Fundamentals and practicalities of MPP*

Greg Astfalk
CONVEX Computer Corp.
PO Box 833851
Richardson, TX 75083–3851
email: astfalk@convex.com

March 16, 1993

1 Prologue

Massively Parallel Processors, MPP in the sequel, are receiving a large amount of "press" recently. Many organizations are examining their future computations and computer purchasing strategies with MPP in mind. Well what then is MPP? Why is it getting so much press? What are the issues associated with computing on MPP? Will an MPP replace my existing computer? In this paper we explore each of these facets of MPP in some detail. Unlike much of the press, this paper will explore the key issues with *realism* in mind. The field of MPP is very broad, very deep, and *very* technical, thus we can not hope to cover it in great detail in this paper. Such an effort would require a tome of epic proportions. What we attempt is to discuss the most important issues in sufficient detail to reveal the fundamental nature of the problems and benefits. It is hoped that the reader will gain a sense of what is truly involved in dealing with MPP. The literature on the subject can then be read with a more balanced perspective. Rather than trying to explain each "new" item of jargon that arises in what follows, we provide a glossary of terms to which we refer the reader.

*To appear somewhere; probably in *The Leading Edge*

Although this author works for a computer company that is developing an MPP we strive to maintain objectivity. An advantage of working for a computer company is that our views are formed, and sharpened, by exposure to many users, applications, algorithms and codes. The goal here is not to sell CONVEX at the expense of "bashing" other vendors of MPP machines. When another vendor's product is mentioned it is to make a specific technical point.

2 What is MPP?

Before going forward we should define what it is we are dealing with. MPP from its very definition tells us that the processor is "parallel" and "massive." The "massive" means multiple processors (CPUs) within a machine. As to what number constitutes massive, we leave that definition until a subsequent section. The goal of the "parallel" is to have more than one of these processors working, simultaneously, on a single problem.

More precision is required in the definition just given. Assume we have 100 workstations in our organization and that they are all connected via a local area network (LAN). Does this collection of CPUs constitute an MPP? The answer is yes and no. If we program the collection of machines to meet the above mentioned criteria of working *in concert* and *simultaneously* on a *single* problem then, sure, it is an MPP of a sort. If each workstation is simply a computing appliance on an individual's desk and the LAN only furnishes a connection for email or NFS then we are not describing an MPP. While we don't say too much about it in this paper such a collection is being called a "distributed computer" these days.

The true MPP is a machine with a much tighter integration among the CPUs. In particular the CPUs are connected by a high-speed, low-latency interconnection and having a single operating system for all the CPUs. We will give more details on the specifics of MPP architectures in what follows.

3 Why MPP?

To set the stage for the remainder of this paper we want to point out the meaning of what is undoubtedly a new acronym to the reader; NWP. It

means; Nobody Wants Parallelism. Isn't this a bit of heresy coming from an employee of a company that is developing an MPP? Before answering that question lets state the corollary to NWP; EWP. This means Everybody Wants Performance. As contradictory as these acronyms sound they really do summarize the situation. We will be enumerating a number of reasons behind "NWP" in this paper.

Users have an essentially insatiable demand for computer resources to run larger and more complicated models in their respective fields. These fields could be chemistry, engineering, physics, mathematics, economics, and many others. What these end-users would really like to have is, (a) a machine that has a single CPU, (b) is "sufficiently fast," and (c) has "enough memory." This is what we'd call the *perfect* MPP[1]. Sufficiently fast means of course that it will run their existing, and planned, application(s) in a reasonable time. With problems getting continually larger and more complicated this means that the computer will need to get faster over time. The size of applications, in terms of the computer memory required, is growing larger with time. Larger dimensionality of problems, say from 2D to 3D, additional physics in the models, and the need for more accuracy all come at the expense of larger memory requirements and more powerful processors.

