will succeed in stealing the frame because of the locking mechanism. Thus we only need to consider the case of one thief.

1. Both the thief and the victim attempt to obtain different frames from the deque concurrently. In this case both are successful, and they do not interfere.

```
push() {
  T++;
}

pop() {
   T--;
   if (H > T) {
       T++;
       lock(L);
       T--;
       if (H > T) {
            T++;
            unlock(L);
            return FAILURE;
        }
        unlock(L);
    }
    return SUCCESS;
}
```

(a)

```
steal() {
   lock(L);
   H++;
   if (H > T) {
       H--;
       unlock(L);
       return FAILURE;
   }
   unlock(L);
   return SUCCESS;
}
```

(b)

Figure 10.8: **(a)** Pseudocode of the actions performed by the worker/victim in the TH protocol. **(b)** Pseudocode of the actions performed by the thief in the TH protocol.

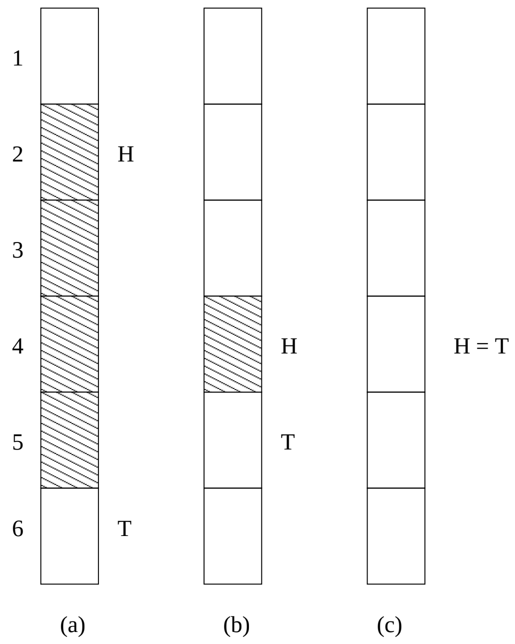


Figure 10.9: The three cases of the ready deque in the TH protocol. A shaded block indicates the presence of a frame at a certain position in the deque. The head and the tail are marked by T and H.

```
push() {
  T++;
}

pop() {
  T--;
  if (E > T) {
     T++;
     lock(L);
     T--;
     if (H > T) {
       T++;
       unlock(L);
       return FAILURE;
     }
     unlock(L);
   }
   return SUCCESS;
}
```

(a)

```
steal() {
  lock(L);
  E++;
  H++
   if (H > T) {
    E--;
    H++;
    unlock(L);
    return FAILURE;
  }
  unlock(L);
  return SUCCESS;
}
```

(b)

Figure 10.10: **(a)** Pseudocode of the actions performed by the worker/victim in the THE protocol. **(b)** Pseudocode of the actions performed by the thief in the THE protocol.

2. The deque contains only one frame. Either victim and thief will get the frame if the other is not making an attempt to obtain it. If both victim and thief try to get the frame, the protocol guarantees that at least one of them discovers that $\mathsf{T} > \mathsf{H}$. If the thief discovers that $\mathsf{T} > \mathsf{H}$, it restores H to its original value and retreats. If the victim discovers $\mathsf{T} > \mathsf{H}$, it will restart the protocol after acquiring the lock L, which guarantees that it will get the frame without interference from any thief unless a thief has already stolen the frame.

3. The deque is empty. A thief always fails to steal, and the victim also fails to pop the frame. The control passes to the Cilk runtime system.

The TH protocol contributes little to the work overhead. Pushing only involves updating T. Successfully popping a frame involves only 6 operations — 2 memory loads, 1 memory store, 1 decrement, 1 comparison, and 1 (predictable) conditional branch. In the case where both thief and victim simultaneously try to grab the same frame, the cost incurred by the lock can be considered as part of the critical-path overhead and does not influence the work overhead.

10.2.2 THE Protocol

The THE protocol extends the TH protocol to support signaling of exceptions to a worker. The index H in the TH protocol plays two roles: it marks the head of the deque, and it marks the point that the worker cannot cross when it pops. In the THE protocol, the two roles of H are separated into two variables: H, which now only marks the head of the deque, and E, which marks the point that the victim cannot cross. Figure 10.10 shows the pseudocode of the THE protocol.

The variable E is used to signal the worker that an exception has occurred. For example setting $\mathsf{E} = \infty$ causes the worker to discover the exception at its next pop. At this point, it returns the control to the run time system, which can then handle the exception appropriately. Different high values of E are used for different exceptions. This mechanism is used to implement abort in Cilk. The limitation of this approach is that the worker is notified of an exception only after it returns from a `spawn` for the first time after the exception is signaled. Thus if a procedure does a lot of computation before returning, it might continue to compute for some time before discovering the `abort`. Although no bounds are placed on how long it takes for a sub-computation to stop after the exception is signaled, the exception mechanism seems to work well in practice.

6.895 Theory of Parallel Systems Lecture 11

Project Proposals I

Lecturer: Charles Leiserson *Scribe:*

6.895 Theory of Parallel Systems Lecture 12

Project Proposals II

Lecturer: Charles Leiserson *Scribe:*

6.895 Theory of Parallel Systems	Lecture 13

Implementation of Memory Consistency

Lecturer: Bradley C. Kuszmaul — *Scribe: Seth Gilbert, Xie Yong*

Lecture Summary

1. *Sequential Consistency*
   We review the definition of sequential consistency.

2. *Uncached Consistency Algorithm*
   We present a simple algorithm and prove that it guarantees sequential consistency.

3. *Overlapping Operations and Caching*
   We discuss the difficulties introduced by allowing overlapping operations at a single processor, and by using caches.

4. *The MESI Protocol*
   We present the MESI protocol, and algorithm that takes advantage of caching to provide an efficient, sequentially consistent memory.

13.1 Memory Consistency

We first review the definition of sequential consistency in Section 13.1.1. We then define a system model in Section 13.1.2, and present an algorithm for implementing the usual memory operations, `load` and `store`, in Section 13.1.3. We then prove that this algorithm guarantees sequential consistency in Section 13.1.4.

13.1.1 Sequential Consistency

We review here the definition of sequential consistency, otherwise known as strong consistency.

Definition 13.1 *Assume that G is a DAG representing a multithreaded program, and that α is an execution (i.e., a single run) of G. The execution α is* sequentially consistent *if and only if there exists a topological sort (i.e., a total sort), S, of G such that for every load, the most recent (in the total order S) store to that memory location wrote the value that the load received.*

Note that there can be more than one topological sort of the DAG demonstrating sequential consistency.

13.1.2 System Model

We consider a model which contains of a set of processors, a set of memory modules and a communication network, as shown in Figure 13.1. We assume the following:

- The memory is organized into words.
- Processors execute `load` and `store` instructions on words.
- Processors send messages to the memory using the network.
- The memory modules perform `read` and `write` operations on the memory.

Example 11 Let the memory module M_i contain the set of all memory locations in which the low-order two bits of the page number are i. On a system with 2^{12} words per page, that means that M_i contains any memory address that when written in binary looks like:

`XXXX XXXX XXXX XXXX XXab XXXX XXXX XXXX`

where $i = 2 * a + b$, and $0 \leq a, b < 2$. M_i, then, consists of every address X for which

$$(X \gg 12) \ \& \ 3 == i$$

13.1.3 Uncached Consistency Algorithm

It turns out that it is easy to implement sequential consistency, if at each processor load and store operations do not overlap and processors do not cache memory values.

For this section and Section 13.1.4, then, we assume that each processor can only perform a single load or store operation at any given time. That is, once a processor begins an operation, it must wait until the operation completes before it can begin a new processor. We now present a simple algorithm in which there is no caching:

- `load`$(x)_i$: When processor i receives a `load` request for location x, it sends a message to the memory module containing the address x, asking for the current contents. Processor i then waits for the reply containing the value.
- `read`$(x)_j$: When memory module j receives a `read` request for location x, it reads the context of location x and sends a reply.
- `store`$(x, v)_i$: When processor i receives a `store` request to place value v in location x, it sends a message containing x and v to the memory module containing x and waits for a reply indicating that the operation is complete.
- `write`$(x, v)_j$: When the memory module j receives a write request, the memory writes v into location x and sends a reply indicating that the operation is complete.

13.1.4 Proving Sequential Consistency

In this section, we show that the Uncached Consistency Algorithm implements sequential consistency. To prove this, we need to first construct a topological sort of the program DAG, and then show that every `load` operations returns the most recently stored value to the same location in the total order.

Definition 13.2 *We sort the load and store operations in the same order in which the memory modules acutally process the corresponding messages. If the times at which two operations are processed by the memory modules overlap, then either order is allowed.*

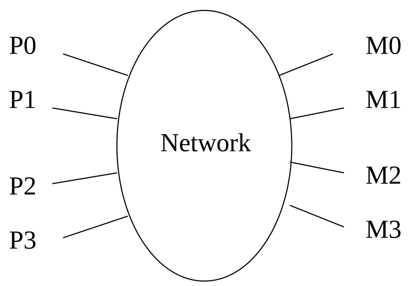


Figure 13.1: Model for processors to access memory through communication network.

More formally, we define sort S *as follows: given two operations in the DAG, i and j, we say that $i < j$ if the memory module accessed by i completes the operation associated with i before the memory module accessed by j completes the operation associated with j. If the operations overlap, we arbitrarily order i and j, for example, we let $i < j$ if the address associated with i is smaller than the address associated with j.*

Claim 13.3 *Sort* S *is a topological sort of the program DAG G where G=(V,E).*

Proof To prove that Sort S is a topological sort of the DAG, G, we need to show:

$$\forall\, i \in V, j \in V, \quad i \prec j \text{ (in } G) \quad \Rightarrow \quad i < j \text{ (in S)}\ .$$

If $i \prec j$, then i sends a message to the memory, and waits for the reply before allowing any of its successors to run. The memory acutally processes the operation before sending the reply, and therefore j cannot start execution until the memory operation for i has finished. The memory operation for j cannot start until j starts. Therefore, the memory operations for i finish before the memory operation j starts, and $i < j$ in the sort S. Hence the sort S is a topological sort of the program DAG. □

Claim 13.4 *Every load observes the most recent store to the same location in the total order.*

Proof Consider a particular location X. The total order of all memory operations induces a total ordering of the operations on memory location X. Hence if i and j are two operations on the same memory location, where i occurs before j, then $i < j$ in the total order. For each load, the memory returns the value most recently stored value, and hence every load observes the most recent store to that location. □

Theorem 13.5 *The* UNCACHED CONSISTENCY ALGORITHM *implments sequential consistency*

Proof Combining Claims 13.3 and 13.4, we have shown that Sort S is a topological sort of the program DAG, G and every load observes the most recent store to the same location in the total order. Therefore, by defintion of sequential consistency, the UNCACHED CONSISTENCY ALGORITHM implements sequential consistency. □

13.2 More Complicated Memory

Although the UNCACHED CONSISTENCY ALGORITHM implements sequential consistency, it has two problems:

- Operations at a single processor may not overlap.
- Processors are not allowed to cache memory values.

In Sections 13.2.1 and 13.2.2 we discuss why these two problems are difficult to solve. We then present the MESI algorithm in Section 13.2.3, which takes advantage of caches to implement sequential consistency more efficiently.

13.2.1 Overlapping Operations

An overlapping operation occurs when a processor begins executing a new operations prior to receiving a reply from an earlier operations. Allowing overlapping operations lets the processor take advantage of the parallelism in operations, which is illustrated in the following example.

Example 12 Assume there is no overlapping of operations.

```
R1 = *X;
R2 = *Y;
R3 = R1 + R2;
*z = R3;
```

The critical path of this code fragment is:

- 3 round-trip delays through the network +
- 2 read operations +
- 1 write operation +
- 1 add operation

If we can fetch X and Y in parallel, we can save one read and one round trip delay, leading to a critical path of:

- 2 round-trip delays through the network +
- 1 read operations +
- 1 write operation +
- 1 add operation

Notice that round-trip delays are significantly larger than the other operations.

It is very tough, however, for hardware to detect whether the algorithm allows such parallelism. As a result, we cannot perform overlapping operations, since the algorithm might rely on sequential consistency. The following example illustrates the problem with overlapping.

Example 13 Initially all memory locations are 0.

P_0	P_1
$*X = 1$	
$*Y = 1$	
	$R_1 = *Y$ // gets 1
	$R_2 = *X$ // might get 0

Assume that the operations are allowed to overlap at processor P_0. It is possible, then, that the contents of R_2 could be 0 at the end of the execution of this code fragment. This can happen when the message for the second operation, which sets $*Y = 1$ arrives at its memory module before the message for the first operation, which sets $*X = 1$ arrives at its memory module. In this case, R_1 can return the value 1, while R_2 returns the value 0. (At some later point, the first operation will complete and $*X = 1$. This may, however, be too late.) Notice that sequential consisteny has been violated: the execution DAG orders the writing of $*X$ earlier than the writing of $*Y$, while the memory modules order the operations in the opposite order.

To solve this problem, we can try to fix the network. We can keep the packets in-order between a processor and a memory module (pair-wise First-In-First-Out). This does not fix the problem, however. If all the packets are delivered in the order in which they are sent, then we can avoid this problem. This solution, though, is not scalable.

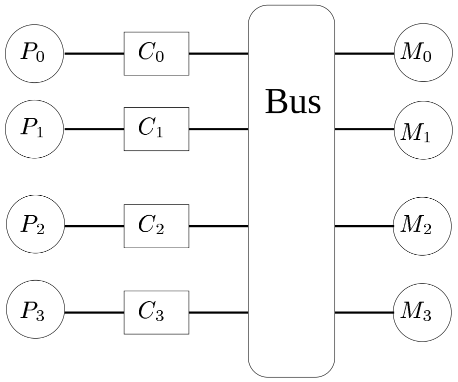


Figure 13.2: Diagram of a system with caches. The circles on the left represent processors, the circles on the right represent memory modules. The squares represent caches, and the oval represents a bus.

13.2.2 Caching

We now consider the possibility of caching memory locations. Consider again Example 12. If we allow caching, the critical path is reduced to:

- 1 read +
- 1 write +
- 1 add

in the case where all the values are cached. Caching, however, introduces a new problem. Memory may be modified by another processor, while the local cache is not updated, as shown in the following example.

Example 14

P_0	P_1
	$R_1 = *X$ // gets 0, and stores it in the cache on P_1
$*X = 1$	
$*Y = 1$	
	$R_2 = *Y$ // gets 1, strong consistency implies *X=1
	$R_1 = *X$ // still sees 0 because it is in the cache

The local cache of $*X$ on P_1 is not updated when $*X$ is updated to 1, so processor P_1 cannot see the changes of $*X$ in the memory.

13.2.3 MESI

The MESI protocol takes advantage of caches to implement a sequentially consistent memory more efficiently. Our system, as before, consists of processors, memory modules, and a network. Each processor, however, has a cache associated with it. (See Figure 13.2.) We also assume that the network is implemented by a bus. This means that every processor sees every message that is sent on the network. This allows processor to silently update their state in response to ongoing operations. Note that this also implies that only one processor can broadcast at a time, so a separate protocol is needed to handle contention. As a result, this does not scale all that well. However, this is the basis for more complicated algorithms that do scale.

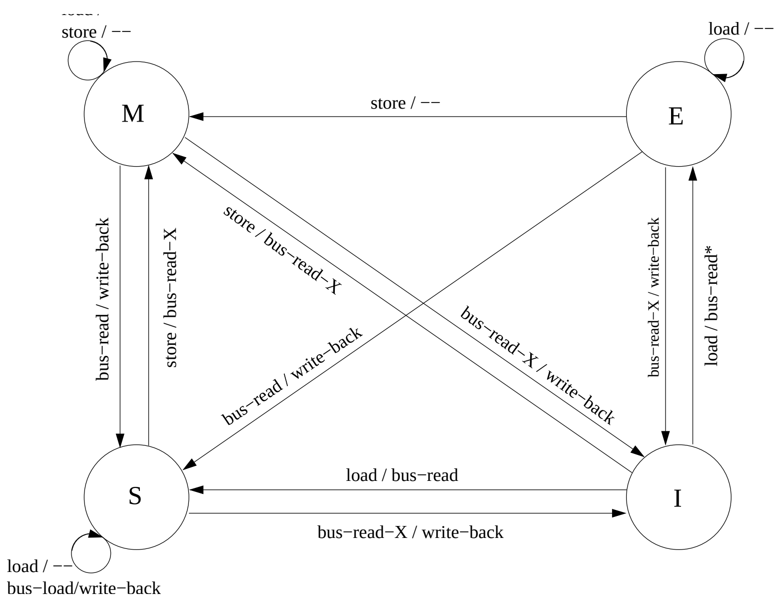


Figure 13.3: State diagram for the MESI protocol. Circles represent states, arrow represent transitions. Arrow are labeled `input / response`. The `input` is either a processor action (i.e., `load` or `store`) or a bus action (i.e., `bus-read` or `bus-read-X`). The responses are bus actions (i.e., `bus-read`, `bus-read-X`, or `write-back`).

The caches are (typically) organized into lines, each containing several words of data. As a result, if the algorithm chooses to cache location 100, it also fetches (and caches) locations 101, 102, and 103. This is useful, since many programs exhibit *spatial locality*: often, when a program performs an operation on a given location in the memory, it is quite likely to perform another operation on a nearby location soon afterwards. There are two types of caches.

- *Write-through caches* immediately notify the main memory of every `store` operation. This ensures that the value in the cache is always the same as the value in the main memory. (This leads to simple memory algorithms.)

- *Write-back caches* do not always immediately notify the main memory of a `store` operation. Sometimes, the cache maintains a new value, unknown to the main memory. Eventually, the value will get "written-back" to the main memory, but this may take a long time. This is more efficient, since it saves network bandwidth when the value does not really need to be written back.

The MESI protocol assumes that the system contains write-back caches. It maintains two extra bits of data for every line in the cache to determine the state of that line of the cache. Figure 13.3 presents the state transition protocol enforced by the algorithm. There are four possible states:

- *[M]odified*: No other cache in the system contains this line in its cache in the M, E, or S state. The cache contains a new value that needs to be written-back to main memory.

- *[E]xclusive*: No other cache in the system contains this line in its cache in the M, E, or S state. The cache contains the same value as main memory.

- *[S]hared*: No other cache in the system contains this line in its cache in the M or E state.

- *[I]nvalid*: The cache line contains no data.

A `load` operation can take place if the appropriate cache line is in the M, E, or S state. That is, a processor can read the data whenever it is available in the cache. A `store` operation can take only only if the appropriate cache line is in the E or S state. That is, a processor can only modify the memory when it it is the only cache maintaining a copy of the data. The operations are implemented as follows:

- `load`$(x)_i$: If processor i does not have the location x cached in the M, E, or S state, it sends a message on the bus requesting the location x in the S state. (This is notated in Figure 13.3 as a *bus-read*.) Since the message is broadcast on the bus, all other processors see this message. If some processor, j, sees this message and discovers that it has cached x in the M state, then it writes x back to memory and sets the state of x to S. (This is notated in Figure 13.3 as *write-back*.) If processor i sees the value written back to memory, it takes the value written and stores it in its own cache in the S state. (This is referred to as "snooping" the bus.) If no processor writes back a value to memory, then the memory module sends a message back to processor i indicating the current value of x, Processor i then caches the value of x and sets the state to S.

- `store`$(x, v)_i$: If processor i has location x in the E state, it sets the state to M and modifies the cached copy, setting it to v. If the state of x is already M, then again i can simply modify the cached copy. If x is in the S or I state, then i broadcasts a message on the bus requesting exclusive access to x. (This is notated in Figure 13.3 as a *bus-read-X*.) Every processor that has x in the S or E state sets the state of the cache line to I. If some processor j has x in the M state, then j writes x back to memory. If no processor writes the value back to memory, then the main memory broadcasts the value of x on the bus. Either way, processor i sees the value of x on the bus and updates its cache, setting the state to M. Processor i can then modify the value to v directly in the cache.

Further details of the MESI protocol can be found in Figure 13.3. Consider the following example, where $*X$ and $*Y$ are initially zero:

Example 15

	P_0	P_1
(1)		$R_1 = *Y$
(2)	$*X = 1$	
(3)	$*Y = 1$	
(4)		$R_1 = *Y$

In line 1, P_1 sets R_1 to zero, and sets the cache-line containing Y to S or E. In line 2, processor P_0 updates the value of X, and as a result the cache-line containing X is set to the M state. In line 3, processor P_0 similarly wants to update the value of Y. However, P_0 first has to obtain exclusive access to that cache-line. Processor P_0 therefore broadcasts on the bus a message requesting exclusive access to Y. When processor P_1 sees this request, it broadcasts its value of Y, and sets the state to I. Processor P_0 records this value of Y and sets the state to M. It then modifies the value directly in the cache. In line 4, processor P_1 wants to read Y. As a result, P_1 broadcasts a request for shared access to Y. Processor P_0 sees this request, and writes the value of Y back to main memory, setting its state locally to S. Processor P_1 snoops the value, and records the value of Y in its cache, setting it to state S.

Aura Rayuela Julio Cortazar

Gabriel Garcia Marquez

*The response `bus-read*` indicates that the bus action resulted in an exclusive read.

6.895 Theory of Parallel Systems	Lecture 14

Competitive Snoopy Caching

Lecturer: Bradley C. Kuszmaul — *Scribe: Advait D. Karande and Sriram Saroop*

Lecture Summary

1. *Setup for snoopy caching algorithms*
   This section briefly describes the arrangement of the processors, caches, memory and their interaction by means of the bus.
2. *Two strategies to ensure correctness while updating replicated blocks*
   This section introduces the two commonly used strategies, pack rat and exclusive write, which are employed to ensure correctness while writing a replicated block. We also compare the two algorithms, and briefly introduce the algorithm suggested by Goodman.
3. *Introduction to Competitive Algorithms*
   We define the terms - online algorithms and c-competitive algorithms. This sections also illustrates competitive algorithms by means of an example (due to Larry Rudolph).
4. *Competitive Snoopy Caching*
   In this section, we introduce competitive algorithms for snoopy cache multiprocessor systems and provide one such algorithm for direct-mapped caches.

14.1 Setup for snoopy caching algorithms

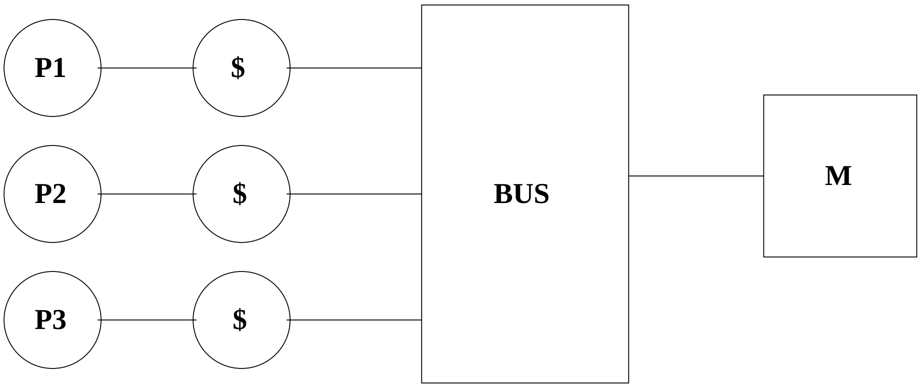


Figure 14.1: Setup for competitive snoopy caching

The competitive snoopy caching algorithm uses a MESI like protocol. The set up of the system is illustrated in Figure 14.1. We have processors each of which has a cache. The caches and the memory are interconnected through a bus. We observe the following characteristics in the system:

- Caches and Memory can see all the messages in the bus.
- Only one entity, either cache($) or memory(M) can send a message at a time.
- Processors(P) talk to the caches($) which in turn talk to the bus.

