Is Parallelism for You?
Rules-of-Thumb for Computational Scientists and Engineers

Cherri M. Pancake
Oregon State University
pancake@cs.orst.edu

Published in *Computational Science and Engineering,* Vol. 3, No. 2 (Summer, 1996), pp. 18-37.

Abstract

Parallel computing sounds straightforward: apply multiple CPUs in order to solve bigger, more complex problems and get results faster. Unfortunately, the experiences of computational scientists and engineers show that the price-tag for parallelism is high. It is possible to spend months of effort, only to find that the parallel program runs slower on six CPUs than the original version did on one. Should you make the investment? This article provides practical rules-of-thumb for predicting if parallelism is likely to be worthwhile, given the nature of your application and the amount of effort you want to invest.

Contents

Introduction

Parallelism is an intuitive and appealing concept. Consider a computational science or engineering problem you've been working on. If executing it on a single CPU yields results in, say, 10 hours, why not use 10 CPUs and get the results after just an hour?

In theory, parallelism is that simple-applying multiple CPUs to a single problem. For the computational scientist, it overcomes some of the constraints imposed by single-CPU computers. Besides offering faster solutions, applications that have been parallelized-converted into parallel programs-can solve bigger, more complex problems whose input data or intermediate results exceeded the memory capacity of one CPU. Simulations can be run at finer resolution. Physical phenomena can be modeled more realistically.

In practice, however, parallelism carries a high price tag. Parallel programming involves a steep learning curve. It is also effort-intensive; the programmer must think about the application in new ways and may end up rewriting virtually all of the serial (single-CPU) code. What's more- whether "parallel" refers to a group of workstations or to a top-of-the-line high-performance computing system, a parallel computer's runtime environment is inherently unstable and unpredictable. The techniques used for debugging and tuning the performance of serial programs do not extend easily into the parallel world. It is perfectly possible to work months on parallelizing an application, only to find that it yields incorrect results or that it runs slower now than before.

How do you know whether or not to make the investment? The purpose and nature of your application are the most important indicators of how successful parallelization will be. Your choice of parallel computer and plan of attack will have significant impact, too, not just on performance but also the level-of-effort required to achieve it. This article is an informal introduction to the factors that influence parallel performance. It offers practical, basic rules of thumb that can help you predict if parallelism might be worthwhile, given your application and the effort you want to invest. The techniques I present for estimating likely performance gains are drawn from the experiences of hundreds of computational scientists and engineers at national labs, universities, and research facilities.[1] The information is more anecdotal than experimental, but it reflects the very problems that must be overcome if parallel programming is to yield useful benefits.

Pre-conditions for Parallelism

Basically, your application's purpose is a good indicator of how much effort you're likely to invest in improviing its performance. Unless you have a burning desire to learn parallel programming, your performance needs should be used

as a "precondition" test. Three factors establish an application's performance objectives. As Figure 1 illustrates, these fall into a spectrum reflecting what you might gain through parallelization.

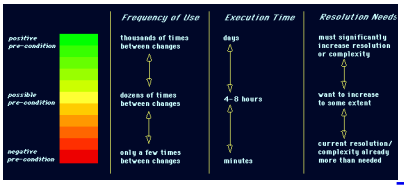


Figure 1. Precondition test: how much performance do you need?

First, how frequently will the application be used before changes are needed? If the answer is thousands of times between revisions, this is a highly productive application that probably merits significant programmer effort to improve its performance. A program that must change frequently, on the other hand, will not let you amortize the time invested in those improvements.

The second factor is the time currently needed to execute the application. Let's assume you now wait days to get your results. Reducing that time to a fraction may improve your professional productivity significantly. In contrast, if you can measure runtime in minutes, you are unlikely to be satisfied with the payoff in terms of performance improvement versus effort required. Note that theses are relative measures. If your application is a real-time emergency management system, even a few seconds' improvement might be significant.

Third, to what extent are you satisfied with the current resolution or complexity of your results? If the speed or memory capacity of serial computers constrains you to a grid whose units are much coarser than you want-say, representing the ocean surface in 10-degree units, when what you really need is a granularity of 2 degrees-parallelism may be the only feasible way to break that constraint.

According to the experiences of other scientists and engineers, your needs should rate at least one "white" in Figure 1's spectrum before you even consider investing the effort to parallelize your application. Conversely, even one "black" factor should be interpreted as an indication that your performance needs probably do not merit much parallelization effort. Further, note that even three whites do not guarantee that parallelism will pay off; they simply indicate that you need parallelism's potential power. Apply the rules of thumb described in this article to determine if the effort you must invest will be small enough to make the whole process worthwhile.

How Your Problem Affects Performance

The nature of the problem is the key contributor to ultimate success or failure in parallel programming. In particular, data access patterns and associated computation indicate how easy or difficult it will be. Geoffrey Fox was the first

researcher to study how the characteristics of applications constrain their performance. He established that most technical applications fall into one of three categories, which he called *problem architectures*, and that each is suited to certain types of parallel computers.[1,2] Here, I extend Fox's concept to a fourth category, *pipeline parallelism*, and describe how you can use problem architecture to help determine how likely you are to achieve respectable performance-and at what cost.

Consider a seismic imaging problem.[2,3] Data on responses to seismic shock waves are gathered at field sites, then computed to derive contour plots of the subsurface geological structure at each site. The computation can be a sequence of serial jobs, each computing an image from one input data set; or parallelism can be introduced by having multiple data sets processed at the same time, as portrayed in Figure 2.

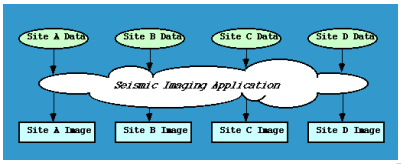


Figure 2. An example of perfect parallelism: seismic imaging.

From the parallel programmer's perspective, this is the simplest problem style, referred to as perfect (or "job-level") *parallelism*. Fundamentally, the calculations on each data set are wholly independent. That is, the images could be computed on independent machines running copies of the application, as long as the appropriate input data were available to each copy. It's easy to achieve significant performance gains from applications fitting this style of parallelism, so they are sometimes called "embarrassingly parallel" (but no programmer should be embarrassed to have one).

