

Parallel Processing Concepts



Saleh Elmohamed

Video Introduction

View with [modem](#) or [broadband](#) connectionRead the [text transcript](#)

Table of Contents

1. [Overview and Goals of Parallel Processing](#)
2. [Why Use Parallel Processing to Reach These Goals?](#)
3. [Taxonomy of Architectures](#)
4. [Terminology of Parallelism](#)
5. [Models of Memory Access](#)
6. [Converting From Serial to Parallel Execution](#)
7. [Costs of Parallel Processing](#)
8. [Parallel Processing at the Theory Center](#)
9. [Parallel Programming Example](#)
10. [Conclusions](#)

[Quiz](#) ☐ [Evaluation](#) ☐ [Navigation Guide](#)

Table of Contents	1	2	3	4	5	6	7	8	9	10	Less Detail
-------------------	---	---	---	---	---	---	---	---	---	----	-------------

1. Overview and Goals of Parallel Processing

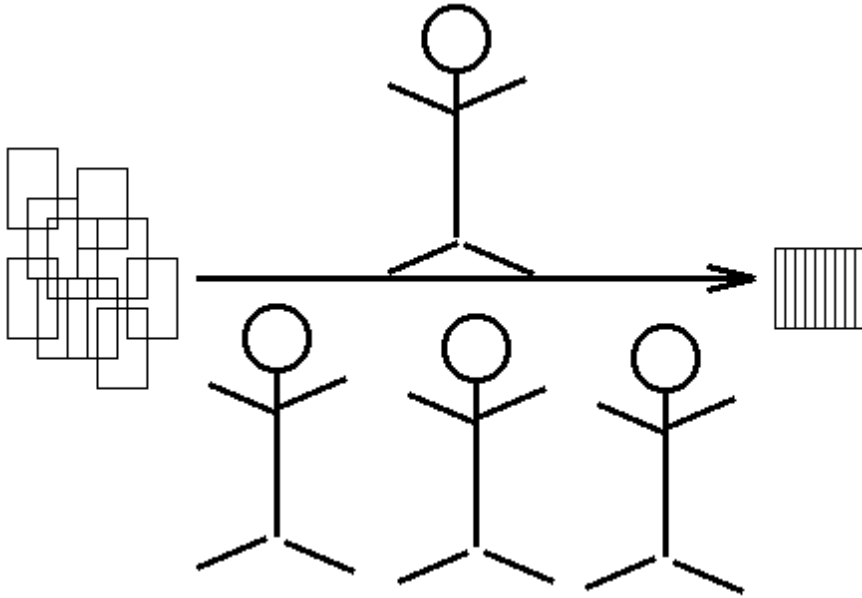
Overview of Parallel Processing

Parallel processing is the use of multiple processors to execute different parts of the same program simultaneously.

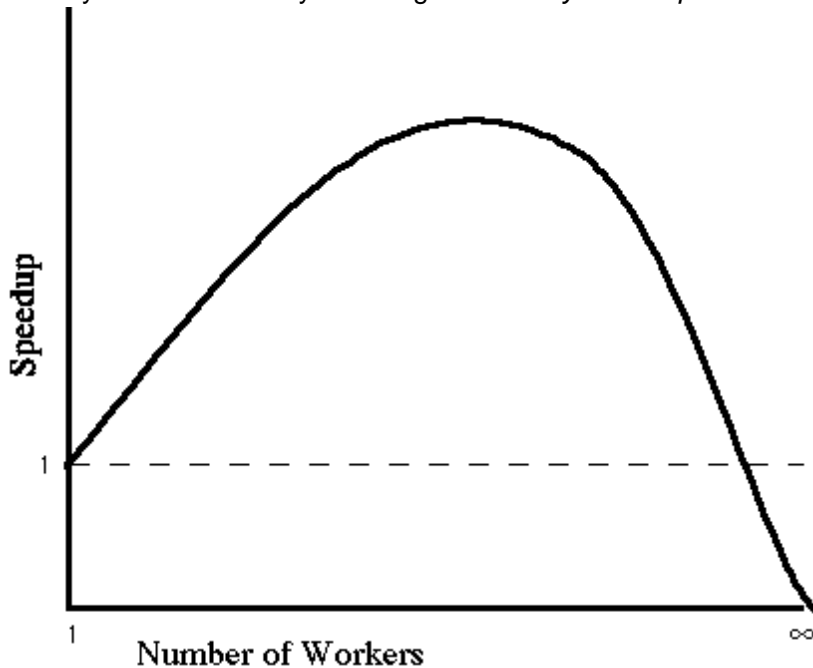
The main goal of parallel processing is to reduce wall-clock time. There are also other reasons, which will be discussed later on in this section.

Imagine yourself having to order a deck of playing cards: a typical solution would be to first order them by suit, and then by rank-order within each suit. If there were two of you doing this, you could split the deck between you and both could follow the above strategy, combining your partial solutions at the end; or one of you could sort by suit, and the other by order within

suits, both of you working simultaneously. Both of these scenarios are examples of the application of *parallel processing* to a particular task, and the reason for doing so is very simple: reduce the amount of time before achieving a solution. If you've ever played card games that use multiple decks, you've almost certainly engaged in *parallel processing* by having multiple people help with the tasks of collecting, sorting and shuffling all of the cards.



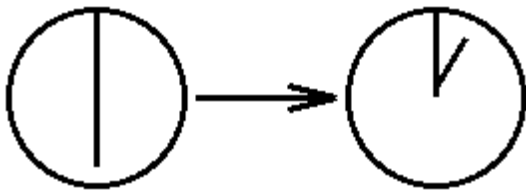
You can use this analogy to see indications of both the power and the weakness of the *parallel* approach, by taking it gradually to its extreme: as you increase the number of helpers involved in a particular task, you'll generally experience a characteristic *speedup* curve demonstrating how up-to-a-certain-number of helpers is beneficial, but any over that simply get in each others' way and reduce the overall time to completion. Consider, for example, how little it would help to have 52 people crowding around a table, each responsible for putting one particular card into its proper place in the deck -- this is exactly what is meant by the adage *Too many cooks spoil the broth*.



If you'd like to do an exercise which will drive home some of the aspects of both the power and the limitations of parallel programming, click [here](#).

1.1 Goals of Parallel Processing

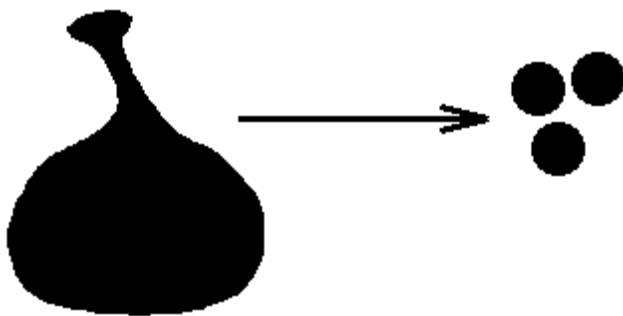
1. Reduce Wall Clock Time



In the just-stated goal, you'll notice that it isn't simply **time** that's being reduced, but **wall-clock time**; other kinds of "time" could have been emphasized, for example *CPU time*, which is a count of the exact number of CPU cycles spent working on your job, but **wall-clock** is considered to be the most significant because it's what you-the-researcher want to spend as little of as possible waiting for the solution: your own time. What is considered to be "acceptable" differs with the situation, and involves characteristics of the user, the particular code being run, and the system it's being run on. But in all cases, it is safe to assume that the user is generally going to be more pleased the faster a solution is produced.

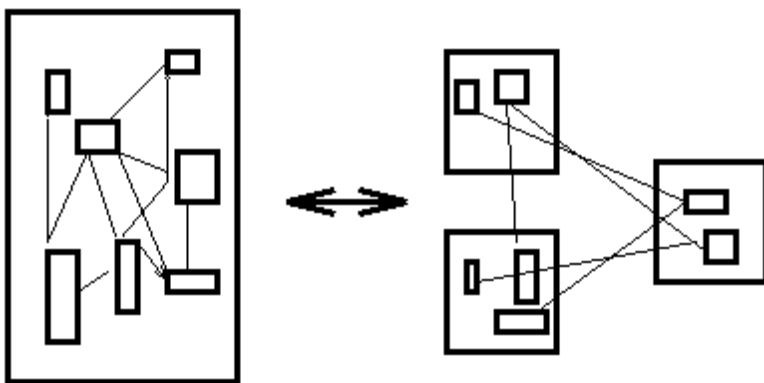
It should be pointed out that *reducing the wall-clock time to solution* is only one of many possible goals that might be of interest; some others might be:

2. Cheapest Possible Solution Strategy



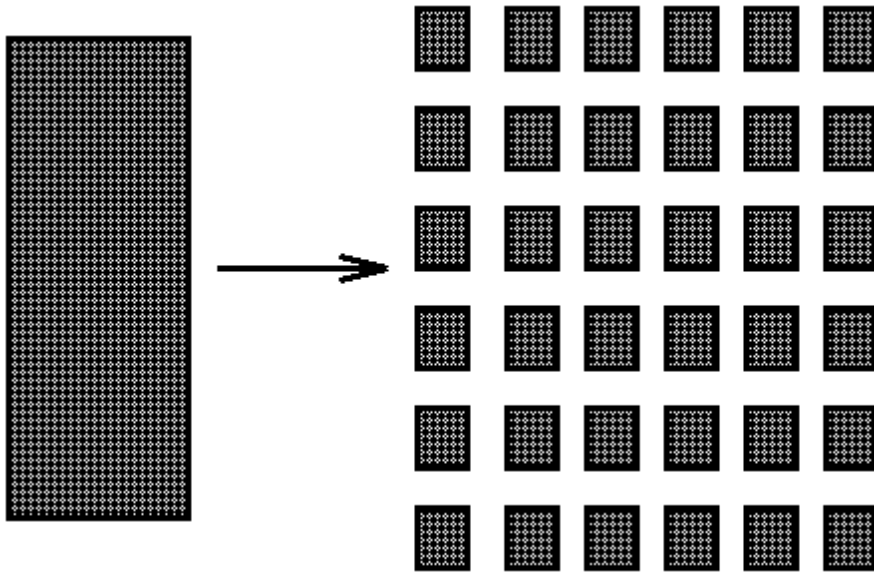
Running programs costs money; different ways of achieving the same solution could have significantly different costs. If you're in a fiscally tight situation, you may have no reasonable recourse to a parallel strategy if it costs more than your budget allows. At the same time, you may find that running your program in parallel across a large number of workstation-type computers could cost considerably less than submitting it to a large, mainframe-style mega-beast.