14.1.1 Why do caches work ?

Localities of Reference:

- *Temporal Locality* : It is the property that if a processor refers(reads/writes) to a memory location once, it is likely to refer to the same memory location again in the near future.
- *Spatial Locality* : It is the property that if a processor accesses(reads/writes) a location v, it likely to access locations that are near v , like v-1, v+1, v-2, v+2 etc.

It has been observed that programs do exhibit temporal locality and spatial locality. Since they exhibit spatial locality, caches are organized as blocks of say size p. Hence, whenever a processor requests for a single memory location, it receives the memory location and p-1 neighboring locations through the bus.

Let us compare the cost in terms of the number of cycles for different values of p. We assume that it takes one cycle for every request to be sent and one cycle for every byte of data that is got back. We consider two cases wherein 1 byte and 128 bytes are requested respectively.

- *1 byte requested*: Let us first consider a request for 1 byte. If p=1, it takes an unit cycle to send the request and an unit cycle for the memory location(1 byte) to come back. For larger p (p>1), it takes an unit cycle to send the request and p cycles for the memory location(1 byte) and its p-1 neighbors to come back. Hence, it takes a total of p+1 cycles.
- *128 bytes requested*: We proceed to consider another case wherein 128 bytes of data are requested. In the case when p=1, the processor sends a request for each byte, which takes an unit cycle each time, and receives 1 byte for every request, which again takes an unit cycle each time. Hence, for 128 bytes, it takes 128 request unit cycles and 128 bytes of data response unit cycles, amounting to a total of 256 cycles. This corresponds to an *uncached read*. In the case when p=128, it takes 1 request unit cycle, because of which a block of p=128 bytes is returned. Hence this takes 129 cycles. However, since the rest of the 127 bytes are already cached, no more cycles are needed. This corresponds to a *cached read*.

Therefore, for a reasonably large cache block size, we notice an improvement by a factor of 2 in the number of cycles needed.

14.2 Two strategies to ensure correctness while updating replicated blocks

14.2.1 An introduction to the Pack Rat and Exclusive Write algorithms

Blocks are sometimes stored at more than one processor, since we use caching to store the blocks at each processor's local cache. Hence whenever such a replicated block is being updated, we need to ensure that we communicate this change to other processors that have this replicated block. There are two commonly used strategies that address this problem.

- *Pack Rat* :

  If the processor knows that someone else has a copy of the block, it goes ahead to write onto the bus, and also *updates everybody* who has a copy. The processor incurs an unit cycle for writing the memory location, and since there is no reply, that is the only cost it incurs. Thus, a pack rat operation requires just 1 cycle.

- *Exclusive Write* :

  The processor which is writing to a memory location invalidates all processors that hold a copy of the block to which the memory location belongs to. This means that the exclusive write operation incurs only 1 unit cycle, as the only cost incurred is the communication cost for invalidating the cache-blocks of all the other processors.

14.2.2 Good cases for pack rat and exclusive write

Let us consider a couple of good cases for each algorithm.

- *Good case for exclusive-write* : Suppose a given processor wants to perform k consecutive writes, then exclusive-write would incur a lesser cost (just 1 cycle) since all other processors have been invalidated for the first write itself. Whereas pack-rat would have to update all other processor-caches for every write (incurring a cost of k cycles) which evidently is more expensive.

- *Good case for pack-rat* : Suppose that another processor wants to read a memory location after a processor has performed a write. In this case, exclusive-write would incur p+1 cycles for the read, as this processor's block in the cache would have been invalidated by the previous processor's write on the replicated block. Thus, it needs to read the entire cache block of p bytes. Whereas pack-rat would incur 0 cycles, as the previous processor would have updated its cache block.

14.2.3 A bad case for exclusive write

repeat k times
    P_0 writes to location v
    P_1,...,P_{n-1} read location v

The total number of read cycles needed for this task in *exclusive-write* is *k(n-1)p*. This is because the n-1 processors have to read the cache block that contains location v in every iteration as that cache block is invalidated in each iteration.

However, the total read cost incurred by *pack-rat* is only *k+np*, since the n processors have to load their caches only initially (cost = np), and the updates cost k cycles over all iterations. There are no further read costs incurred since the cache blocks of the processors are updated each time P_0 writes to location v. For large k, $np \ll k$. Considering the ratio of the performance of exclusive-write to that of pack-rat as k $\rightarrow \infty$, we get :

$$\lim_{k\rightarrow\infty} \frac{k(n-1)p}{k+np} = \lim_{k\rightarrow\infty} \frac{(n-1)p}{1+(np/k)} = (n-1)p \tag{14.1}$$

Hence in this case, exclusive-write is worse than pack-rat by a factor of (n-1)p.

14.2.4 A bad case for pack rat

P_0 reads location v
P_1 reads location v
repeat w times
    P_0 writes to location v

The first two reads by P_0 and P_1 each incur p+1 cycles in both exclusive-write and in pack-rat as their cache blocks have to be loaded initially. The w writes by P_0 incurs a cost of w cycles in pack-rat because in every iteration P_0 has to update all other replicated cache blocks which contain v. Whereas it incurs only one cycle in exclusive-write since the invalidation takes up only one cycle and does not have be repeated for each iteration.

Hence, the total cost for *exclusive-write* is (p+1)+(p+1)+1= 2p+3.

The total cost for *pack-rat* is (p+1)+(p+1)+w= w+2p+2.

Hence for large w,

$$\lim_{w \to \infty} \frac{w + 2p + 2}{2p + 3} \to \infty \tag{14.2}$$

This is an unbounded limit on the ratio of the performances.

14.2.5 Goodman's algorithm

Goodman's algorithm [10] alternates between the two strategies.

```
On first write, update everything else
On second write, invalidate everything else
```

It sounds fine for the first bad case that affected exclusive-write in Section 14.2.3 because Goodman's algorithm acts like pack-rat for the first write. But it is still unclear if this is an optimal strategy.

14.3 Introduction to Competitive Algorithms

Consider an algorithm that takes an input sequence, σ and produces an output sequence τ.

Definition 14.1 (Online algorithm) *An algorithm is online if it can produce the i^{th} output, τ_i having seen only the first i inputs: $\sigma_1, \sigma_2, ..., \sigma_i$. Otherwise, the algorithm is considered to be offline.*

The *intuition* is that online algorithms cannot look into the future.

Definition 14.2 (Competitive algorithm) *An online algorithm, A is c-competitive (or competitive) if and only if $\exists$ d $\forall$ sequences σ, such that: cost(A) $\leq$ c.cost(optimal offline algorithm)+d , where c and d are independent of the sequence, σ.*

Definition 14.3 (Strongly c-competitive algorithm) *A c-competitive algorithm is strongly c-competitive for a given c such that no better c can be found to satisfy the definition for c-competitiveness.*

14.3.1 An example of a competitive algorithm (Due to Larry Rudolph)

Let us consider the ski-rental problem. Suppose that skis cost \$10 to rent and \$100 to buy. What would the optimal strategy be?

We can construct a 2-competitive algorithm in the following manner. The strategy would be to rent until we spend the purchase price and then proceed to buy the ski. This would mean that we would rent the ski 10 times and only after that do we proceed to buy the ski. Obviously, the best strategy would be to rent the ski if we plan to go skiing less than 10 times. The algorithm that we constructed follows this as well. In the case when we plan to go for skiing more than 10 times, we would be spending 200\$, whereas the optimal offline strategy had we known the future would have been to purchase the ski beforehand for 100\$. Therefore, in the worst case our algorithm is at most a factor of 2 off the optimal offline strategy. Hence, our algorithm is 2-competitive.

14.4 Competitive Snoopy Caching

In a snoopy cache multiprocessor system, each processor has a cache memory in which it can store blocks of data. (Refer to Figure 14.1) The *snoopy* cache of each processor actively monitors the activity on the bus and in its processor and dynamically determines which blocks of data to keep and which ones to discard. This strategy is called *block retention strategy.*

Karlin et al. [11] introduced competitive snoopy caching algorithms for various cache architectures. A cache architecture is defined by several parameters such as, whether the cache is direct-mapped or associative, whether it snoops on single variables or on blocks of data and so on. We will focus on the competitive algorithm for *direct-mapped* caches.

The characteristics of direct-mapped snoopy cache model are as follows.

- Direct-mapped: Caches are characterized by a hash function $h(B)$, which gives a unique cache-line in which block B would reside. An example of a simple hash function would be $h(B) = b0 \textit{ modulo } c$, where b0 is the block number of B and c is the cache size in terms of the number of cache lines. In other words, each memory block maps to a unique cache-line in the cache. This is in contrast to *k-way set associative* or *fully associative* caches, wherein a memory location may be mapped to multiple cache-lines.
- Each read or write request of a processor for a memory variable is satisfied only after its cache has the block containing the variable.
- Caches snoop only on variables and not on the blocks.

We now provide a setup for analyzing the Direct-Mapped Snoopy Caching algorithm (**DSC**).

We let p be the size of each cache-line in bytes, $[v]$ denote the the block containing variable (i.e. memory location) v. Usually, $[v]$ would be calculated as $\lfloor \frac{v}{p} \rfloor$.

Processors perform a sequence of memory transactions, each of which can be of one of the two types:

- $\mathrm{READ}_i(v)$: Processor i requests the value of variable v.
- $\mathrm{WRITE}_i(v)$: Processor i requests that a new value be written to variable v.

In response to a request by processor i, its cache performs one or more actions from the following domain. Associated with each action is the cost, which reflects the time spent or bus cycles used in performing the actions.

$Fetchblock(i, B)$:	Block B is added into cache i. Since it requires fetching p bytes over the bus, it costs p.
$Writeback(i, B)$:	Block B is sent over the bus to the main memory. Again, this requires sending p bytes over the bus and costs p.
$Drop(i, B)$:	The cache-line containing block B is made empty. The action costs 0.
$Supply(i, v)$:	When block $[v]$ is present in cache i, it supplies variable v to processor i. The action costs 0.
$Updatelocal(i, v)$:	When block $[v]$ in cache i is the only cached copy of the block, variable v is updated in the cache. The action costs 0.
$Updateglobal(i, v)$:	When block $[v]$ is in cache i, variable v is updated in the cache and is broadcast on the bus. This action requires sending unit data over the bus and costs 1.
$Finishwrite(i)$:	It occurs at the end of a write request. Its cost is 0.

14.4.1 Direct-mapped snoopy caching algorithm (dsc)

In the direct-mapped snoopy cache model described above, the only freedom a cache has has is in deciding when it should drop a block. It makes this choice using the dsc algorithm.

The main data structure used in this algorithm is an array of counts $W_i[B]$, which is used to decide when to drop block B from cache i. The values in $W_i[B]$ range from 0 to p. The algorithm is presented below.

To read variable v on processor i [$\mathrm{READ}_i(v)$]:

B := [v]
if $W_i[B] == 0$

```
        then GETBLOCK(i, B);
        else W_i[B] := min(p, W_i[B] + 1);
    Supply(i, v);
```

To write variable v from processor i [WRITE$_i(v)$]:

```
    B := [v];
    if W_i[B] == 0
        then GETBLOCK(i, B);
        else W_i[B]:=p;
    if ∃j ≠ i such that W_j[B] ≠ 0
        then Updateglobal(i, v);
             W_j[B] := W_j[B] − 1;
             if W_j[B] == 0
                then Drop(j, B);
        else Updatelocal(i, v);


  Procedure GETBLOCK(i, B)
    if ∃B' such that h_i(B') = h_i(B) ∧ W_i[B'] ≠ 0
        then          ▷ B collides with B', so drop B'
             if W_i[B'] == 0
                then Writeback(i, B);
             W_i[B'] := 0;
             Drop(i, B');
    Fetchblock(i, B);
    W_i[B] := p;
  end GETBLOCK;
```

Let's look at some of the characteristics of this algorithm. It maintains two invariants.

- $W_i[B] = 0$ if and only if block B is not in cache i.
- If i is the last processor to modify a dirty block B , then $W_i[B] = p$.

A block is fetched into a cache only when the corresponding processor performs a read or write to a variable in the block and the block is not in the cache. After fetching block B into cache i, its count for that processor ($W_i[B]$) is set to p. It is dropped from cache i, when the count $W_i[B]$ reaches zero. In case of a read operation, if the required block is found in the cache, then its count is incremented by 1 up to a maximum of p. Similarly, on a cache hit during a write operation, the count for the corresponding block $W_i[B]$ is set to p. Thus, a cache hit on read or write operation increases the likelihood that the corresponding block will remain in that cache. Moreover, during a write operation, if the required block B is replicated, then the writing processor i needs to broadcast the update on the bus. Here each other cache containing the block is guilty of causing this bus cycle. Hence, a write to a replicated block decrements the count in one of the other caches (say j) sharing the block. Thus a cache hit on write operation to a shared block increases the likelihood that the block will be dropped from other caches holding the block.

Now we state the following theorem about the competitiveness of this algorithm.

Theorem 14.4 *Algorithm dsc is strongly 2-competitive, that is, for any sequence σ and any online or offline algorithm A,*

$$C_{dsc}(\sigma) \leq 2 * C_A(\sigma) + k$$

The constant k depends only on the initial cache states of dsc and A, and is zero if all caches are initially empty.

A formal proof of this theorem will be presented in the next lecture. Intuitively, this algorithm is similar to the ski-rental algorithm mentioned in Section 14.3.1, which is also a 2-competitive algorithm. We maintain a (replicated) block in a cache until the overhead of maintaining it becomes equal to the cost involved in fetching it from the memory.

6.895 Theory of Parallel Systems	Lecture 15

Snoopy caching and the Spin-Block Problem

Lecturer: Bradley C. Kuszmaul and Michael Bender *Scribe: Sharad Ganesh and Neelkamal*

Lecture Summary

1. *Competitive snoopy cache proof*
   This section discusses the proof by induction of snoopy bus-based caching algorithm.

2. *Directory-based caching systems*
   This section gives a brief overview of the algorithm and its scalability as compared to the snoopy cache algorithm.

3. *Randomized algorithm for the Ski-rental Problem*
   This section introduces the Spin-Block problem, which is a continuous version of the Ski-buy problem.

15.1 Strategy for the proof (Snoopy caching)

A competitive algorithm that is within a factor of 2. The idea is to keep these buckets such that:

The economics of the algorithm:

$W_i(B) = p$; whenever we have to fetch block B from memory(M).

$W_j(B) --$; when i writes and j has money in its cookie-jar. Whenever I do a write, I decrement somebody else's cookie jar. At some point of time they are going to put money into the cookie-jar, so I am actually going to charge them something. If nobody has any money left, then they all have to invalidate their copies in their caches and then I will not have to do a broadcast in future.

$W_i(B) = 0$; when I drop a block. A reason I could drop a block is the possiblility of a conflict with another block or maybe somebody else decremented my cookie-jar count to 0.

$W_i(B) ++$; We increment the count upto a point(only upto p), when we read a block.

Some facts about the algorithm

It does tell us when there is going to be a switch from a write-through to a write-back. Basically, you do a switch from pack-rat, where everybody tries to hang onto everything, to the exclusive write where I can do writes to my local cache and not tell anybody. Basically, I have to broadcast(keep updating everybody) as long as everyone is reading.

We are not too far off from the optimal, because everytime I did one of these operations that add money, there was already doing an operation that cost me that much (anyway $W_i(B) = p$ cost p cycles). So i'm putting money into the cookie-jar(i.e. p).
So, anybody who is going to do work and steal money from my cookie-jar, that work just charge it to me. Otherwise, anytime I increment it, because I did a read, its saying if it weren't in the cache, I'd be happy to pay something to have it gotten into the cache, because the read would really have cost me a lot more.

15.1.1 Strategy

Define a potential function, which function that tells us how far off our algorithm is from the offline algorithm A.
This is a function of the cache states of algorithm A(which is the one we are trying to beat - to come up within a factor of 2 of A). Those cache states after processing i steps in sequence of operations(reads and writes). The sequence we are talking about is:

- τ: which is the ops for A for the first read/write (tagged "A")
- ops for dsc for the first read / write (tagged "dsc")
- supply / update at the end of the first read / write. (tagged "both" - because both have to do it)

Basically both the algorithms are constrained in the sense that, when they see the first read, they can do whatever they want to, but you have to end with the supply which gives the value to the processor.

We will be taking all the operations for A and tag them as specified in the bracket as tagged "A". It is important to know the order in which the ops happened.

15.1.2 What is this potential function ?

$$\phi(t) = \sum_{(i,B)\in S_A} (w_i(B) - 2p) + \sum_{(i,B)\notin S_A} -w_i(B). \tag{15.1}$$

where: $(i, B) \in S_A =¿$ A has B in cache i at step t. Basically, the set of things that A caches.

Recall $w_i(B) = 0$ iff it is not in cache i. So, no matter how big the memory is, there is only a certain amount of non-zero values in the sum in equation (1).

15.1.3 Proof by induction of the following statement

$$COST_{dsc}(t) - 2COST_A(t) \leq \phi(t) - \phi(0). \tag{15.2}$$

where:
$COST_{dsc}(t)$: cost of doing dsc upto time t
$COST_{dsc}(t)$: cost of doing A upto time t
RHS : Amount the potential function has increased since we started.

Theorem 15.1

$$COST_{dsc}(t) \leq 2COST_A(t) + k$$

where: k is essentially the difference of the potential functions.

If cache started out with 0, i.e. completely empty, then k = 0. But, if cache started out with some value, that may favor A, k covers for that.

By construction:

$$\phi_t \leq 0. \tag{15.3}$$

Basically, the $W_{i's}(B)$ are between 0 and p. ($0 \leq w_i(B) \leq p$).
Then we take some point of those numbers, subtract 2p and then $-w_i(B)$ makes it more negative.
So, we are taking a negative number + something having to do with the initial state.
Just let,

$$k = -\phi(0)$$

If the caches are completely empty at the beginning of the algorithm, $\phi(0)$ will turn out be be 0. So, we will later see that this will actually prove $k = 0$, if the cache starts out empty.

Proof
Base case for induction (t=0)

LHS:Neither algorithm has done anything yet, so their costs are both 0.
RHS: $\phi(0) - \phi(0) = 0$.

Both sides are 0.

Inductive case: Basically, we want to show something about the changes in the costs and how they are related to the change in potential function. We would want to show:

$$\Delta COST_{dsc} - 2\Delta COST_A \leq \phi$$

where:

$\Delta COST_{dsc}$ denotes the change in cost of dsc.
$\Delta COST_A$ denotes the change in cost of A.

We present a case analysis for the above:
Idea of the analysis:

We have the sequences, that we'd discussed earlier, and for everything in the sequence the property holds. i.e. We have to show that, if one of them drops a block or does a fetch or an update, the changes should have this property.

We will go through some of the cases, but for the rest you can refer to the Goodman's paper [9].

Case 1:
Step: A does a fetchblock(i, B), which is fetching over the bus to get a block into cache i of block B.

$$\Delta COST_A = p$$

$$\Delta COST_{dsc} = 0$$

This is because, dsc did'nt do anything

Now we have to show:

$$\Delta\phi \geq 2p$$

Before the action: $(i, B) \notin S_A$
This is because, none of these snoopy caches do a fetch unless they do a read / write. Moreover, they don't a fetch if they already got it cached. So, before the action it was'nt in the cache.

After the action: $(i, B) \in S_A$
After the action, it is in the cache.

So,

$$\Delta\phi = w_i(B) - 2p - wi(B)$$

Therefore, $\Delta\phi = -2p$
Therefore, whatever A does if it does a fetch, potential function does the right thing.

Case 2:
A drops (i, B)

At this point 'A' has more freedom to do something different than we did. 'A' might drop something that you didn't drop because we were still paying something and so could'nt drop it. A might decide to drop something and let the slot be occupied by something else. In that case, I won't be incurring any cost for update later.

$$\Delta COST_A = 0$$

$$\Delta COST_{dsc} = 0$$

Now we have to show:

$$\Delta\phi \geq 0$$

So we have,
$w_i(B) + 2p - w_i(B)$
The second $-w_i(B)$, because it is no longer in the cache.

$$\Delta\phi = -2w_i(B) + 2p$$

The above term is always greater than or equal to 0, because $0 \leq W_i(B) \leq p$.

Case 3:
dsc does a fetchblock(i, B)

$$\Delta COST_{dsc} = p$$

$$\Delta COST_A = 0$$

Now we have to show:

$$\Delta\phi \geq p$$

Therefore here, $wi(B)$ changes from 0 to p.
Before it was'nt in the cache($wi(B) = 0$), and then when we did a fetch we set($wi(B)$ = p).
Since we did A's operations first, we must have (i, B) in A's cache at the end of A's ops. Both algorithms can be supplied from the cache. The key point is since we did A's ops first, we know it is in the cache.

Therefore,

$$\Delta\phi = p$$

For the other cases, you can read the Goodman's[1] paper. □

15.2 Directory based caches

The problem with MESI is the bus.

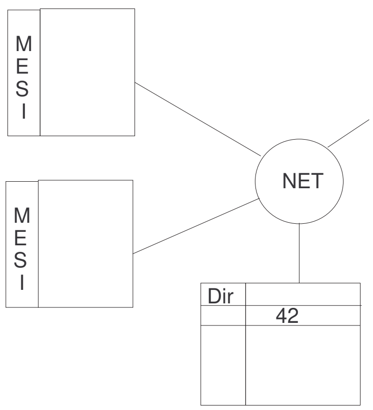


Figure 15.1: MESI network setup

Some processor wants to obtain E(exclusive) access, So send a message to M saying you want E access.

Case 1: Nobody has it. Memory says you have got it and changes the state.

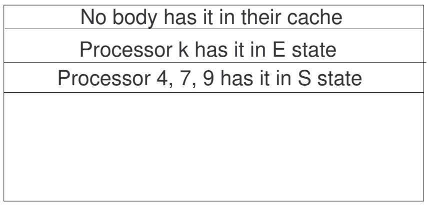


Figure 15.2: A typical directory entry

Case 2: Some other proc has it in E state. M sends message to the other processor. The other processor sends it to M. If it has been modified, it has to send it back to M. In any case M sends it to P. Real machines do a shortcut sending a message directly to the processor.

Directory based cache analysis
None.

15.3 Spin-Block Problem

There are basically two ways in which a process waits for a lock

1. Spin: keep on waiting in a nop loop till get the lock Cost = spin time
2. Block: You ask the lock manager to give u a lock and while he manages for the lock u go on and do your work and return after some time. Cost: fixed cost 'c'

15.4 Continuous ski rental problem

A simple algorithm based on randomized algorithm strategy.

15.4.1 Spin

```
1. until time c/2
2. With probability P, block
3. With probability 1-P, keep spinning until 'c' spins.
4. Then block
```

Definition 15.2 *f(t) = expected cost of waiting t units of time.*

$$f(t) = \begin{cases} t, & t < c/2; \\ p(c + c/2) + (1-p)t, & c/2 \le t < c; \\ p(c + c/2) + (1-p) * 2c, & t \ge c. \end{cases}$$

Goal: Choose p to minimize competitive ratio (1+ α)
Setting the inequalities to equalities and solving:

$$f(t) = c/2 + pc = (1+\alpha)c/2 \tag{15.4}$$

$$f(t) = p(3/2c) + (1-p) * 2c = (1+\alpha)c \tag{15.5}$$

$$p = 2/5 \tag{15.6}$$

$$1 + \alpha = 9/5 \tag{15.7}$$

Good News: (1+ α) is less than 2.