Now suppose that the images are not completely independent; perhaps substructure responses are being simulated in a series of time-step, as shown in Figure 3. Data from different time steps are used to generate images showing change over time. Data produced by the simulation must be rendered in a three-dimensional volume, then formatted for graphical display. If this application were carried out serially, the simulator's output data sets would serve as input to the volume-rendering program, whose output would in turn serve as input to the formatting application. Parallelism can be introduced by overlapping processing, so that volume rendering begins as soon as the first time step's data are available. Then, while the simulator produces the third data set, volume rendering proceeds on the second data set, and the first is formatted and displayed.

Figure 3. Example of pipeline parallelism: simulation of earth substructure.

This model is called *pipeline parallelism*, since data is effectively "piped" from one computational stage to another. The key characteristic is that results are passed just one way through the pipe (that is, the simulation of the next time step does not require information from the volume-rendering or formatting stages). Start-up is delayed initially as data become available at each stage, so overall performance gains will depend on the relative number of time steps to be processed once all points along the pipe are active. Pipeline parallelism also introduces potential problems. If the stages are not all computationally equivalent, faster stages will overtake the slower ones, finishing sooner. One solution is to execute computationally intensive stages on faster CPUs, balancing the work precisely can be quite difficult. Either way, the programmer must accommodate a possibly unequal work load with tests to check when input data are ready and to ensure that buffer or disk space can hold output data. For this reason, pipeline parallelism is not as simple as perfect parallelism.

In many applications, results cannot be constrained to a one-way flow among processing stages. Consider, for example, an atmospheric dynamics problem .[3, 4] The data represents a 3D model of the atmosphere, where an occurrence in one region influences areas above and below the disturbance, and perhaps to a lesser extent, those on either side. Over time, the effects propagate to an ever-larger area extending in all directions; even the disturbance's source may experience reverberations or other movements from neighboring regions. If this application were executed serially, calculations would be performed across all the data to obtain some intermediate atmospheric state, then a new iteration would begin. Parallelism is introduced with multiple CPUs participating in one iteration, each applying the calculations to a data subset (see Figure 4). Each iteration is completed across all data before the next iteration begins.

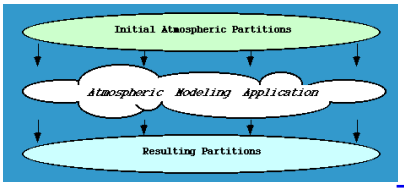


Figure 4. Example of fully synchronous parallelism: simulation of atmospheric dynamics.

This is called *fully synchronous parallelism*, meaning that-at least conceptually-each calculation is applied synchronously (or simultaneously) to all data. The key here is that future computations or decisions depend on the results of all preceding data calculations. Usually, there aren't enough CPUs to apply a calculation to all data at the same time, so each CPU iterates through a subset. If the subsets are not homogeneous, the computational intensity will vary on different CPUs. For example, a disturbance in the uppermost stratum starts by modifying data representing the upper layers, while lower layers are unaffected. This spatial variation means that if each CPU applies calculations to a subset representing a horizontal stratum, only one or two CPUs actually perform intensive work at this point. Meanwhile, synchronicity demands that the other CPUs cannot proceed to the next set of calculations, so they must wait

until the busy ones catch up.

Alternatively, if CPUs apply calculations to vertical regions, computational work may be uniformly distributed of at this point in the program, but this will be offset at later points when computation varies along the horizontal dimension instead. Consequently, fully synchronous parallelism requires more programmer effort than pipeline parallelism to achieve good performance.

The fourth style of parallelism is illustrated by a related application, which models the diffusion of contaminants through groundwater (Figure 5). Initially, only the groundwater partitions close to the contamination source are affected, but over time the contaminants spread, building up irregular areas of concentration. The amount of computation depends on the amount of contaminant and the geophysical structure, so it varies dramatically from one partition (and time step) to another. In a serial program, this means that time step length will be irregular and perhaps unpredictable. Parallelism is introduced by dividing the work among multiple CPUs at each time step. During early time steps, each CPU may apply calculations to just a few partitions and the computation's duration may be brief because concentrations are low; later, as concentrations build up and progressively affect more partitions, a single CPU may perform many more computations on many more partitions at each step.

Figure 5. Example of loosely synchronous parallelism: contaminant flow through groundwater.

This is an example of *loosely synchronous parallelism*. When each time step ends, CPUs that have finished their work must wait for the others to complete before sharing intermediate results and going on to the next time step. Thus, this style's key characteristic is that the CPUs each do parts of the problem, exchanging information intermittently. Loosely synchronous parallelism, combining the difficulties faced in pipeline and fully synchronous parallelism, is the most difficult to program. The need to exchange information among CPUs (here, at time step boundaries) requires tests so that one CPU can determine when the others's data and ready and can avoid overwriting values not yet used. These CPUs effectively proceed at their own rates between those exchanges. With loosely synchronous parallelism, it's very difficult to distribute computational work evenly among the CPUs, since the work load now varies both temporally and spatially.

Analyzing your problem's architecture may seem an unnecessary exercise, but it will help you to decide if parallelism is worth it. First, consider how your application uses data. Classify your application as perfect, pipeline, fully synchronous parallelism (the case studies present examples of how this is done.) Then determine how the computational characteristics will influence effort-to-parallelize by applying the following rules-of-thumb:

Rule of Thumb (1)

If your application fits the model of perfect parallelism, the parallelization task is relatively straightforward and likely to achieve respectable performance.

Rule of Thumb (2)

If your application is an example of pipeline parallelism, you have to do more work; if you can't balance the computational intensity, it may not prove worthwhile.

Rule of Thumb (3)

If your application is fully synchronous, a significant amount of effort is required and payoff may be minimal; the decision to parallelize should be based on how uniform computational intensity is likely to be.

Rule of Thumb (4)

A loosely synchronous application is the most difficult to parallelize, and probably is not worthwhile unless the points of CPU interaction are very infrequent.