3. Local versus Non-Local Resources



"Locality," here, usually refers to either "geographical locality" or "political locality." The former is just another way of saying that you want all of your processes to be "close" to one another in terms of communications; the latter indicates that you only want to use resources that are administratively open to you. Both can have a bearing on "cost;" e.g., the more communications latency incurred by your application, the longer it will run, and the more you might be charged, while use of resources "owned" by other organizations may also carry charges.

4. Memory Constraints



As researchers become increasingly computationally sophisticated, the complexity of the problems they tackle increases proportionally (some would say *superlinearly*, that researchers are *forever* trying to bite off more than their computers can chew). One of the first resources to get exhausted is local memory -- especially for **Grand Challenge** level projects, the amount of memory available to a single system is rarely going to be sufficient to the computational and data-storage needs encountered during application runs.

This situation is greatly alleviated by having access to the aggregate memory made available by distributed computing environments: working storage (*main memory*) requirements can be spread around the various processors engaged in the cooperative computation, and long-term storage (tape and disk) can be accommodated at different locations (indeed, data security can be enhanced by arranging for multiple copies to be maintained at distinct locations in the distributed environment).

You can probably think of others; basically, any limited resource can be considered as the object of optimization, if it is deemed to be the most important quantity to conserve. In most cases involving large-scale computation, however, *user time*, as measured by the clock on the wall, is considered to be the single most valuable resource to be conserved.

[Table of Contents](#)
[1](#)
[2](#)
[3](#)
[4](#)
[5](#)
[6](#)
[7](#)
[8](#)
[9](#)
[10](#)
[Less Detail](#)

2. Why Use Parallel Processing to Reach These Goals?

Accepting *reduction of wall-clock time* as the fundamental goal of our activities, why necessarily focus on *parallel programming* as the means to this end? Aren't there other approaches that can also yield fast turnarounds? Yes, indeed there are, the most significant being the oldest: "beef up that old mainframe," make the standalone single-processor design larger (e.g., increase the amount of memory it can directly address) and more powerful (e.g., increase its basic wordlength and computational precision) and faster (e.g., use smaller-micron etching technology, packing more transistors into less space, and coupling everything with larger and faster communications pathways).

This approach, though, can only be pushed so far, and indications are getting stronger and stronger that fundamental limitations are going to put permanent roadblocks up before too much longer. Three of these are:

- Limits to transmission speed

The speed of light is 30 cm/nanosecond, which means that, even using strictly optical communications methods, high-speed modules must be placed relatively close to one another in order not to lose synchronization; copper wire, however, is still the most prevalent means of building communication pathways, and there the limit is 9 cm/nanosecond, making physical proximity just that much more important in a single-processor design.

- Limits to miniaturization

Processor technology is currently capable of placing millions of transistors within fingertip-sized modules, and using sub-micron (1e-6 meters) width layers in building the chips; obviously, the more units you can pack into smaller and smaller

areas, the more computational power you've achieved. However, you can only decrease the size of these components so far, even though there are now efforts directed at both molecular- and atomic-level component structures; regardless, there will be a physical component size established below which it will be impossible to assemble effective units.

- Economics limitations:

It is increasingly expensive to make a single processor faster. The most common strategies for increasing speed involve:

1. Faster Processors

Current technology is pushing towards the gigahertz range for clocks. Standard, leading edge processors today utilize 1GHz and better, and soon these processors will be the standard pieces of hardware design. But there is a price: in order for the higher-frequency clock-signals to be effective, the other parts of the processor utilizing those signals must be capable of doing significant work in the same frequency domain -- if you have an instruction unit that is capable of executing 1 instruction in a 20MHz cycle, and you boost the clock rate to 1GHz, you haven't done anything about the absolute speed of the instruction unit: it will still need the same amount of time as before, and will now issue instructions at the rate of 1 for every 50 ticks, rather than 1 for every 1 tick. In order for higher clock frequencies to translate into faster processors, the other components of the processor must also be modified to operate at a higher rate, and this process can be very, very expensive.

2. Higher-density Packaging

Just as clocks are getting faster and faster, transistors are getting smaller and smaller, and more of them than ever before can be packed together in a very small area. This has a number of benefits: making it possible to put much more functionality on a chip, greatly reducing the communication time between cooperating units, etc. But, just as with clocks, these advances come with their own costs; in the case of transistors, this generally comes from advances in etching technology, and having to deal with heat dissipation.

Etching technology refers to the manner in which the substrate (the alternating layers of conductive/non-conductive materials, upon which the transistors are fixed, and through which the communication between transistors occurs) is prepared for the matrix of transistors and "wires" connecting them. As transistor densities increase, the complexity of the etching process greatly increases, not only in terms of the fine detail that must be maintained, but also due to the huge increase in complexity in the communication network that must be accommodated.

Heat dissipation is one of the primary roadblocks encountered in the development of higher density packaging. All electronic activity produces waste heat as an unavoidable byproduct, and the more closely transistors are packed together, the more heat per unit area gets produced.

3. Thinner Substrates

Also playing a role in "higher-density packaging", decreasing the thickness of the substrate layers can have a marked effect on, among other things, the length of communication lines, this leading directly to communication speed. The thinner you make the substrates, the "closer together" the transistors are...but, by the same token, the closer they are, the more likely they'll be to generate electrical and magnetic interference and waste heat. So, in order to bring a new, thinner substrate into standard use, a great deal of research, engineering and manufacturing effort must be expended on insulators and effective heat sinks, among many other things.

For all these reasons and many more, the introduction of a new generation of processors is a very costly enterprise.

- Fairly fast processors are inexpensive.

The other side of the "evolutionary expense" coin we just flipped insists that *any recent-generation processor already on the market must already be reasonably fast ("recent-generation") and inexpensive ("on the market")*. It only stands to reason that a chip maker would put millions of dollars into the development of a new chip *only if they thought that they'd be able to generate sufficient revenue from volume sales*, and the higher the volume, the lower the cost of an individual chip. Also pushing this curve is the fact that there are a number of chip makers out there, all competing for those volume sales, and trying to reduce their prices as much as possible in order to clear their inventories. So, even if we can't have the absolutely latest, fastest, most gee-whiz chip to hit the streets, we **can** get a number of chips that are within an order of magnitude of being just as good.

- Bottom line: Use More, Relatively Inexpensive Processors in Parallel

So, we have a handful of processors one or two generations older than the hottest thing yet unveiled; we didn't pay all that much for them, and if you just added up their individual performance figures, you'd beat the numbers that are touted for the latest-and-greatest processor. But what gives you the opportunity to *realize* the potential you see in the long string of additions? **Putting all those processors in parallel.**

[Table of Contents](#) [1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#) [8](#) [9](#) [10](#) [Less Detail](#)

3. Taxonomy of Architectures

		DATA STREAM	
		Single	Multiple
INSTRUCTION STREAM	Single	Single Instruction Single Data SISD	Single Instruction Multiple Data SIMD
	Multiple	Multiple Instruction Single Data MISD	Multiple Instruction Multiple Data MIMD

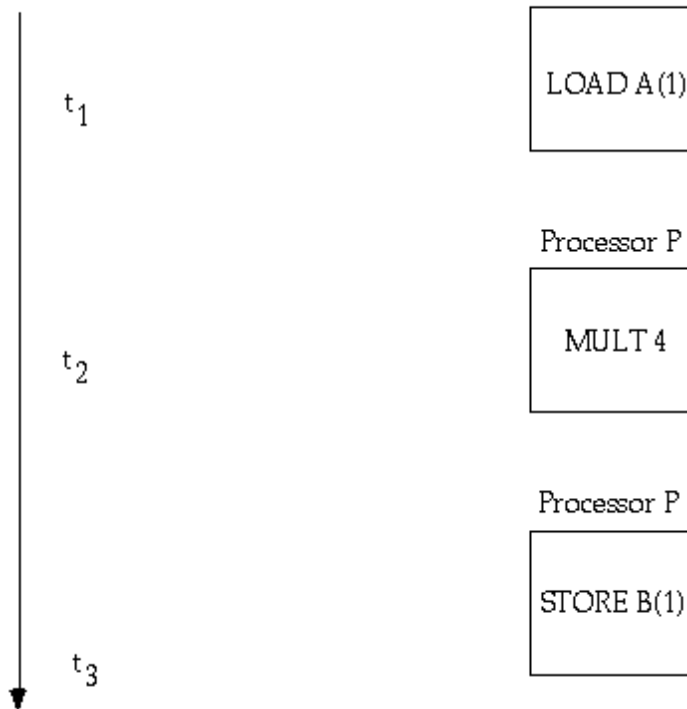
To set a foundation for our examination of parallel processing, we need to understand just what kinds of processing alternatives have already been identified, and where they fit into the "parallel picture", if you will. One of the longest-lived and still very reasonable classification schemes was proposed by Flynn, in 1966, and distinguishes computer architectures according to how they can be classified along two independent, binary-valued dimensions; *independent* simply asserts that neither of the two dimensions has any effect on the other, and *binary-valued* means that each dimension has only two possible states, as a coin has only two distinct flat sides. For computer architectures, Flynn proposed that the two dimensions be termed **Instruction** and **Data**, and that, for both of them, the two values they could take be **Single** or **Multiple**. The two dimensions could then be drawn like a matrix having two rows and two columns, and each of the four cells thus defined would characterize a unique type of computer architecture.