Definition 15.3 ***Function*** $\Pi(t)$ ***= density function of time before a processor should block.***

Expected cost of waiting Q steps.

$$f(q) = \int_{t=0}^{q} dt \prod(t)(t+c) + q\int_{t=q}^{\inf}(dt\Pi(t)) \tag{15.8}$$

Solving the above equation:
Idea: choose $\Pi(t)$ **to minimum competitive ratio:**

$$f1(q) = (q+c)\Pi(q) + \int_{t=q}^{\inf}(dt\Pi(t)) - q\Pi(q) \tag{15.9}$$

$$f1(q) = c\Pi(q) + \int_{t=q}^{\inf}(dt\Pi(t)) - q\Pi(q) \tag{15.10}$$

$$f2(q) = c\Pi1(q) - c\Pi(q) = 0 \tag{15.11}$$

$$\Pi(q) = A\exp q/c \tag{15.12}$$

Calculating A

$$\int_{t=0}^{c} dt\Pi(t) = 1 \tag{15.13}$$

$$A\int_{t=0}^{c} \exp(t/c)dt \Rightarrow AC(e-1) = 1 \tag{15.14}$$

$$\Rightarrow A = 1/(C(e-1)) \tag{15.15}$$

Calculating 1+α

$$f(C)/C = A/C \times \int_{t=0}^{c} dt(t+C)\exp(t/c) \tag{15.16}$$

$$= A/C \times C^2 \tag{15.17}$$

$$= e/(e-1) \simeq 1.59 \tag{15.18}$$

Note: If you make this discrete space solution, then also u will get the same value

6.895 Theory of Parallel Systems	Lecture 16
Interconnection Networks	
Lecturer: Charles Leiserson	*Scribe: Kenneth Barr and Ben Adida*

Lecture Summary

1. *Basics*
   Important definitions are presented so that we can discuss some simple topologies

2. *Hypercubes*
   Two definitions and a routing scheme

3. *Butterfly (FFT) Networks*
   A cube-connected cycles network is presented. It is equivalent to a butterfly network, which is easier to visualize. Properties of the butterfly network are discussed

16.1 Basic Interconnection Networks

16.1.1 "Ideal" Parallel Computer and Definitions

Figure 16.1 (left panel) depicts an ideal parallel computer. It's ideal in the sense that every processor, P, can communicate directly with every memory module, M. If the processors and memory modules are combined as in a distributed shared memory computer, the graph looks like the right panel of Figure 1.

A ***diameter*** of a network is the longest shortest-path between two nodes. The diameter of the ideal parallel computer is $\Theta(1)$, but building such a computer would prove difficult for several reasons. First, there are $\Theta(n^2)$ wires in the system since each node must communicate to $\Theta(n)$ other nodes and a given node has degree $\Theta(n)$. Not only is the total number of wires large, but the number of wires at a particular node (its degree) is large as well. Furthermore, the fact that the degree depends on N makes it difficult to build uniform, modular nodes.

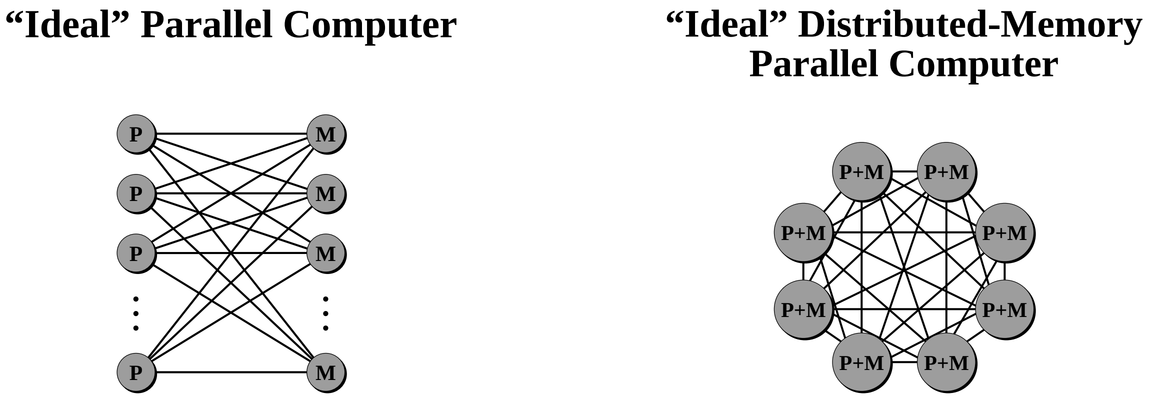


Figure 16.1: Ideal Parallel Computers

Network	Diameter	Comment
Linear array	$\Theta(n)$	Simplicity is nice, but only feasible for small networks
2D-Mesh	$\Theta(\sqrt{n})$	$\Theta(\sqrt{n})$ is an improvement over the linear arary, but can still be large
Tree	$\Theta(\lg n)$	

Table 16.1: Diameters of common networks

Figure 16.2 shows several low/constant degree networks and their diameters are listed in Table 16.1.

Theorem 16.1 *If a network has $O(1)$ degree, it has $\Omega(\lg N)$ diameter.*

Proof Begin by listing the number of nodes at a given distance from a particular node (in terms of degree, d):

Dist	0	1	2	3	k
Number of nodes	1	d	$d(d-1)$	$d(d-1)^2$	$\Theta(d^k)$

If there are N nodes, $\Theta(d^k) \geq N$ which implies $k = \Omega(\lg_d N)$. □

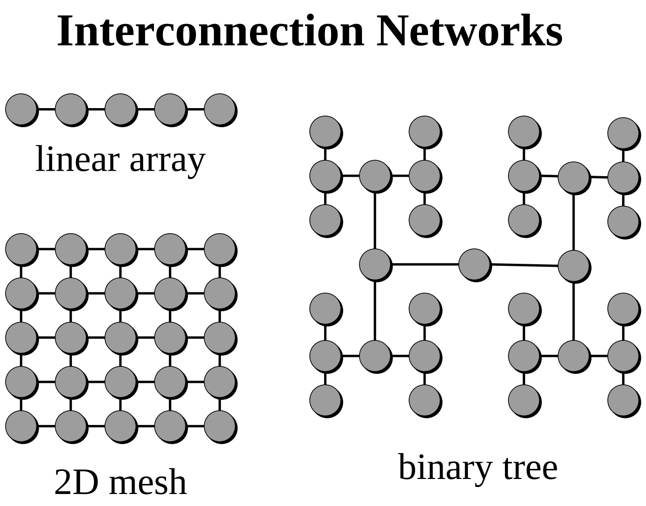


Figure 16.2: Common Networks

Of the networks we've looked at so far, the tree and ideal network have the lowest diameters, but are they good routing networks? Unfortunately, these metrics of *wires*, *degree*, and *diameter* do not take into account network congestion or hotspots. It's easy to see how congestion might be a problem in the tree: if all nodes on the left want to communicate with nodes on the right, all messages must go through the root where they are serialized. Another problem (which will be addressed in a different lecture) occurs when many processors want to access the same node. If nodes can handle ≈ 1 message at a time, a "hotspot" is created.

To start thinking about congestion, we must first define ***minimum bisection width***. Minimum bisection width (or simply bisection width) refers to the minimum number of edges that must be removed to partition a network into halves (or within one, for networks with an odd number of nodes). Bisection width for various networks are shown in Table 16.2.

Knowing the bisection width (BW), we can prove a result on the expected routing time.

Network	BW	comment
Tree	1	this is by severing a link at the root. Of course you could chop off all leaves, but that would be *maximum* number of edges.
Linear array	1	
2D-Mesh	$\sqrt{n}$	
3D-Mesh	$n^{2/3}$	edge of cube is $N^{1/3}$ nodes long. A slice through middle of cube would sever $N^{1/3^2}$ connections.

Table 16.2: Bisection Widths

Theorem 16.2 *Consider an N node network with diameter d. If N messages are sent to random destinations from N processors, $E[routing\ time] = \Omega(N/BW + d)$.*

Proof We expect $\Theta(N)$ messages to cross BW wires. If each wire ships ≤ 1 message in unit time, $time \geq \Theta(N)/BW$. We also note that $E[Time] = \Omega(diameter)$ (proved in the next lecture). Note that $(x+y)/2 \leq max(x,y) \leq x+y$. The lower bound on expected time is $max(N/BW, d)$ which we can express $\Omega(N/BW + d)$. □

16.2 Hypercubes

The next network we will examine is the hypercube. Hypercubes have a large bisection width and short diameter. It is simple to figure out routing on a hypercube. Unfortunately, they do not have constant degree as we will see in the following construction.

16.2.1 Hypercubes explained

A hypercube's recursive construction is shown in Slide 5, but it may be more convenient to define the hypercube based on the binary representation of its nodes. Each of the n nodes is represented by $\lg n$ bits: $\langle b_{d-1}, b_{d-2}, ..., b_0\rangle$. To form the network, node $\langle b_{d-1}, b_{d-2}, ..., b_0\rangle$ is connected to the nodes:
$\left\langle \overline{b_{d-1}}, b_{d-2}, ..., b_0 \right\rangle$
$\left\langle b_{d-1}, \overline{b_{d-2}}, ..., b_0 \right\rangle$
$\left\langle b_{d-1}, b_{d-2}, ..., \overline{b_0} \right\rangle$.

That is, two nodes are connected if their Hamming Distance (number of bit positions in which they differ) is 1.

16.2.2 Routing on a Hypercubes

Using binary representation, routing on a hypercube is achieved by comparing the starting and destination node addresses, sending the message along the dimension of each differing bit. Specifically, an `XOR` of the starting and destination addresses yields routing instructions, where a 1 at a particular bit position indicates the need to route along the dimension associated with that position.

Start	10111010
End	01101110
XOR	11010100

We note the following properties for a hypercube:

Diameter	$\lg N$
Time	$\Omega(\lg N)$
Degree	$\lg N$
BW	$N/2$
# of wires	$\Theta(N \lg N)$

16.2.3 Cube-Connected Cycles

Hypercube networks still have variable node degrees, which can be a problem for network modularity. ***Cube-Connected Cycles*** (CCC) offer a structure with many properties similar to those of hypercubes, but with a fixed degree.

The construction of a CCC network is shown in Slide 7, which clearly shows that each node's degree is in fact 3 (even when we move to higher dimensions). Routing is still performed by `XOR`ing the binary representations of starting and destination nodes, routing within the cycle for a 0 bit, and routing out on the cube for a 1 bit.

We note the following properties for a CCC network:

Cycles	$N = n \lg n$ nodes
Degree	$\Theta(1)$
Diameter	$\Theta(\lg N)$
BW	$\Theta(n) = \Theta(N/\lg N)$
# of wires	$\Theta(N \lg N)$

16.3 Butterfly (FFT) Networks

16.3.1 Construction

A ***Butterfly Network***, also called an FFT network, is constructed as shown in Slide 8, with n input nodes and n output nodes. The network can be **direct** or **indirect**, depending on whether the last layer of nodes is in fact same as the first layer. (MORE DETAILS HERE on this direct vs. indirect –ben).

Given the construction of the network, given n input nodes, the number of total nodes is $N = n \lg n$. We note the following properties:

Diameter	$\Theta(\lg N)$
Degree	$\Theta(1)$
BW	$\Theta(n) = \Theta(\frac{N}{\lg N})$

A butterfly network is, in fact, exactly the same as a CCC network, a fact overlooked by the original authors.

(SOMETHING HERE ABOUT DECOMPOSING A BUTTERFLY network –Ben)

16.3.2 Routing and Congestion

Routing on a butterfly network is much like on a hypercube, using the specific order of the dimension. Specifically, if we `XOR` the starting and destination binary addresses, a bit value of 0 indicates a straight routing, while a bit value of 1 indicates a cross routing.

(We note, in the construction of a butterfly network, the presence of binary trees rooted at each input and at each output.)

Packet routing from a specific source $x_{d-1}x_{d-2}...x_0$ to a specific destination $y_{d-1}y_{d-2}...y_0$ progresses from major to minor bit, as follows:

- $x_{d-1}x_{d-2}...x_0$
- $y_{d-1}x_{d-2}...x_0$
- $y_{d-1}y_{d-2}...x_0$

- ...
- $y_{d-1}y_{d-2}\cdots y_0$

Routing takes $d = \lg n$ steps, but there remains a fairly congested worst-case scenario. The worst-case routing scenario occurs for packets originating at sources $x_1x_2x_3x_4 0000$ going to destinations $0000x_1x_2x_3x_4$. There are $\sqrt(n)$ such packets, and all must go through the 00000000 halfway point.

6.895 Theory of Parallel Systems	Lecture 17

Butterfly Network Analysis and The Beneš Network

Lecturer: Charles Leiserson — *Scribe: Jason Hickey and Tyeler Quentmeyer*

Lecture Summary

1. *Network with N Nodes*
   This section proves part of the lower bound on expected routing time for an arbitrary N node network.
2. *The Beneš Network*
   This section motivates, introduces, and analyzes the Beneš connection network.
3. *Routing on Butterfly Networks*
   This section establishes that most routing problems take only $O(\lg N)$ time on butterfly networks.

17.1 Network with N Nodes

In this section, we prove a lemma that completes the proof of the ***Network with N Nodes*** theorem given in lecture 16.

Theorem 17.1 *For an arbitrary network with N nodes, the expected routing time to simultaneously send a single message from each processor to a random processor is*

$$E[\mathit{Routing\ Time}] = \Omega(N/BW + \mathit{diameter})$$

where BW is the minimum bisection width.

Proof

Lemma 17.2 *The expected routing time for the N random messages problem laid out in theorem 1 is* $\Omega(N/BW)$.

Proof Refer to the notes for lecture 16 for proof. □

Lemma 17.3 *Each node v in a N node network chooses a destination* $dest(v)$ *uniformly at random. Then,* $E[\mathit{Routing\ Time}] = \Omega(d)$*, where d is the diameter.*

Proof Find node x such that $Pr\{\delta(x; dest(x)) \geq d/2\} \geq 1/2$ where $\delta(x, y)$ is the shortest distance from node x to node y. Then,

$$\begin{aligned}
E[\text{time to route } x \text{ to its destination } dest(x)] &= \sum_k k Pr\{\delta(x; dest(x)) = k\} \\
&\geq \sum_{k \geq d/2} k Pr\{\delta(x; dest(x)) = k\} \\
&\geq d/2 \sum_{k \geq d/2} Pr\{\delta(x; dest(x)) = k\} \\
&= d/2 Pr\{\delta(x, dest(x)) \geq d/2\} \\
&= d/2 \\
&= \Omega(d)
\end{aligned}$$

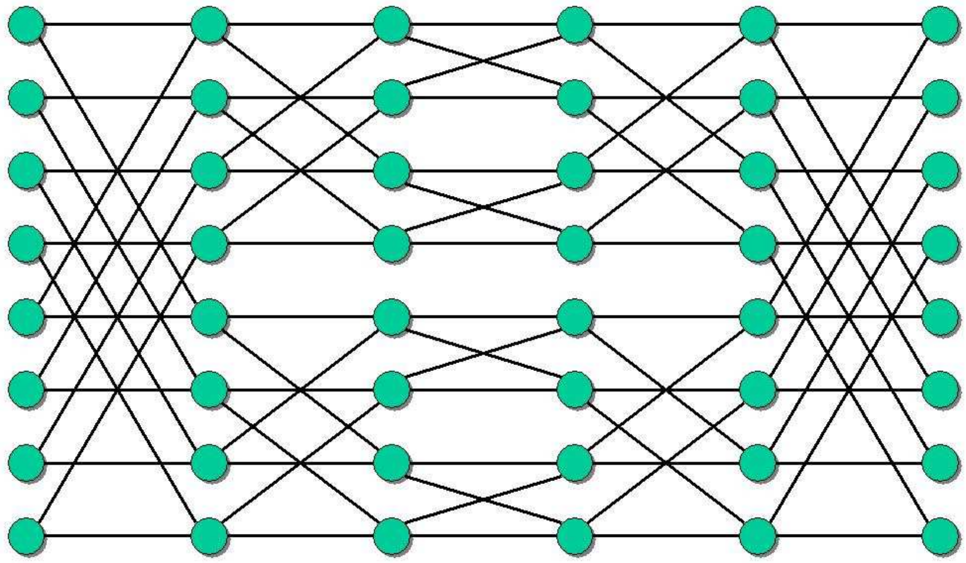


Figure 17.1: Example of a Beneš Network. Nodes aligned vertically are in the same level and nodes aligned horizontally are in the same row.

To find such an x, let $\delta(x, x') = d$, and define $S = \{v : \delta(x, v) < d/2\}$ and $S' = \{v : \delta(x', v) < d/2\}$. We know $S \cap S' = \varnothing$ since for all $z \in S \cap S'$, $\delta(x, x') \le \delta(x, z) + \delta(z, x') < d/2 + d/2 < d$ gives a contradiction, implying no such z can exist. Therefore, at least one of $\overline{S}$ and $\overline{S'}$ has $\ge N/2$ nodes. Without loss of generality, assume that $|\overline{S}| \ge N/2$. Thus, $dest(x) \in \overline{S}$ with probability $\ge 1/2$, i.e., $Pr\{\delta(x, dest(x)) \ge d/2\} \ge 1/2$. □

From lemma 2 and lemma 3, we know that $E[\text{Routing Time}] = \Omega(N/BW)$ and $E[\text{Routing Time}] = \Omega(diameter)$. Therefore, we can conclude that $E[\text{Routing Time}] = \Omega(N/BW + diameter)$. □

17.2 The Beneš Network

17.2.1 Worst Case Problems for the Butterfly Network

We saw last lecture that the worst case congestion for the butterfly network is $\sqrt{n}$ for an n input network. This was motivated by considering the problem where inputs $x_1x_2x_3x_4 0000 \longrightarrow$ outputs $0000 x_1x_2x_3x_4$. There are many such bad inputs that cause $\sqrt{n}$ congestion for the butterfly network. For example, consider the problem of computing the matrix transpose where each node represents an element of a matrix and we wish to compute $(i, j) \longrightarrow (j, i)$.

Matrix transpose and other operations that elicit the worst case congestion in a butterfly network are common in programs written for supercomputers. It is therefore a real problem to build supercomputers with butterfly networks because typical applications suffer serious performance penalties.

17.3 The Beneš Network

This section introduces the Beneš network, a network that allows provably congestion-free communication from inputs to a permutation of the outputs. A Beneš network is constructed by overlapping the low-order

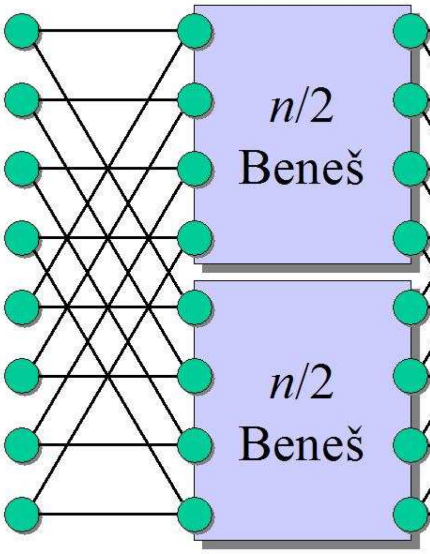


Figure 17.2: An example Beneš Network with the middle $d-2$ levels replaced with two half-sized Beneš networks.

cycles of two butterfly networks. An example is shown in figure 17.1.

Theorem 17.4 *Any n-permutation can be routed (off-line) on an n-input Beneš network with node-disjoint paths.*

Proof The proof uses the fact that Beneš networks are decomposable into smaller Beneš networks. Let $n = 2^d$. The proof is based on induction on d. For $d = 1$, the network is trivial and the theorem is obvious on examination.

As the inductive step, assume that the theorem holds for a $n = 2^d$ input Beneš network. Observe that the middle $d-2$ levels of the Beneš network form two Beneš networks, one on top and one on bottom, each with $n/2 = 2^{d-1}$ inputs, as shown in figure 17.2. By assumption, any input to one of the sub-Beneš networks can be routed to any output of the same subnetwork without congestion. Therefore, all we need to do is determine whether each input is to be routed through the top or the bottom subnetwork and show that the subnetwork outputs can be correctly routed to the final outputs without congestion.

The only constraint we must observe to avoid congestion is that connected pairs at level 1 and at level d do not overlap. That is, each butterfly at the first and last level must either route packets straight or crossed: butterflies cannot send both input packets to a single output node. Therefore, by observation, the two inputs comprising a butterfly at the first level must be routed into different subnetworks and the two inputs comprising a butterfly at the last level must come from different subnetworks. It is this final constraint that is important.

Define a butterfly pair as

$$bfly(i) = \begin{cases} i + n/2 & (i \leq n/2), \\ i - n/2 & (i > n/2). \end{cases}$$

The butterfly pair is the other node (either input or output) that defines a butterfly in the network, $bfly(i) = j$ and $bfly(j) = i$. Define $\pi(i)$ as the permutation that maps inputs to outputs: $\pi(i) = j$ denotes that input i should be routed to output j and $\pi^{-1}(j) = i$ denotes that output j should be routed from input i.

We now show how to route inputs into the upper and lower subnetworks in a way that satisfies our constraints. We start by routing the first input through the upper subnetwork and connecting it to the correct output, $\pi(1)$. Next, we fulfill the constraint at the last level by routing the first path's output butterfly pair, $bfly(\pi(1))$, through the lower network and back to its correct input, $\pi^{-1}(bfly(\pi(1)))$. We satisfy the new constraint at the input by routing the appropriate input, $\pi^{-1}(bfly(\pi(1)))$, through the upper subnetwork and to its correct output. We continue going back and forth through the upper subnetwork when connecting an input to an output and through the lower subnetwork when connecting an output to an input until we have defined a completed loop. Since the first path went through the upper subnetwork and the final return path went through the lower subnetwork, the input constraint on the original path is satisfied. If the first cycle of paths doesn't include all nodes, we pick an unrouted node and repeat this process until all inputs are routed. Following this algorithm, half of the inputs are routed through the top subnetwork and half of the inputs are routed through the bottom subnetwork without congestion.

We have satisfied the inductive hypothesis by showing that given a $n/2 = 2^{d-1}$ input congestion free Beneš network, we can construct a $n = 2^d$ input congestion free Beneš network. By induction, we have proved the theorem. □

As a side note, since we always have a choice of how to route the first input, we can remove a switch from each inductive level of the butterfly network without affecting its performance characteristics. Consequently, for a $n = 2^d$ input network, we can remove d switches from the network without loss of generality.

Corollary 17.5 *An n-input Beneš network can simulate any n-node, degree-d network in $O(d \lg n)$ time.*

Proof Let G be an n-node network with maximum degree d. Let H be an n-input Beneš network. We prove the corollary by showing how to simulate G on H. Identify the ith node of G with the ith input of H to simulate node computation. The tricky part is showing how to simulate communication on edges in G using H's network. We do this by identifying $d+1$ subsets of the edges in G such that for each subset, we can route packets from input i to output j for each edge (i,j) in the subset of G with at most $O(\lg n)$ congestion. Each subset is routed in a separate round. $d+1$ subsets with $O(\lg n)$ congestion each gives us a total of $O(d \lg n)$ time to simulate G.

We now construct $d+1$ disjoint and spanning subsets of edges in G. Construct a bipartite graph $\Gamma_G = (U, V, E)$ where $U = \{u_1, u_2, ...u_n\}$, $V = \{v_1, v_2, ...v_n\}$, and $E = \{(u_i, v_j) | (i,j) \text{ is an edge of } G\}$. Communication from i to j in G is represented by edge (u_i, v_j) in Γ_G. Since the maximum degree in G is d, the maximum degree in Γ_G is also d, making Γ_G a d-regular bipartite graph. Therefore, we can construct an edge-coloring of Γ_G using $d+1$ colors (see Leighton's *Introduction to Parallel Algorithms and Architectures* for proof). Each set of edges with a given color forms one of the $d+1$ subsets.

Let S be the kth of the $d+1$ subsets. For each edge (i,j) in S, we simulate communication on (i,j) by sending a packet from input i to output j on H during round k. Since the edges incident to i and the edges incident to j are all colored differently, we know that no two packets originate from or are delivered to the same node during the same round. Therefore, S can be routed with our $O(\lg n)$ bound. One round for each of the $d+1$ subsets with $O(\lg n)$ routing time each completes the proof. □

17.4 Routing on Butterfly Networks

Even though the butterfly network has a routing time of $\Omega(\sqrt{n})$ on certain permutations (including many "interesting" permutations), most routing problems take only $O(\lg n)$ time. The following theorem covers arbitrary N-packet routing problems on butterfly networks, not just those that route inputs to outputs.