Note that you may need to analyze how computation (as well as data) is dispersed over the lifetime of an execution. This information may be useful even if you decide not to parallelize, since it provides valuable insight into serial performance. For our purposes, a general understanding of problem architecture is essential for determining if your application is likely to perform well on the type(s) of parallel computer available to you.

How Your Machine Affects Performance

Generally, a parallel computer is any collection of processing elements connected by some type of communication network. (Here, the processing elements are referred to as CPUs for simplicity, but they involve memory as well.) Also known as multicomputers, such systems encompass a range of sizes and prices, from a group of workstations attached to the same LAN to an expensive high-performance machine with hundreds or thousands of CPUs connected by ultra high speed switches. Clearly, CPU speed, capacity, and communication medium constrains the performance of any parallel application. But from the programmer's perspective, the way in which multiple CPUs are controlled and share information may have even more impact, influencing not just the ultimate performance results but also the level of effort needed to parallelize an application.

Figure 6 shows a basic "family tree" for parallel computer architectures. The

control model dictates how many different instructions can execute simultaneously. The terms SIMD (single instruction, multiple data) and MIMD (multiple instruction, multiple data) date from parallel computing's early days; both are still in evidence, although no longer the only distinguishing feature of parallel computers. *Memory model* indicates how many CPUs can directly access a given memory location. All CPUs access a single memory in shared-memory computers, whereas distributed-memory computers make use of a separate memory for each CPU. Memory is shared among small groups of CPUs in symmetric multiprocessor (SMP) computers, but when groups are clustered to form larger systems each group's memory remains isolated. The *programming model* refers to restrictions on the number of executables (object images) that can participate in a parallel execution. In the multiple-program, multiple-data model, the programmer creates a separate executable for each CPU; for the single-program, multiple-data model, all intstructions to be carried out by all the CPUs are combined into a single executable. Programming models are discussed in more detail in a later section.

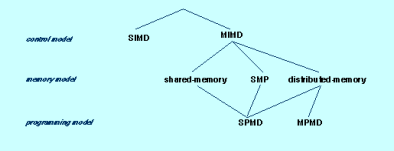


Figure 6. "Genealogy" of parallel computing systems.

The interaction of control model and memory model results in four classes of parallel computer architecture: SIMD, shared-memory, distributed-memory, and SMP. Each of these is described individually below; Table 1 provides a summary of that information.

Characteristic	SIMD	Shared-memory	Distributed-memory	SMP cluster
Memory	Shared	Shared	Distributed	Shared by CPUs on a node but distributed across nodes in a cluster
Instruction sequencing	Single stream	Multiple streams	Multiple streams	Multiple streams
Number of executables	Single	Single (SPMD)	Single or multiple	Single or multiple
Examples	TMC CM-2, MasPar	Cray C-90, Fujitsu, IBM ES/9000	Cray T3D, IBM SP2, Intel Paragon	Convex Exemplar, SGI PowerChallenge, Sun Sparcserver
Highlights	Efficient, relatively easy to program	Fast, large memory	Versatile, cost-effective	Versatile, cost-effective
Lowlights	Fits few problems	Expensive	Hard to use efficiently	Hard to use multiple nodes efficiently
Major programming hurdle	Using array operations efficiently	Protecting access to shared data	Minimizing communication costs	Protecting access to shared data (within a node), minimizing off-node communications

Table 1. Summary of parallel computer architectures.

SIMD Multicomputers

On a SIMD multicomputer, sometimes called a processor array, all CPUs execute the same instruction in "lockstep" fashion- examples are MasPar's MP-2 and Thinking Machines' Connection Machine (CM). Figure 7a illustrates the general concept: a single control unit tracks the current instruction, which the CPUs apply simultaneously to different operands.

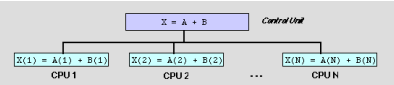


Figure 7a. Comparison of parallel computing architectures: SIMD multicomputer.

The control unit is the programmer's key to both the benefits and the costs of

parallelization. SIMD machines are relatively easy to program and use memory efficiently. Whenever the program uses Fortran90-style array operations or makes calls to the array functions library, the compiler automatically generates parallel code. The main programming hurdle is to cast basic calculations as array operations. If your application doesn't fit the fully synchronous model, it will be difficult or impossible to parallelize it for a SIMD architecture.

Achieving good performance can be quite difficult, even if the application apparently fits the model. When an instruction involves arrays as operands (as in Figure 7a), the control unit appears to cause all CPUs to execute the instruction on the appropriate element pairs in one step. In actuality, however, few operations involve arrays whose dimensions exactly match the number of CPUs. Most instructions require that the CPUs iterate through groups of elements. If the number of elements isn't an integral multiple of the number of CPUs, the "extra" CPUs will effectively lose cycles while the last elements are processed.

Other performance problems are tied to lost, or wasted, CPU effort. When an operation is conditional (for example, dividing vector **a** by vector **b** only where the element of **b** is nonzero), all CPUs actually perform the operation; the results are simply discarded from any CPU where the condition proves false. The worst case occurs for a scalar operation (such as the addition of two floating-point numbers), since all CPUs redundantly perform the operation even though only one copy of the result is needed. The condition represents a serial bottleneck, since the machine's hundreds or thousands of CPUs are effectively reduced to the power of a single CPU. Just a few of these can counteract all the performance gains realized by array operations.

Shared-memory multicomputers

Unlike SIMD machines, MIMD multicomputers provide each CPU with its own control unit. At any moment during execution, different CPUs may execute different instructions. This lets CPUs proceed through calculations at different rates, but it also means that the programmer cannot necessarily assume anything about the relative order in which a given instruction is executed on two different CPUs.

On a shared-memory multicomputer, the CPUs interact by accessing memory locations in a single, shared memory, exemplified by traditional supercomputers such as Cray Y/MP and C-90s, IBM ES/9000s, and Fujitsu VPs. They tend to be the fastest, largest, and most expensive form of parallel computers. Although more difficult to program than SIMD machines, shared-memory multicomputers offer a more natural fit with a much larger range of applications.