There are several reasons why the single-CPU machine cited above is "perfect." First, this architecture is very simple and well understood. Every student of a computer science course has enough of a grasp of this type of machine to understand how it works. There is another reason why users would define the perfect MPP to be a single CPU: it is easy to program. We have a forty year legacy of thinking about how to program computers that have a single CPU. Finally there is the tremendous inertia of the existing base of software. To paraphrase Carl Sagan, there are "billions and billions" of lines of existing code that were written with this sequential machine in mind. This body of code will not go away and it will not be re-written—at least not for many years. It is here for a long time to come and therefore must be dealt with by the MPP machines and the users of such machines.

We can assume the user has, and will continue to have, access to a current high-end supercomputer to run his application. To examine what is happening to supercomputer performance, we have plotted in Figure 1 the peak

[1]Note that this conflicts with our earlier definition of a MPP.

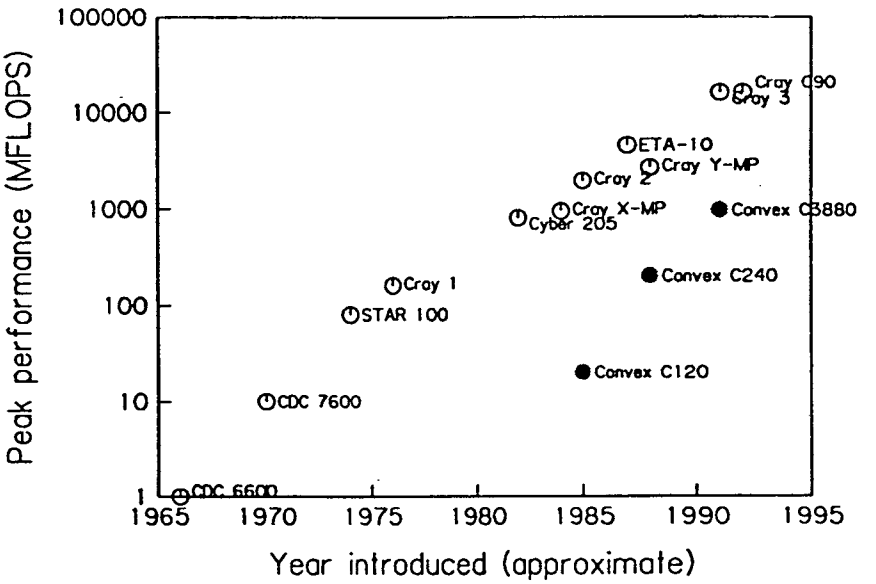

Figure 1: Peak performance, in MFLOPS, for various supercomputers of their day. Note that the vertical axis is logarithmic.

performance[2] vs. year of introduction for our collection of supercomputers *de-jour*. That is, each machine represented was considered to be the leading supercomputer of its time[3].

From a quick glance at Figure 1 the initial reaction is probably that there really isn't a problem with supercomputers meeting the growing demand for performance. It shows exponential growth. This is quite impressive—it appears that computing power, as measured by peak MFLOPS, is increasing many-fold per year. This trend has continued for the past 20+ years.

Figure 1 is a deceiving picture however. Some of the machines displayed in Figure 1 have more than one CPU. What we really need to do is replot this data with the effect of multiple CPUs removed. While they might not be considered massively parallel (we discuss that in more detail later) some do

[2]Peak performance is never achieved in practice. Another way to view peak performance numbers are; (1) the speed of the computer with no software on it, (2) the guaranteed never to exceed speed, and (3) the speed-of-light for this machine, remembering from physics nothing goes faster than this!

[3]The solid points are for the three generations of CONVEX; not the supercomputer of the day but, hey, this author works for CONVEX.