Single Instruction, Single Data (SISD)

$$B(I) = A(I) * 4$$

→ LOAD A(I)
MULT 4
STORE B(I)

TIME:



This is the oldest style of computer architecture, and still one of the most important: all personal computers fit within this category, as did most computers ever designed and built until fairly recently. *Single instruction* refers to the fact that there is only one instruction stream being acted on by the CPU during any one clock tick; *single data* means, analogously, that one and only one data stream is being employed as input during any one clock tick. These factors lead to two very important characteristics of **SISD** style computers:

- Serial

Instructions are executed one after the other, in lock-step; this type of sequential execution is commonly called *serial*, as opposed to *parallel*, in which multiple instructions may be processed simultaneously.

- Deterministic

Because each instruction has a unique place in the execution stream, and thus a unique time during which it and it alone is being processed, the entire execution is said to be *deterministic*, meaning that you (can potentially) know exactly what is happening at all times, and, ideally, you can exactly recreate the process, step by step, at any later time.

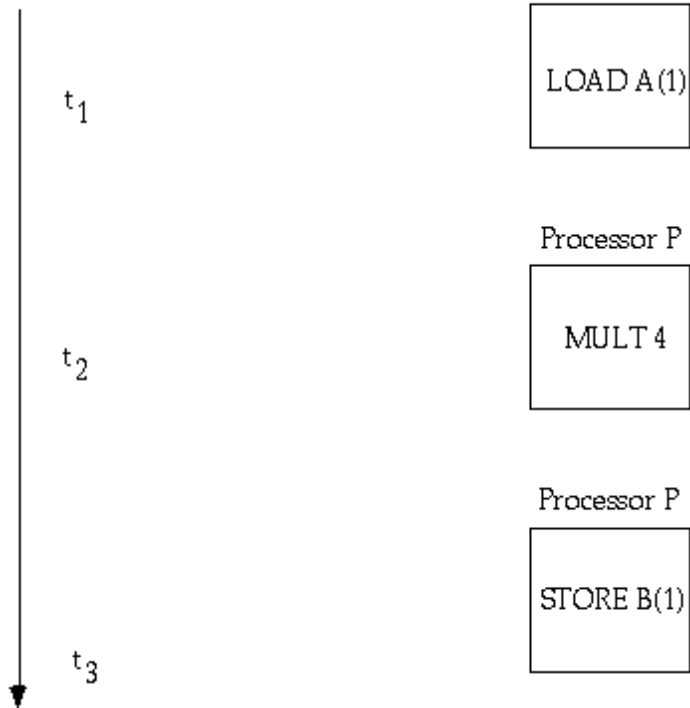
- Examples: Most non-supercomputers

Most computers commonly available today are of the **SISD** variety: all personal computers, all single-instruction-unit-CPU workstations, mini-computers, and mainframes.

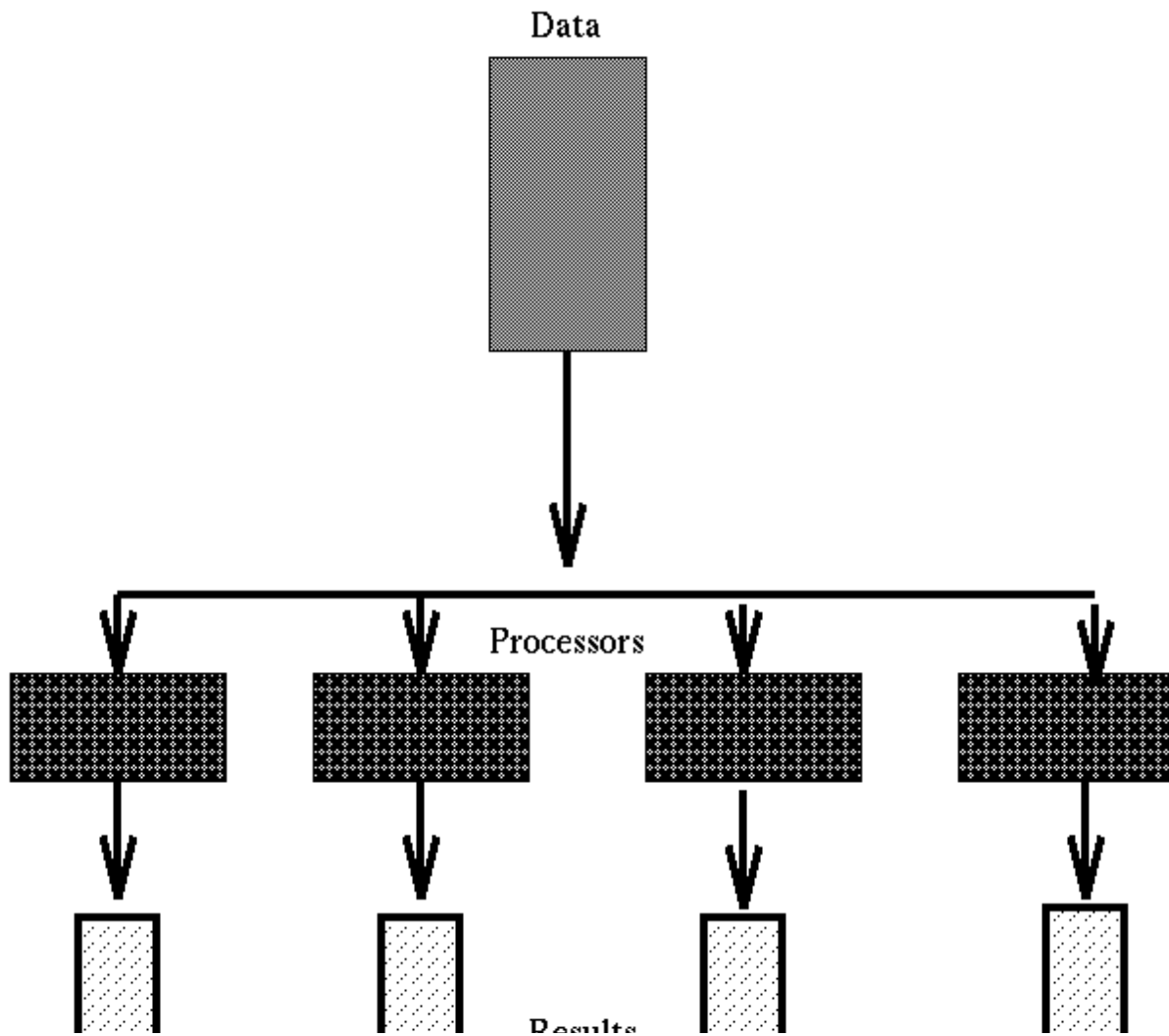
$$B(I) = A(I) * 4$$

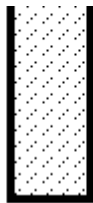
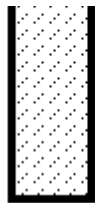
→ LOAD A(I)
MULT 4
STORE B(I)

TIME:

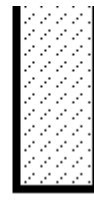
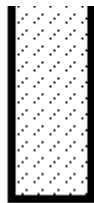


Multiple Instruction, Single Data (MISD)





Results

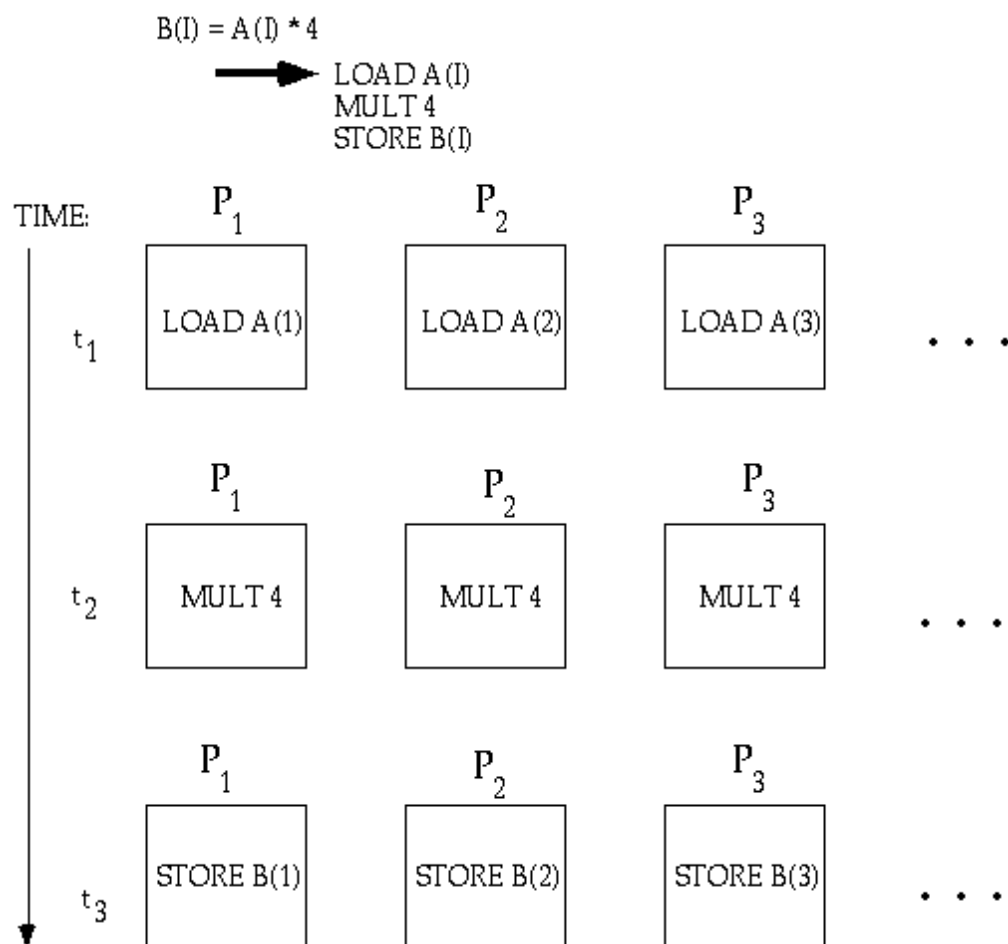


Few actual examples of computers in this class exist; this category was included more for the sake of completeness than to identify a working group of actual computer systems. However, special-purpose machines are certainly conceivable that would fit into this niche: multiple frequency filters operating on a single signal stream, or multiple cryptography algorithms attempting to crack a single coded message. Both of these are examples of this type of processing where multiple, independent instruction streams are applied simultaneously to a single data stream.