As shown in Figure 7b, each CPU executes its own instruction, applied to operands stored in the shared memory. Rather than specifying array operations-though these may become more common as Fortran90 parallel compilers

become generally available-the programmer uses compiler directives on computationally intensive loops. This process is similar to preparing programs for vector processing and will be familiar to some computational scientists and engineers. The basic idea is to take advantage of program loops that perform a large number of calculations (typically applying the same calculations to multiple elements of arrays). A parallel compiler converts the loop into collection of loops that will be performed by multiple CPUs, each applying the calculations to a subset of the data. At execution time, each CPU proceeds through its instructions, accessing shared-memory locations without knowledge of other CPU's activities.

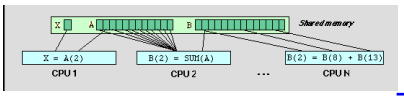


Figure 7b. Comparison of parallel computing: shared-memory MIMD multicomputer

The balance between CPU speed and communication speed is critical.

Shared-memory accesses can be a potential source of race conditions, where program results are sensitive to specific memory accesses ordering-in effect, it's a race to see which CPU arrives first. Figure 7b indicates this, where two CPUs each attempt to modify the current value of B(2); the final value will depend on the relative order of the two store operations. Since relative timing can vary from subtle changes in the runtime environment, a program with a race condition may appear to work normally, perhaps for extended periods, then suddenly "blow up" or produce inconsistent results.[6] A major part of the programmer's time is likely to be spent identifying potential races and safeguarding shared data through a locking mechanism that excludes other CPUs from access when a data value is being modified. Frequent locking adversely affects performance as CPUs are forced to wait their access turn, so the trick is to provide just the right amount of protection.

Shared data protection is not the only area requiring programmer effort. As with vector computing, the performance of shared-memory parallelism largely depends both on the size and of intensity computational loops (see Levesque[7] for examples applying to both vector and shared-memory machines) and on the compiler's analysis capabilities. The programmer may have to restructure loops to help the compiler recognize potential parallel code. For some applications, it is impossible to restructure calculations enough to achieve good performance. This is particularly true of fully synchronous problems like the atmospheric dynamics example, where data accesses are sporadic and highly interdependent.

Distributed-memory multicomputers

On distributed-memory multicomputers, too, each CPU executes its own instruction stream, but as the name implies, memory isn't shared. Instead, each CPU is has a private memory. Most current high-performance parallel machines

have distributed memory: for example, Cray T3D, IBM SP-2, Intel Paragon, and Meiko CS-2. Based on workstation microprocessor technology, these systems are versatile and very cost-effective. Their major disadvantage is their inherent difficulty in efficiently using resources.description of symmetric multiprocessors.)

Figure 7c illustrates how distributed memories operate. To interact or to share information, the CPUs send each other messages, typically over high-speed switches. As shown, the vector A referenced by one CPU is not in the same location as that referenced by other CPUs. If data are read-only, they can be copied into all the CPUs' memories and accessed quickly, with no need to lock out other CPUs. When there is no particular need to share, arrays can be split up and stored across multiple memories so that, for example, each CPU's vector A actually represents one column of a large array.

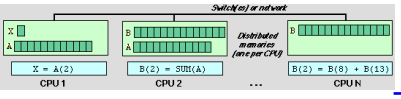


Figure 7c. Comparison of parallel computing: distributed-memory MIMD multicomputer

To share data, however, the program must explicitly send them back and forth among the CPUs. This leads to potential race conditions, since it takes time to propagate one CPU's updates to the copies stored at other CPUs. Distributed-memory systems are also prone to livelock, where a CPU waits for data that never arrive, or deadlock, where two or more CPUs are stuck waiting for each other. Compilers can analyze a program to detect all possible locations where races, livelock, or deadlock might occur, but they do so conservatively, typically estimating a hundred or more "potential" problems for every real error. Distributed-memory programs tend to be harder to debug and test than SIMD or shared-memory programs.[8]

In terms of performance, the balance between CPU speed and communication speed is critical, for reasons elaborated in "Setting Realistic Expectations," below. Current technology results in relatively fast CPUs being coupled with relatively slow communications. (Note that the same model applies to workstation clusters, which essentially are distributed-memory multicomputers with ultra-slow communications.) The key to obtaining performance is thus the programmer's ability both to minimize communication, in terms of interaction points and the data transferred at each interaction, and to time them so that the CPUs are kept busy. For a perfectly parallel application, this may be trivial. But pipeline and loosely synchronous applications will achieve respectable performance only if there is relatively little data to exchange and/or relatively long time periods in which to effect the exchanges. Fully synchronous applications are entirely unsuited to this type of system.

SMP's and SMP clusters

So-called symmetric multiprocessor machines recently joined the parallel

computing marketplace. They also use workstation microprocessor technology, but couple several CPUs (typically four or eight) with a shared memory. The word "symmetric" refers to the fact that each CPU can retrieve data stored at a given memory location in the same amount of time. SMPs resemble shared-memory multicomputers, but are slower and less expensive, with less CPU power. Examples include SGI's PowerChallenge and Sun's Sparcserver product lines.

It is also possible to cluster SMPs into larger groups with correspondingly more CPU power, as shown in Figure 7d. The resulting configuration behaves much like a distributed-memory multicomputer, except that each node actually has multiple CPUs sharing a common memory (Convex's Exemplar best illustrates this, since the cluster is connected by a high-performance switch; there also are a growing number of SGI and Sun clusters).

To date, the major performance successes have been scored by programmers who treat SMPs as a collection of distinct, small-scale shared-memory systems. With the exception of the Exemplar, the performance of the networks/switches connecting the SMPs has been disappointing. Parallelism involving even moderate numbers of CPUs tends to be bounded in performance by the communication speed (typically comparable to that of a workstation cluster). When assessing an application's likely performance, SMP clusters should be treated as shared-memory multicomputers if your entire application can fit on one SMP node, or as distributed-memory multicomputers if it requires CPUs distributed across the cluster.