Theorem 17.6 *Consider the N^N N-packet routing problems on an N-node (n-input, where $n = \Theta(N/\lg N)$) butterfly network. At least $N^N(1 - 1/N^{\Omega(1)})$ of these problems can be routed in $O(\lg N)$ time.*

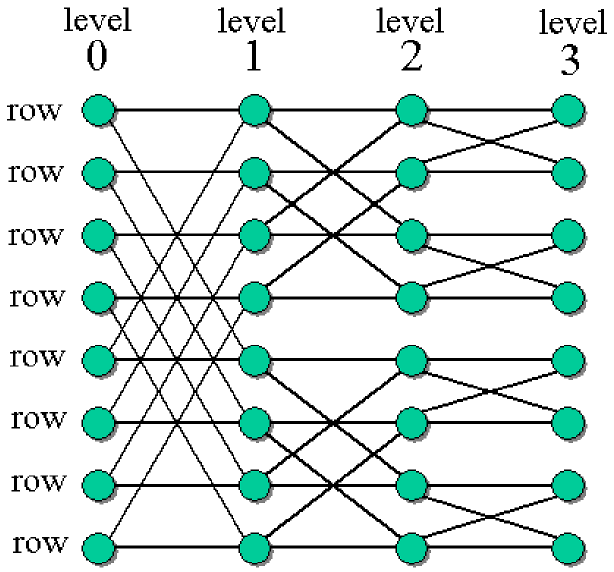


Figure 17.3: Depiction of a butterfly network. Nodes aligned vertically are in the same level and nodes aligned horizontally are in the same row.

Proof We will see the proof of a weaker result than is needed to establish the theorem. The proof establishes a congestion bound that leads to an $O(\lg^2 n)$ time result.

WLOG, we route packets in three phases (figure 17.3 shows the rows and levels in a butterfly network):

1. Route packets straight across rows in the network to the corresponding outputs (the rightmost node in each row).
2. Use greedy input to output routing, to get each packet to the correct row in the network.
3. Route packets straight across to the final destination.

We analyze the congestion at the output nodes at each row of the network in Phase 1. All packets from a given row end up at the output node of that row at the end of the phase. Each row contains $\lg n$ nodes, and $\lg n = O(\lg N)$, so the congestion in Phase 1 is $O(\lg n)$.

For Phase 2, we consider a node x at the kth level of the network (where the inputs are the 0th level and the outputs are the $(\lg n)$th level). By symmetry, the x is equivalent to any other node at the kth level. The number of packets that can reach x during Phase 2 is $2^k \lg n$ since there is a path to each node at the kth level of a butterfly network from 2^k input nodes and each input node has at most $\lg n$ packets at the beginning of Phase 2. Thus,

$$\text{worst case congestion at a node } x \text{ at the } k\text{th level} = 2^k \lg n.$$

The probability that a given packet passes through node x is at most 2^{-k}. This result follows from the binary tree property of butterfly networks and the fact that the destination of each packet is chosen independently and uniformly at random from all the nodes in the network. (See figure 17.4 for a depiction of the binaray tree property). Thus,

$$\Pr\left[\text{a given packet passes through node } x\right] \leq 2^{-k}.$$

Consider any set of r specific packets. Since the destinations of the packets are independent of one another, the probability that all the packets pass through x is given by

$$\Pr\left[\text{all } r \text{ packets pass through } x\right] \leq (2^{-k})^r = 2^{-kr}.$$

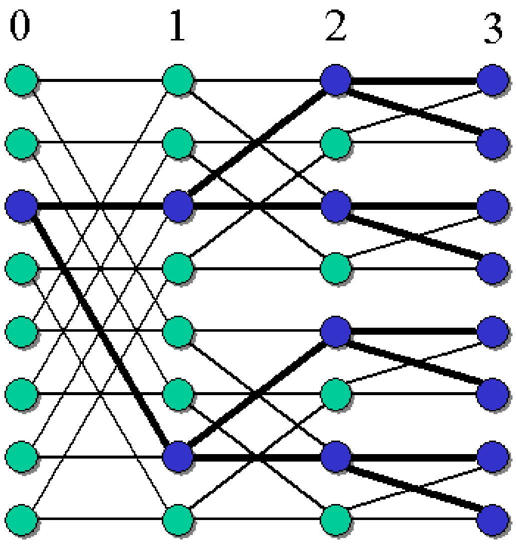


Figure 17.4: Binary tree property of a butterfly network. Each choice on a path from an input node to an output node is the head of a binary tree.

The probability that at least r packets pass through x is bounded by the number of ways to choose r packets from the total number of packets multiplied by the probability that all r packets pass through x:

$$\Pr[\geq r \text{ packets pass through } x] \leq \binom{2^k \lg n}{r} 2^{-kr}.$$

Note: This argument overcounts. If $r+\Delta$ packets pass through x, this event is counted $\binom{r+\Delta}{r}$ times within the $\binom{2^k \lg n}{r}$ ways.

Using the formula $\binom{a}{b} \leq \left(\frac{ea}{b}\right)^b$, we have

$$\begin{aligned}\Pr[\geq r \text{ packets pass through } x] &\leq \left(\frac{e2^k \lg n}{r}\right)^r 2^{-kr} \\ &= \left(\frac{e \lg n}{r}\right)^r.\end{aligned}$$

If we choose $r = 2e \lg N$, then we obtain

$$\begin{aligned}\Pr[\geq r \text{ packets pass through } x] &\leq \left(\frac{1}{2}\right)^{2e \lg N} \\ &\leq N^{-2e} \\ &\leq \frac{1}{N^{5.4}}.\end{aligned}$$

By Boole's inequality, the probability that *any* node has more than $r = 2e \lg N$ packets is $\leq N$ times the probability that a given node has more than r packets:

$$\begin{aligned}\Pr[\geq r \text{ packets pass through some node}] &\leq N\left(\frac{1}{N^{5.4}}\right) \\ &= N^{-4.4}.\end{aligned}$$

Therefore, $\geq N^N(1 - 1/N^{4.4})$ routing problems see $\leq 2e \lg N$ congestion. Since there are $O(\lg N)$ levels and each node at a level has at most $O(\lg N)$ congestion, the total congestion for Phase 2 is $O(\lg^2 N)$. At the end of Phase 2, there are $O(\lg N)$ packets at each output node with high probability. Therefore, at each node in Phase 3 there is $O(\lg N)$ congestion with high probability. Thus, the overall time bound is $O(\lg^2 N)$ for at least $N^N(1 - 1/N^{\Omega(1)})$ routing problems. □

Corollary 17.7

$$\begin{aligned} E[\textit{routing time}] &= O(\lg N)(1 - 1/N^{4.4}) + O(N)(1/N^{4.4}) \\ &= O(\lg N). \square \end{aligned}$$

6.895 Theory of Parallel Systems	Lecture 18
Hypercube Networks-III	
Lecturer: Charles Leiserson	*Scribe: Sriram Saroop and Wang Junqing*

Lecture Summary

1. *Review of the previous lecture*
   This section highlights the worst case routing on the butterfly network.

2. *Valiant's Algorithm*
   This section describes the idea behind Valiant's algorithm for solving a hard problem on the butterfly network.

3. *Ranade's Algorithm*
   This section briefly introduces Ranade's algorithm.

4. *Shuffle-exchange graph*
   This section explains the properties of the Shuffle-exchange graph.

5. *DeBruijn Network*
   This section introduces the DeBruijn Network, and illustrates the DeBruijn sequence with the aid of a card trick.

6. *Bisection Width for Shuffle-Exchange graph*
   This section provides the proof for deriving the bisection width for the shuffle-exchange graph.

18.1 Review of the previous lecture

- Routing on an N-node (i.e. $N/\lg N$ input)butterfly takes $O(\lg N)$ time with high probability, assuming all routing problems are equally likely.

- Some problems are still hard. For example, the Matrix transpose problem is hard. If we consider the routing from $X_1X_2X_3X_40000 \to 0000X_1X_2X_3X_4$, essentially $\sqrt{n}$ inputs have to route through a common intermediate node, 00000000. This routing step is a subproblem of the matrix transpose problem wherein $(i,j) \to (j,i)$. The indices i and j are represented by bits and are concatenated to form the input. Therefore, matrix transpose incurs $\sqrt{n}$ congestion. Hence, we find that the butterfly network has a bad worst case O($\sqrt{n}$), even though it has a good average case.

18.2 Valiant's Algorithm

Valiant's Algorithm is a *permutation routing* algorithm. The idea is to convert one hard problem into two easy ones. The routing is performed in two phases:

- **Phase 1:** Firstly, we route from the sources to random intermediate nodes.

- **Phase 2:** Then, we route from the intermediate nodes to the destination nodes.

Both phases 1 and 2 are *random routing problems* which are good cases for the butterfly network. Hence, there is no hard routing problem. We can however be *unlucky*, but it is quite *unlikely* that we are unlucky. The *unluckiness* does not correspond to the choice of input because we cannot put a bound on the choice of input. However, we can bound probabilistically how likely we could have a bad routing problem. An adversary cannot elicit the worst case behavior. In fact, the adversary cannot provide a bad routing problem because every problem is equally bad. In conclusion, by randomizing the chosen intermediate nodes which are destinations in Phase 1 and sources in Phase 2, we have essentially distributed the congestion, thereby making the sub-problems easy ones.

18.3 Ranade's Algorithm

It is a routing algorithm which runs in $O(\lg N)$ time with high probability. It has some interesting properties like bounded queues, which are queues of size $O(1)$. The proof is through means of an elegant delay sequence argument.

18.4 Shuffle-exchange graph

Description

Definition 18.1 (Shuffle Exchange Graph) *Let $d \epsilon N$. The d-dimensional shuffle-exchange SE(d) is defined as an undirected graph with node set $V = [2]^d$ and an edge set $E = E_1 \cup E_2$ with*

$E_1 = \{((a_{d-1}, ..., a_0), (a_{d-1}, ..., \overline{a_0}))|(a_{d-1}, ..., a_0)\epsilon[2]^d, \overline{a_0} = 1 - a_0\}$

and

$E_2 = \{((a_{d-1}, ..., a_0), (a_0, a_{d-1}, ..., a_1))|(a_{d-1}, ..., a_0)\epsilon[2]^d\}$.

It is based on an N node hypercube, where $N = 2^d$. Figure 18.1 shows an 8-node shuffle-exchange graph.

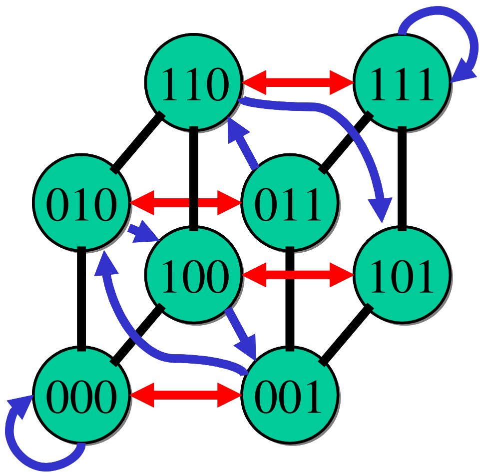


Figure 18.1: Shuffle-exchange graph

Each node is labelled with a unique $\lg N$ bit string. A node labelled a $= a_{\lg N-1}, ..., a_0$ is linked to a node labelled b $= b_{\lg N-1}, ..., b_0$ by a *shuffle edge* if rotating one position to the left or right yields b, i.e., if either b $= a_0, a_{\lg N-1}, a_{\lg N-2}, ..., a_1$ or b $= a_{\lg N-2}, a_{\lg N-3}, ..., a_0, a_{\lg N-1}$.Two nodes labelled a and b are linked by an *exchange edge* if a and b differ in only the least significant (rightmost) bit, i.e., b $= a_{\lg N-1}, a_{\lg N-2}, ..., \overline{a_0}$. In Figure 18.1, shuffle edges are the blue edges, and exchange edges are the horizontal red edges. Figure 18.2

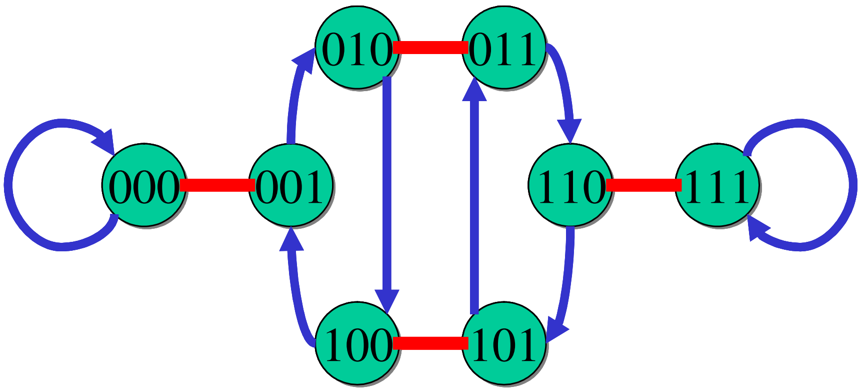


Figure 18.2: Shuffle-exchange graph: Another view

shows another view of the same 8 node shuffle-exchange graph. To summarize, if a node is a d-bit binary number, *exchange edges* are between $< b_{d-1}, b_{d-2}, ..., b_1, 0 > \leftrightarrow < b_{d-1}, b_{d-2}, ..., b_1, 1 >$ and *shuffle edges* are from $< b_{d-1}, b_{d-2}, ..., b_1, b_0 > \rightarrow < b_{d-2}, ..., b_0, b_{d-1} >$, which corresponds to a left-cyclic shifting of the bits.

Properties

- Degree = 3 (Out-degree = 2 , In-degree = 2)
- Number of edges $\leq$ 2N(in the case of an undirected graph). According to the Hand-shaking lemma, sum of the in-degrees = number of edges. Since there are N nodes, each of which has an in-degree of at most 2, the total number of edges would be at most 2N.
- Diameter $\leq$ 2 lg N. Since there are $\lg N$ bits in a node, in order to route from one node to another, even in the worst case would correspond to traversing at most $\lg N$ shuffle edges and $\lg N$ exchange edges. Hence the diameter $\leq$ 2 lg N.
- Bisection Width = $O(N/\lg N)$. Deriving the Bisection Width is not trivial as is illustrated by the proof in Section 18.6.

It can be seen that every property of the butterfly is also a property of a shuffle-exchange graph. However, the properties were easier to prove in the case of the butterfly because it was more intuitive.

A Perfect (Out) Shuffle

The connection pattern in Figure 18.3 is called a *perfect shuffle* because it is like shuffling two halves of a card deck, so that the cards from the two decks interleave perfectly. We notice that the top and bottom cards which correspond to bit patterns 000 and 111 do not change positions, whereas the other positions go a position which is double in value, that also corresponds to a left shift of one bit [12]. For example, 001 goes to 010 that is double of it. Hence, we can conclude that **Shuffle(X) $\equiv$ 2X mod (N-1)**. This shuffle, wherein the top and bottom cards retain their positions is called a *Perfect Out-Shuffle.*

A Perfect (In) Shuffle

In a perfect in-shuffle as shown in Figure 18.4 the top and the bottom cards are *in* the deck rather than *out*, which means that they do not retain their positions.

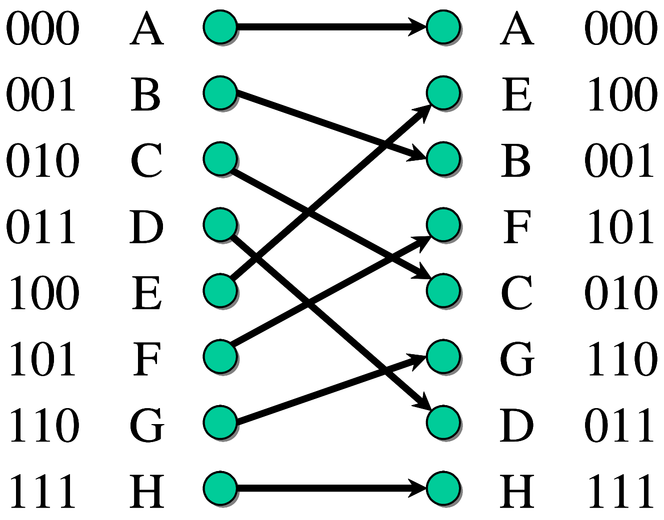


Figure 18.3: A perfect shuffle

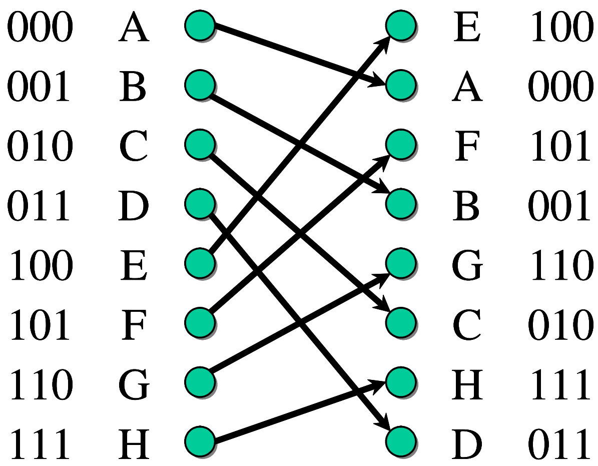


Figure 18.4: A perfect in-shuffle

A perfect in-shuffle can be viewed as a perfect out-shuffle followed by an exchange, which is illustrated in Figure 18.5.

A Card trick

Consider a deck of 8 cards which correspond to the 8 nodes shown earlier in the shuffle-exchange graph. Suppose a card is chosen at random and placed in any location in the deck, given a required destination location, we can perform a certain number of perfect shuffles(in or out) in order that we route the card to the required destination. Let us consider an example. Suppose that the randomly chosen card is placed at the last position (corresponding to 8-1 = 7 = 111), and that the required destination position is 3 (corre-

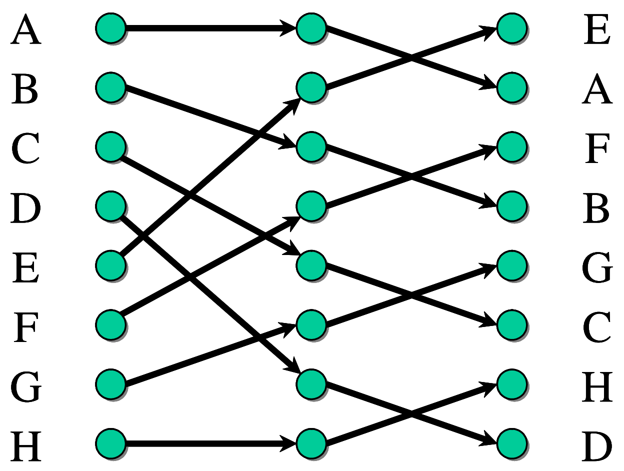


Figure 18.5: In-shuffle = Out-shuffle + Exchange

sponding to 3-1 = 2= 010). The claim is that 3 perfect shuffles is sufficient to place the last card in the third position. It can be seen from Figure 18.6 that starting with an in-shuffle, followed by an out-shuffle, and lastly performing an in-shuffle again would route the card at 111 to the position 010. Let us proceed to trace through the sequence of operations. The first in-shuffle actually corresponds to a shuffle followed by an exchange. We take the redundant shuffle edge from 111 we go back to 111 and then take the exchange edge to 110. The second operation which is an out-shuffle just corresponds a shuffle from 110 which takes us to 101. The last operation which is an in-shuffle takes us from 101 to 011 following the shuffle edge and then from 011 to the required position 010 following the exchange edge. This sequence is illustrated by the green edges in Figure 18.6. In order to obtain the required sequence of operations, we first take the XOR of the source and the destination. The bits which are 1 in the resultant bit string correspond to the bits that have to be inverted by taking exchange edges at appropriate rotations. Hence, we perform an in-shuffle for each 1 in the bit string, and perform an out-shuffle for each 0 in the resultant bit-string.

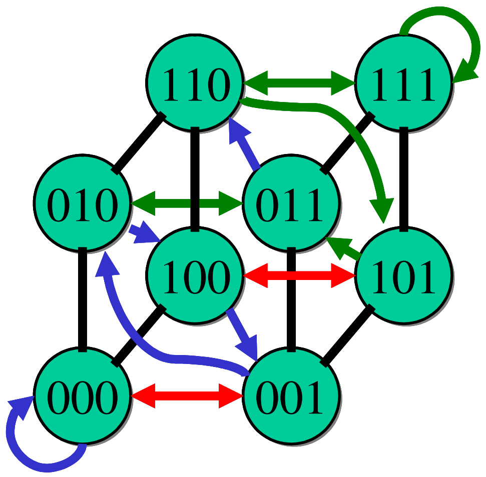


Figure 18.6: Required path in the shuffle-exchange graph (Shown in green)

18.5 DeBruijn Network

In this section, we look at the ***DeBruijn graph***. This network is based on the Hypercube. In fact, we may convert the previous Shuffle Exchange graph to the DeBruijn graph by contracting the exchange edges. Consider a 3-bit node as an instance. We use the first 2 bits as the prefix, and then combine all nodes with the same prefix. For instance, node $\langle 1,1,0\rangle$ and $\langle 1,1,1\rangle$ will merge into one node $\langle 1,1\rangle$, and so on. Then we collapse the Shuffle-exchange network along the exchange edges to obtain the DeBruijn Network (as shown in Figure 18.7).

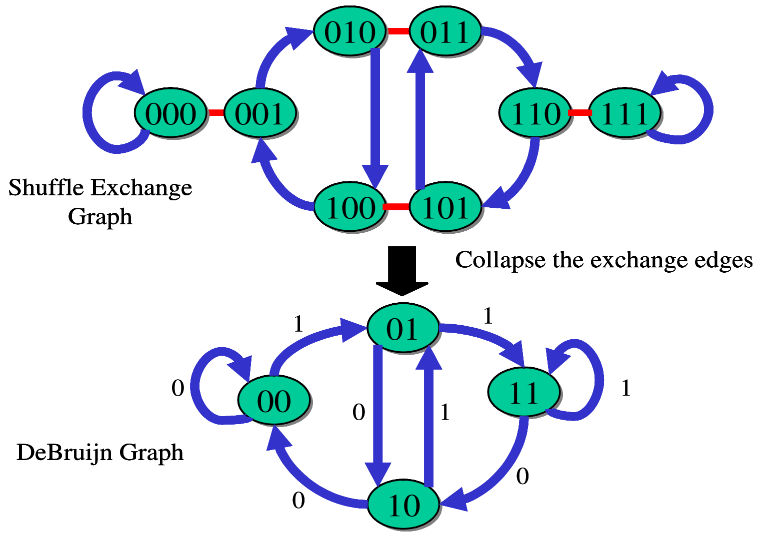


Figure 18.7: Converting the Shuffle-exchange graph to the DeBruijn graph

In one DeBruijn network, we have:
Node = d-bit binary number
Out-shuffle edge $\langle b_{d-1}, b_{d-2}, ...b_0\rangle \to \langle b_{d-2}, ...b_0, b_{d-1}\rangle$
In-shuffle edge $\langle b_{d-1}, b_{d-2}, ...b_0\rangle \to \langle b_{d-2}, ...b_0, \overline{b_{d-1}}\rangle$
The shuffle edges can also be represented as 0-edges or 1 edges based on whether the higher order bit, b_{d-1} is a 1 or a 0, like:
0-edge is $\langle b_{d-2}, ...b_0, 0\rangle$
1-edge is $\langle b_{d-2}, ...b_0, 1\rangle$

Degree = 4 (out-degree = in-degree = 2)
Number of edges $\leq$ 4N
Diameter = lgN
Bisection width = O(N / lgN).
We had shown that an (n-1)-dimension DeBrujin network is obtained by collapsing the exchange edges of an n-dimensional S-E network, which has a bisection width of O(N/lgN) (which will be proved in Section 18.6). Since the DeBrujin network is a reduced form of the S-E network, the DeBrujin network has the same upper bound of the bisection width = O(N/lgN).