A less technological, but perhaps more cosmopolitan example, was suggested by a participant in the Cornell Theory Center's Virtual Workshop:

"I thought of another example of a MISD process that is carried out routinely at [the] United Nations. When a delegate speaks in a language of his/her choice, his speech is simultaneously translated into a number of other languages for the benefit of other delegates present. Thus the delegate's speech (a single data) is being processed by a number of translators (processors) yielding different results."

Single Instruction, Multiple Data (SIMD)



A very important class of architectures in the history of computation, *single-instruction/multiple-data* machines are capable of applying the exact same instruction stream to multiple streams of data simultaneously. For certain classes of problems, e.g., those known as *data-parallel* problems, this type of architecture is perfectly suited to achieving very high processing rates, as the data can be split into many different independent pieces, and the multiple instruction units can all operate on them at the same time.

● Synchronous (lock-step)

These types of systems are generally considered to be *synchronous*, meaning that they are built in such a way as to

guarantee that all instruction units will receive the same instruction at the same time, and thus all will potentially be able to execute the same operation simultaneously.

- Deterministic

SIMD architectures are deterministic because, at any one point in time, there is only one instruction being executed, even though multiple units may be executing it. So, every time the same program is run on the same data, using the same number of execution units, exactly the same result is guaranteed at every step in the process.

- Well-suited to instruction/operation level parallelism

The "single" in *single-instruction* doesn't mean that there's only one instruction unit, as it does in **SISD**, but rather that there's only one instruction **stream**, and this instruction stream is executed by multiple processing units on different pieces of data, all at the same time, thus achieving parallelism.

The most advanced parallel processor arrays that are in production today:

- The Cambridge Parallel Processing Gamma II Plus;

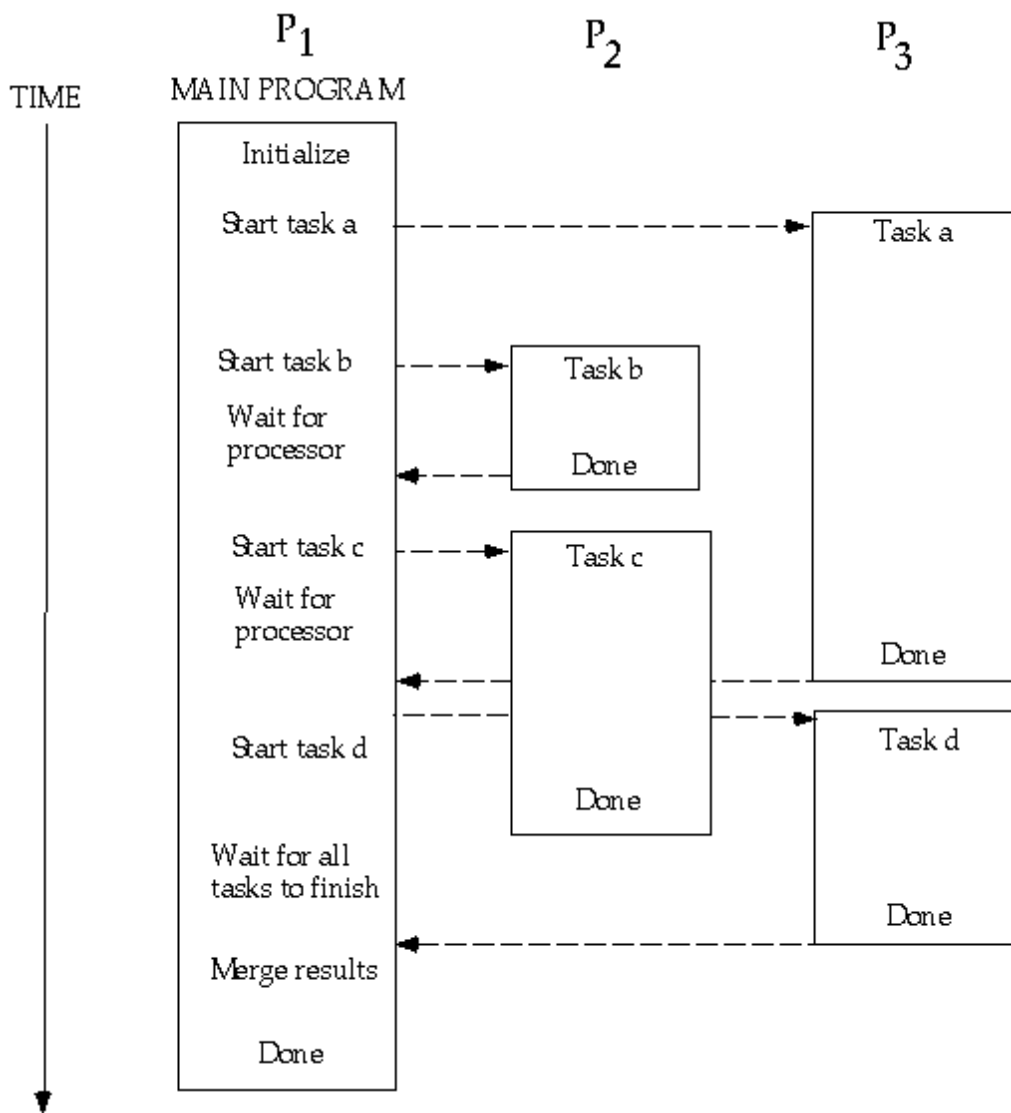
The Gamma 1000 models 1024 processors are ordered in a 32×32 array, while the Gamma 4000 has 4096 processors arranged in a 64×64 square. As in all processor-array machines, the control processor (called the Master Control Unit (MCU) in the Gamma-II) has a separate memory to hold program instructions while the data are held in the data memory associated with each Processing Element (PE) in the processor array.

- The Quadrics Apemille;

The Apemille is a commercial spin-off of the APE-1000 project of the Italian National Institute for Nuclear Physics and a successor to the APE-100 systems. The systems are available in multiples of 8 processor nodes where up to 16 boards can be fitted into one crate or in multiples of 128 nodes by adding up to 15 crates to the minimal 1-crate system. The interconnection topology of the Quadrics is a 3-D grid with interconnections to the opposite sides (so, in effect a 3-D torus). Communication is controlled by the Memory Controller and the Communication Controller which are both housed on the backplane of a crate. The TAO language has several extensions to employ the SIMD features of the Quadrics. Firstly, floating-point variables are assumed to be local to the processor that owns them, while integer variables are assumed to be global.

Note: The face of HPC is changing very quickly. While few years ago a lot of **SIMD** vector pipeline supercomputers had been produced, few of them are produced today. However some of them are still in production run. You may find [here](#) the list of major **SIMD** platform that may be still available at some supercomputing centers.

Multiple Instruction, Multiple Data (MIMD)



Many believe that the next major advances in computational capabilities will be enabled by this approach to parallelism which provides for multiple instruction streams simultaneously applied to multiple data streams. The most general of all of the major categories, a MIMD machine is capable of being programmed to operate as if it were in fact any of the four.

- Synchronous or asynchronous

MIMD instruction streams can potentially be executed either synchronously or asynchronously, i.e., either in tightly controlled lock-step or in a more loosely bound "do your own thing" mode. Some kinds of algorithms require one or the other, and different kinds of MIMD systems are better suited to one or the other; optimum efficiency depends on making sure that the system you run your code on reflects the style of synchronicity required by your code.

- Deterministic or non-deterministic

MIMD systems are potentially capable of deterministic behavior, that is, of reproducing the exact same set of processing steps every time a program is run on the same data. A number of other factors, for example, how multiple messages at a receiver are handled, go into the actual determination of this characteristic, but, if it is important for the system to be deterministic, there is nothing in the nature of MIMD parallelism that fundamentally precludes it.

- Well-suited to block, loop, or subroutine level parallelism

The more code each processor in an MIMD assembly is given domain over, the more efficiently the entire system will operate, in general. This is largely due to communications requirements, particularly synchronization, which are characteristically less stringent at levels above instruction-oriented parallelism.

- Multiple Instruction or Single Program

MIMD-style systems are capable of running in true "multiple-instruction" mode, with every processor doing something different, or every processor can be given the **same** code; this latter case is called **SPMD**, "Single Program Multiple Data", and is a generalization of **SIMD**-style parallelism, with much less strict synchronization requirements.

● MIMD Examples

The following are representative of the many different ways that MIMD parallelism can be realized:

○ Asynchronous

All of these systems implement MIMD parallelism in terms of more or less loosely coupled instruction streams:

- ❑ IBM SPx, or clusters of workstations, using PVM, MPI etc.

The nature of message-passing libraries, especially general ones applicable to many different kinds of processors, makes the resulting parallel system much more suited to coarser-grained, loosely-coupled tasks.

- ❑ Multiple vector units working on one problem (e.g., Fujitsu VPP5000)

Vector-parallel systems are very efficient at *data parallel* tasks, where each vector unit is given responsibility for computations involving one unique segment of the overall data.

- ❑ Hypercubes (e.g., nCube2S) and Meshes (e.g., Intel Paragon)

The *hypercube* is one of a whole family of network architectures that provide multiple connection points among the processors, typically allowing each processor to be directly connected to 8 or more processors. Meshes do much the same kind of thing, and come in 2- or 3-dimensional configurations, the former looking like a simple flat grid, the latter like a box.

○ Synchronous: e.g., IBM RS/6000 (up to 4 instructions per cycle), NEC SX-5.

Processors are now capable of executing multiple instructions every cycle, although not every cycle is so filled. This capability allows these processors to execute a number of related instructions simultaneously, for example, the multiplication of two floating point values already loaded into registers, the integer addition corresponding to the array location in memory of the next floating point value to be added, and the fetch of that value. This type of parallel processing, however, requires a sophisticated compiler with the ability to order instructions in such a way as to both preserve necessary instruction precedence and recognize instruction independence.