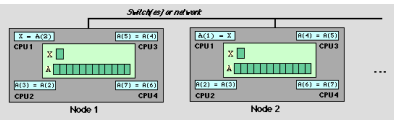


Figure 7d. Comparison of parallel computing: cluster of symmetric multiprocessors (SMPs)

Matching problem to machine

In general, then, each type of parallel computer is appropriate for applications with certain characteristics. If an inappropriate match is made, the programmer will certainly be forced to expend excessive effort, with possibly disappointing performance results. The following rules of thumb summarize the interaction between application model and machine type:

Rule of Thumb (5)

A perfectly parallel application will probably perform reasonably well on any MIMD architecture, but may be difficult to adapt to a SIMD multicomputer.

Rule of Thumb (6)

A pipeline style application will probably perform best on a shared-

memory machine or clustered SMP (where a given stage fits on a single SMP), although it should be adaptable to a distributed-memory system as well, as long as the communication network is fast enough to pipe the data sets from one stage to the next.

Rule of Thumb (7)

A fully synchronous application will perform best on a SIMD multicomputer, if you can exploit array operations. If the computations are relatively independent, you might achieve respectable performance on a shared-memory system (or clustered SMP if a small number of CPUs is sufficient). Any other match is probably unrealistic.

Rule of Thumb (8)

A loosely synchronous application will perform best on a shared-memory system (or clustered SMP if a small number of CPUs is sufficient). If there are many computations between CPU interactions (see "Setting Realistic Expectations"), you can probably achieve good performance on a distributed-memory system as well.

How Your Language Affects Performance

The programming language you use will obviously affect the effort required to parallelize your application. What's more, extreme variation in compiler capabilities and run-time support environments means that the language will also constrain the performance you can hope to attain. The type of programming model, shown as the lowest level in machine genealogy in Figure 6, is often a key indicator of both effort and performance.

With a SPMD model, each CPU will execute the same object code. On a SIMD multicomputer, exactly the same instructions will be executed in lockstep synchrony. On MIMD systems, the CPUs have individual copies of the program and proceed through it at differing rates, perhaps executing entirely different instructions sequences (for example, subject to If conditions). Either way, the programmer has only one program to track, which can be an advantage for debugging. There may well be a performance cost, particularly on MIMD systems. All data and all instructions to be accessed by any CPU effectively must be accessible to all CPUs, increasing the memory required and often degrading memory access time as well.

In contrast, the MPMD model lets each CPU to have a distinct executable. (Note that since this conflicts with basic SIMD computing concepts, the model applies only to MIMD machines). Many experienced parallel programmers prefer MPMD for two reasons. First, it utilizes memory space more efficiently. Code space requirements are reduced for pipeline and loosely synchronous

applications, where CPUs typically execute totally different code. Data space can also be reduced for programs with large arrays, since the programmer can subdivide them in portions accessible to just those CPUs that really need them. Second, the programmer can split up the functionality of different computational stages into separate programs, to be developed and debugged independently or reused as components of other programs. But it becomes harder to deal with some types of errors and performance problems, as it's difficult for programmers to conceptualize how the activities of independent CPUs might influence one another.

Strictly speaking, "programming model" is a feature of programming languages, rather than parallel computers. Many machines described here, however, impose the SPMD model on the programmer because their operating system and tools view a parallel program as a single entity, and cannot report information on multiple executables. While it may be possible to run multiple executables in MPMD fashion on a predominantly SPMD system, the operating system and tools will consider them as a collection of unrelated programs. The programmer may have to forego many aspects of system support, including consolidated I/O, use of debuggers, and access to program-wide timing information.

Table 2 lists the parallel languages and libraries available (see the literature[9,10] for surveys of language features). The programmer rarely has much real choice, however. Except for the libraries, all languages enforce a particular programming model. Most are also limited to particular machine types (and perhaps manufacturers). Message-passing libraries are the most broadly available, having been ported across all the MIMD architectures. This means that message-passing applications are the most portable; on the other hand, the programmer essentially sacrifices compiler error detection capabilities and may inhibit compiler optimizations.[11]

Paradigm	Characteristics	Examples	Model
Control-parallel	Computational "work" (loop iterations or subroutines) is split up for assignment to processors. Processors periodically synchronize their activities (for example, at end of parallel loop).	IBM's Parallel Fortran Fujitsu's VP Fortran Cray's Autotasking Fortran ANSI X3H5 Fortran†	MIMD/SPMD MIMD/SPMD MIMD/SPMD MIMD/SPMD
Data-parallel	Data domain (usually array) is subdivided and "ownership" is assigned to individual processors. Typically, the owner performs computations for its elements and provides copies to other processors as needed.	High Performance Fortran† TMC's CM-Fortran MasPar's MPF Data Parallel C† TMC's C* MasPar's MPL	MIMD/SPMD SIMD SIMD MIMD/SPMD SIMD SIMD
Message-passing	Each processor executes on its own, when two processors need to share data values or coordinate their activities, one sends a message and the other must receive it. A subroutine library supports send, receive, and other operations. Very low-level, virtually no error detection.	PVM library† MPI library† Intel's NX library p4 library TCGMSG library Parasoft's Express library IBM's MPL library Fortran M	MIMD/MPMD MIMD/MPMD MIMD/MPMD MIMD/MPMD MIMD/MPMD MIMD/MPMD MIMD/SPMD MIMD/SPMD
Combined	Two or three of the above can be combined (for example, control-parallel subroutines that send messages to each other).	pC++ Cosmic's Fortran Cosmic's C	MIMD/MPMD MIMD/SPMD MIMD/SPMD

† A language standard; supported, or intended to be supported, across multiple manufacturers' machines.

Table 2. Varieties of parallel programming languages available.

Once you determine your application and machine, you will probably be limited to just a couple of parallel language/library choices. This will be further constrained by such factors as your expertise in Fortran versus C, access to colleagues who have used the parallel language, the ability to call other scientific or math library routines you need, and the availability of public-domain languages on your particular system (for example, PVM, MPI, p4, pC++, Data Parallel C, Fortran M).[4]

The rule-of-thumb that applies to language selection, then, is quite simple:

Rule of Thumb (9)
With few exceptions, you don't pick the language; it picks you.