Another Card Trick
This card trick is related to DeBruijn networks. We consider a deck of 8 cards. Suppose that the deck is cut

3 times, and then 3 cards are chosen consecutively. Now, if the number of black cards chosen is known, then we can find which 3 cards were chosen ! Knowing the number of black cards provides us 3 bits of information, and it is only 3 bits of information that we need in order to get the position of a given card. According to the listing of the 3 card sequences in Figure 18.8, we find that any sequence of 3 cards corresponds to a different pattern of cards. Any kind of pattern only occurs once there. So 3 bits information is enough for us to predicate what cards are in hand. For example, if the pattern is 100, then it shows that the cards in hand are 6 clubs, 3 hearts and A hearts. By performing an Eulerian tour of the DeBruijn network in Figure 18.7, one of the DeBruijn sequences we get is shown in Figure 18.8.

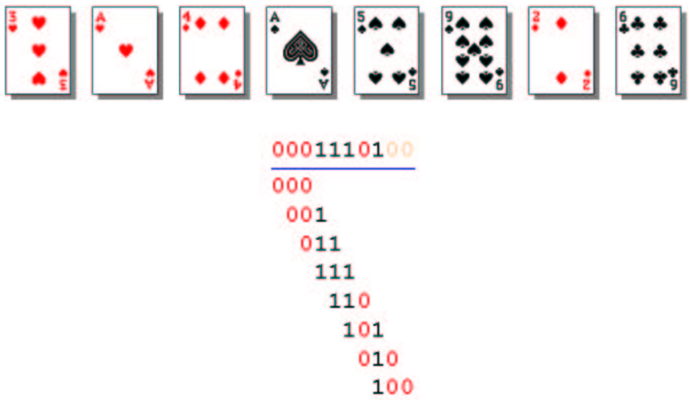


Figure 18.8: Another Card trick (DeBruijn Sequence)

18.6 Deriving the Bisection Width for the Shuffle-Exchange Graph

In this section, we will prove that the bisection width of S-E network is O(N/lgN). The proof is based on the complex-plane diagram of the Shuffle Exchange network (as shown in Figure 18.9). The dotted edges are the exchange edges and the solid edges stand for the shuffle edges.
$N = 2^d$
Let w_d=$e^{2\pi i/d}$ be d-th complex root of unity. Hence, w_d^d=1

Definition 18.2 $\sigma(b_{d-1}, b_{d-2}, ...b_0) = \sum_{k=0}^{d-1} b_k w_d^k$

A few lemmas which describe the properties of the S-E network are described below. The lemmas form the crux of the proof.

Lemma 18.3 *Exchange edges have unit length.*

Proof The least significant bits of nodes connected by exchange edges are complements of one another.

$$\sigma(b_{d-1}, b_{d-2}, ...1) = \sigma(b_{d-1}, b_{d-2}, ...0) + 1w_d^0 \tag{18.1}$$

□

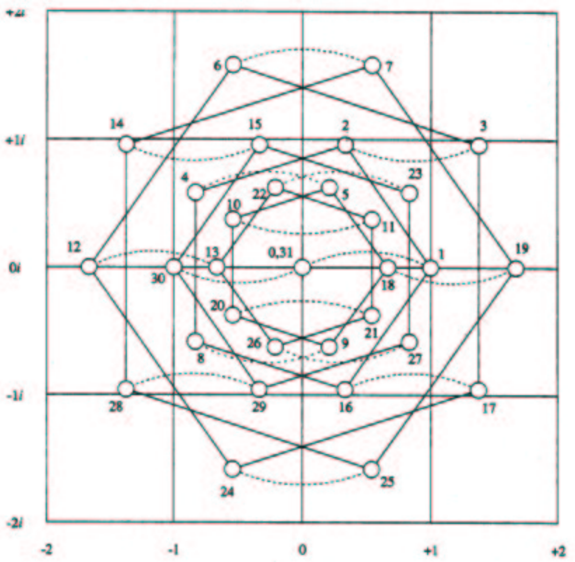


Figure 18.9: Complex plane diagram of the Shuffle-exchange network

Lemma 18.4 *Shuffle edges form* necklaces*, all of whose nodes are equidistant from origin:*

Proof

$$w_d\sigma(b_{d-1}, b_{d-2}, ...b_0) = w_d \sum_{k=0}^{d-1} b_k w_d^k = \sum_{k=0}^{d-1} b_k w_d^{k+1} = \sigma(b_{d-2}, ...b_0, b_{d-1}), since\ w_d^d = 1 \tag{18.2}$$

Thus, $\|\sigma(b_{d-1}, b_{d-2}, ...b_0)\| = \|\sigma(b_{d-2}, ...b_0, b_{d-1})\|$.

Most nodes are in *full* necklaces , which have d nodes, but some are in *degenerate* necklaces, which have lesser nodes.

For example:
For N=16 : {0000}, {1111}, {0101,1010} are *degenerate* necklaces.

□

Lemma 18.5 *All degenerate necklaces are mapped to origin.*

Proof Suppose: $w_d^k\sigma(x) = \sigma(x)$ with k<d
Each multiplication by w corresponds to a shuffling which in turn rotates the necklace (1/d)th each time.
Since we have $w_d^k \neq 1$ (k<d), we infer that $\sigma(x)$=0.

□

Lemma 18.6 *O(N/lgN) nodes are mapped to origin.*

Proof

$$\sigma(b_{d-1}, b_{d-2}...b_0) = 0 \Rightarrow \sigma(b_{d-1}, b_{d-2}...\overline{b_0}) = \pm 1 \tag{18.3}$$

The necklace of $\langle b_{d-1}, b_{d-2}, ..., \overline{b_0} \rangle$ has lgN nodes and there are at most two of them lying at $\pm$ 1. Therefore, there are at most N/lgN full necklaces. This means that there are at most 2N/lgN nodes are at ± 1. Hence $\leq$ 2N/lgN nodes are at 0.

□

Lemma 18.7 *O(N/lgN) nodes are mapped to real line*

Proof At most 2 nodes from each full necklace and O(N/lgN) nodes of degenerate necklaces at origin map to the real line.

□

Lemma 18.8 *O(N/lgN) edges touch real line.*

Proof There are at most 4 shuffle edges per full necklace that corresponds to O(N/lgN) edges. In addition there are O(N/lgN) shuffle edges at origin and O(N/lgN) horizontal exchange edges. Hence, O(N/lgN) edges touch real line.

□

Lemma 18.9 *Real line forms bisection.*

Proof

$$\sigma(b_{d-1}, b_{d-2}...b_0) + \sigma(\overline{b_{d-1}}, \overline{b_{d-2}}...\overline{b_0}) = \sum_{k=0}^{d-1} (b_k + \overline{b_k}) w_d^k = \sum_{k=0}^{d-1} w_d^k = \frac{w_d^d - 1}{w_d - 1} = 0, \ \textit{since}\ w_d^d = 1 \tag{18.4}$$

Thus, the number of nodes above the real line equals the number of nodes below the real line. And the nodes on the real line are divided in half arbitrarily. □

6.895 Theory of Parallel Systems	Lecture 19

Squish Routing and Disk-drive Hardware

Lecturer: Bradley C. Kuszmaul *Scribe: Advait D. Karande and Zhang JiaHui*

Lecture Summary

1. *Squish routing*
   We begin by defining what a squish message pattern is.

2. *Squish theorem*
   In this section, we prove that squish routing can always be performed without collisions on a butterfly network.

3. *An application of squish routing*
   This section explains how one-to-one routing can be performed as a sequence of squish routings.

4. *Implementing squish routing*
   This section explains how squish routing can be implemented using parallel prefix (scan) operation.

5. *Analysis of Scan and Routing*
   The cost of scanning in squish-up/squish-down and the bound on the routing time with N processors are analyzed.

6. *Disk Drives-Hardware*
   This section introduces some basic concepts of a disk drive, its operations (read and write), seek time and spin speed.

19.1 Squish Routing

A **squish** message pattern is defined as follows. Consider a set of k processors out of n available processors ($k < n$). These processors are labeled S_i ($0 \le i \le k-1$) such that

$$0 \le P_{S_0} < P_{S_1} < ... < P_{S_{k-1}} < n$$

where P_{S_i} indicates the ID of the processor labeled S_i. P_{S_i} sends a message to processor with ID i. Figure 19.1 shows a squish pattern, wherein processors 1, 2 and 4 send messages to processors 0, 1 and 2 respectively.

Now consider a butterfly network with $n = 8$ processors. Figure 19.2 shows a squish involving 5 sources on such a network.

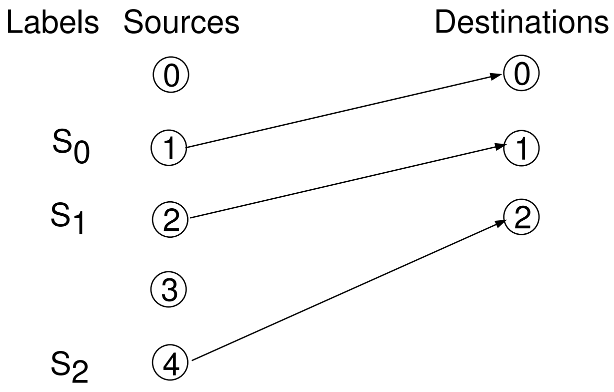


Figure 19.1: An example of a squish pattern.

It is apparent that the messages in the squish have been routed without any collisions. In fact, we will now prove that squish routing can always be performed on a butterfly network without any collisions.

19.2 Squish theorem

Theorem 19.1 *Squish routing can always be performed without any edge-conflicts on a butterfly network.*

We begin by defining the terms necessary for the proof.

Definition 19.2 *A **semi-contraction** is a one-to-one routing from a set of k source processors $\{s_0, s_1, ..., s_{k-1}\}$ to k destination processors $\{d_0, d_1, ..., d_{k-1}\}$ such that processor s_i sends a message to d_i and*

$$\forall i, j : |s_i - s_j| \geq |d_i - d_j| \tag{19.1}$$

Claim 19.3 *Every squish is a semi-contraction.*

Lemma 19.4 *Semi-contractions route without an edge-conflict on a butterfly network.*

Proof By Contradiction. We show that if a conflict exists, then the defining equation of a semi-contraction (Equation (19.1)) no longer holds.

Consider a butterfly network with n processors running from minor to major cycles, similar to the one shown in Figure 19.2. Assume that each processor ID is given by $r = lg(n)$ bits. Since the network runs from minor to major cycles, a route from source to destination is found by sequentially setting or resetting bits in source processor ID from LSB to MSB. Now we consider a semi-contraction for this network.

Suppose there exists an edge-conflict. This implies that there exist i and j such that the route from s_i to d_i shares at least an edge with the route from s_j to d_j. Let l be the dimension of the shared edge as shown in Figure 19.3.

Let the bit-pattern of the processor IDs of the two source nodes be as follows:

$$\begin{aligned} s_i &: b_{r-1}b_{r-2}...b_l b_{l-1}...b_0 \\ s_j &: a_{r-1}a_{r-2}...a_l a_{l-1}...a_0 \end{aligned}$$

Since the routes from both s_i and s_j reach point Y as shown in Figure 19.3, the most significant bits up to dimension l of s_i and s_j must be the same. That is:

$$b_{r-1}b_{r-2}...b_l = a_{r-1}a_{r-2}...a_l$$

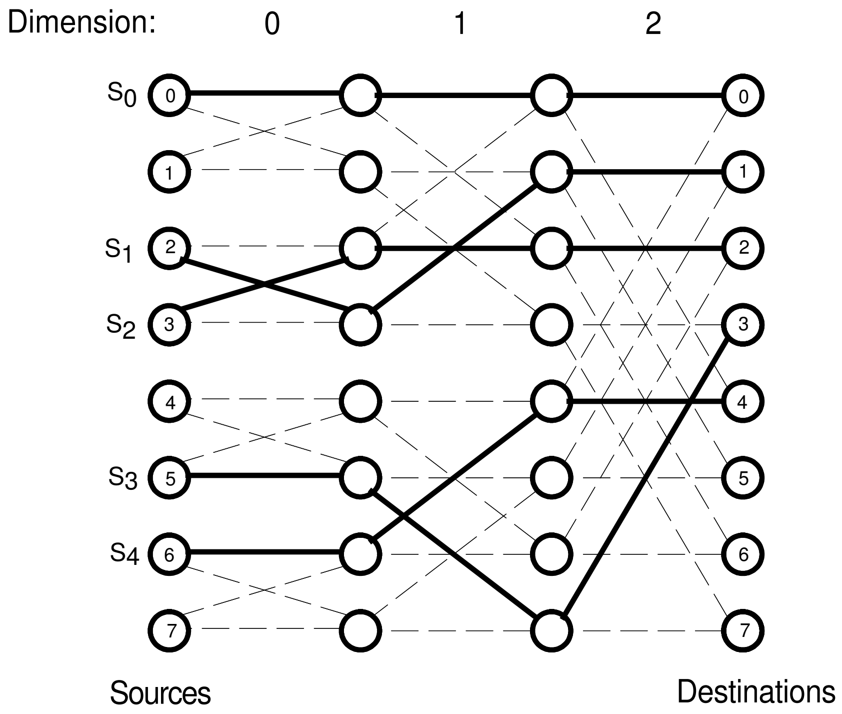


Figure 19.2: A squish pattern on a butterfly network with $n = 8$ inputs. Nodes 0, 2, 3, 5 and 6 send messages to nodes 0, 1, 2, 3 and 4 respectively. Thin, dashed lines indicate the underlying network connections, while bold lines indicate the message routes.

Therefore, the distance between the IDs of the two source processors is:

$$\begin{aligned} |s_i - s_j| &= \left| \sum_{m=0}^{r-1} (b_m - a_m) \cdot 2^m \right| \\ &= \left| \sum_{m=0}^{l-1} (b_m - a_m) \cdot 2^m \right| \\ &\leq \left| \sum_{m=0}^{l-1} 2^m \right| \\ &= 2^l - 1 \end{aligned} \tag{19.2}$$

Similarly, let the bit-patterns of the destination processors be represented as follows.

$$\begin{aligned} d_i &: f_{r-1} f_{r-2} \ldots f_l f_{l-1} \ldots f_0 \\ d_j &: e_{r-1} e_{r-2} \ldots e_l e_{l-1} \ldots e_0 \end{aligned}$$

Since the routes to both d_i and d_j have reached the same point Y (Figure 19.3), the $l+1$ least significant bits in the IDs of these processors must be the same. That is:

$$f_l f_{l-1} \ldots f_0 = e_l e_{l-1} \ldots e_0$$

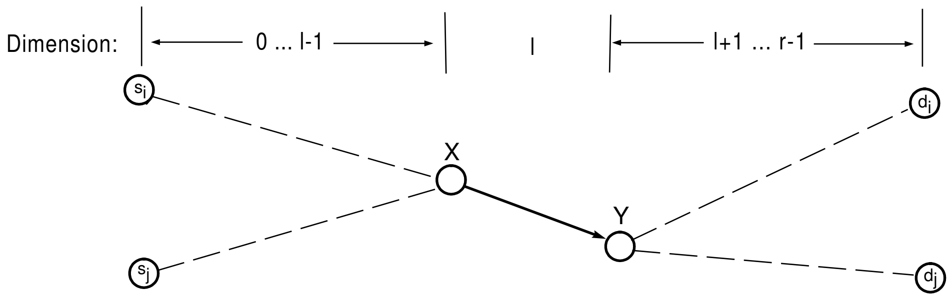


Figure 19.3: An edge-conflict on an $n = 2^r$ input butterfly network - Routes $s_i \to d_i$ and $s_j \to d_j$ share an edge at dimension l.

Therefore, the distance between the IDs of the two destination processors is:

$$\begin{aligned} |d_i - d_j| &= \left|\sum_{m=0}^{r-1} (f_m - e_m) \cdot 2^m\right| \\ &= \left|\sum_{m=l+1}^{r-1} (f_m - e_m) \cdot 2^m\right| \end{aligned}$$

Replacing m with $i = m - (l+1)$:

$$|d_i - d_j| = \left|\sum_{i=0}^{r-l-2} (f_{i+l+1} - e_{i+l+1}) \cdot 2^i \cdot 2^{l+1}\right| \quad (19.3)$$

$$= 2^{l+1} \cdot \left|\sum_{i=0}^{r-l-2} (f_{i+l+1} - e_{i+l+1}) \cdot 2^i\right| \quad (19.4)$$

Given that the two destination IDs must differ by at least 1 bit, the absolute value of summation in the above equation must be at least 1. Hence,

$$|d_i - d_j| \geq 2^{l+1} \quad (19.5)$$

From Equations (19.2) and (19.5),

$$|s_i - s_j| < |d_i - d_j| \quad (19.6)$$

which contradicts the defining equation of a semi-contraction. □

The proof of Theorem 19.1 follows from Lemma 19.4 and Claim 3.

19.3 An application of squish routing

Squish routing can be used to perform one-to-one routing on a butterfly or a hypercube. The key idea is to decompose one-to-one routing into a sequence of squish-routings.

Consider an r-dimensional hypercube (or a butterfly with 2^r inputs). The ID of each node is given by bit-sequence $b_{r-1}b_{r-2}...b_0$. An algorithm to implement one-to-one routing on such a network is as follows.

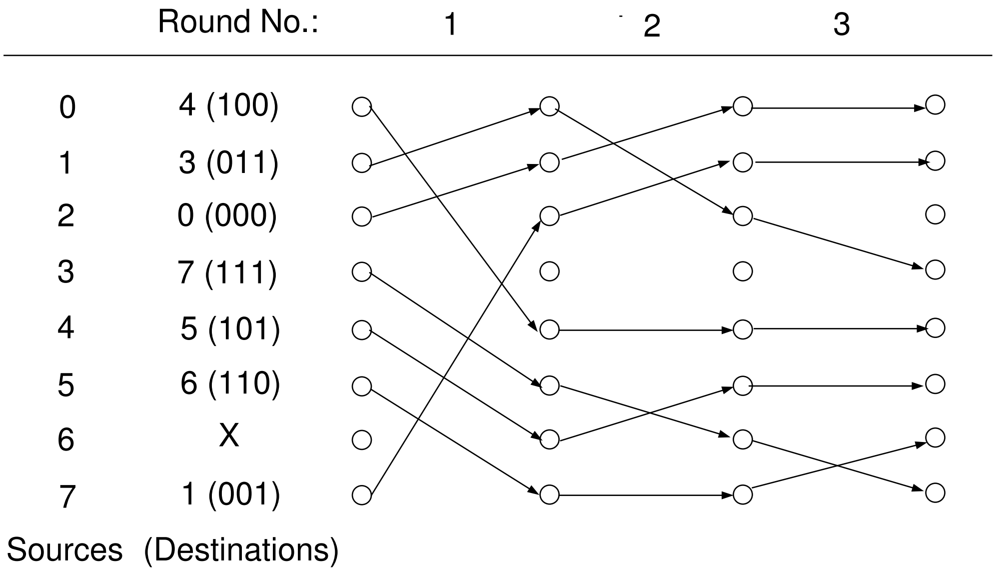


Figure 19.4: One-to-one routing as a sequence of squish routings. Processors 0, 1, 2, 3, 4, 5 and 7 send messages to processors 4, 3, 0, 7, 5, 6 and 1 respectively. The routing completes after 3 rounds.

For each dimension i = r-1 **downto** 0
A. Squish down all messages whose i^{th} bit in the destination ID is 0.
B. Squish up all messages whose i^{th} bit in the destination ID is 1.

The one-to-one routing, thus, completes after r rounds of squish routings. Figure 19.4 provides an example.

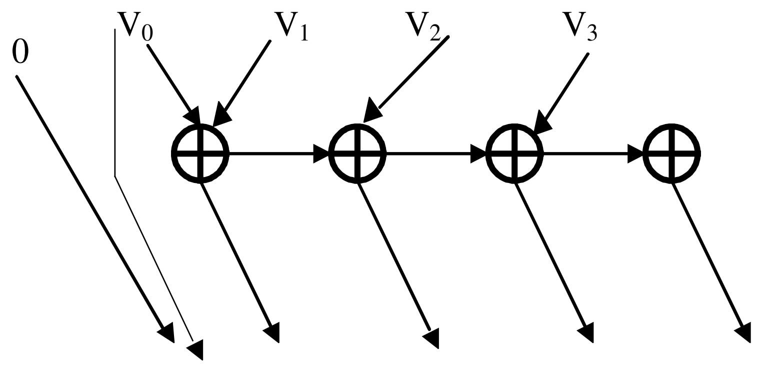


Figure 19.5: An example of DAG that computes the vector ω.

19.4 Implementing squish routing

In order to implement squish routing on a hypercube, for each source processor, a destination processor must be determined in each round. This involves computing an enumeration of the set $S = \{S_i\}$. We define a bit vector v such that:

$$v_i = \begin{cases} 1 & \text{if } i \in S \\ 0 & \text{otherwise} \end{cases} \tag{19.7}$$

The destination processor for each source processor i $(i \in S)$ in a round is the number of 1's in v before i. It is found by computing the prefix sum, ω, of vector v, where:

$$\omega_i = \sum_{j=0}^{i-1} v_i$$

We can compute ω shown in Figure 19.5. Notice that there is no parallelism in this DAG.

The work and critical path for this computation are as follows.

$$T_1 = O(n)$$
$$T_\infty = O(n)$$

Can we do it faster? Yes, for example, by using a tree structure as shown in Figure 19.6.

The work and critical path for this tree structure are:

$$T_\infty = O(lg(n))$$
$$T_1 = O(n^2)$$

Here the work is proportional to the number of adders. If we reuse the adders/results, we can even achieve

$$T_1 = O(nlg(n))$$

The parallel prefix operation on a vector is also known as a scan operation.

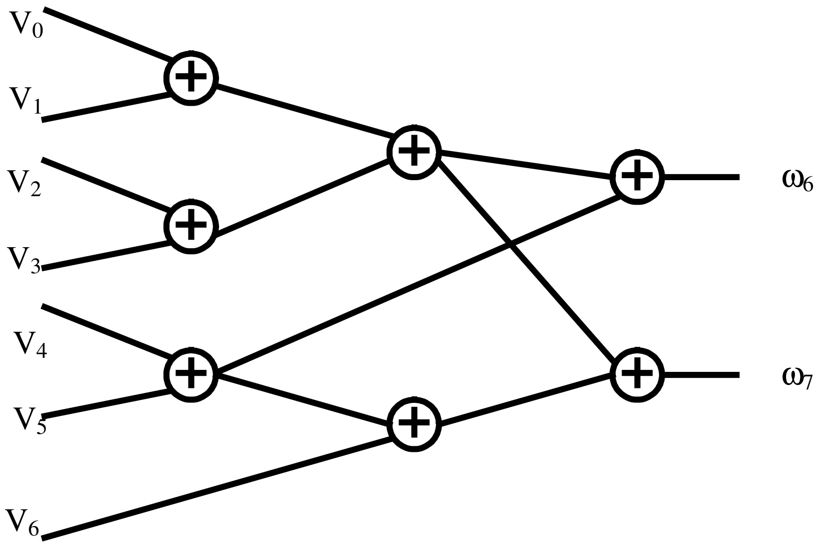


Figure 19.6: An example of a tree-structure that computes the vector ω.

19.5 Analysis of Scan and Routing

Time to perform a scan is $O(lg(n))$:

depth of graph $= lg(n)$
The wires carry O(1) bits/unit time
The cost of addition: $lg(n)$ to add 2 n-bit numbers

Now we will show that the time to perform a scan is $O(lg(n))$, if we pipeline the bits through the network. How does the pipeline work? Let's illustrate it with a diagram. In Figure 19.7,we are trying to add bits 0 1 1 0 1 1 0 1, and the expected result is 101 (5 in decimal).