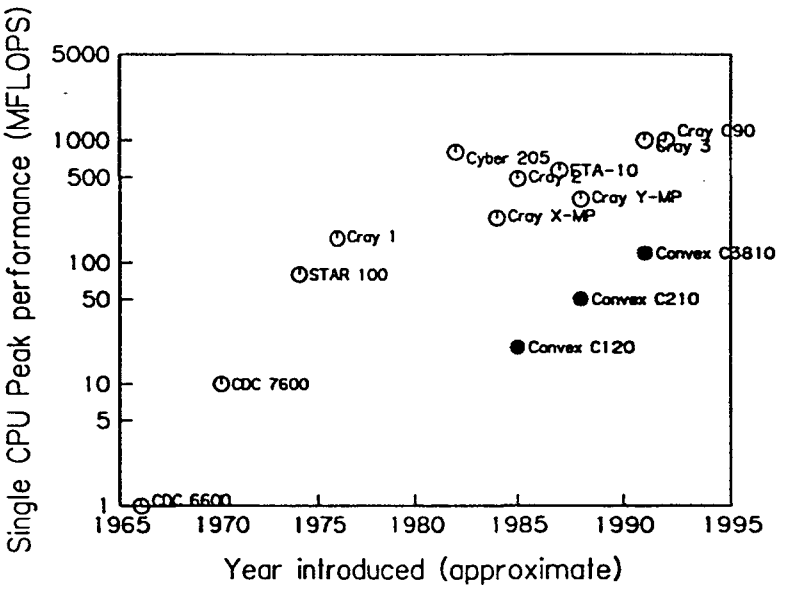

Figure 2: Peak performance for the individual CPUs of the supercomputers of their day. Note that the vertical axis is logarithmic.

have more than one CPU and are therefore parallel processors. In Figure 2 we take the exact same machines and simply plot the performance for a single CPU. (Note the difference in the scale of vertical axis between Figure 1 and Figure 2.) When we do this we can see that the peak performance for a *single* CPU has been leveling off since about 1986.

The unfortunate thing is that single CPUs are thus not going to meet the demand for computing power in the future. We will skip a long discussion about the specifics that inhibit it. Suffice it to say that, without a major revolution in computer design, *physics* will limit the cycle time of a CPU to approximately 1 nanosecond. It isn't that computer vendors don't want to make cycle times smaller but rather that the laws of nature become limiting. As a small taste of this point consider that the speed of light translates to 11.9 inches per nanosecond. The electrical pulses that move through the computer's conductor paths, if they are to convey information, are speed-of-light bound[4]. Conductor path length is then a serious consideration in

[4]We point out that electrical signals moving in circuit paths and wires do not move at the speed of light but some fraction of that. A close approximation would be 0.6 times

computer design these days. That's easy you say—make it smaller. Great, but then other physics enter the picture; heat generation per volume, lithography limitations, ballistic electrons, etc.

The cycle time of the fastest machines today is at 2–3 nanoseconds. Thus we are most likely within a factor of two or three of the "ultimate" in single CPU speed. The demand for resources is insatiable and it is common for users to seek a factor of 10 or 100 increase in computing power. It is likely that when they get that increase they will ask for yet *another* factor of 10 or 100! In view of this, a factor of 2 or 3 improvement in cycle time for a single CPU computer is insignificant.

Let's now discuss another aspect of the highest performance computers. Consider the case of the Cray C90. This is the highest performance supercomputer available today. For a 16-CPU system the cost is 25–30 million dollars. It has a peak performance of 16000 MFLOPS. Thus its price-performance is $1700 per peak MFLOPS. At the other end of the spectrum let's consider a Hewlett-Packard Model 735 workstation which costs $40000 and has a peak performance of 198 MFLOPS. Its price-performance is $200 per peak MFLOPS. Don't miss the point that although the supercomputers are poorer in price-performance then workstations they can handle the large problems that people need to solve, including the large storage, time-to-solution, fast i/o and multiuser throughput. The workstation is generally a single-user appliance with much lower capacities than the supercomputers. Vendor expectations are to have machines that offer price/performance of less than $100 per peak MFLOPS in 3–5 years.