Setting Realistic Expectations

Computer scientists may find parallel programming to be interesting in itself, but that's not the objective of most scientists and engineers. As Boeing's Ken Neves said, "Nobody wants parallelism. What we want is performance".[12] If applying 50 CPUs to a task doesn't yield results much sooner than a single CPU, the computing resource is used inefficiently. Even more important, the fact that an application can execute across 50 CPUs means that someone has expended time and energy parallelizing it. Failure to attain reasonable performance with a reasonable level of effort wastes human productivity, too.

To avoid that kind of failure, assess the application's potential before deciding about parallelization. This assumes that your problem lends itself to parallelism, that your machine offers a reasonably good fit to that problem, and that you know what language will be used. It also presupposes that you have an existing serial program that already implements your application; I will refer to this as the baseline. Strict devotees of parallel programming claim that a new parallel program should be built from scratch, but this is unrealistic for most users. (Surveys of experienced parallel programmers show that 59 percent modify or compose programs from existing code; the 31 percent who start from scratch are typically computer scientists and applied mathematicians.[8]) Moreover, a solid baseline program provides a built-in mechanism for checking the validity of the parallel program's results (does it yield the same results as the serial code does for all sets of inputs?), as well as the basis for measuring performance improvements (how much faster is version X than the baseline?).

However, a sloppily implemented baseline must first be cleaned up if it is to provide realistic estimates of future performance. Although this may involve a significant amount of work (for example, restructuring Common blocks if a large application redefines them at many points), the investment is guaranteed to pay off, since it will improve the serial version's maintainability-and perhaps its performance-even if you decide not to parallelize. If you do proceed, a clear, robust code will be absolutely essential in order to produce a reliable parallel implementation.

Performance estimates are based on timings of the baseline program. Insert calls to the system library to obtain wall-clock readings just before and after the portion(s) of the application with potential for parallelism (based on the information in the preceding sections); collectively, these represent the

potentially parallel code. In addition, insert timing calls as the program's first and last statements, so that you can also determine whole code time. Figure 8 shows where timing calls would be placed to measure a simple simulation program. Exclude the input and output phases from the potentially parallel portion, since they represent serial bottlenecks (I/O cannot be performed in parallel on most machines). Identify other major operations that must be executed serially and execute them, too.

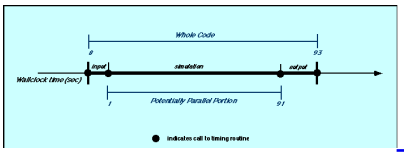


Figure 8. Timing the baseline program to estimate likely parallel performance: whole-code versus potentially parallel timings.

The goal of parallelism, clearly, is to reduce the whole code time so that results are produced faster. Equally clearly, performance gains can only be made by reducing the amount of time spent in the potentially parallel portion, since this is the only area where multiple CPUs can really be applied. Ideally, the entire simulation portion of the example could execute in parallel.

The timing results obtained by executing the baseline program make it possible to calculate the program's *parallel content*, *p*, defined as a proportion:

$$p = \frac{\textit{potentially parallel time}}{\textit{whole code time}} = \frac{90}{93} = 0.9677$$

This indicates that 96.8 percent of the code is potentially parallelizable, while only 3.2% is necessarily serial content. To understand the impact of those figures, Amdahl's law is applied to calculate the theoretical speedup as a function of the parallel content (p) and the number of CPUs that will be used (N):

$$\textit{theoretical speedup} = \frac{1}{(1-p)+(p/N)} = \frac{1}{.0323+(.9677/N)}$$

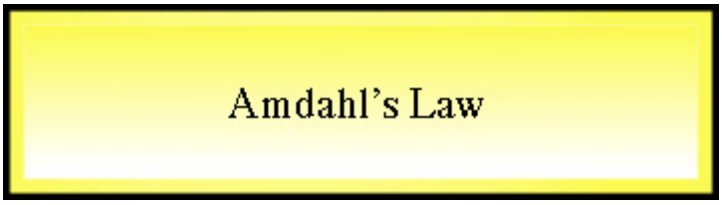


Figure 9a shows how this theoretical speedup changes for increasing numbers of *N*. It is compared with *ideal speedup*, which reflects the ideal that applying *N* CPUs to a program should cause it to complete *N* times faster. Obviously,

between ideal and theoretical speedup there is a gap that widens as *N* increases. The gap size is solely a function of the program's serial content. This suggests that for every program, it will not be worthwhile to go beyond some number of CPUs. As Table 3 shows, even applying an infinite number of CPUs to the example will achieve at most a 30-times speedup.

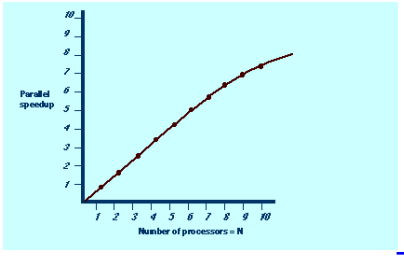


Figure 9a. Estimating parallel performance: theoretical speedup differs from ideal speedup as a function of the program's serial content

Number of CPUs	Theoretical speedup
1	1.000
2	1.937
3	2.818
4	3.647
5	4.428
6	5.167
7	5.863
8	6.525
9	7.152
10	7.752
...	...
∞	30.959

Table 3. Theoretical speedup, assuming a parallel content of 96.77 percent.

Note that the curves may change as the problem size increases (for example, when the time steps in the simulation double). If increasing problem size is essentially equivalent to increasing the amount of parallelizable computation, the potential parallel content will increase. This, in turn, will improve the curve for theoretical speedup, diminishing the gap from ideal speedup. However, if increasing problem size also increases the length of the serial bottlenecks, the gap may widen. You should consider how much size variation is likely for your application, and estimate its effect on theoretical speedup.