In Figure 19.7, we can see that there is no pipeline delay; after $lg(n)$ clock ticks, the first bit comes out. After another $lg(n)$ clock ticks, the last bit comes out.

Now, we will analyze the time required for squish-routing on N processors, $T(N)$. Here M is defined as the message size in bits.

Squish up:
scan — $lg(n)$
squish — $M + lg(n)$

Squish down:
scan — $lg(n)$
squish — $M + lg(n)$

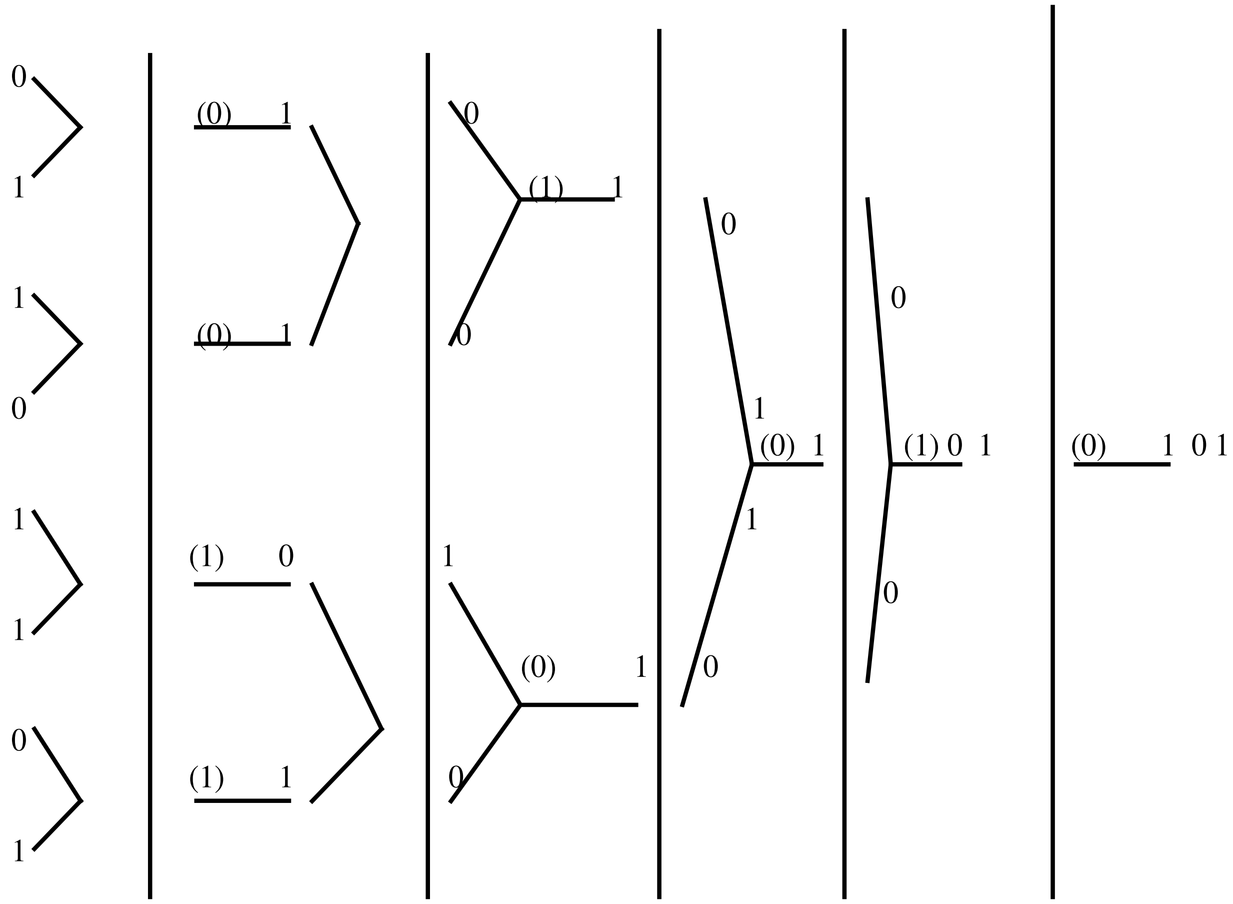


Figure 19.7: Adding the 8 bits 0 1 1 0 1 1 0 1 by pipelining the bits through the network. Bits in parentheses are carry bit.

So, we obtain the recurrence

$$T(N) = O(M) + O(lg(n)) + T(N/2)$$

Here, M is the message size, which is $\Omega(lg(N))$, since we at least need to encode the destination address in the message. Therefore we have:

$$\begin{aligned} T(N) &= O(M) + O(lg(n)) + T(N/2) \\ &= O(M) + T(N/2) \\ &= O(M) + O(M) + T(N/4) \\ &= O(M) + O(M) + O(M) + T(N/8) \\ &= \ldots \\ &= O(Mlg(N)) \end{aligned}$$

The time required for squish-routing on N processors with a message size of M bits is therefore $O(Mlg(N))$.

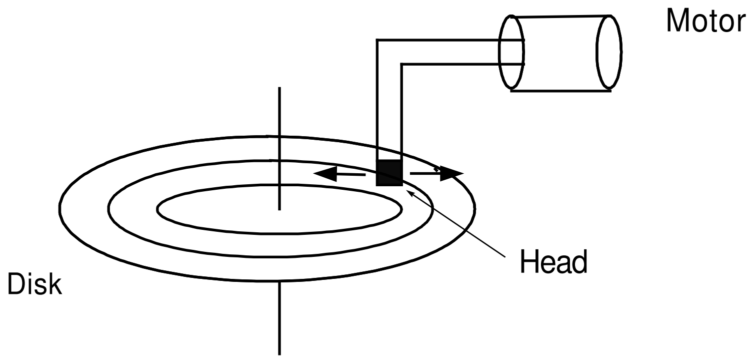


Figure 19.8: A disk drive with its head controlled by a motor.

19.6 Disk Drives - Hardware

Figure 8 shows a Disk Drive with its head controlled by a motor. How does it work?

In order to read/write a bit from/to a disk, the head is placed above the region of the disk that represents the bit-position.

- Write operation: The head consists of an electromagnet. It writes the bit-value by magnetizing the region with appropriate polarity.
- Read operation: The head also includes a coil. The value of a bit is determined by the direction of the current the bit-region induces in the coil.

In modern hard disks, the read operation is based on the Giant Magneto Resistance effect (GMR). The head includes a read sensor, whose resitance is affected by the magnetic orientation of bit regions. Such read sensor is highly sensitive and supports high-density , high-capacity disks. The GMR effect was discovered in late 1980's. It took about 10 years for it to appear in commodity disks.

As a disk spins, the motor moves the head back and forth ("seek") radially above the disk, and it can access any spot on disk. The motor is a linear induction motor. Initially, hard disks used to have stepper motors. For such hard disks, the relationship between the seek time, t, and the distance traveled, x, is $t = \Theta(x)$. With linear induction motors, head accelerates and decelerates to let the seek time be of the order of $\Theta(\sqrt{x})$.

Today, the average seek time is 4.9 ms for a read, and 5.4 ms for a write. The best case seek time is around 1 ms and the worst case seek time is around 9ms. The rotation speed ranges from 5400 rpm to 10000 rpm.

In high performance applications, which access the data very frequently, the data is placed on the outer tracks of the disk to reduce the seek time.

6.895 Theory of Parallel Systems	Lecture 20
Permuting Data on Parallel Disks	
Lecturer: Charles Leiserson	*Scribe: Sajindra Jayasena, Sharad Ganesh*

Lecture Summary

1. *Parallel Disk Systems*
   In this section, an overview of parallel Disk systems is given with a brief look at practical examples
2. *Problem of permuting Disk Blocks*
   In this section,we investigate how a disk access permutation is performed which would materialize high level of parallelism
3. *Conflict Graphs*
   This section looks at the theoretical aspects on how a non-conflicting schedule could be created for parallel disk access
4. *Hall's Theorem*
   Here we look at Hall's theorem and its application to a non blocking scheduling

20.1 Parallel Disk Systems

In this section we will briefly look at parallel disk systems.

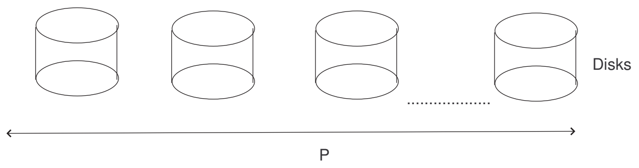


Figure 20.1: Parallel disks

One class of parallel disk systems that's used widely is RAID, which stands for **R**edundant **A**rray of **I**nexpensive **D**isks. In RAID, error correction is done across the disks and multiple disks can be accessed concurrently. RAID defines several levels of configurations that provide a combination of properties like level of performance,fault tolerance,redundancy etc that the user can select depending on his requirement.
Some example of levels are:

1. RAID 0: Striping - The controller stripes data across multiple drives in the RAID subsystem. RAID 0 has the best performance and capacity of any RAID level, but the lowest availability (no fault tolerance).
2. RAID 1: Mirroring - In RAID 1, the controller duplicates all data from one drive to another. RAID 1 provides complete data redundancy, but at the cost of doubling the required data storage capacity. RAID 1 has the highest availability but lowest capacity, since twice the number of drives are required.

3. RAID 3: Striping with Dedicated Parity - RAID level 3 provides disk striping and complete data redundancy through a dedicated parity drive. The stripe size must be 64KB if RAID 3 is used. RAID 3 handles data at the block level, not the byte level, so it is ideal for networks that often handle very large files, such as graphic images.

4. RAID 5: Striping with Rotational Parity - The controller stripes blocks of data and parity information across all drives. The parity block is distributed among the drives of the array, giving a more balanced access load across the drives. The parity information is used to recover the data if one drive fails, and explains why this method is the most popular.

Another type of configuration is JBOD which stands for **J**ust a **B**unch **O**f **D**isks.This is just a collection of disks without any high level inter disk management system.
Current disk timings : Generally following are the current timing for data access and data transfer in modern disk systems.

1. Data access times: $\simeq 10^{-2}$ seconds

2. Data Transfer Rate $\simeq 10^{8}$ words per seconds.

it is clear that disk access takes most of the time compared to disk transfer.Therefore the main strategy is to **reduce or parallelize the disk accesses as much as possible**.

One convenient engineering assumption is:
Instead of having free writing across the disk , the disk is broken down into large fixed size blocks. For example each block might consist of 1000+ words or possibly million words.This is to amortize the cost of data access over a larger number of words.Therfore it would make the cost per word relatively lesser.
Now we will look at an example and visualize how it is modelled in a disk.

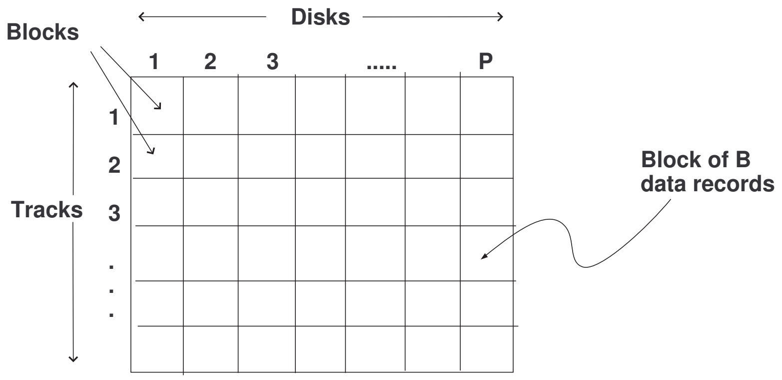


Figure 20.2: Disk block layout

In Figure 2, there are P disks and N data records.
Therefore to store N data records, the number of tracks needed are :
Number of Tracks = N / P.B
This is also equal to the number of IOs' to read all the data in N records.This is true because in parallel disk systems , at each unit time one block per each disk can be read. For example:
At Time T_1 : read track 1
At Time T_2 : read track 2
....
At Time T_n : read track n

This simplifies the model to a Block read , without the need to look into the intricacies of each block.
Typically a computer has a certain size of memory. For example it might be able to hold M data records where the general assumption is that $M \gg P.B$. (i.e in reality the size of memory is significantly large compared to the amount of data that could be read at unit time from P parallel disks)

20.2 Problem of permuting Disk Blocks

Now we will look at how an actual disk access works and how it should be scheduled in such a manner in obtaining maximum parallelism in disk access.
Here we will look at a permutation which is :

1. Offline (Fixed in advance)
2. Not In Place

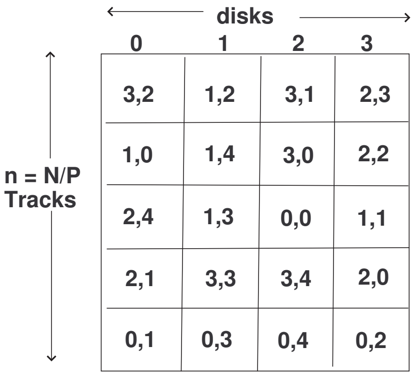


Figure 20.3: Disk access permutation

Here in figure 3, each block would hold 1 record. Therefore the number of tracks required to hold N data records is $\mathbf{N/P}$ where P is the number of Disks in the parallel system. But when there are multiple data records per block the calculation would be slightly more complex.
In the diagram ,each block specifies where it should be written to .For example the top left block specifies that the data in that block should be written to the second track of third Disk.
Objective of scheduling :
The objective of selecting a proper access sequence for the given permutation is to be able to direct the set of disks to read a set of disk blocks in a column and write those disk blocks in such a manner that the disk blocks that are read would allow the disks to write them back in parallel , thus achieving the optimal parallelism.
For example as in the diagram, if the last track is read in parallel, when it comes to writing them , they all have to be written to a single column.But the issue here is they can only be written one at a time ,since all those blocks belong to same disk.Therefore the objective is to find an access sequence where a set of blocks could be read at the one unit of time in parallel and could be written at in one unit of time in parallel. i.e in fewer IOs' as possible.
Now let's look at the following schedule.For simplicity lets assume that the read is performed in a linear fashion to a file and would be written back to a different location in the disks.

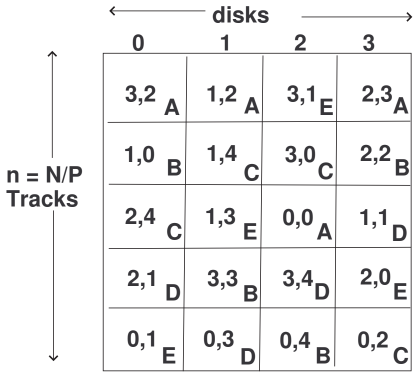


Figure 20.4: Disk block access sequence

By simulating through the schedule it is evident that reads and writes are scheduled in a fashion that all writes can also be performed in parallel. The permutation is done in exactly **N/P** parallel IOs'.
How well could we perform this in general? The following section presents the theorem for it.

Theorem 20.1 *It should be possible to permute access of N data records with P disks with* ***O(N/P)*** *parallel IOs.*
No matter what permutation is given this can be scheduled to guarantee that it could be performed in N/P parallel Disk IOs'.
note : The ***O(N/P)*** *is given since it needs to perform a read as well as a write.*

Now let's look at how we could model and analyze this using Graph theory .

20.3 Conflict graph

Idea:

Set up a source disk and a destination disk. There is a vertex for every disk. The graph models the permutations required and helps us see what the conflicts are.

The degree of this graph is $n = N/P$. The smallest degree of any vertex is also $n = N/P$.

In fact all degrees are the same n = N/P.
This is a bi-partite graph which is n-regular, since all degrees are the same(=n).

Claim:

Any d-regular bi-partite multigraph can be edge colored with d colors.

Definition 20.2 (Edge coloring) *Color the edges in such a way that all the edges coming into a given vertex have a different color.*

Claim: Each vertex will require at least d-colors and we argue that it is sufficient.

In general you can show that if d is the degree of any arbitrary graph it can be colored with at most

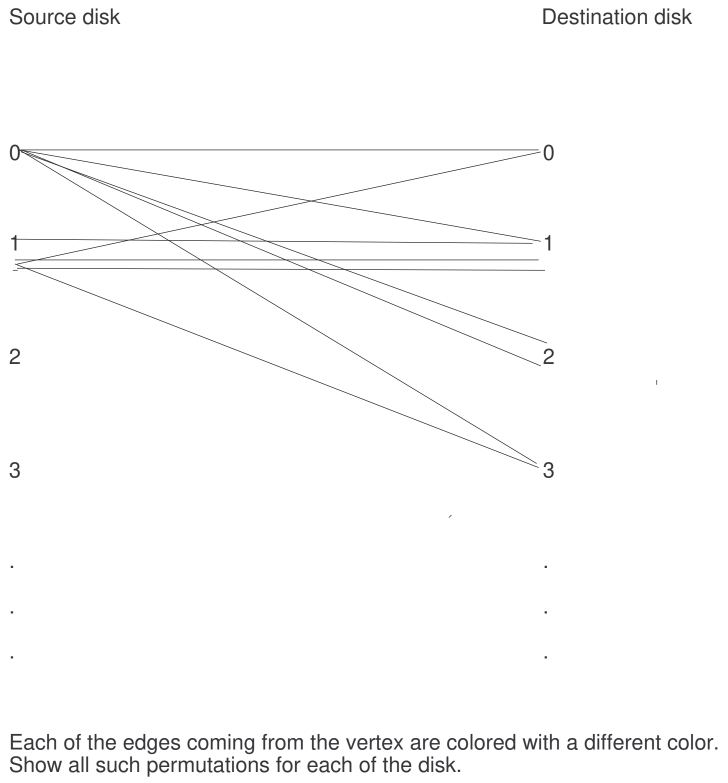


Figure 20.5: Conflict Graph

$d + 1$ colors. The fact that, whether you can color it with d-colors is NP-Complete. When the graph is colored, each coloring matches the vertices on the left with the vertices on the right.

```
Color = step at which the block is moved.
```

Method of edge coloring:

```
Find a perfect matching(everything on the left is matched to
something on the right).

Color the edges in the matching using color 1.

Remove all edges of color 1. (Now we have a $d-1$ regular
bi-partite multigraph. The degree of every vertex has dropped by
exactly 1).

Recur.
```

To know that a perfect matching exists in a d-regular bi-partite graph, we present Hall's theorem.

20.4 Hall's theorem -(General bi-partite graph)

Theorem 20.3 *Claim: If it is a d-regular bi-partite graph, a perfect matching always exists.*

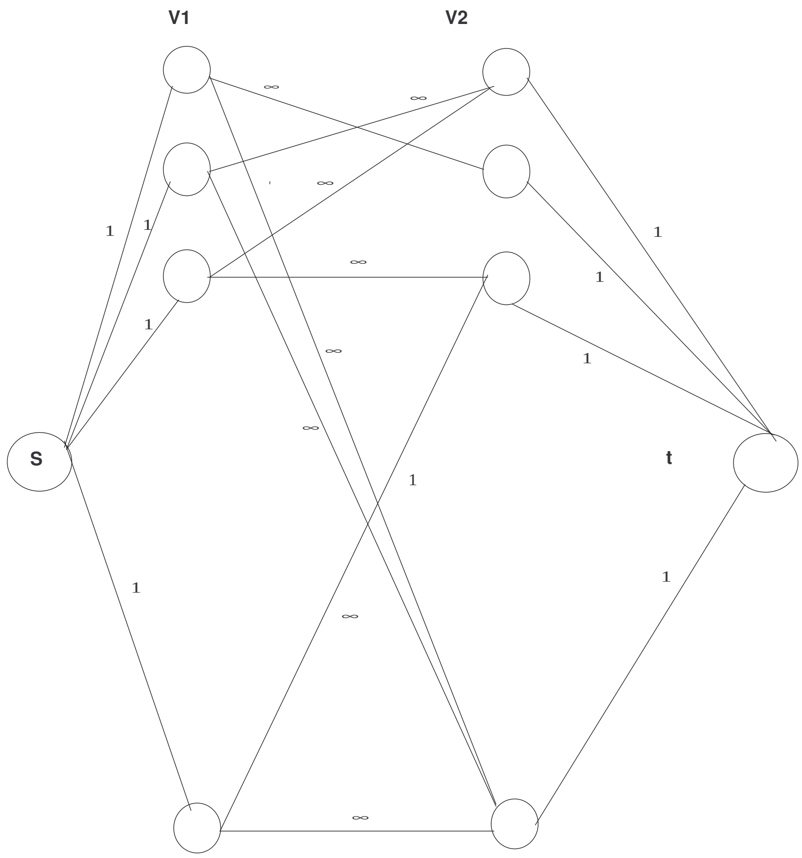


Figure 20.6: Flow graph

For any particular bi-partite graph, we have,

A set of vertices $A \subseteq V_1$, and let $N(A) \subseteq V_2$ be the set of neighbors of A. For any bi-partite graph a perfect matching exists if:

$$|N(A)| \geq |A| \forall A \subseteq V_1.$$

Proof of Hall's theorem

To find a maximum matching, one way to do that is to build a flow network shown in Figure: 20.6. If the value of the flow is n, then we have a perfect matching.
Proof
Let f be the max flow.

$$|f| = C(S,T)$$

For some cut (S, T) by the max-flow min-cut theorem. S has to be in one set and T in another, according to the max-flow min-cut theorem. If you have a maximum flow, there must be a cut, whose capacity is equal to the cut. If there is none, then you can find another path to route the flow.

Let, $A = S \cap V_1$, since the edges from vertices in V_1 to vertices in V_2 have infinite capacity.

$$N(A) \subseteq S$$

The max possible flow on this network is n. Flow across any cut in the graph is n. At the same time, the minimum cut must have the value of the flow, and cannot be infinite.
Also,

$$N(V_1 - A) \subseteq T$$

$$C(S,T) \geq |V_1 - A| + |N(A)|$$

$$|V_1 - A| + |A|$$

$$= |V_1| = n$$

$\square$

The flow value is n, everyone gets matched up, which proves Hall's theorem.

6.895 Theory of Parallel Systems	Lecture 21

Sorting and Permuting on Sequential and Parallel Disks

Lecturer: Michael Bender — *Scribe: Pham Duc Minh and Jim Sukha*

Lecture Summary

1. *DAM Model*
   This section introduces the DAM memory model and presents the asymptotic bounds for the number of memory transfers required to sort and permute N records on disk.

2. *I/O Analysis of Mergesort*
   This section presents an analysis of the number of I/O's required for serial and parallel mergesort.

3. *Sorting Lower Bound*
   This section proves the lower bound for the number of memory transfers required for sorting N records.

4. *External Permutation Lower Bound*
   This section proves the lower bound for the number of memory transfers required for permuting N records.

5. *Cache-Oblivious Sorting*
   This section briefly describes a cache-oblivious sort, funnelsort.

21.1 DAM Model

Consider the problem of sorting N records stored on a disk. To analyze the time required for this problem, we use the DAM model proposed by Aggarwal and Vitter [1]. This model assumes that M records can fit into internal memory, and B records (blocks) can be transfered between internal memory and disk in one memory transfer. The data is stored on P disks, so we can concurrently transfer P blocks in one unit of time. The setup of the DAM model is illustrated in Figure 21.1.

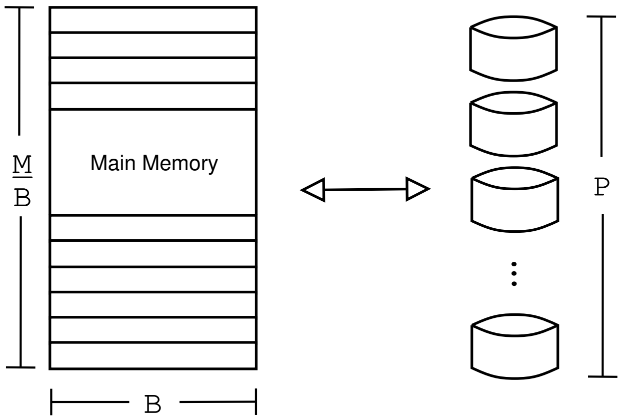


Figure 21.1: Setup of the DAM model.