[Table of Contents](#)
[1](#)
[2](#)
[3](#)
[4](#)
[5](#)
[6](#)
[7](#)
[8](#)
[9](#)
[10](#)
[Less Detail](#)

4. Terminology of Parallelism

Parallel processing has its own lexicon of terms and phrases, emphasizing those concepts that are considered to be most important to its goals and the ways in which those goals may be achieved. The following are some of the more commonly encountered ones. They are listed in an order for you to learn them, assuming you do not know any, so you can start with the first and then build up to the rest.

Task

A logically discrete section of computational work.

This is a somewhat loose definition, but adequate for this introduction. For now, think of a task as computational work you can describe simply, such as, "calculate the mean and standard deviation of 100,000 numbers," or "calculate a Fast Fourier transform."

Parallel Tasks

Tasks whose computations are independent of each other, so that all such tasks can be performed simultaneously with correct results.

Serial Execution

Execution of a program sequentially, one statement at a time.

Parallelizable Problem

A problem that can be divided into parallel tasks. This may require changes in the code and/or the underlying algorithm.

Example of Parallelizable Problem:

Calculate the potential energy for each of several thousand independent conformations of a molecule; when done, find the minimum energy conformation

The above is an example of a problem that is amenable to parallelization; each of the conformations is independently determinable, while the calculation of the minimum such conformation is itself a *parallelizable problem*.

Example of a Non-parallelizable Problem:

Calculation of the Fibonacci series (1,1,2,3,5,8,13,21,...) by use of the formula:

$$F(k + 2) = F(k + 1) + F(k)$$

A *non-parallelizable problem*, such as the calculation of the Fibonacci sequence above, would entail **dependent** calculations rather than **independent** ones -- notice how calculation of the $k + 2$ value uses those of both $k + 1$ and k , hence those three terms **cannot** be calculated independently, nor, therefore, in parallel.

Types of Parallelism

There are two basic ways to partition computational work among parallel tasks:

- **Data parallelism**: each task performs the same series of calculations, but applies them to different data. For example, four processors can search census data looking for people above a certain income; each processor does the exact same operations, but works on different parts of the database. You can read more on data parallel computing [here](#).
 - **Functional parallelism**: each task performs different calculations, i.e., carries out different functions of the overall problem. This can be on the same data or different data. For example, 5 processors can model an ecosystem, with each processor simulating a different level of the food chain (plants, herbivores, carnivores, scavengers, and decomposers).
-

Observed Speedup

Observed speedup of a code which has been parallelized =

$$\frac{\text{wall-clock time of serial execution}}{\text{wall-clock time of parallel execution}}$$

One of the most widely used indicators of parallelizability, the calculation of **observed speedup** is both intuitively satisfying, and potentially misleading; the former because a well-parallelized code **can** be shown to run in a fraction of the time that it takes the serial version, the latter because, in many respects, this is a comparison of apples and oranges: the codes are different, they perform different tasks, the algorithms may be entirely distinct. Still, there is no discounting the fact that a good job of parallelization will be evident in the amount of wallclock time it has saved the user; what is debatable is the converse: if it is **not** evident that a lot of time has been saved, is it because the problem itself is not parallelizable, or because the parallelization simply wasn't done well? This, by the way, is where parallel profiling tools, covered later in this presentation, can help tremendously.

Synchronization

The temporal coordination of parallel tasks. It involves waiting until two or more tasks reach a specified point (a sync point) before continuing any of the tasks.

- Synchronization is needed to coordinate information exchange among tasks; e.g., the previous example finding minimum energy conformation: all of the conformations had to be completed before the minimum could be found, so any task that was dependent upon finding that minimum would have had to wait until it was found before continuing.
 - Synchronization can consume wall-clock time because processor(s) sit idle waiting for tasks on other processors to complete.
 - Synchronization can be a major factor in decreasing parallel speedup, because, as the previous point illustrates, the time spent waiting could have been spent in useful calculation, were synchronization not necessary.
-

Parallel Overhead

The amount of time required to coordinate parallel tasks, as opposed to doing useful work.

Parallelization doesn't come free, and one of the most insidious costs is the time and cycles put into making sure that all of those separate tasks are doing what they're supposed to be doing. Things that are simply taken for granted in serial execution, or that don't apply, take on special significance when there are many tasks instead of just one; the three most commonly encountered coordination tasks are:

- Time to start a task

This involves, among other things:

1. identifying the task
2. locating a processor to run it
3. loading the task onto the processor
4. putting whatever data the task needs onto the processor
5. actually starting the task

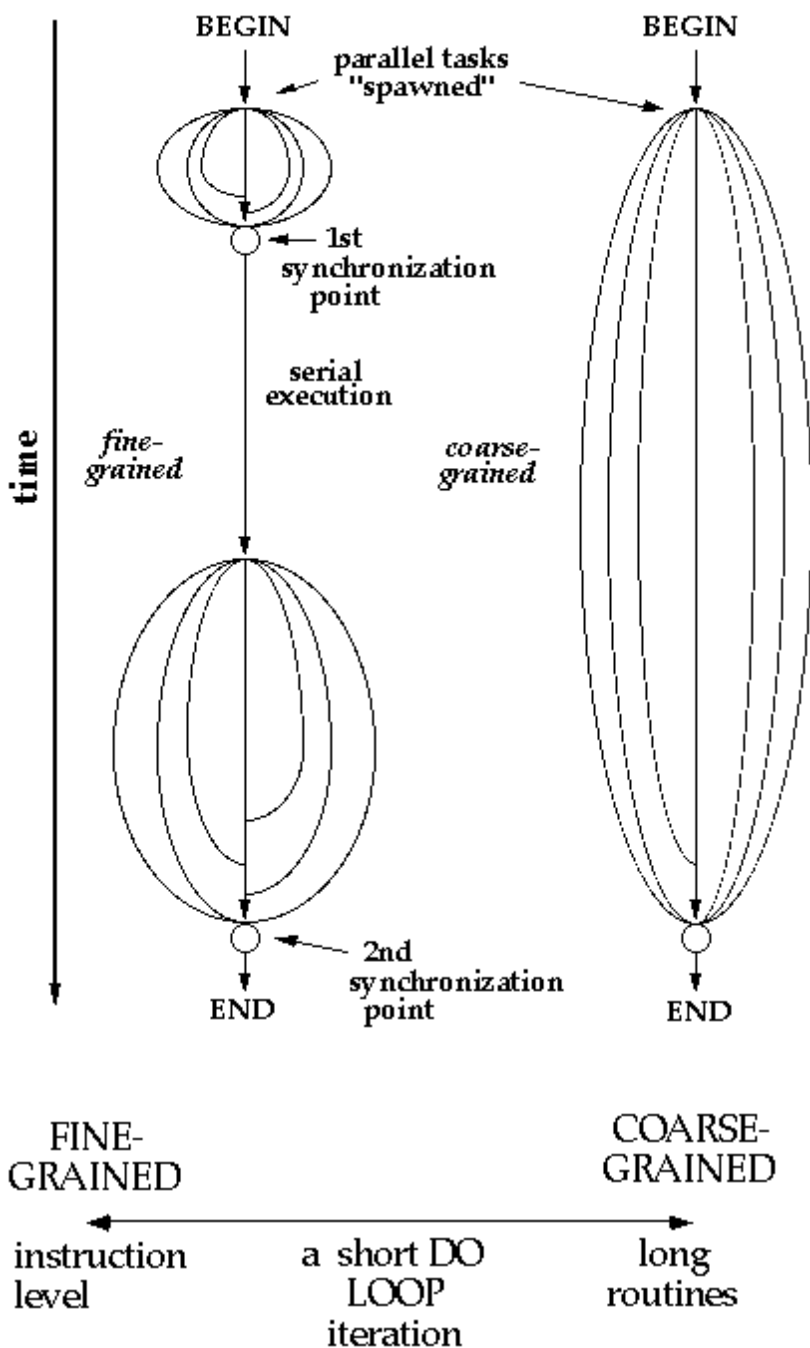
- Time to terminate a task

Termination isn't a simple chore, either: at the very least, results have to be combined or transferred, and operating system resources have to be freed before the processor can be used for other tasks.

- Synchronization time, as previously explained.
-

Granularity

A measure of the ratio of the amount of computation done in a parallel task to the amount of communication.



- Scale of granularity ranges from fine-grained (very little computation per communication-byte) to coarse-grained (extensive computation per communication-byte).
- The finer the granularity, the greater the limitation on speedup, due to the amount of synchronization needed. Remember from the very beginning of this module about considering how hard it would be to coordinate the activities of 52 people, all trying to help sort a single deck of cards?

Massively parallel system

A parallel system with many processors. "Many" is usually defined as 1000 or more processors.

Scalable parallel system

A parallel system to which the addition of more processors will yield a proportionate increase in parallel speedup. Whether or not this increase occurs typically depends on some combination of:

- Hardware

One of the most significant bottlenecks to scalability lies in the capability of the communications network -- adding more processors to a network that is already filled to capacity will **not** yield the expected increase in speedup, and could in

fact do just the reverse.

● Parallel Algorithm

Some algorithms are better suited to parallelism (i.e., more *scalable*) than others, and there are even economies of scale within parallel ones, i.e., algorithms that work well at certain scales of parallelism work poorly at higher scales, and vice versa. Generally speaking, *scalable* implies $O(n)$ growth ("on the order of n "; i.e., *linear*) with data-size n , and preferably the growth should be $O(\log n)$.

It should be emphasized here that great care should be taken to match the algorithm with the actual problem, and both of these with the actual size of the machine on which the problem will be attacked using that particular algorithm. A particular algorithm may appear to work quite nicely on the given problem when run on a small number of nodes, or on a part of the problem over the entire target machine, but when all three production versions are brought together, you may find that your initial tests were in fact misleading, and what worked well at a small scale works not at all at the desired scale.

