Mathematics 18.337, Computer Science 6.338, SMA 5505

Parallel Scientific Computing

Spring 2002

Lecturer: Alan Edelman[1]

MIT

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

Preface

These lecture cover a mixture of material that we feel students should be exposed to. Our primary focus is on modern numerical algorithms for scientific computing, and also on the historical trends in architectures. At the same time, we have always felt that students (and the professors) must suffer through hands-on experience with modern parallel machines. Some students enjoy fighting new machines; others scream and complain. This is the reality of the subject.

During 1994 and 1995 our students programmed the 128 node Connection Machine CM5. This machine was the 35th most powerful computer in the world in 1994, then the very same machine was the 74th most powerful machine in the spring of 1995. By December 1995, the machine sunk to position 136. The machine was still in the top 500 list when it was retired in the summer of 1997. The year 1996 was a transition period. We used the IBM SP-2 and Boston University's SGI machines. In 1997 and 1998 we moved over to SUN's network of SUN Ultra Enterprise symmetric multiprocessors housed at MIT's laboratory for Computer Science.

In 2002, this class joined the Singapore-MIT alliance. Our class is being videotaped and watched on both sides of the world. TA Ron Choy has built a 16 processer Beowulf cluster that is being used on the MIT side. Tony Wen has helped modernize our notes.

The early versions of the notes were coauthored by Shang-Hua Teng and Robert Schreiber, I would like to thank our numerous students who have written and commented on these notes and have also prepared many of the diagrams. We also thank the students from Minnesota SCIC 8001 and the summer course held at MIT, Summer 6.50s (also taught by Rob Schreiber) for all of their valuable suggestions. These notes will probably evolve into a book.

Meanwhile, we are fully aware that in the current form, the notes are incomplete, contain mathematical and grammatical errors, and do not cover everything we wish. They are an improvement over last year's notes, but not as good as next year's notes will be. I view these notes as a basis on which to improve, not as a completed book.

It has been our experience that some students of pure mathematics and theoretical computer science are a bit fearful of programming real parallel machines. Students of engineering and computer science are sometimes intimidated by mathematics. The most successful students understand that computing is not "dirty" and mathematical knowledge is not "scary" or "useless," but both require hard work and maturity to master. The good news is that there are many jobs both in the industrial and academic sectors for experts in the field!

A good course should have a good theme. We try to emphasize the fundamental algorithmic ideas and machine design principles. We have seen computer vendors come and go, but we believe that the mathematical, algorithmic, and numerical ideas discussed in these notes provide a solid foundation that will last for many years.

Contents

Chapter 1

Introduction

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

1.1 A General Look at the Industry

This is a science course at MIT, but somehow it seems appropriate to wonder how special purpose parallel computation really is. Maybe this sort of analysis more properly belongs in the introduction to a business school case study.

Is it correct to conclude that there will never be an enormous market for parallel computers? We are certain that parallel computers will be everywhere. The editorial below may not immediately substantiate my conclusion, but it does illustrate the idea in the famous quote by Shaw: “You see things; and you say, ‘Why?’ But I dream things that never were; and I say, ‘Why not?’

> **Editorial on markets for technology:** We probably all have our favorite examples. How many of us have heard that there will never be a large market for computers. Only “nerds” would have a computer in the living room. I for one have systematically heard that 1) only nerds would communicate by electronic mail, 2) French people would never embrace electronic mail, and 3) lawyers would never accept electronic mail. I suspect that 100 years ago people might have thought that nobody would want a telephone in their living rooms. The surest way to be wrong is to reason that nobody is buying it now hence nobody will ever buy it.

Some books might start with a machine and a programming language. This is a mistake. The machines have been changing and the programming languages are maturing, but still in 1996 the subject is in a state of flux. Many machines are still not as usable as people would want. Quite likely, too many companies have been trying to create their own proprietary products.

> **Editorial on proprietary products:** Look at the fast evolving world wide web. For years we wondered why we were all content to live with `ftp`. One of us used to ask why my bank account balance was unavailable from my workstation. The other now wonders why his telephone is not an integrated part of his workstation. Somehow, just now, the infrastructure and the public’s perception is right for the internet to catch on, not only for techies, but for everybody.