Unfortunately, theoretical speedup is rarely achieved by a parallel application. There will actually be an *observed speedup* curve that exhibits a widening gap from theoretical speedup (Figure 9b), reflecting the external overhead's effect on total execution time. This overhead comes from two sources, both essentially beyond the programmer's control: the additional CPU cycles expended in simply managing parallelism, and delays, or wasted time, spent waiting for I/O, communications among CPUs, and competition from the operating system or other users. Theoretical speedup does not consider these factors.

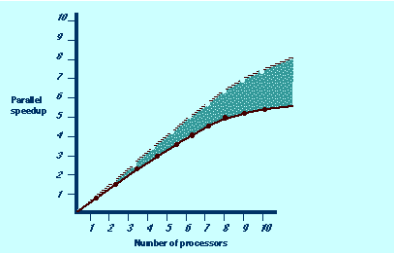


Figure 9b. Estimating parallel performance: observed speedup will fall well below theoretical speedup, due to environmental factors and imperfect concurrency

Another lack of precision in theoretical speedup is that it assumes *perfect concurrency*. Parallel code run on 5 CPUs will speed up 5 times only if all CPUs simultaneously (a) start the parallel portion, (b) perform all coordination activities (such as exchanging data), and (c) complete their calculations. Combined, this is perfect concurrency, shown in Figure 10a. It assumes that computational intensity is completely homogeneous, which may be almost true for dense linear algebra, but certainly won't be for sparse or irregular problems. It also assumes that the CPUs are identical and have identical access to all the limiting resources, such as memory and the communication network.

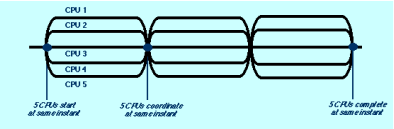


Figure 10a. Concurrency: perfect concurrency, where all CPUs begin, interact, and complete at the same time

What actually happens is *imperfect concurrency* (Figure 10b), because CPUs find it necessary to wait for access to each other or to resources. Some factors responsible for poor concurrency are within the control of the programmer, but some aren't:

- ***Uneven computational intensity across CPUs***: This can be improved by careful programming, but the nature of the application itself may be the source of the problem.
- ***CPUs waiting for information controlled by other CPUs*** (such as shared variables or messages): Experienced parallel programmers spend most of their efforts ensuring that data are "produced early, consumed late" to minimize this wait, but some applications simply require excessive interaction.
- ***Vagaries of the runtime environment*** (such as competition from other users, system interrupts, I/O delays, network "hiccups"): The average user can do little, other than schedule off-hour program runs.

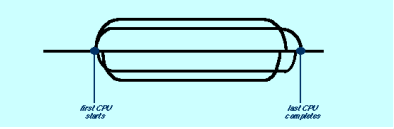


Figure 10b. Concurrency: slight variations in timing affect concurrency and cause the program to fall short of theoretical speedup

Concurrency worsens as the number of CPU interaction points increases relative to the amount of computation performed, which gives rise to *program granularity*. A coarse-grained program requires many computations between each point of CPU interaction, while a fine-grained one performs proportionately few computations. Consider, for example, a loop or subroutine containing many instructions. If the CPUs executing it reference and modify values scattered through a single matrix, the program will be fine-grained, because the CPUs must be notified whenever another CPU updates a value. If each CPU applies the operations to a different matrix, the code will be coarse-grained. As the number of instructions shrinks-or the need to share updated values increases the granularity becomes finer.

On a shared-memory computer, it is difficult to calculate a priori the minimum granularity to achieve acceptable performance. For distributed-memory computers (including networks of workstations and, to a lesser extent, clustered SMPs), however, you can get a crude approximation based on its published CPU speed and communication properties. Most hardware vendors publicize two measures message-passing performance. *Latency* is the time, typically measured in microseconds, spent initiating a message transmission. *Bandwidth* is the speed, typically in Mbytes per second, at which message data are transmitted. Essentially, latency represents the fixed overhead of a message communication; the same cost is incurred to set up any message, regardless of its length. Bandwidth represents the variable overhead, because the cost incurred to transmit a message is a function of message length. Nominally, then, the cost of sending a message can be described as:

$$message\ time = latency + \frac{message\ size}{bandwidth}$$

The real "cost" of sending a message, however, is the number of CPU cycles wasted as a program waits to send/receive a message. Quite simply, a CPU that is spending even a few cycles idling, rather than doing useful computation, will not show good performance. By considering what each communication is actually costing in terms of lost CPU power, you can predict the granularity level necessary to achieve reasonable performance on a specific parallel computer. A *message-equivalent* [13] measures the approximate number of floating-point operations that could be executed in the time needed to send one message 1,024 bytes long:

$$message\ equivalent = CPU\ speed * [latency + (1K / bandwidth)]$$

where *CPU speed* is the so-called peak speed of a single CPU in Mflops, latency is assumed to be in microseconds, and bandwidth in Mbytes per second. (Peak CPU speed in an unrealistic measure but serves as a useful basis for calculating this crude approximation of needed granularity.)

Table 4 shows the values calculated for five current parallel computers. It is clear that system A (actually a collection of workstations connected by Ethernet) will require an extremely coarse-grained program if the CPUs are to do anything more useful than wait for communications. In contrast, system C (a parallel computer highly tuned for fast communications) would tolerate almost a hundred times as many points of communication. System B (a so-called general-purpose parallel computer) falls between the two. Systems D (an SMP) and E (a cluster of those SMPs connected by a high-speed switch) show just how much impact the communication speed really has.

	Peak CPU (Mflops/sec)	Latency (microsec)	Bandwidth (Mbytes/sec)	Message-equivalent (flops)
Parallel computer A	100	2,000	1	300,000
Parallel computer B	200	300	8	85,000
Parallel computer C	100	20	50	4,000
Parallel computer D	150	5	30	5,700
Parallel computer E	150	25	10	18,750

Table 4. "*Message-equivalent*" *approximations calculated for five existing parallel computers, indicating how many floating-point operations should occur between CPU interactions for good performance.*

Note that none of these systems would really tolerate a medium- or fine-grained program. Good performance requires that computation exceed the message-equivalent on a regular basis, so each CPU would need to perform tens (or hundreds or millions) of thousands of operations between interaction points to attain good performance.