● Your actual code

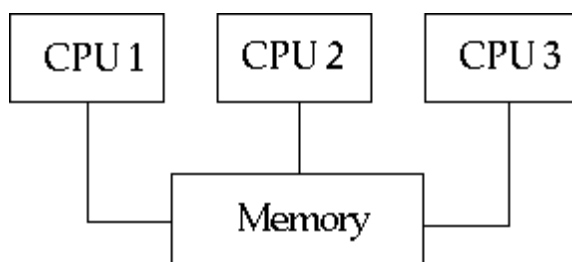
Even if you use an algorithm well suited to your purposes, the way you implement it can determine how much parallelism is actually expressed in your application. Easy ways to negate the usefulness of a good algorithm include using unsuitable data objects, or failing to take into consideration how your programming language structures things like multi-dimensional arrays in order to maximize the efficiency of addressing natural blocks of data.

Table of Contents 1 2 3 4 5 6 7 8 9 10 Less Detail

5. Models of Memory Access

Memory access refers to the way in which the working storage, be it "main-memory", "cache-memory", or whatever, is viewed by the programmer. Regardless of how the memory is actually implemented, e.g., if it's actually remotely located but is accessed as if it were local, the *access method* plays a very large role in determining the conceptualization of the relationship of the program to its data.

5.1 Shared Memory



Think of a single large blackboard, marked off so that all data elements have their own unique locations assigned, and all the members of a programming team are working together to test out a particular algorithmic design, all at the same time...this is an example of *shared memory* in action:

● The same memory is accessible to multiple processors

All processors associated with the same shared memory structure access the exact same storage, just as all the programmers in the above example used the same unique data-element location on the blackboard to record any changes in those values.

● Synchronization is achieved by tasks' reading from and writing to the shared memory.

In just the same way that the programmers would have to take turns writing into the blackboard locations, so the processors have to take turns accessing the shared memory cells. This makes it easy to implement *synchronization* among all of the tasks, by simply coding them all to watch particular locations in the shared memory, and not do anything until certain values appear...of course, you also have to arrange for those values **to** appear.

● A shared memory location must not be changed by one task while another, concurrent task is accessing it.

If one programmer is trying to use a value from the blackboard to calculate some other value, and sees another programmer begin to write over the one being copied, screams and shouts and thrown chalk and erasers can keep the needed value from being overwritten until it's no longer needed. Processors use more polite means of achieving the same ends, sometimes called *guards* or *spin-locks*: these are shared variables associated with the location in question, and a task can be programmed not to change the location before first gaining sole ownership of the *guard*; if all tasks have been programmed so that sole ownership of the *guard* is required before either reading or writing the associated location, this guarantees that no task will be attempting to read while another is busy changing that same value.

- Data sharing among tasks is fast (speed of memory access)

One of the most attractive features of shared memory, besides its conceptual simplicity, is that the time to communicate among the tasks is effectively a factor of a single fixed value, that being "the time it takes a single task to read a single location." There are, of course, limitations to this sharing...

- Disadvantage: scalability is limited by number of access pathways to memory

If you have more tasks than connections to memory, you have contention for access to the desired locations, and this amounts to increased latencies while all tasks obtain the required values. So the degree to which you can effectively scale a shared memory system is limited by the characteristics of the communication network coupling the processors to the memory units. See the [Dining Philosophers problem](#) for a discussion of this.

- User is responsible for specifying synchronization, e.g., locks

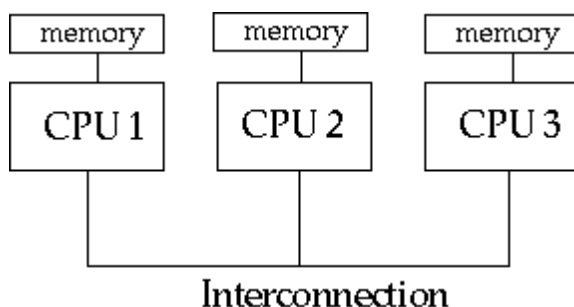
Any time resources are shared among multiple parties, whether these parties be human, canine, insect or computer, there will have to be some form of control imposed. In the case of shared memory, this "control" takes the form of different ways in which synchronization is established and enforced. *Spin-locks*, as previously described, are one common form of enforcing synchronization; others include *barriers*, *mutexes*, and system-specific entities. What is common to all of them is that their use is all under the control, and hence the responsibility, of the user.

- This model is often referred to as SMP, for Symmetric MultiProcessors, because a common implementation is for several processors of the same type ("symmetric") to access the same shared memory.
- Examples

A number of commonly encountered multi-processor systems implement a shared-memory programming model; examples include:

- NEC SX-5;
- SGI Power Onyx/ Origin 2000;
- Hewlett-Packard V2600/HyperPlex;
- SUN HPC 10000 400 MHz ;
- DELL PowerEdge 8450.
- [Cache coherence and memory consistency](#) are two of the issues that need to be understood when dealing with parallel computing.

5.2 Distributed Memory



The other major distinctive model of memory access is termed *distributed*, for a very good reason:

- Memory is physically distributed among processors; each local memory is directly accessible only by its processor.

Just as you're used to when buying a plain computer, each component of a *distributed memory* parallel system is, in most cases, a self-contained environment, capable of acting independently of all other processors in the system. But in order to achieve the true benefits of this system, of course there must be a way for all of the processors to act in concert, which means "control"...

- Synchronization is achieved by moving data (even if it's just the message itself) between processors (communication).

The only link among these distributed processors is the traffic along the communications network that couples them; therefore, any "control" must take the form of data moving along that network to the processors. This is not all that different from the shared-memory case, in that you still have control information flowing back to processors, but now it's from other processors instead of from a central memory store.

- A major concern is data decomposition -- how to divide arrays among local CPUs to minimize communication

Here is a major distinction between shared- and distributed-memory: in the former, the processors don't need to worry about communicating with their peers, only with the central memory, while in the latter there really isn't anything **but** the processors. A single large regular data structure, such as an array, can be left intact within shared-memory, and each cooperating processor simply told which ranges of indices are its to deal with; for the distributed case, once the decision as to index-ranges has been made, the data structure has to be *decomposed*, i.e., the data within a given set of ranges assigned to a particular processor must be physically *sent* to that processor in order for the processing to be done, and then any results must be *sent back* to whichever processor has responsibility for coordinating the final result. And, to make matters even *more* interesting, it's very common in these types of cases for the **boundary values**, the values along each "outer" side of each section, to be relevant to the processor which *shares* that boundary.

Try One

Play with an exercise intended to demonstrate the differences in the ways these two models deal with *boundary values*.

5.3 Distributed Memory: Some Approaches

Distributed memory is, for all intents and purposes, virtually synonymous with *message-passing*, although the actual characteristics of the particular communication schemes used by different systems may hide that fact.

- [Message-passing](#) approach: tasks communicate by sending data packets to each other

Messages are discrete units of information, *discrete* meaning that they have a definite identity, and can be distinguished from all other messages ... well, that's the theory, at least. In practice, one of the most common programming errors is to forget to actually **make** the messages distinctly different, by giving them unique identifiers or *tags*. Regardless, parallel tasks use these messages to send information and requests for same to their peers.

- Overhead is proportional to size and number of packets (more communication means greater costs; sending data is slower than accessing shared memory.)

Message-passing isn't cheap: every one of those messages has to be individually constructed, addressed, sent, delivered, and read, all before the information it contains can be acted upon. Obviously, then, the more messages being sent, the more time and cycles spent in servicing message-oriented duties, and the less spent on the actual tasks that the messages are supposed to be subservient to. It is also clear from this portrayal that, in the general case, *message-passing* will take more time and effort than *shared-memory*.

Having said that, it should be pointed out that shared memory scales less well than message passing, and, once past its maximum effective bandwidth utilization, the latency associated with message-passing may actually be lower than that encountered on an over-extended shared memory communications network.

- Effective message-passing schemes overlap calculations & message-passing.

There are ways to decrease the overhead associated with message-passing, the most significant being to somehow arrange to do as much valuable computation as possible while communication is occurring. The most easily conceived method of doing this is to have two completely separate processors, each dedicated to either

computation or communication, and coupled via a dual-ported DMA (*direct memory access*) in order to cooperate. This is something of the nature of *shared-memory* being put in the service of *distributed-memory*, and **does** require a multi-processor configuration for a single entity in the distributed system.

Other schemes involve time-slicing between the two tasks, or *active-waiting* where a processor waiting for a communications event, such as receipt of an awaited message or acknowledgment of delivery of a sent message, arranges for a preemptive signal to be generated when the event occurs, and then goes off and does independent computation. These alternatives require considerably more sophistication in the control programs than simply sitting and twiddling one's thumbs until the communication process completes, but can be made to be very effective.

○ Examples:

The following are few examples of common types of *message-passing* systems:

- ❑ **MPP** ([ASCI White](#), [ASCI Red](#), [ASCI Blue-Pacific](#)), Clusters of Workstations / [PCs](#);

Utilizing high bandwidth switch networking architecture, the nodes within such systems use a special very low-level message-passing library for the lowest-level (i.e., most efficient, fastest) mechanism for communicating with one another. On top of this is built the **MPI /PVM** message-passing facility.

- ❑ hypercube machines (e.g., nCube2S)

Hypercube architectures typically utilize off-the-shelf processors coupled via proprietary networks (both hardware and message-passing software); the processor, as is the case in the above example, can also be proprietary, and often is when the decision is to trade price for performance, especially communications performance.

- **ccNUMA** (**C**ache **C**oherent **N**on-**U**niform **M**emory **A**ccess) machines : systems that have a rather small (up to 16) number of RISC processors that are tightly integrated in a cluster, a Symmetric Multi-Processing (SMP) node (SGI Origin 2000). The processors in such a node are virtually always connected by a 1-stage crossbar while these clusters are connected by a less costly network.

ccNUMA approach provide the programmer with a shared-memory programming model, even though the actual design of the hardware and the network was distributed. A very sophisticated address-mapping scheme insured that the entire distributed memory space could be uniquely represented as a single shared resource. The actual communication model utilized at the lowest level, however, was in fact "message-passing", but that "lowest level" was not accessible to the programmer.