> However, an interesting phenomenon is developing. Various internet providers are attempting to be the sole owners of the internet. They are learning the hard way that attempts to own the market are doomed to failure. The realists will understand that the web is public, and a common infrastructure benefits all. Attempts to dominate the internet are doomed to failure. Bill Gates, beware!

Parallel computers do not yet have a common infrastructure, but things are improving. Nevertheless parallel machines continue to be more painful to use than serial machines.

> **Editorial: What do scientists and students with scientific applications want on a parallel machine?** People get religious about programming languages, and among the students, even the engineering students, C is growing in importance over Fortran. Nevertheless if we had to pick four things that would make a large number of people very happy we would have to list
>
> - A reliable High Performance Fortran Compiler
> - No difficulties using the Message Passing Interface MPI
> - A high quality portable common debugging and visualizing mechanism such as Thinking Machines Corporation's Prism product. It is crucial that every machine support the same product.
> - A non-proprietary performance monitoring interface whose timings one can trust and allow for predictions. Vendors just do not seem to be able to solve the engineering question of providing the user with the answer to the question of just how long did the program take to execute.

[To be written: discussions about parallel libraries]

Which specific machines did we use in our class? Last year (1995) we used MIT's 128 processor CM-5, and a 12 processor SP-2. This year we expect to use the SP-2 and some Silicon Graphics machines currently located at Boston University. If we are lucky, the MIT lab for Computer Science will receive a donation of an interesting symmetric multiprocessor that we will use in our class.

Most of you may have either seen the 1993 movie or read the 1990 Michael Crichton novel *Jurassic Park*. The 1993 movie shows the CM-5 in the background of the control center and even mentions the Thinking Machines Computer, but you could easily miss it if you do not pay attention. The older novel has the dinosaur theme park controlled by a Cray vector supercomputer. Many in this field believe that the Cray belonged on the dinosaur side of the *Jurassic Park* fence!

In fact, in 1996, both vector supercomputers and proprietary, custom-engineered parallel computers may be on their way to extinction. The investments required to keep pace with evolving hardware technology can't be justified by the size of the market. At Cray, for example, the vector supercomputer (the T90) at 60 gigaflops and 8 gigabytes, is overshadowed by the new parallel machine (the T3E), based on the DEC alpha chip, which offers up to 1.2 teraflops and 4 terabytes. Kendall Square Research designed its machine from the ground up in order to exploit novel ideas about caching and memory architecture: they are now out of business. Tera Computer Company, which has followed the same business model as KSR, will very likely suffer the same fate. And the first big parallel computer company, Thinking Machines Corporation, once located on expensive Cambridge real estate near MIT, has trimmed itself, moved to less-expensive Bedford, and has emerged from bankruptcy protection. Their latest announcement has them teaming up with Sun Microsystems for software for the next generation of machines.

1.2 The Web as a Tool for Tracking Machines

For the latest info on the parallel machine market, the World Wide Web is unmatched. The address `http://www.cs.cmu.edu/~scandal/vendors.html` currently (January 1996) points to Alta Technology. Chen Systems, Convex Computer Corporation, Cray Research, Digital Equipment Corporation, Fujitsu, Hewlett-Packard, Hitachi, IBM, ICE, Intel, Meiko, NEC, Parsytec, Siemens Nixdorf, TERA Computer, and Thinking Machines Corporation.

1.3 Scientific Computing and High Performance Computing

There is no formal definition of *parallel scientific computing* and we do not intend to give one here. However, the goal of parallel scientific computing is very well defined, that is, to find faster solutions to larger and more complex problems. Parallel scientific computing is a highly interdisciplinary area ranging from numerical analysis and algorithm design to programming languages, computer architectures, software, and real applications. The participants include engineers, computer scientists, applied mathematicians, and physicists, and this team is growing.

High-performance parallel computers have become a fundamental tool for computation intensive tasks in real applications. Computer architecture is driven by technology. The advance of very large scale integration (VLSI) makes it possible to develop faster computers that have smaller physical volume. The new technology makes feasible massive parallelism. To sustain increases in computation demands, the industry must embrace multiprocessing in almost all computers, either in the format of tightly coupled multiprocessor supercomputers, or loosely coupled workstation clusters.