Another aspect that is not obvious from Figures 1 and 2 is the development cost of these machines. In order to develop a machine that is at the cutting-edge of technology the design, development, and manufacturing costs are very high. Additionally the risk of a cutting edge technology not working correctly when first implemented is large. Certainly the high development costs are reflected in the cost of the machine (this is, after all, business). Compounding this is the pyramid effect; as the price of the computer increases (the height of the pyramid) the number of potential buyers of the machine decreases (the area of a plane through the pyramid). Consider the case of Cray Computer Corp. that is developing the Cray-3. At the expected price of this machine there are only a few sites world-wide that could

---

the speed of light.

afford to buy it. We leave the specifics to the interested reader to pursue via published information in the popular press.

Summarizing the points of this section we have the answers to the title of this section, "Why MPP?". Current computer designs can't progress fast enough to meet demand, there is a limit to the performance achievable from single CPUs, and very high-end machines are quite costly. The only logical path to the performance levels that some applications require is to bring multiple processors to bear on the problem.

4 What is "massive"?

For what follows we must define some terms that are often points of confusion. Most important is the definition of "massive" in massively parallel processor. The literature sometimes states that 10 or fewer processors is "modest", 10 to 500 is "moderate" and greater than 500 is "massive."[5] These exact numbers vary with author. We believe that the absolute value of the number of CPUs is not the *key* parameter in defining whether a machine is massive or not. If the machine in question has the property of being *scalable* then it potentially qualifies as an MPP. Granted, it is hard to argue that if your machine has only 2 or 4 CPUs, but is scalable, that it is massive. However we think it less important to focus on attributing a threshold number of processors in order to be considered "massive." The fact is that most MPPs in use today have 32 or fewer processors. Likewise if the algorithm in your application code used only 2 CPUs, is that a scalable and massively parallel algorithm?

Having said it, we should now define *scalable*. Per Webster's the definition is, "capable of being increased in a graduated series." We have seen others adopt the definition of scalable as a machine which is "field upgradable without significant changes to the operating system, as seen by the user." These do not quite suit our needs so we define, in the context of MPPs, scalable as, "resources can be added without introducing a fundamental bottleneck."

Let's consider a machine which connects all of its CPUs via a simple bus. Since this single bus must be arbitrated for, acquired, and then used by an individual CPU it introduces a bottleneck in that only one CPU can be utilizing it at a time. The effective bandwidth of the bus, as perceived

[5] Steve Wallach, the founder of CONVEX, refers to greater than 1000 processors as "masochistic."

by any one CPU, is reduced by a factor of $1/p$, where p is the number of processors. Such a machine is not truly scalable.

This brings up the question about whether today's classic, high-end supercomputers are scalable. The answer is no. Such machines generally rely on a crossbar to connect the CPUs to the memory. Crossbars, to not be a bottleneck, need to sustain a bandwidth that can allow all the CPUs to operate at peak performance by supplying as much data as is required by all of the CPUs. This implies that the bandwidth of the crossbar must be able to increase in proportion to the number of CPUs, each of which has very large bandwidth requirements. This isn't possible. This is not the same as saying that today's high-end machines aren't good; it just states that their architecture is not scalable to large[6] numbers of processors. This is another perspective that leads us to believe that MPP is the future architecture for high-end computing.

There is a lot of work to be done in order to get users and applications to MPP computing. Overtly focusing on how many processors are necessary is distracting. It is foolish to think that we can jump from today's state of 4–8 processors to thousands of processors in one leap. The evolution, and the increase in the number of processors, will occur over time at a pace dictated by technical developments, not the hype of the popular press.

5 MPP architectures

There are currently a number of different architecture types in the field of MPP. A classification of the types of architectures would help to make discussing the MPPs easier. The best place to begin is with the so-called Flynn taxonomy[22]. This is a grouping of computers into four classes. The classes are identified by acronyms. They are:

- SISD—Single Instruction stream, Single Data stream
- SIMD—Single Instruction stream, Multiple Data stream
- MISD—Multiple Instruction stream, Single Data stream
- MIMD—Multiple Instruction stream, Multiple Data stream

[6]Our specific definition of "large" is a bit nebulous. Current high-end shared-memory machines have at most 16 CPUs. Given this, then 32 CPUs looks like a "large" number.