Table of Contents	1	2	3	4	5	6	7	8	9	10	Less Detail
-------------------	---	---	---	---	---	---	---	---	---	----	-------------

6. Converting From Serial to Parallel Execution

There are a number of bottlenecks typically encountered in the transition from serial processing to parallel processing. One of the most pernicious is that of **mindset**: people who have been in the computing business for a long time are understandably reluctant to have to learn a new way of designing their codes, and an efficient parallel algorithm often has little similarity to an efficient serial algorithm. The very first task in the conversion effort is to step way back from the existing serial application, and re-examine the **intent** it was written to serve: *can this task be effectively and efficiently performed in parallel, and, if so, how best can that be accomplished?*

Very often existing serial code has to be almost completely ignored, and the parallel version written virtually from scratch. This can be a major commitment of resources, and for some dusty-deck codes the projected return from such an investment is often considered to be insufficient to warrant the effort.

However, once the decision has been made to move from serial to parallel, the real nitty-gritty work of code conversion can very often be helped along by application of the growing number of automatic tools, well seasoned by the manual use of hard-learned rules of thumb.

6.1 Automatic vs. Manual Conversion

For years, ever since the first parallel system was constructed, the parallelization of existing codes has been largely the realm of manual conversion. The ultimate future goal of parallel support is to build tools capable of accepting the before-mentioned *dusty-deck* serial code, and returning a perfectly parallelized program suitable for execution on a particular state-of-the-art parallel system.

○ Automatic parallelization

SURPRISE!!! We're not there yet ... but we're a lot closer now than we were just a few years ago, and the effort is gaining a lot of momentum. A number of significant factors affecting parallelization and effective speedup have been identified and automated in new compilers,

- ❑ Many compilers attempt to do some automatic parallelization, especially of DO-loops

As will be shown in more detail later, **DO-loops** are natural candidates for parallelization ... just seeing one doesn't guarantee that you'll be able to parallelize, obviously, but they're a clear marker of where to start looking. Modern parallel compilers immediately examine DO-loops for the appropriate characteristics, basically *independence* of one type or another, and do whatever they can to take advantage of what is often called "natural parallelism" exhibited by the code.

- ❑ Significant speedups usually require going beyond automatic level

But, just as indicated, things aren't as automatic yet as we'd like them to be. There are still too many interlocking factors, too complex to be captured in code, that require even the best automatic tools to be used in conjunction with well-trained conversion experts. At the very minimum, it is usually necessary to alter the code so as to expose parallelism to the compiler.

○ Manual parallelization

The bulk of parallelization is still the realm of the human programmer, and this necessary resource cannot be emphasized too strongly:

- ❑ Programmer must spend time to parallelize

Expect this to be a time-consuming process, and budget for it ...if you expect to simply have to change a few lines, then you've likely got a nasty shock in store.

- ❑ Some possible actions ("Rules of thumb")

Over the years, a body of hard-learned wisdom has grown regarding how one can most efficiently extract parallelism from serial seeds; here are a few:

- ❑ Remove inhibitors to parallelization

Unnecessary serialization, for example, making all processes wait while one of them does something that could have been put off until a later required serial section.

Re-arranging of loop-indices, to minimize inter-loop dependency.

- ❑ Insert constructs or calls to library routines

Some packages of often-used algorithms, for example, linear algebra routines, have already been parallelized. If your application has a need for such tasks, use someone else's work if at all possible.

Constructs are commented-out keywords that are intended to be read by pre-processors and code-generators, allowing the programmer to indicate what kinds of parallelism should be attempted in certain parts of the program. Sometimes **directives**, which insist that things be done a certain way, and sometimes **suggestions**, indicating potentially useful directions, *constructs* often make it possible for the programmer to leave a section of serial code completely alone and still have it parallelized because of the actions taken upon recognition of the commented-out information.

- ❑ Run code through preprocessors

When *parallel constructs*, such as explained above, are used, it is typically possible to have the

resulting parallel code displayed rather than simply compiled. Being able to look at what has been done by automatic means, as reflected in the modified source code, gives the programmer the opportunity both to learn how parallelism can be implemented, and to check a particular case against human knowledge and intuition.

❑ Restructure algorithm

Always be willing to re-examine the design upon which your application is based -- sometimes a simple change, such as moving a calculation outside of a loop, can have dramatic effects on parallelization.

❑ [Software tools](#) that are available to assist at CTC

- ❑ ClusterCoNTroller Batch System ;
- ❑ MPI/Pro software;
- ❑ C/C++ and Fortran Compilers;
- ❑ Cornell Multitask Toolbox for Matlab;
- ❑ Other parallel libraries;

As indicated earlier, there is a small but growing number of software tools focused on providing assistance in the parallelization effort. These are language-dependent (mostly oriented towards Fortran), and still limited in their scope, but they can be very useful when appropriately applied. Here, "Parallel Compiler" means a compiler that understands parallel constructs and/or automatically extracts parallelism.

More information on these tools can be found in later tutorials.

6.2 Converting Serial Code to Parallel: General Dependencies

In this section, we'll spend some time discussing a major factor relevant to successful code parallelization: dependency.

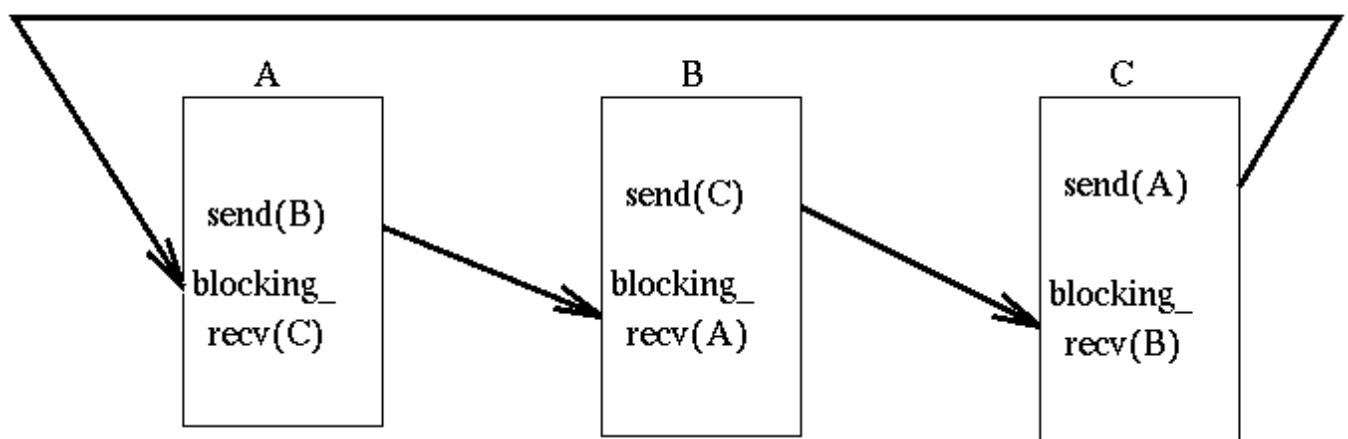
General Dependencies

- Definition: A **DEPENDENCE** exists between statements when the order of the statements' execution affects the results of the program. A **DATA DEPENDENCE** results from multiple use of the same location(s) in storage.

As an example of *data dependence*, consider the following: $a=b+1$; $c=4-a$; $b=2a-c$;

Here we have three statements all of which contain *data dependencies*, and which are *execution dependent* as well.

Communication Dependence



For example, if a group of parallel tasks post *sends* to one another before executing *blocking receives*, they all will likely (assuming no other problems in the code) obtain their expected data; if however, they all do *blocking receives* before their *sends*, then none of them will receive their data, as none of them will be able to get past the *receive* in order to *send* ... this demonstrates a **dependence** between the *sends* and *receives*.

- The concepts of dependence in parallel processing are the same as those in vector processing; the consequences of certain dependencies, however, are different.

This has to do with how the two different forms of concurrency, *parallelism* and *vectorization*, view dependencies in multiple-

loop situations, i.e., when you have one loop within another. Without going into a great deal of detail, let's leave it with the following:

- ❑ *parallel* concurrency prefers to have any *loop-carried dependencies* (see below) at the inner-most loop, because that means that the outer-most one can be used for parallelization, while...
 - ❑ *vectorized* concurrency prefers just the opposite, so that there are no inconsistencies within each long string of vectors being operated on simultaneously.
- Dependencies are acceptable within parallel tasks but not between parallel tasks. Unavoidable dependencies between tasks require synchronization.

Within a parallel task, things are being executed serially, and the rules concerning dependencies are well-understood; dependencies **between** parallel tasks are major sources of erroneous execution, due to the inability to guarantee a particular time sequence of execution. Therefore, whenever a dependency is discovered between parallel tasks, the safest way to handle it is to force a synchronization between the involved tasks so that you know exactly where each one is in its execution stream, and **then** allow them to go on into the dependent code.

6.3 DO-LOOP Dependencies

Since *DO-loops* are prime candidates for very effective parallelization, identification of dependence is very important, even more so the determination of whether or not the dependence found will render the loop incapable of the desired parallelization.