Scientific numerical computing plays a critical role in the development of high-performance parallel computers. It provides a class of problems that are so demanding of computational cycles, a class of problems that do not require completely "general purpose" machine, yet are rich in content and in real applications, and a class of problems that give raise challenging scientific and engineering activities ranging from mathematical modeling, algorithmic design to software/hardware/architecture development. Scientific applications were the main motivation for the development of supercomputers in the past decade. Now, parallel machines have begun to find more applications in symbolic computing, large-scale databases, and multimedia environments.

Conversely, parallel computers are the most important platforms for scientific simulation. To perform meaningful simulation within a reasonable time-span computational scientists have to make trade-offs between accuracy, time, and space. The larger memory a machine has, the more powerful it is, the less restriction and simplification we need to impose on the numerical approximation of real problems and their physical domains, and the more accurately we can solve real problems.

1.4 A Natural 3D Problem

The most challenging problems in parallel scientific computing are those in three or higher dimensions. One simple reason is that problems in higher dimensions are of a very large scale.

Parallel scientific computing itself is a three dimensional discipline whose three major dimensions are

$$\begin{aligned} x &= \text{scientific and engineering applications} \\ y &= \text{mathematical algorithms} \end{aligned}$$

$$z \ = \ \text{parallel architecture and programming}$$

Our goal in the course is to provide a reliable road map to the projection of our field into the (y, z) plane!

1.5 Components of a Parallel System

The components of a parallel system are not just its hardware, architecture, and network topology. They include its operating system, parallel languages and their compilers, scientific libraries and other software tools.

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

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

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

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

1.6 Scientific Algorithms

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

1.7 Applications

Parallel (numerical) methods have been used to solve problems in virtually all areas of science and engineering:

- Structural analysis and mechanical engineering.
- Computational chemistry and material science
- Computational particle physics
- Computational fluid and plasma dynamics, air and space vehicle design, and space science.
- Environment and Earth science: climate modeling, glaciology, weather forecasting, an ocean modeling.
- Structure biology and medical imaging, signal processing, drug design
- ECAD and circuit validation
- Economics and finance
- Optimization, searching and databases
- Oil and gas exploration

1.8 History, State-of-Art, and Perspective

The supercomputer industry started when Cray Research developed the Cray-1, in 1975. The massively parallel processing (MPP) industry emerged with the introduction of the CM-2 by Thinking Machines in 1985. The industry is represented by several companies including Cray Research (the T90 and the T3E), Intel Supercomputer System Division (the Paragon), Thinking Machines Corporation (the CM-5 and the Global), IBM (the SP-2), HP (the Convex Exemplar), MasPar (the MP-2), Tera (the MTA, which is also the subway system in Boston), and Silicon Graphics (the PowerChallenge). The MPP industry has received substantial support from the U.S. government. The *US High Performance Computing and Communication* program was designed to explore technologies for building and using such machines. Most MPP installations are at government research centers (e.g., NASA Ames and Los Alamos) and universities and supercomputer centers (NCSA, Pittsburgh, MSI, etc).

Achieving high performance for numerical computing for scientific applications are the main goals. Indeed, most of the benchmarks are numeric oriented such as dense linear algebra computations, as represented by the heated race for the top of Linpack benchmarks between Intel and Thinking Machines.

High performance parallel computers have begun to play an important role in science and engineering development. At the same time, the vendors of parallel supercomputers have engaged in a brand of hype unusual in science and engineering. Now that the cold war has ended, the supercomputing market in major universities and government centers is close to being saturated, at least for the moment. (Nevertheless, the U.S. Department of Energy has, this past year, purchased a massively parallel supercomputer that far exceeds the capacity of any previously built.) The parallel computing industry is evolving. There are more and more "real" demands from commercial companies. Hence the cost/efficiency trade-off is more relevant. More friendly programming environments are desired. Other non-numeric applications such as in large databases, optimization, business and finance are in greater demand. And of course the parallel computing industry is more competitive. Workstation clusters bring new challenges to tightly coupled MPP. The parallel computing field and industry are maturing. The application domain of parallel computing is expanding.

1.9 Parallel Computing: An Example

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

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

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

(Illustrate on tree.)

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

This is an example of what is often known as a *reduce* operation. We can replace the addition operation with any associative operator to generalize. (Actually, floating point addition is not associative, leading to interesting numerical questions about the best order in which to add numbers. This is studied by Higham [**?**, See p. 788].)