Our goal is to minimize the number of memory transfers required to sort N records. Since the time required for an I/O is many orders of magnitude larger than the time required for an internal memory access, in this model, we neglect the cost of manipulating a block already inside internal memory.

The DAM model differs somewhat from the cache-oblivious memory model proposed in [7] because the DAM model requires the user to manage memory explicitly. In the cache-oblivious model, parameters such as the block size are unknown, and the system undertakes the burden of memory management. The cache-oblivious model also only considers the case $P = 1$.

We have asymptotically tight bounds on the number of memory transfers required for the problems of sorting and permuting N records:

Theorem 21.1 *The average-case and worst-case cost to sort N records is $\Theta\left(\frac{N}{PB}\frac{\log(1+N/B)}{\log(1+M/B)}\right)$ block transfers.*

Theorem 21.2 *The average-case and worst-case cost to permute N records is $\Theta\left(\min\left\{\frac{N}{P}, \frac{N}{PB}\frac{\log(1+N/B)}{\log(1+M/B)}\right\}\right)$ block transfers.*

In Section 21.3 and Section 21.4, we prove the lower bounds in Theorem 21.1 and Theorem 21.2, respectively.

Note that these bounds involve expressions with the term $\log(1 + N/B)$ or $\log(1 + M/B)$. Although the 1 is technically required, for clarity, in the sections that follow we omit the 1 in our analysis.

21.2 Mergesort Analysis

To get a sense of the complexity of the bound stated in Theorem 21.1, in Section 21.2.1, we analyze the number of memory transfers required for a normal merge sort algorithm when $P = 1$. Although this simple algorithm is not optimal, in Section 21.2.2 we show that the optimal number of memory transfers can be attained using a parallel mergesort.

21.2.1 Normal Mergesort

We first analyze the performance of a normal mergesort using the DAM model. A simple mergesort algorithm, illustrated in Figure 21.2, works by splitting the N records into half, recursively sorting each half, and merging the two sorted lists. As we see in Theorem 21.3, the cost of this algorithm is not optimal.

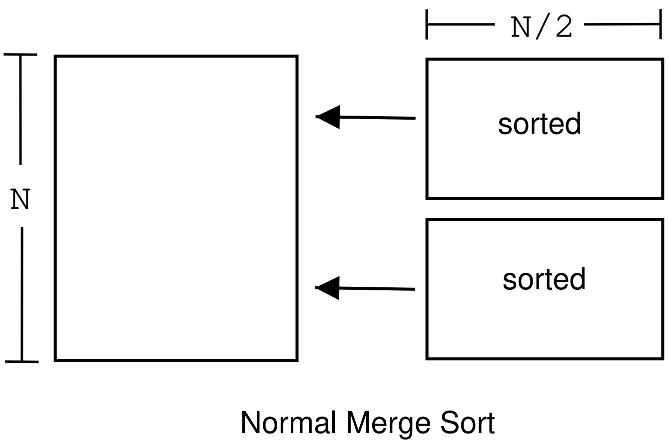


Figure 21.2: A simple mergesort algorithm.

Theorem 21.3 *Normal merge sort requires $O(\frac{N}{B}\lg\frac{N}{B})$ memory transfers.*

Proof The number of memory transfers satisfies the recurrence

$$T(N) = 2T\left(\frac{N}{2}\right) + O(\frac{N}{B}) \tag{21.1}$$

$$T(B) = 1. \tag{21.2}$$

In the recursive step we split the problem into two subproblems of half the size. The merging operation requires $O(N/B)$ memory transfers because we scan both arrays of length $N/2$ once to read in the data, and we require only $O(N/B)$ transfers for the output of the merge. In the base case a problem of size B will certainly fit into internal memory, so we sort the block using only one memory transfer to read the block in.

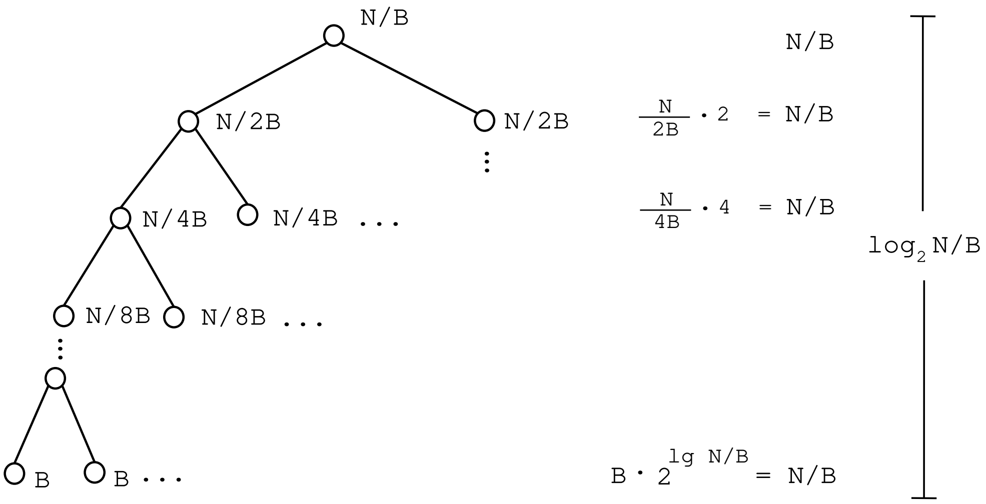


Figure 21.3: Recursion tree for normal mergesort.

This recurrence can be solved using a recursion tree, as illustrated in Figure 21.3. At each level, there are twice as many problems of half the size compared to the previous level. Therefore, the cost at each level is N/B. If we recursed all the way to a base case of 1, then we would have $\lg N$ levels. However, since we stop at problems of size B, we only have $\lg N - \lg B$, or $\lg(N/B)$ levels. This analysis gives us the stated bound of $O(N/B \lg(N/B))$. □

Note that this analysis is actually independent of the size of internal memory, M. With a more clever algorithm we might be able to improve the bound slightly, but it would still not match the optimal bound from Theorem 21.1. The normal mergesort is off by a factor of $\lg(M/B)$ from optimal.

21.2.2 Parallel Mergesort

We improve the simple mergesort by increasing the number of arrays we merge at once. Consider the algorithm illustrated in Figure 21.4 that divides the array into M/B problems and recursively sorts arrays of size $\frac{N}{M/B}$. This parallel mergesort for the case $P = 1$ requires an optimal number of memory transfers.

Theorem 21.4 *Parallel mergesort requires $\Theta\left(N/B \log_{N/B}(N/B)\right)$ memory transfers.*

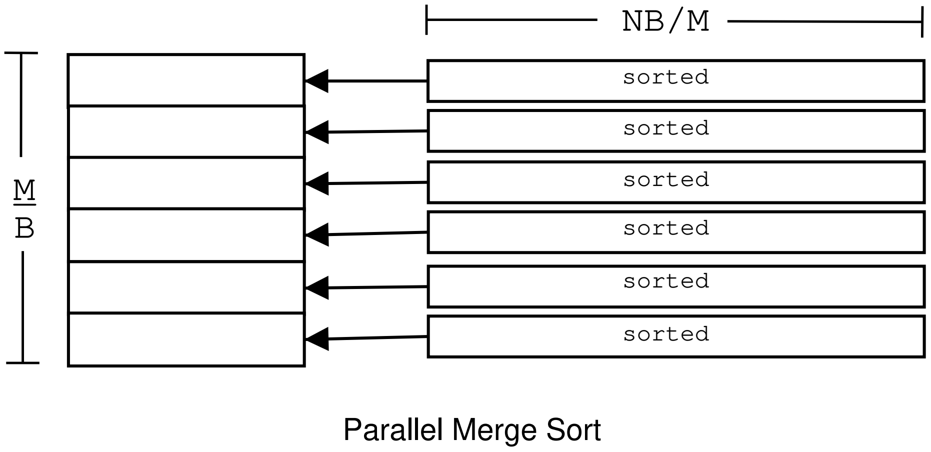


Figure 21.4: A parallel mergesort algorithm.

Proof The algorithm satisfies the recurrence

$$T(N) = \frac{M}{B} T\left(\frac{N}{M/B}\right) + O\left(\frac{N}{B}\right) \tag{21.3}$$

$$T(B) = 1. \tag{21.4}$$

The recursive step splits the problem into M/B problems, each of size $N/(M/B) = NB/M$. As with the normal mergesort, the merge step for each problem requires $O(N/B)$ memory transfers because we only need to scan through each of the sorted sub-arrays once.

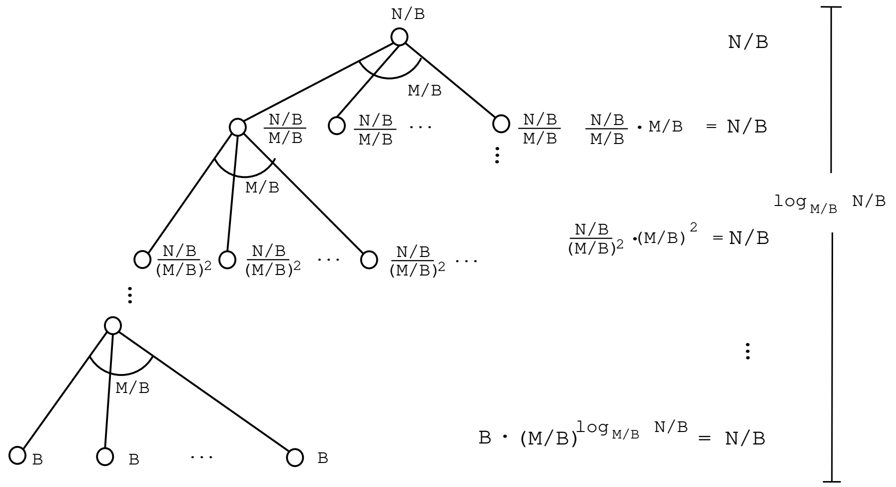


Figure 21.5: The recursion tree for parallel mergesort.

In the recursion tree shown in Figure 21.5, we see that each level still has cost $O(N/B)$. However, because we have more subproblems, the number of levels required is now only $\log_{M/B} N - \log_{M/B} B$. Adding up the costs of all the levels, we find that the number of memory transfers required is $\Theta\left(\frac{N}{B}\log_{N/B}\left(\frac{N}{B}\right)\right)$. □

The bound for this mergesort algorithm is equal to the optimal bound shown in Theorem 21.1. However, this particular parallel mergesort does not immediately work for $P > 1$ if P is not constant. In this case, it becomes important to specify how exactly the merge step is being done. Although we do not present the details here, it is in fact possible to make the parallel mergesort work for non-constant P.

21.3 Lower Bound for Sorting

In this section we prove the lower bound from Theorem 21.1 for the number of I/O's required to sort.

Theorem 21.5 *External sorting requires* $\Omega\left(\frac{N}{B}\log_{M/B} N/B\right)$ *I/O's.*

Proof This bound assumes a comparison-I/O model, where the only operation we are allowed to use in internal memory is to compare values. With this assumption, we prove the theorem using an information theoretic argument. At the beginning of the computation, we have no information about the data, so all $N!$ orderings of the data are consistent with what we know. After each memory transfer, we learn through comparisons, so we reduce the number of possible orderings by eliminating those that are inconsistent with the new information. We have enough information to sort the list when only one ordering is consistent with all our information. To prove the theorem, we show that we need $t = \Omega\left(N/B \log_{M/B} N/B\right)$ memory transfers to narrow down the possibilities to one consistent ordering.

Stating this argument more formally, let S_i denote the set of possible orderings of the N elements that are consistent with the information we have after i memory transfers. A lower bound on $|S_i|$ is the smallest the set of consistent orderings can be after i memory transfers.

At the beginning of the computation, $|S_0| = N!$. We want to find the smallest i such that our lower bound is $|S_i| \geq 1$. This i represents the smallest number of memory transfers we can do while still getting enough information to eliminate all but one ordering.

After each memory transfer, we compute a new lower bound for the size of S_i. There are two types of memory transfers:

- *Case 1*: We know the order of elements in internal memory, but not the order of the block B being input.

  In this case, we learn the correct ordering of the elements within the block. Before the transfer, there were $B!$ possible orderings of the elements, but after the transfer we can sort the block and eliminate all but one of these orderings.

  We also learn where to interleave the elements in the block into internal memory. Before the memory transfer, the B elements in the block could have been interleaved between the other $M - B$ elements in internal memory arbitrarily. Using a stars and bars argument, the number of possible interleavings is the number of ways to place B bars between $M - B$ stars, or $\binom{M-B+B}{B}$. After the input we can eliminate all but one interleaving. This case is illustrated in Figure 21.6.

  Combining these two pieces of information, we know before the memory transfer, there are

  $$B!\left(\frac{M - B + B}{(M - B)!B!}\right) = B!\binom{M}{B}$$

  possible orderings of the elements in internal memory consistent with our information. After the transfer we reduce this to just one.

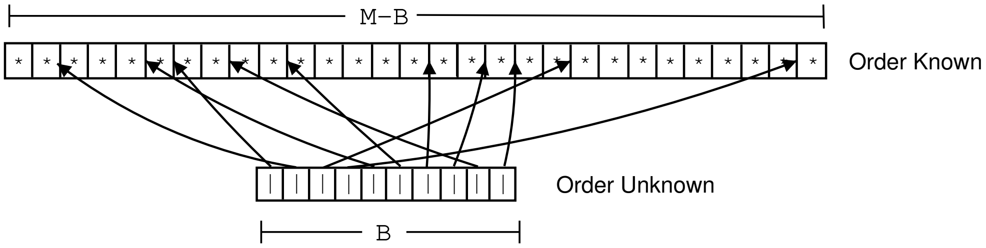


Figure 21.6: Case 1: the order of elements in B is unknown. After a memory transfer we learn the order of elements in the block and the correct place to interleave them into internal memory.

Recall that

$$|S_i| = \text{ \# of consistent orderings after } i \text{ memory transfers.}$$

From our analysis, we know that our lower bound on the size of this set changes according to the formula

$$B!\binom{M}{B}|S_i| \geq |S_{i-1}|$$
$$|S_i| \geq \frac{|S_{i-1}|}{B!\binom{M}{B}}.$$

If we start with $|S|$ consistent orderings, then after t transfers of this type, the number of remaining orderings is at least

$$|S_t| \geq \frac{|S|}{\left(B!\binom{M}{B}\right)^t}. \tag{21.5}$$

- Case 2: The order of records in both main memory and the input block are already known. This can happen if the input block was previously output to external memory.

  In this case, we only learn how to interleave the elements in the block into internal memory. This case is illustrated in Figure 21.7. The analysis is the same as in Case 1, except we lose the $B!$ term because we already knew how to order the block before the transfer.

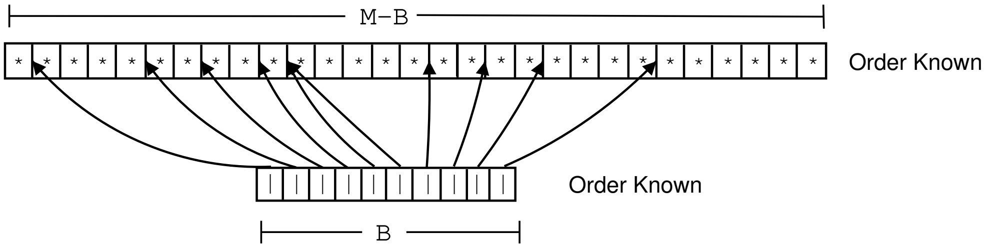


Figure 21.7: Case 2: The order of elements in B already known. After the memory transfer we only learn how to interleave the elements in the block into internal memory.

Therefore, after the transfer, our lower bound on the size of $|S_i|$ changes according to the formula

$$|S_i| \geq \frac{|S_{i-1}|}{\binom{M}{B}}. \tag{21.6}$$

We claim that Case 1 can occur at most N/B times, because there are at most N/B blocks of data to read in. After we have read N/B different blocks, every element has been in internal memory at least once. Combining the two cases, we have the following lemma:

Lemma 21.6 *After t input operations, at least $N!/\left(\binom{M}{B}^t (B!)^{N/B}\right)$ orderings are consistent with the available information.*

Proof $|S_0| = N!$ orderings are possible initially. In the worst case, of our t inputs, N/B of the blocks correspond to Case 1 and $t - N/B$ blocks correspond to Case 2. After reading in the blocks for Case 1, we have

$$|S_{N/B}| \geq \frac{|S_0|}{\left(\binom{N}{B} B!\right)^{N/B}}.$$

After reading in the rest of the blocks, we get

$$|S_t| \geq \frac{|S_{N/B}|}{\binom{N}{B}^{t-N/B}}.$$

Combining the two cases, we get the desired result,

$$|S_t| \geq \frac{N!}{\binom{N}{B}^t (B!)^{N/B}}.$$

□

Recall that our goal is to narrow down the number of possible orderings to 1. To get a lower bound, we need to find the smallest value of t that makes it possible for us to have $|S_t| = 1$. From Lemma 21.6, we see that for this to be the case, t must satisfy

$$\frac{N!}{\binom{N}{B}^t (B!)^{N/B}} \leq 1. \tag{21.7}$$

To work with (21.7), we use the following useful formulae:

- *Stirling's Approximation*: $n! \approx \sqrt{2n\pi}\left(\frac{n}{e}\right)^n$
- $\log(x!) = \Theta(x \log x)$
- $\log \binom{M}{B} = \Theta(B \log \frac{M}{B})$

Rearranging the inequality and taking log of both sides, we get

$$\frac{N!}{\binom{M}{B}^t (B!)^{N/B}} \leq 1$$

$$\binom{M}{B}^t (B!)^{N/B} \geq N!$$

$$t \log \binom{M}{B} + \frac{N}{B} \log (B!) \geq \log (N!).$$

Applying our formula for $\log\binom{M}{B}$, we get

$$tB\log\left(\frac{M}{B}\right)+\frac{N}{B}\log\left(B!\right)\geq\Omega\left(\log(N!)\right).$$

Using Stirling's approximation, and simplifying, we have

$$\begin{aligned}tB\log\left(\frac{M}{B}\right)+\frac{N}{B}B\log(B)&\geq\Omega\left(N\log N\right)\\ tB\log\binom{M}{B}&\geq\Omega\left(N\log\frac{N}{B}\right).\end{aligned}$$

Finally, we reduce this expression to the desired bound,

$$\begin{aligned}t&\geq\Omega\left(\frac{N}{B}\frac{\log\frac{N}{B}}{\log\frac{M}{B}}\right)\\ t&=\Omega\left(\frac{N}{B}\log_{M/B}\left(\frac{N}{B}\right)\right).\end{aligned}$$

This proves the lower bound for t, the number of memory transfers required to sort N elements in external memory. $\square$

In the I/O literature, results are typically expressed in terms of the variables $m=M/B$, and $n=N/B$. We can rewrite our lower bound in this standard notation,

$$t=\Omega\left(n\log_m n\right). \tag{21.8}$$

21.4 Lower Bound for External Permutations

In this section, we prove the lower bound stated in Theorem 21.2. The memory model we use is similar to the one for the sorting problem, and requires the following assumptions:

- External memory is comprised of N blocks of size B for a total memory size of NB. An I/O moves a single block.
- I/O's are simple. A transfer of elements is the only allowed operation. This means the lower bound we prove may not hold if more complicated operations are possible. This model disallows an I/O that might introduce new elements or duplicates existing ones. This assumption is justified because we can ignore any non-simple I/O's that are not needed in finding the permutation.

Under these assumptions, we can view main memory and disk as one big extended array, as illustrated in Figure 21.8. We can essentially ignore all the other memory blocks because they do not contain any of the elements we are trying to permute.

A ***permutation*** is defined as the order of elements in the extended array, ignoring any empty spaces. This setup implies that there is exactly one permutation at all times during the computation. We prove the lower bound from Theorem 21.2:

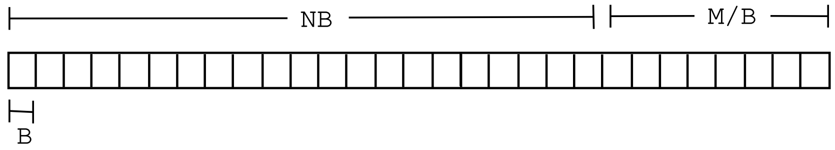


Figure 21.8: Memory model for external permutation problem.

Theorem 21.7 *Rearranging N elements according to a given permutation requires $\Omega\left(\min\left(N, N/B \log_{M/B} N/B\right)\right)$ I/O operations.*

Proof

We can do a permutation either by placing each element directly in its correct destination, or by sorting the list. We analyze the number of memory transfers required for each approach.

- One method for doing a permutation, as illustrated in Figure 21.9, is to move each element from the input array directly to its correct destination in the output array. The input array is contiguous, so reading the elements in order requires only N/B memory transfers. However, a permutation can send elements to an arbitrary locations in the output array, so in the worst case we require a memory transfer for every element. Therefore, the cost of this method is $\Omega(N)$ I/O operations.

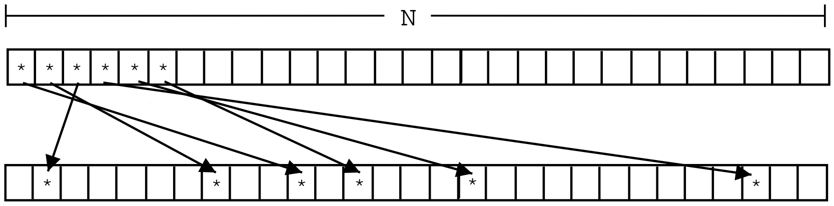


Figure 21.9: One method for doing a permutation is to scan the input array and move each element directly to the correct destination.

- The other method for doing a permutation is to sort the elements of the array. The order in our sort is the ordering imposed by the desired output permutation. To prove the lower bound on the number of I/O's required by this method, we use an information-theoretic argument similar to the one presented in Section 21.3.

  Instead of starting with $N!$ possible orders and eliminating those that are inconsistent with what we learn from a memory transfer, we start with one permutation and count how many different permutations we can realize after each memory transfer. We can solve an arbitrary permutation problem when we can realize $N!$ different permutations.

  Let V_i be the number of different permutations we can realize after i memory transfers. An upper bound on V_i represents the maximum number of different permutations we can realize.

  With no transfers, we only have the initial ordering, so $V_0 = 1$. After each memory transfer, we determine how our upper bound on V_i changes. As before, there are two types of memory transfers:

  - *Case 1: New block*: If the block has never been read into memory, then we can generate new permutations by rearranging the order of the elements in the block. Therefore, we can increase the number of realizable permutations by a factor of $B!$.

We can also generate new permutations by interleaving the elements from the block into different places in internal memory. This increases the number of realizable permutations by a factor of $\binom{M}{B}$.

We obtained the same factor of $B!\binom{M}{B}$ in Case 1 of the proof of the sorting lower bound. However, when doing a permutation, we can also choose which of the N blocks from the extended array we transfer. This increases the number of realizable permutations by an extra factor of N.