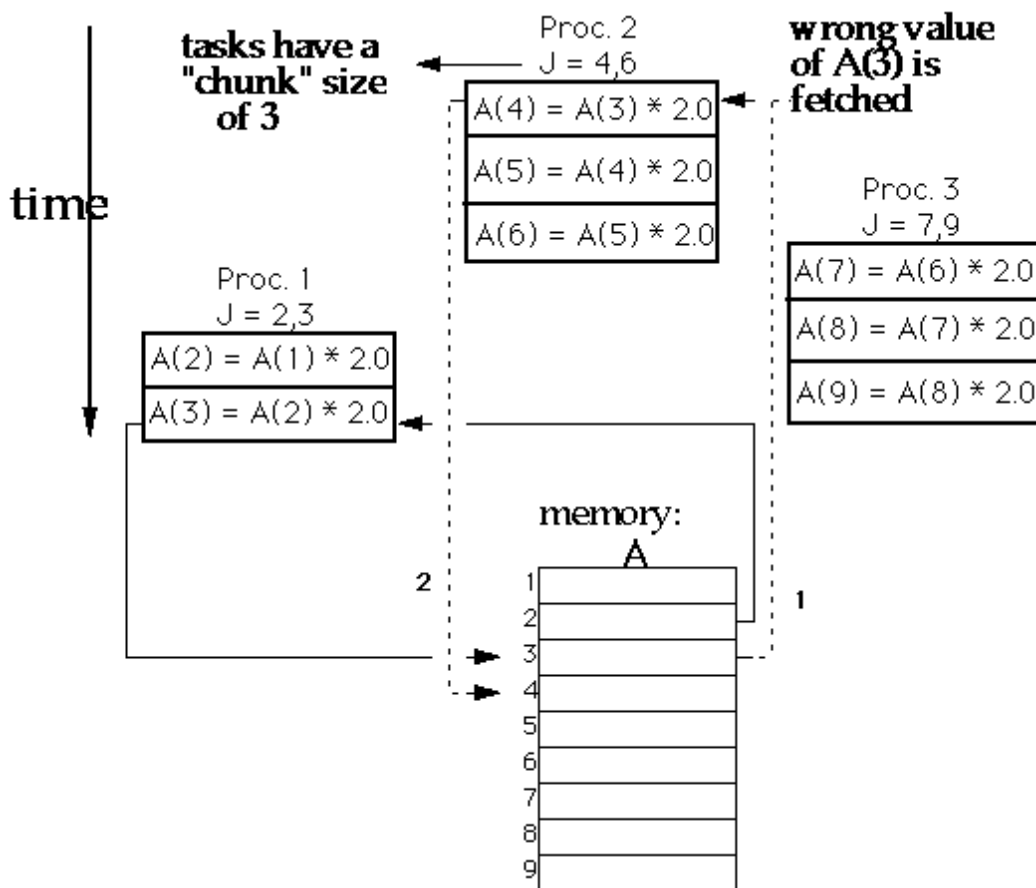
- A **LOOP-CARRIED DEPENDENCE** exists when a storage location is used by a statement or statements, with at least one write, during different iterations of the loop.

This *cannot parallelize*, because the order of execution of the loop's iterations is important for correct results.

The important point, here, is that *iteration-based* parallelization, where different execution-streams are going to be responsible for different portions of the iteration-space of the loop, are going to wind up using incorrect values for the overlapping variable.

Here's a piece of code demonstrating this:

```
DO 500 J = 2,9
  A(J) = A(J-1) * 2.0
500 CONTINUE
```



- A **LOOP-INDEPENDENT DEPENDENCE** exists when a storage location is used by successive statements within an iteration, but not by different iterations.

This *can parallelize*, because loop iterations can be executed independently and asynchronously. As all references to the same location are being executed by the same instruction stream, hence serially, there is no possibility for untimely access.

7. Costs of Parallel Processing

By this point, I hope you will have gotten the joint message that:

1. Parallel processing can be extremely useful, but...
2. ... TANSTAAFL (There Ain't No Such Thing As A Free Lunch)

I.e., you can get great rewards from parallelizing, but you'll likely sweat blood getting there; now, that's not always the case, but it's better that you assume it will be, and be pleasantly surprised when it goes quickly and smoothly, than expecting everything will go smoothly and ending up mired to your neck in problems.

Here are some of the more significant ways that you can expect to spend time and encounter problems:

- Programmer's time

As the programmer, your time is largely going to be spent doing the following:

- ❑ Analyzing code for parallelism

The more significant parallelism you can find, not simply in the existing code, but even more importantly in the overall task that the code is intended to address, the more speedup you can expect to obtain for your efforts.

- ❑ Recoding

Having discovered the places where you think parallelism will give results, you now have to put it in. This can be a very time-consuming process.

○ Complicated debugging

One of the nice things about serial code is that, in the end, there's only one instruction at a time being executed, and you could, if you had to, get an instruction-level dump of the whole thing and stand a good chance of finding that last, elusive bug. Debugging a *parallel* application is at least an order of magnitude more infuriating, because you not only have multiple instruction streams running around doing things at the same time, you've also got information flowing amongst them all, again all at the same time, and **who knows!?!** what's causing the errors you're seeing?

It really is that bad. Trust me.

Do whatever you can to avoid having to debug parallel code:

- ☐ consider a career change;
- ☐ hire someone else to do it;
- ☐ or *write the best, self-debugging, modular and error-correcting code you possibly can, the first time.*

If you decide to stick with it, and follow the advice in that last point, you'll find that the time you put into writing **good, well-designed** code has a tremendous impact on how quickly you get it running correctly. Pay the price up front.

○ Loss of portability of code

Serial code is serial code; sure, different serial machines have different dialects of your favorite serial language, but standards committees and software developers' need for portability are forcing a very welcome commonality in most of them.

But, when you convert your code to parallel, there ain't no goin' back -- what you end up with (assuming you are **not** using parallel constructs hidden inside comments) will never again run on a serial machine ... well, that's not entirely true, in all cases, but it's better that you expect to have to support two completely different packages, one for serial environments, and one **each** for every different kind of parallel environment you want your application to be able to run on.

Things **are** getting a little brighter on this score, however: there are standards efforts underway (at least for Fortran) to insure portability of parallel programs. For example, *High Performance Fortran (HPF)* runs on a variety of platforms, and the entire *MPI* effort was directed at being able to provide message-passing portability on top of a wide range of underlying transport environments.

○ Total CPU time greater with parallel

A good job of parallelization will end up reducing the **wall-clock** time you spend waiting for your application to finish; however, the bill you get back from your service center is **not** likely to be based on time as measured by your trusty Timex ... if it were, they wouldn't stay in business very long. No, they're going to add up the CPU time you racked up **over all of the processors you used** (after all, if you used them, no one else could, right?), and bill you for that.

Even on a per-CPU basis, you're going to see that a parallel task runs up a higher bill than the equivalent serial one; as previously explained, this increase is due to the additional instructions and time required to:

- ☐ Initialize and terminate tasks
- ☐ Communicate among tasks

○ Replication of code and data requires more memory

Your service center bill may take into consideration things other than CPU cycles, such as how much disk and main memory you use. A serial task uses a fixed amount of memory. A skillfully written parallel one will distribute most of it across the processors, but there will always be some values that are replicated in all tasks and some buffers used for communication that will make the total memory used by a set of parallel tasks greater than what the serial task had used.

○ Other users might wait longer for their work

The extra CPU time, disk space, and memory that your parallel application requires will not be available to other users of the system while you are using them. Show consideration for your fellow parallelizers -- use only the resources you actually need, and only for as long as you actually need them.

8. Parallel Processing at the Theory Center

Experimenting with parallel processing since 1986

The Theory Center has been in the parallel arena for almost 15 years; in fact, our first director, Nobel Laureate Ken Wilson, convinced Floating Point Systems, until then known world-wide for their vectorization units, to build one of the first US full-capability multiprocessor hypercubes, the late, lamented *T-Series*. Stories abound concerning this first venture into untamed waters; ask Greg Burns or Andy Pfeiffer about the meaning behind the term *boat-anchor*, or about the significance of the little string that ran across the floor and under the wall.

All early explorers have similar kinds of stories, and they all go together to make up the rich history we're still in the process of constructing. The Theory Center has continued to pursue parallelism, and has convinced many initially recalcitrant people and companies, some of the latter being among the largest in the world, to join the parade. A detailed description of the previous projects and supported tools you can find [here](#).

High Performance Cluster of PCs

New PC technologies offer a low price possibilities for high performance computing based on the cluster of PCs. In 1999 CTC had established the [AC3 Velocity](#) Cluster and had started to experiment (along with [NCSA](#)) various parallel tools and applications based on Microsoft Windows OS. Scheduling of parallel and serial work on the CTC Velocity Cluster is accomplished with the CTC developed [Cluster CoNTroller™](#) for NT system. A large number of [software applications](#) are available for the researchers that utilize AC3 Velocity Cluster.

9. Parallel Programming Example

This section provides access to a sample program that demonstrates parallel techniques. A simple program uses the Message Passing Interface (MPI) to send the message "Hello, world" from one task to several others. The same program runs on each node, determining whether it is a sender or receiver through a variable named "me."

- [Hello world code \(FORTRAN version\)](#)
- [Hello world code \(C version\)](#)

The equivalent program in HPF runs on each node, in parallel sets up the message and determines its own identity, and then sends that value ("me(i)") to node 0 where it is printed along with the message.

- [Hello world code \(HPF version\)](#)



10. Conclusions

Here are some of the most important things you should take with you from this presentation:

Parallel processing can significantly reduce wall-clock time.

The whole reason for getting involved in parallelism is to reduce the time you spend waiting for your results. The characteristics of algorithms virtually insure that there will be points in most applications where significant savings can be achieved by judicious use of parallelism.

Writing and Debugging Software is More Complicated

Parallelism involves at least an order-of-magnitude more complexity in your code -- this need not be evident in the code that **you** write, but will certainly be evident in the size of the executable that results. The important aspect, of course, is the **conceptual** complexity that has been added, which has immediate implications for its debugging and maintenance.

Parallelization is not yet fully automatic

You'll have to assume that whatever parallelization is needed, **you'll** have to provide. There are specific situations where you'll be able to get away with as little as adding a few *parallel-directives* as comments in your still-serial source, but this is not yet a commonly-encountered scenario.

Overhead of parallelism costs more CPU

Parallelism doesn't come for free; not only do **you** have to do more work, but so does the computing system, and parallelism itself involves additional effort in terms of process-control: starting, stopping, synchronizing, and killing. Besides this, parallelism requires that, in general, both code and data exist in multiple places, and getting them there involves additional time as well as the additional space needed to hold them.

Decide which architecture is most appropriate for a given application

The characteristics of your application should drive your decision as to how it should be parallelized; the form of the parallelization should then determine what kind of underlying system, both hardware and software, is best suited to running your parallelized application.

Table of Contents	1	2	3	4	5	6	7	8	9	10	Less Detail
-------------------	---	---	---	---	---	---	---	---	---	----	-------------

Quiz

Take a multiple-choice quiz on this material, and submit it for grading.

Evaluation

Please complete this short evaluation form. Thank you!