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

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

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

> **Four Bad Parallel Algorithms for Computing $1 + 2 + 3 + \cdots + 10^6$ on Four Processors**
>
> 1. Load imbalance. One processor has too much to do, and the others sit around waiting for it. For our problem, we could eliminate all the communication by having processor zero do all the work; but we won't see any parallel speedup with this method!
>
> 2. Excessive communication.
>
>   Processor zero has all the data which is divided into four equally sized parts each with a quarter of a million numbers. It ships three of the four parts to each of

the other processors for proper load balancing. The results are added up on each processor, and then processor 0 adds the final four numbers together.

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

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

   We are thus requiring a sequential computation: our binary summation tree is maximally unbalanced, and has height $1,000,000$. It is always useful to know the critical path – the length of the longest path in the dataflow graph – of the computation being parallelized. If it is excessive, change the algorithm!

4. A scheduling problem. For this kind of performance bug, we get a better illustration by changing the problem. Let there be a 1000×1000 matrix, distributed to the four processors by columns, so that processor zero owns all of columns 1 through 250. The job is to add the columns, producing a vector of length 1000. The constraint, which makes this problem interesting, is that within each row we are required to do the addition left-to-right! So the computation in each matrix row is sequential, and the parallelism comes from the fact that there are 1000 independent row sums to be computed.

   The bad algorithm has processor zero do all of its row sums, then send them all in a message of 1000 words to processor one, etc. We have chosen to defer sending the sum of the first rows until we compute the sum of the last rows, thereby sequentializing an otherwise parallel task. The right idea is to schedule high-priority tasks (the sending of sums to the next processor) sooner than lower priority tasks. The fact of the latency term A in message cost makes the optimum scheduling here tricky – single-word messages may not be ideal either.

1.10 Exercises

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

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

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

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

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

Chapter 2

Computer Architecture

Anybody who uses a computer ought to have an understanding of basic architectual components such as memory, registers, caches, and buses.

The actual data that a computer works with is stored as binary digits (0's and 1's) on memory chips. This collection of chips is often denoted *main memory* (or DRAM in the PC ads). The minimum number of bits (binary digits) that can be accessed by one request to memory is known as a *word*. Word sizes are almost always one *byte* (meaning eight bits) long. Single precision numbers are usually four words long, while double precision are eight words long. Computers access multiple words simultaneously.

If M is the total number of words available on a machine, then a unique word may be specified by an address from 0 to M-1.

If words are always eight bits wide, what is all this media fuss about 32 bit technology, or 64 bit technology? These numbers (32 or 64) represent $\log_2(M)$, i.e., with 64 bit technology, there are potentially $M = 2^{64}$ words available on the machine. The truth is that there are often fewer words actually available, but these many are at least potentially available. In a given United States area code, we might say we have seven digit "technology", but not all 10^7 numbers have been given away at any one time. It is always good to leave room for more. Furthermore, some memory addresses may not point to words in memory at all, but rather to other components where you may want to send bits.

To actually work with data, a processor must load it into its own *registers*. Registers are special storage areas on a processor that may be quickly accessed.

Words travel from memory to registers via *buses*. A bus is a piece of hardware that allows the various components of a computer to talk to each other.

It was well understood in the 1960s that the time taken to load data from memory into registers slowed down machine execution time. Most memory simply can not release the data fast enough for the processor to work with the data. Therefore a device designed to accelerate the average transfer time called a *cache* was invented.

Caches are small amounts of fast memory which contain copies of a small subset of the data in main memory. When a processor wants to load data from memory, the cache is checked to see if the data is there. If so we get a fast load which is called a *cache hit*. If not, we get a slow load called a *cache miss*. When a cache miss occurs, a contiguous block of data containing this word of data is loaded into cache anticipating that having the data and its neighbors in cache is likely to increase the chance of a cache hit the next time.

Therefore, it is of utmost important to design algorithms that exhibit this sort of locality whenever possible.

I Parallelism

Chapter 3

Parallel Machines

****

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

As we write this in February 1996, we fear that what we are about to write may well be outdated by next year. [Looking at it in February 1997, we realize it's not so bad.] No doubt this is the most ephemeral section of our notes. The rapid pace of change is exciting to watch, but frustrating to write down. No matter what the pace of change, it is impossible to make intelligent decisions about parallel computers right now without some knowledge of their architecture. For more advanced treatment of computer architecture we recommend Kai Hwang's *Advanced Computer Architecture* and the soon to be completed *Parallel Computer Architecture* by Gupta, Singh, and Culler.