Recall that

$$V_i = \# \text{ of realizable permutations after } i \text{ memory transfers}$$

Combining all these factors, we have the relationship

$$V_i = N(B!)\binom{M}{B}V_{i-1}. \tag{21.9}$$

– *Case 2: Already-read block*: Since the block has been read into memory at least once, we can not generate new permutations by reordering the block. This case is similar to Case 2 in the proof of the sorting lower bound. In this case, we do not have a factor of $B!$, so

$$V_i = N\binom{M}{B}V_{i-1}. \tag{21.10}$$

Once again, the claim is that Case 1 happens at most N/B times. By the same logic as in Lemma 21.6, after t memory transfers, we know

$$V_t \leq (B!)^{N/B}\left(N\binom{M}{B}\right)^t. \tag{21.11}$$

We want V_t, the number of different permutations that we can realize after t memory transfers, to be at least $N!$. Therefore, we want the smallest t such that

$$(B!)^{N/B}\left[\binom{M}{B}N\right]^t \geq N!. \tag{21.12}$$

Taking the log of both sides and using Stirling's approximation, we get

$$\begin{aligned}
\frac{N}{B}\log(B!) + t\left(\log\binom{M}{B} + \log N\right) &\geq \log(N!) \\
N\log B + t\left(B\log\left(\frac{M}{B}\right) + \log N\right) &\geq \Omega(N\log N) \\
t\left(B\log\left(\frac{M}{B}\right) + \log N\right) &\geq \Omega\left(N\log\left(\frac{N}{B}\right)\right).
\end{aligned}$$

This expression can be rewritten as

$$t \geq \Omega\left[\frac{N\log(N/B)}{B\log(M/B) + \log N}\right].$$

For this value of t, there are two cases to consider. If $\log N \leq B\log(M/B)$, then we get the bound

$$t \geq \Omega\left(\frac{N}{B}\log_{M/B}\left(\frac{N}{B}\right)\right). \tag{21.13}$$

If $\log N > B \log\left(\frac{M}{B}\right)$, then this implies $B \ll \sqrt{N}$. Then, we have

$$\begin{aligned}\frac{N\log(N/B)}{2\log N} &= \frac{N\log N - N\log B}{2\log N}\\ &= \Omega\left(\frac{1}{2}\left[N - N\frac{\log B}{\log N}\right]\right)\\ &= \Omega\left(\frac{1}{2}\left(N - \frac{N}{2}\right)\right)\\ &= \Omega\left(N\right).\end{aligned}$$

Taking the minimum of the lower bounds for the two different methods, we prove that the number of I/O's require to do a permutation is

$$\Theta\left(\min\left\{\frac{N}{P}, \frac{N}{PB}\frac{\log(N/B)}{\log(M/B)}\right\}\right) \tag{21.14}$$

□

21.5 Cache-Oblivious Sorting

Although the parallel merge sort presented in Section 21.2.2 achieves the optimal number of memory transfers, it requires the user to know the value of B. In the case of $P = 1$, it is actually possible to achieve the optimal number of memory transfers from Theorem 21.1 without knowing the block size B. In this section, we briefly describe funnelsort, a cache-oblivious mergesort presented in [7].

The difficult part of doing an I/O-efficient mergesort is determining how many elements to merge during the recursive step. As we saw in Section 21.2.1, a merge that only splits the problem in half does not achieve the optimal bound. In general, if we split the problem into too few sub-problems, we do not get the optimal number of memory transfers because we have to recurse too many times before our base case fits completely into cache. On the other hand, if we split the problem into too many sub-problems, then when doing a k-way merge, k may be larger than the number of blocks that can fit in cache. In this case, we do not use the cache efficiently when we merge. To do a cache-oblivious sort, we require a k-way merge, with k chosen so that it is not too large or too small.

In the funnelsort algorithm, illustrated in Figure 21.10, we take an input array of size N, split the array into $N^{2/3}$ sub-arrays of length $N^{1/3}$, sort the sub-arrays, and then merge the data to obtain a N element sorted list.

This method requires a ***funnel***, a black box algorithm that merges. See Figure 21.11. The funnel presented in [7] requires $O\left(\frac{k^3}{B}\log_{M/B}\frac{k^3}{B} + k\right)$ memory transfers and $O(k^2)$ space.

The algorithm also makes a tall-cache assumption, that $M = \Omega(B^2)$. This assumption only requires that the number of blocks that can fit into cache, M/B, be larger than the block size B.

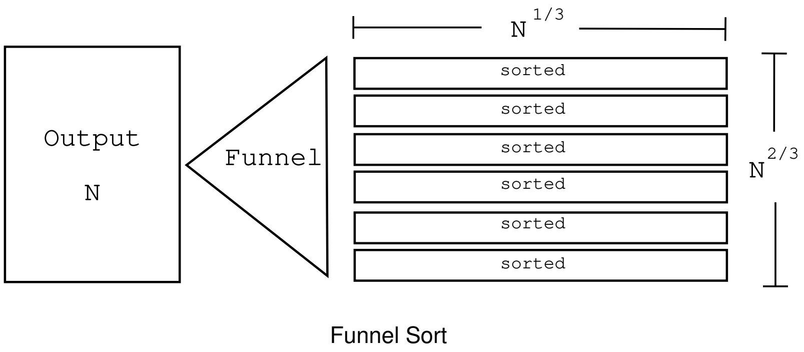


Figure 21.10: Illustration of funnel sort.

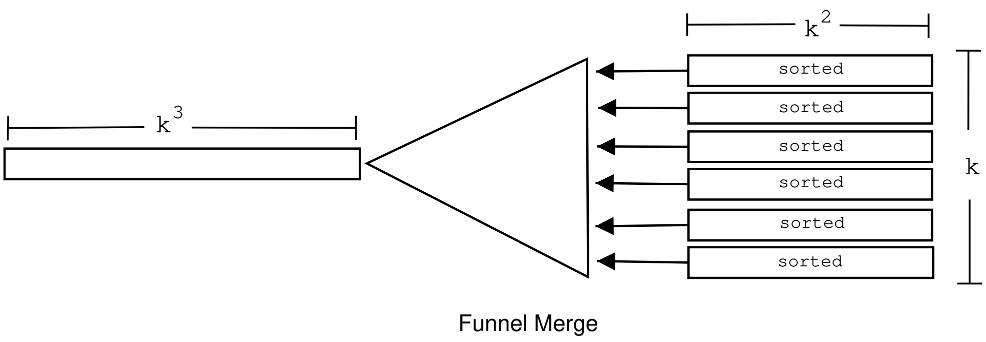


Figure 21.11: A funnel that takes k sorted input arrays, each of length k^2, and outputs a sorted list of length k^3.

6.895 Theory of Parallel Systems	Lecture 22
Pick a Winner	
Lecturer: Tom Leighton	*Scribe: He Yuxiong and Sajindra Jayasena*

Lecture Summary

1. *The Pick-a-Winner game*
   The section describes the rules of the pick-a-winner game and shows two simple observations.

2. *Strategies to maximize the profit*
   This section describes the optimal strategy, analyzes its performance, shows the key intuitions and sketches the proof of the performance guarantee theorem.

3. *Pick-a-Winner - K player Game*
   This section looks at how the game and strategy vary with multiple players

4. *Applications*
   This section looks at some applications that can be modelled using Pick-a-Winner single player and multi-player games

In the lecture, we consider a class of problems in which a decision-maker is required to select from among numerous dividend-paying options. The goal of the decision-maker is to select an option (or a set of options) that will have the best future return. The goal of the lecture is to show how to use probabilistic method to get the performance guarantee without making probabilistic assumptions.

22.1 The Pick-a-Winner game

The section describes the rules of the pick-a-winner game and shows two simple observations.

Pick-a-winner game has one player. The player is required to select one from among n dividend-paying options. The selection can be made on any day within d days. Once the player makes the decision, he is not allowed to change it. The profit of the player is equal to the dividends paid by the option after it has been selected. The goal of the game is to maximize the profit.

In the online version of this game, the player only knows the past dividend payments but he does not know the future payment schedule. In addition, the past and the future schedule may not be correlated. In fact, the future is decided by the adversary who knows the player's strategy and seeks to minimize the profit of the player. The question is how much money the player can make.

We can make two simple observations based on the rules of the game.

Observation 22.1 *The profit for any deterministic player is 0, even if $P_{opt} = \$d$.*

Proof Before the player makes any decision, the adversary pays a dividend for every option on every day. After the player makes a selection, the dividends are stopped for the selected option. The adversary can still pay a dividend for all the other options on every day. So even if $P_{opt} = \$d$, the profit of the deterministic player is 0. □

Observation 22.2 *By choosing a random option at the start, we can get an expected profit $P_{opt}\ /n$.*

Proof Since the adversary only knows our strategy but not the random number, we have a $1/n$ chance of selecting the optimal option. □

22.2 Strategies to maximize the profit

This section describes an optimal strategy, analyzes its performance, shows the key intuitions and sketches the proof of the performance guarantee theorem.

An **Optimal Strategy** is described below:
Player selects target profit P and confidence level $1-q$. We label the options as $O_1, O_2 \ldots O_n$. Each day, the player checks the status of each option to see which one has a dividend. For each x ($x = 1 \ldots n$), if O_x has a dividend on the day, the player flips a coin to decide whether or not to make a selection to O_x . The player will make a selection on O_x with probability $2^{(qi/P)}/(Pn^2)$, where i is the total number of dividend O_x has generated so far. If the selection has been made, the player stops flipping coins and waits for the job to be completed.

There are two important intuitions and facts regarding to the optimal strategy. They are also the main supports for Theorem 22.3

Intuition and Key Fact 1 *The selection probabilities start small and grow quickly. The purpose of the slow start is to reduce the possibility of misleading by adversary for the false start. The objective of fast growth is to ensure that we make some selections before the end of the game with high probability.*

Intuition and Key Fact 2 *The growth must be slow enough to ensure that whichever option is selected, the posteriori probability is high such that the option was selected at least P payments before the final payment.*

Theorem 22.3 *If P_{opt} is known and $P_{opt} > 3P \log n/q$, the strategy will return a profit $\geq P$ with probability $> 1-q$*

Sketch of Proof We will only outline the proof in the class. The proof was based on the two intuitions and following the procedures given in [4].

Let S denote a sample space of coin flips where there will be one coin flip for each pair (x, y) for which O_x has its dividend at day y. If O_x has its i'th dividend at day y, the probability of getting head is $2^{(qi/P)}/Pn^2$. In particular, the player will schedule his job on O_x , at step y if the (x, y) coin is head and if the (x', y') coin is tail for all (x', y') such that $x' < x$ or $y' = y$ and $y' < y$. Let S_{win} denote the subspace of S consisting of sample points for which the player gets its targeted profit P or higher. In order to show the $Pr(S_{win})$, a second space S' is considered for which it is easy to show that $Pr(S')$ is large and that $Pr(S_{win})$ is nearly as large as $Pr(S')$. S' is defined to comprise of the sample points for which there is at least one Head and for which the first d flips for each O_i resulted in Tails. There are two steps involved.

1. Prove $Pr(S') \geq (1 - 1/n)$
   Because of the slow start and the fast growth, this step is to show that the probability of misleading by adversary at the beginning and the probability of not making any choice at the end are small. The probability of making a selection among the Pn flips for which $i \leq P$ is at most

$$Pn(\frac{2^q}{pn^2}) = O(\frac{1}{n})$$

The probability of making no selection at the end of the game is at most

$$\begin{aligned}
Pr[no\ selection] &\leq \prod_{i=1}^{P_{opt}} [1 - (\frac{2^{qi/P}}{Pn_2})] \\
&< [1 - (\frac{2^{q(P_{opt}-P)/P}}{Pn_2})]^P \\
&< [1 - (\frac{n2^{-q}}{P})] \\
&< e^{-\frac{n}{2}}
\end{aligned}$$

So $Pr(S') \geq 1 - O(1/n)$

2. Prove $Pr(S_{win}) \geq (1 - O(P \log n / P_{opt}))Pr(S')$
   It is the step to prove that S_{win} is nearly as large as S'. It is the proof step from Intuition 2 and shows that the probability of making selection at least P payments before the final payment is almost as likely as making a selection before the final payment.

   The proof of this step is to construct an injection from S' to S_{win}. Please refer to lemma 2.2 in [4] for the details.

Combining the results of step 1 and step 2, we are able to prove the theorem. If P_{opt} is known, choose target $P = qP_{opt}/(3 \log n)$ and confidence parameter $1 - q$ as desired, profit P will be obtained with probability $1 - q$. □

Theorem 22.3 shows the performance guarantee when P_{opt} is known. When P_{opt} is unknown, we need to make a guess of it and choose target $P = q2^R/(3 \log n)$ where R is uniformly selected from $[1, \log d]$. 2^R will approximate P_{opt} value among $2, 2^2 \ldots d$ with probability $1/\log d$. So profit $P_{opt}/(c \log n)$ will be obtained with probability $1/(c \log d)$ where c is a fixed constant. Combining the result with that in Theorem 22.3, we know the expected return is $cP_{opt}/(\log n \log d)$ without the knowledge of P_{opt} .

From the theorem and analysis in the section, we know that the past can be combined at random to provide a very good prediction of future performance even if the past does not correlate with the future.

22.3 Pick-a-Winner Game - with Variations

Now let us look at a variation of the previous game. The game has the following conditions and rules.

- K players are playing the game.
- Each player is allowed to select one option.
- There are N options that the players can select.
- There are D days that a game can be played.
- A particular dividend of an option is paid only to the highest priority player holding the particular option.
- If a player needs to change the option ,it will cost 1 $ per change.

let P_i denote the profit of the i^{th} ranked player.
Let us look at an example with 3 players and consider the following observations.

- Player *1* selects D on day *2* , swaps to E on day *5*.
- Player *2* selects D on day *1*, swaps to C on day *3*.
- Player 3 selects E on day *2*. No swaps.

The outcome would be as follows:

Results: P1 = $4, P2 = $0, P3 = $0.

	1	2	3	4	5	6	7	8	9	10
A	$1	$1								$1
B	$1		$1		$1	$1		$1	$1	
C	$1			$1						
D		$1	$1	$1	$1					
E	$1					$1				$1
F	$1					$1		$1		
G					$1					
H										

Figure 22.1: Game Results

After playing the game, now let us try to answer the following questions.

Q.Can Swapping Help ?
Our own intuitive answer would be **NO**. Since the the future outcome cannot be predicted , the future of one option seems to be indistinguishable from the future of another.
But further analysis shows that the answer is **YES**.The basic idea is that once the past is seen , the randomize strategy may dictate change.
The advantages obtained by swapping are :

- Increased profits gained , where the increase in the new income surpassed the swapping cost,
- Increased probability of success,
- Knowledge of P_{OPT} is no longer needed.

Key Idea
One of the key ideas that could be gained is:
Break the time into epochs , and run independent trials in each epochs. It could be observed that with a high probability, most trials would be successful.

Q. Is Coordination needed among players to maximize one's profits?
The answer is **YES**. But very little coordination is needed.
The following items need to be considered:

- As long as no player is allowed to be too greedy, and each player can select in isolation,
- Other players are considered to be part of the adversary,
- At worst, the j^{th} ranked player can choose the j^{th} ranked option.

22.4 Results in Summary

In Figure 2 , summarized results of the single and multiple player Pick-a-Winner games are given.

# Players	# Picks	# Swaps	P_{OPT} known ?	Profit	Prob[success]
1	1	0	NO	C P_{OPT}/log n	1 / log d
1	1	0	YES	C P_{OPT}/log n	.99
1	1	C log n log d	NO	C P_{OPT}	1 - o(1/ n)
1	1	C log n	YES	C P_{OPT}	1 - o(1/ n)
1	O(1)	C log n	YES	(1 − o(1)) P_{OPT}	1 - o(1/ n)
K*	O(1)	C log n log P_{OPT}	NO	(1 − o(1)) P_{OPT}	1 - o(1/ n)

Figure 22.2: Game Results in summary

22.5 Applications

In this section we will be looking at some real life examples where the strategy of Pick-a-Winner game can be applied.

22.5.1 Scheduling a single job on NOWs

The strategy applied in single player Pick-a-Winner game can be applied in solving this problem. The problem formulation is as follows:

- Bob has a job requiring P cycles that he wants to run overnight,
- Bob picks a workstation for his job and assigns the job a background priority,
- The job gets done if the chosen workstation has at least P available cycles during the night.

The Question to answer is : How should Bob choose a Workstation?
Solution:
We will try to model the answer from the single player Pick-a-Winner game with the following mapping.

- Workstations → Options
- Cycles → Days (overnight - d days)
- Availability → Dividends paid
- Target profit → P cycles

The goal is to maximize probability of success, ***1 - q***
In the figure 3, A denotes availability.
Results:

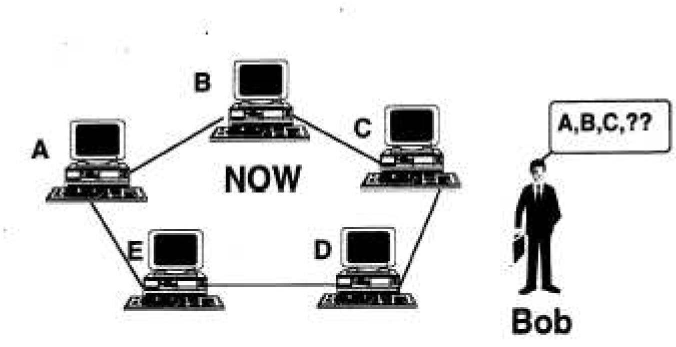


Figure 22.3: which workstation to select?

	1	2	3	4	5	6	7	8	9	10
A	A	A	A	A			A	A	A	A
B	A									
C					A					
D	A									
E			A							
F			A	A	A	A	A			
G					A	A				
H										

Figure 22.4: Availability of Workstations

If some work stations will be available for at least ***3 P*** log ***n*** steps, Bob can get ***c*** log ***n*** jobs of duration ***P*** done with a probability of ***1 - o(1/n)***.
The following pointers should be considered:

- At most 1 work station is used by Bob at any time,
- Bob can kill a job in progress and restart the job elsewhere,
- The high probability of success holds for all patterns of workstations availability,
- If Bob can use $\mathbf{O(1)}$ workstations at a time, then he can attain 99 % efficiency.

22.5.2 Scheduling multiple Jobs on NOWs

Now we will look at scheduling multiple jobs. If there are ***K*** people , each with background jobs to run ,we can use the strategy adopted in ***K-player Pick-a-Winner*** game.

Coordination:
Each person receives a priority , but otherwise works independently.Competing demands for service would also be resolved by priority.
Now Let us look at the results.
Results:

If the j^{th} best workstation will be available for at least T_j steps, then the j^{th} ranked person will get c.T_j cycles with a high probability.
Let us look at another game.

Application: Video Servers

Let's formulate the problem the following way.

- During the day, d customers request movies.
- At most k out of n movies will be shown at a pre-arranged time in each evening.
- Each request is immediately accepted or rejected.
- If a request is accepted, the movie must be shown. Rejected customers are lost.

Goal:

The goal is to maximize the accepted requests thus maximizing profits.
Before looking at the game based solution, let's look at a previous solution proposed by Aggarawal, Garay and Herzberg [2].
According to their solution:

- In the worst case, $\boldsymbol{k}$ requests would be accepted .
- $\boldsymbol{c}\ P_{OPT}\ /\log\ \boldsymbol{n}$ requests would be accepted with some lookahead.

Now let's look at the solution that was obtained from the K-player Pick-a-Winner game. Following are the mappings.

- Movies $\rightarrow$ Options
- Customers $\rightarrow$ Days
- Requests $\rightarrow$ Dividends Paid
- Target Profit $\rightarrow$ Number of Accepted Requests.

Strategy: The strategy is to guess the number of requests R_j for j^{th} most popular movie and set target profit for j^{th} movie to be $\boldsymbol{c}\ R_j/\log\ \boldsymbol{n}$.
Results: The expected number of accepted requests would be =
$\boldsymbol{c}.P_{OPT}\ /\ (\log\ \boldsymbol{n}\ /\log\ \boldsymbol{d}\ )$
The following diagram denotes the requests and movies in the same analogy of playing the K-player game.

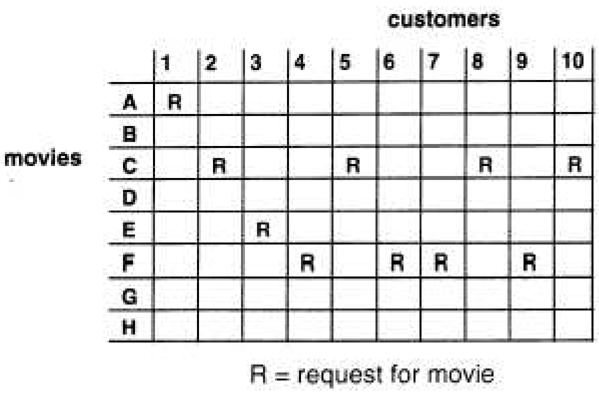


Figure 22.5: Movies Vs. Requests

Application: On-line Set Cover

Problem:
Let's formulate the problem in the following manner.

- Given $\boldsymbol{n}$ Sets $S_1, S_2..., S_n$.
- $\boldsymbol{d}$ items $V_1, V_2, ..., V_d$ arrive one per each step.
- As each item arrives , we are able to distinguish the set(s) the item belong to.

Goal:
The goal is to pick **k** sets to cover as many items as possible. The rewarding mechanism is as follows.

- Credit is given only once for each item, even if it is contained in several sets.
- Credit for an item is given only if the set was selected before or during the step when the item arrives.

Let's look at an example where this strategy can be applied.

Example:
A problem associated with Video server contention and availability, where each customer specifies a collection of movies , any one of which is fine.

Solution:
The solution is the same as for Video servers except that each customer can specify several movies.
The following mapping maps the problem entities to the set cover problem.

- Items $\rightarrow$ Customers
- Sets $\rightarrow$ Movies.

The following results could be obtained.

Results:
Total expected credit would be

$$cP_{OPT}/(\log n \log d)$$

The following diagram denotes the representation of the problem where the request for each movie is denoted by a set S_1 to S_n and customers are represented through the numbering from V_1 to V_d.

In Figure 6, $\in$ denotes each customer's request for a particular movie.

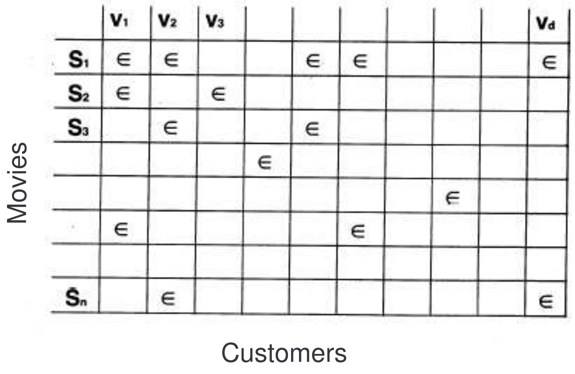


Figure 22.6: Movies Vs. Requests

Bibliography

[1] Alok Aggarwal and Jeffery Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.

[2] S. Aggarwal, J.A. Garay, and A. Herzberg. Adaptive video on demand. proc. 3rd annual european symp. on algorithms.

[3] AMD. Software optimization guide for AMD Athlon 64 and AMD Opteron processors. Technical report, AMD, 2003.

[4] Baruch Awerbuch, Yossi Azar, Amos Fiat, and Tom Leighton. Making commitments in the face of uncertainty: how to pick a winner almost every time. 1996.

[5] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, Cambrdige, MA, November 2000.

[6] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K.H. Randall. An analysis of dag-consistent distributed shared memory algorithms. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Padua, Italy, June 1996.

[7] Matteo Frigo, Charles E. Leiseroson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science, FOCS'99*, October 1999.

[8] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*, October 1999.

[9] James R. Goodman. Using cache memory to reduce processor-memory traffic.

[10] James R. Goodman. Using cache-memory to reduce processor-memory traffic. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 124–131, 1983.

[11] A. R. Karlin, M. S. Manasse, Larry Rudolph, and D. D. Sleator. Competitive snoopy caching. In *Proceedings of the 27th Annual Symposium on the Foundations of Computer Science*, 1986.

[12] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays - Trees - Hypercubes.* Morgan Kaufmann, 1992.

[13] C. E. Leiserson and M. Feng. Efficient detection of determinacy races in cilk programs. In *Proceedinges of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–11, Newport, Rhode Island, June 1997.

[14] Code for sort algorithms. MIT 6.895: Handout 3, 2003.