The SISD class of machine is the most common one and is in fact the class that encompasses workstations, mainframes, VAXs, etc. Deciphering the acronym we have that there is a single stream of instructions that are indexed by the program counter. The data that goes to the processor is fed through a single channel or path from memory or cache. This machine is doing one thing at a time[7].

SIMD is a more interesting class. This machine has a multitude of processors. We have a single stream of instructions, that are indexed by a single program counter, for all of the processors. There are separate streams of data which are fed to each of the processors. All of the processors are executing the exact same instruction, in lock-step, but each is operating on its own data. The classic paradigm for this type of machine is the Connection Machine. This computer can have up to 64K processors. SIMD machines support an approach to parallelism that is called "data parallel." As a good example of an application for this type of architecture consider that we have a picture composed of a number of pixels, say n_{pix} of them. For some image enhancement task we need to perform some set of n_{ops} operations on each pixel in the picture. We "assign" each pixel of data to a processor in the SIMD machine and then the set of operations (*i.e.*, instructions) are applied to all of the pixels, that is, the entire picture, simultaneously. Thus we reduce the time required from $n_{pix}n_{ops}$ for a SISD machine to n_{ops}. This is a very idealized comparison but illustrates the tremendous potential of data parallel computing. A general characterization of the SIMD class of machines is that it has a large number of processors with each individual processor being "weak" in computing power.

As an aside we point out that the vector supercomputers are also in the SIMD class. This is a result of the fact that a single *vector* instruction causes multiple pieces of data, the vector of data, to be operated on. The glaring differences between a vector processor and the Connection Machine is one of the reasons that the Flynn taxonomy has been criticized as being too restrictive[34]. To further confuse the issue consider that the vector processors of today, such as the CONVEX machines, when they have multiple processors, are rightly considered to be in the MIMD class (we describe this

[7]This is a bit of a misnomer. There are many levels of parallelism within even the SISD machines. Our comment applies to the machine executing a single instruction stream at a time.

shortly). So we can have a machine that is a hybrid of sorts!

The MISD class is, to this author's knowledge, empty. No commercial machine has been produced that has multiple instruction streams and a single data stream. If the reader knows of one please let us know! While this author was giving a lecture some years back at the DoD a member of the audience pointed out that in his opinion the DES (Data Encryption Standard) chip is an instantiation of a MISD computer. This author has never bothered to pursue the suggestion, but it sounds plausible.

The most interesting class of MPP machine is the MIMD class. In this class the machines have multiple streams of instructions and multiple streams of data. Thus we can have different processors doing different things simultaneously and working on independent data. In this class we find, in principle, the most generality. As a specific example of this class we can use the Intel iPSC/860 Hypercube. For this machine there are a number of processors (in powers of two) that are connected in a hypercube fashion via its interconnect. Each processor of a MIMD machine has its own memory[8] and is thus a complete computing resource. A process running on this computing resource has a memory space that is confined to the memory associated with that processor. Likewise the other processors in the machine have their own memory and processes running on them have an address space that does not include the memory space on the other processors. Each of the multiple processors can be executing a completely different set of instructions (for example different subroutines or programs). This is the multiple instruction stream portion of the MIMD class. Since the memories are distinct the data that each processor is operating on is different for each processor. This is the multiple data stream part of MIMD. Another perspective might be that the processors in a SIMD machine are synchronous, those in a MIMD machine are operating asynchronously.

We can make a further division of the MIMD class into the message passing sub-class and the shared-memory sub-class. In the message passing sub-class, of which the Intel iPSC/860 described above is a member, if any of the processors need to share or swap data this can only be accomplished by explicitly passing a "message" over the interconnection media. There simply is no other way for the processors to communicate data or synchronize with each other. The act of passing a message between processors takes a long

[8]Well sort of! Read on.

time relative to the processor speed, something like $\mathcal{O}(1000)$ processor cycles. This can be considered a (relatively) high latency interconnect.