One may gauge what architectures are important today by the Top500 Supercomputer list published by Dongarra, Meuer, and Strohmaier. The secret is to learn to read between the lines. There are five kinds of machines on **The November 18, 1996 Top 500 list**:

- Distributed Memory Multicomputers
- Central Memory Symmetric Multiprocessors (SMPs)
- Networks (Arrays) of SMPs
- Vector Supercomputers
- Single Instruction Multiple Data (SIMD) Machines

It seems fair to say the last two (vector machines and SIMD machines) are on the way out (more about this later.)

How can one simplify (and maybe grossly oversimplify) the current situation? Perhaps by pointing out that the world's fastest machines are all distributed memory multicomputers, even the new machine that recently broke the teraflop barrier, but the next batch of machines are symmetric multiprocessors and arrays of SMPs. SMPs are very quickly gaining in popularity, and it is very likely that an array of SMPs will be in the top ten very soon. Perhaps it will be helpful to the reader to list some of the most important machines first sorted by type, and then by highest rank in the top 500 list.

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

Machine	Top 500 First Rank
SMP Arrays	
SGI Power Challenge Array	95
SMP	
SGI Origin 2000	211
Convex SPP1200	264
SGI Power Challenge	288
Digital AlphaServer 8400	296
Vector Machines	
Fujitsu VPP500	8
NEC SX	17
Cray YMP	54
SIMD Machines	
TMC CM-200	196

The trend is clear to anyone who looks at the list. Distributed memory machines capture the high end of the supercomputing market; the next level of supercomputers are arrays of SMPs.

Distributed Memory Multicomputers:

Remembering that a computer is a processor and memory, really a processor with cache and memory, it makes sense to call a set of such "computers" linked by a network a *multicomputer*. Figure 3.1 shows 1) a basic computer which is just a processor and memory and also 2) a fancier computer where the processor has cache, and there is auxiliary disk memory. To the right, we picture 3) a three processor multicomputer. The line on the right is meant to indicate the network.

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

> **Private versus shared memory on distributed memory machines:** It is easy to see that the simplest architecture allows each processor to address only its own memory. When a processor wants to read data from another processor's memory, the owning processor must send the data over the network to the processor that wants it. Such machines are said to have *private memory*. A close analog is that in my office, I can easily find information that is located on my desk (my memory) but I have to make a direct request via my telephone (that is, I must dial a phone number) to find information that is not in my office. And I have to hope that the other person is in his or her office, waiting for the phone to ring. In other words, I need the cooperation of the other active party (the person or processor) to be able to read or write information located in another place (the office or memory).
>
> The alternative is a machine in which every processor can directly address every memory. Such a machine is said to have a *shared address space*, and sometimes informally *shared memory*, though the latter terminology is misleading as it may easily be confused with machines where the memory is physically shared. On a shared address space machine, each processor can load data from or store data into the memory of any processor, without the active cooperation of that processor. When a processor requests memory that is not local to it, a piece of hardware intervenes and fetches the data over

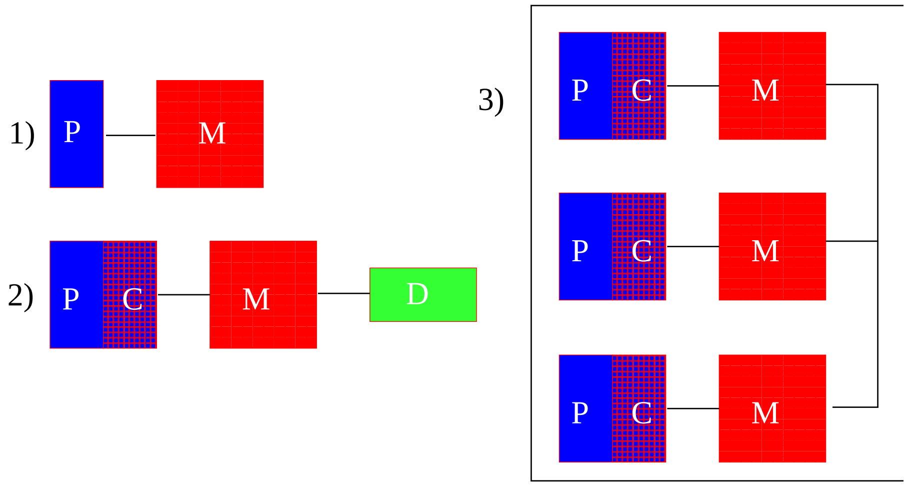

Figure 3.1: 1) A computer 2) A fancier computer 3)A multicomputer

the network. Returning to the office analogy, it would be as if I asked to view some information that happened to not be in my office, and some special assistant actually dialed the phone number for me without my even knowing about it, and got a hold of the special assistant in the other office, (and these special assistants never leave to take a coffee break or do other work) who provided the information.