What is the impact of all these factors on programmer effort? They should be viewed as "warning signals" that alert you to potential problems you are unlikely to overcome, regardless of the effort you are willing to invest. More rules of thumb:

Rule of Thumb (10)

Timings measured on a baseline (serial) version of your application provide a solid starting point for estimating potential payoffs and reliability.

Rule of Thumb (11)

The debilitating impact of serial content on theoretical speedup means that you probably shouldn't consider parallelizing a program with less than 95% parallel content, unless you're already experienced in parallel programming, or unless you will be able to replace a significant portion of the serial version with parallel algorithms that have been proven to be good performers.

Rule of Thumb (12)

Apply your knowledge of the program to estimate how varying problem size will affect the theoretical speedup curve.

Rule of Thumb (13)

Theoretical speedup is only an upper bound on what is possible; the

attained performance will almost certainly be much lower.

Rule of Thumb (14)

Although you can improve concurrency to some extent, it will largely be dependent on the application itself and the average load on the computer.

Rule of Thumb (15)

A coarse-grained program will perform relatively well on any parallel machine; a medium- or fine-grained one will probably be respectable only on a SIMD multicomputer.

Rule of Thumb (16)

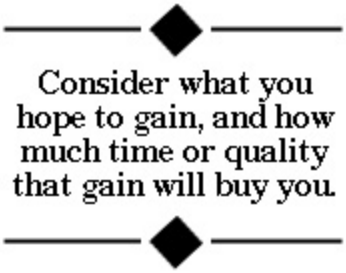
To understand the granularity requirements of a distributed-memory computer, calculate its message-equivalent. To be worth parallelizing, your program probably needs to perform many thousands of floating-point operations between each CPU interaction point.

The three case studies show how applying these 16 rules of thumb can affect your final decision.

How much performance can you really expect to get? Consider an analogy with the physical world[14]: I can't ride my bicycle faster than 40 miles per hour, so that is its peak performance. However, my average speed will depend on environmental conditions, such as my current fitness level, road condition and steepness, amount of traffic, and weather conditions. Some of these are under my control, but most are not. Consequently, my sustained performance is typically 15 miles per hour.

Wild claims about parallel performance abound, typically emanating from the marketing departments of computer manufacturers. Such claims are hard even for experienced parallel programmers to interpret; they often mislead newcomers into unrealistic notions of performance.[15] A fanciful example might be that X Corporation's HypoMetaStellar is a 400 Gigaflops machine. The quoted figure will be aggregate peak performance (that is, the peak CPU speed times the number of CPUs) and is almost worthless in estimating application

performance. The claim may also be substantiated by benchmark results proving the HypoMetaStellar is 10 times faster than any supercomputer, but that too is essentially meaningless for the parallel programmer. What counts is the fraction of peak performance regularly sustained by your application. For most applications, that fraction will probably be only 10-20 percent of peak performance. After all, even highly tuned parallel programs rarely achieve more than 20 percent.



Various other parallel performance metrics are also cited to "prove" that a parallel machine will guarantee your application good performance. As Sahni[16] demonstrates, however, the only reliable performance metric is the parallel runtime for your particular application. That clearly cannot be known in advance. In particular, it cannot be predicted accurately using statistics from any other application, no matter how similar it is in purpose or structure.

Is parallel performance achievable? Absolutely. But it is not easily achieved, nor can it be achieved for every problem. Even more disturbingly, it may require an enormous investment of human effort. Achieved performance depends on five interdependent factors:

1. the degree of parallelism inherent in the application;
2. the parallel computer architecture on which that application executes;
3. how well the language and runtime system exploit that architecture;
4. how effectively the program code exploits the language, runtime system, and architecture; and
5. the runtime environment at the time of execution.

Factor 1 should be considered a precondition for even entertaining the idea of parallelization. Recall that an application's parallel content constrains even its theoretical performance. If there's more than a tiny fraction of serial content, parallelism almost certainly will not be worthwhile. Moreover, changing the algorithm to reduce the application's serial content will have more impact than whatever effort you are willing to invest in tuning. Factors 2 and 5 are probably out of your control, unless you have access to a wide range of parallel computing platforms. Factor 3 is definitely beyond any programmer's control. That leaves factor 4, which essentially boils down to how much effort you're willing to invest in learning and applying parallel skills.

Is parallelism for you? Consider what you hope to gain-quicker access to results, ability to handle larger problem sizes, finer resolution, or increased complexity. Think about how much that gain will buy you in time or quality and what it's worth to you. Balance those considerations against the propensity your application appears to have for parallelism. Factor in the extent to which you think performance should pay off your programming efforts. Then take timings on a cleaned-up version of your serial baseline and use them to estimate the

best performance that could be obtained through parallelization. Assuming there are no counter-indications (such as a mismatch between your problem architecture and the type of machine available to you), parallelism will probably pay off if your upper-bound estimate on future performance is at least five to ten times bigger than what would be minimally worthwhile. Then factor in the extent to which you think performance should pay off your programming efforts.

Theoretically, any problem can be programmed in any language for execution on any parallel computer. Realistically, recognize that if a problem does not lend itself to parallelism, or if it doesn't match your computer's capabilities, parallelization simply won't be worth the effort.

References

Acknowledgments

These guidelines could not have been arrived at without the hundreds of experienced parallel programmers who collaborated in my surveys and interviews at computational science and engineering sites across the US, conducted on behalf of Intel Supercomputer Systems Division, IBM Corp., Convex Computer Corp., and the Parallel Tools Consortium, 1989-1994. Special thanks to Hugh Caffey (formerly at BioNumerik Pharmaceuticals, Inc., and now at Hewlett-Packard) and Rubin Landau (Department of Physics, Oregon State University) for their help in formulating and reviewing the rules-of-thumb.

Back to Pancake home page

Comments to pancake@.cs.orst.edu, 15 October 1996