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

Central Memory Symmetric Multiprocessors (SMPs):

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

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

Microprocessor machines known as *symmetric multiprocessors* (SMP) are becoming typical now as mid-sized compute servers; this seems certain to continue to be an important machine design. On these machines each processor has its own cache while the main memory is central. There is no one "front-end" or "master" processor, so that every processor looks like every other processor. This is the "symmetry." To be precise, the symmetry is that every processor has equal access to

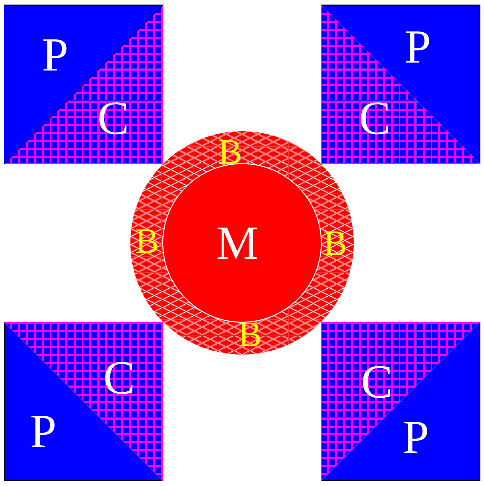

Figure 3.2: A four processor SMP (B denotes the bus between the central memory and the processor's cache

the operating system. This means, for example, that each processor can independently prompt the user for input, or read a file, or see what the mouse is doing

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

A big research issue for shared memory machines is the cache coherence problem. All fast processors today have caches. Suppose the cache can contain a copy of any memory location in the machine. Since the caches are distributed, it is possible that P2 can overwrite the value of x in P2's own cache and main memory, while P1 might not see this new updated value if P1 only looks at its own cache. Coherent caching means that when the write to x occurs, any cached copy of x will be tracked down by the hardware and invalidated – *i.e.* the copies are thrown out of their caches. Any read of x that occurs later will have to go back to its home memory location to get its new value. Maintenance of cache coherence is expensive for scalable shared memory machines. Today, only the HP Convex machine has scalable, cache coherent, shared memory. Other vendors of scalable, shared memory systems (Kendall Square Research, Evans and Sutherland, BBN) have gone out of the business. Another, Cray, makes a machine (the Cray T3E) in which the caches can only keep copies of local memory locations.

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

SIMD machines:

In the late 1980's, there were debates over SIMD versus MIMD. (Either pronounced as SIM-dee/MIM-dee or by reading the letters es-eye-em-dee/em-eye-em-dee.) These two acronyms coined by Flynn in his classification of machines refer to **Single Instruction Multiple Data** and **Multiple Instruction Multiple Data**. The second two letters, "MD" for multiple data, refer to the ability to work on more than one operand at a time. The "SI" or "MI" refer to the ability of a processor to issue instructions of its own. Most current machines are MIMD machines. They are built from microprocessors that are designed to issue instructions on their own. One might say that each processor has a brain of its own and can proceed to compute anything it likes independent of what the other processors are doing. On a SIMD machine, every processor is executing the same instruction, an add say, but it is executing on different data.

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

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

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

A close look at the SIMD argument:
In some respects, SIMD machines are faster from the communications viewpoint. They can communicate with minimal latency and very high bandwidth because the processors are always in synch. The Maspar was able to do a circular shift of a distributed array, or a broadcast, in less time than it took to do a floating point addition. So far as we are aware, no MIMD machine in 1996 has a latency as small as the 24 μsec overhead required for one hop in the 1988 CM-2 or the 8 μsec latency on the Maspar MP-2.

Admitting that certain applications are more suited to SIMD than others, we were among many who thought that SIMD machines ought to be cheaper to produce in that one need not devote so much chip real estate to the ability to issue instructions. One would not have to replicate the program in every machine's memory. And communication would be more efficient in SIMD machines. Pushing this theory, the potentially fastest machines (measured in terms of raw performance if not total flexibility) should be SIMD machines. In its day, the MP-2 was the world's most cost-effective machine, as measured by the NAS Parallel Benchmarks. These advantages, however, do not seem to have been enough to overcome the relentless, amazing, and wonderful performance gains of the "killer micros".

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

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

Vector Supercomputers:

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

Hardware Technologies and Supercomputing:

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

SMP arrays, NOWs, Heterogeneous Clusters:

Whenever anybody has a collection of machines, it is always natural to try to hook them up together. Therefore any arbitrary collection of computers can become one big distributed computer. When all the nodes are the same, we say that we have a homogeneous arrangement. It has recently become popular to form high speed connections between SMPs, known as *SMP arrays*. Any collection of workstations is likely to be networked together: this is the cheapest way for many of us to work on parallel machines given that the networks of workstations already exist where most of us work. Sometimes the nodes have different architectures creating a heterogeneous situation. Under these circumstances, it is sometimes necessary to worry about the explicit format of data as it is passed from machine to machine.

3.0.1 More on private versus shared addressing

Both forms of addressing lead to difficulties for the programmer. In a shared address system, the programmer must insure that any two processors that access the same memory location do so in the correct order: for example, processor one should not load a value from location N until processor zero has stored the appropriate value there (this is called a “true” or “flow” dependence); in another situation, it may be necessary that processor one not store a new value into location N before processor zero loads the old value (this is an “anti” dependence); finally, if multiple processors write to location N, its final value is determined by the last writer, so the order in which they write is significant (this is called “output” dependence). The fourth possibility, a load followed by another load, is called an “input” dependence, and can generally be ignored. Thus, the programmer can get incorrect code do to “data races”. Also, performance bugs due to too many accesses to the same location (the memory bank that holds a given location becomes the sequential bottleneck) are common. [1]

The big problem created by private memory is that the programmer has to distribute the data. “Where's the matrix?” becomes a key issue in building a LINPACK style library for private memory machines. And communication cost, whenever there is NUMA, is also a critical issue. It has been said that the three most important issues in parallel algorithms are “locality, locality, and locality”.[2]

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

3.0.2 Programming Model

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

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

3.0.3 Machine Topology

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

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

---

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

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

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

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

The performance and scalability of a parallel machine in turn depend on the network topology. For example, a two dimensional grid of p nodes has diameter $\sqrt{p}$, on the other hand, the diameter of a hypercube or fat tree of p nodes is $\log p$. This implies that the number of physical steps to send a message from a processor to its most distant processor is $\sqrt{p}$ and $\log p$, respectively, for 2D grid and hypercube of p processors. The node degree of a 2D grid is 4, while the degree of a hypercube is $\log p$. Another important criterion for the performance of a network topology is its *bisection bandwidth*, which is the minimum communication capacity of a set of links whose removal partitions the network into two equal halves. Assuming unit capacity of each direct link, a 2D and 3D grid of p nodes has bisection bandwidth $\sqrt{p}$ and $p^{2/3}$ respectively, while a hypercube of p nodes has bisection bandwidth $\Theta(p/\log p)$. (See FTL page 394)

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

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

3.0.4 Homogeneous and heterogeneous machines

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

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

More than 1000 so called supercomputers have been installed worldwide. In US, parallel machines have been installed and used at national research labs (Los Almos National Laboratory, Sandia National Labs, Oak Ridge National Laboratory, Lawrence Livermore National Laboratory, NASA Ames Research Center, US Naval Research Laboratory, DOE/Battis Atomic Power Laboratory, etc) supercomputing centers

(Minnesota Supercomputer Center, Urbana-Champaign NCSA, Pittsburgh Supercomputing Center, San Diego Supercomputer Center, etc) US Government, and commercial companies (Ford Motor Company, Mobil, Amoco) and major universities. Machines from different supercomputing companies look different, are priced differently, and are named differently. Here are the names and birthplaces of some of them.

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

- Cray-3 (gallium arsenide integrated circuits, multiprocessor, vector) by Cray Computer Corporation, Colorado Spring, Colorado. Now out of the business.