In contrast the shared-memory sub-class within the MIMD class is characterized by just that, shared memory. Here we have some number of processors, but rather than each having its own separate memory there is a single common memory. All the processors have equal, and unbiased, access to all of the memory. Intervening between the processors and memory is some form of switch. This is often in the form of a cross-bar. Neglecting second-order effects any processor can get any memory location in $\mathcal{O}(10)$ processor clock cycles[9]. This is a low latency interconnect. An example of this subclass is given by the CONVEX C-series machines. Here there are a few processors, up to 8, that share a common memory of up to 4 Gbytes. The connection between the processors and the memory is provided by a fast cross-bar. Each of the processors could be executing a separate set of instructions and operating on different data. Note that this is true in spite of the fact that there is only a single common memory for all of the processors.

A general characterization of the MIMD class of machines is that there are a smaller number of processors, relative to the number in the SIMD machines, but each processor can be quite powerful.

Before leaving this discussion we offer in Figure 3 another way to look at the architectures in the Flynn taxonomy. In this figure we show, on the left-hand side, the streams entering the machine. On the right-hand side are the stream(s) leaving the machine. The "+" and "-" are symbolically the instruction stream(s) while the "a,b" and "c,d" are the data streams.

6 TFLOPS and Tbytes

The popular press, largely motivated by the government's High Performance Computing and Communication (HPCC) initiative[29], has popularized the notion of a TFLOPS (Trillion FLoating-point Operations per Second) computer with Tbytes (Terabytes) of memory. One of the thrusts of the HPCC is to have a machine with a TFLOPS of performance and a Tbyte of memory, for solving the so-called "Grand Challenge" problems, available by the end of the decade. The Grand Challenges are problems such as global climate

[9]Actually if the referenced memory locations are contiguous then the access time is $\mathcal{O}(1)$ cycles

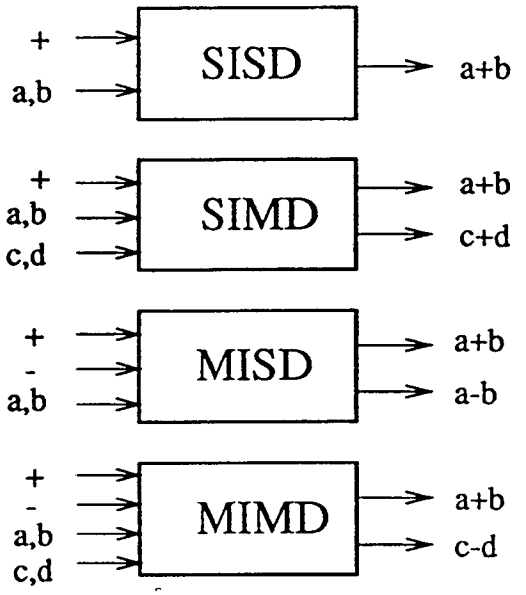

Figure 3: A "data flow" representation of the classes of machines according to Flynn's taxonomy.

modeling, characterizing the human genome, turbulence modeling, quantum chromodynamics (QCD), and many others. In Figure 4 we show how these problems compare to the computing power that is available today. The HPCC is not just the Grand Challenge problems but in many people's minds, and in the popular press, they are synonymous.

All the Grand Challenge problems are characterized by requiring computing resources in the TFLOPS and Tbytes range, but this is not to say that other problems can't use such resources. There are a lot of other problems, many of keen interest to industry, that could in fact use such a powerful machine.

TFLOPS and Tbytes are so large that we should put these numbers into perspective. The few fastest applications today are running, on the fastest machines, at 1–10 GFLOPS. This is still a factor of a hundred to a thousand away from a TFLOPS! Having said this let's consider what it would take to make, using today's technology[10], a machine that has a TFLOPS of peak

[10]Assuming 200 MFLOPS, 50 watts and 5000 dollars per processor and 1 watt and 150 dollars per Mbyte. We also need to add in some money for the interconnect but who knows

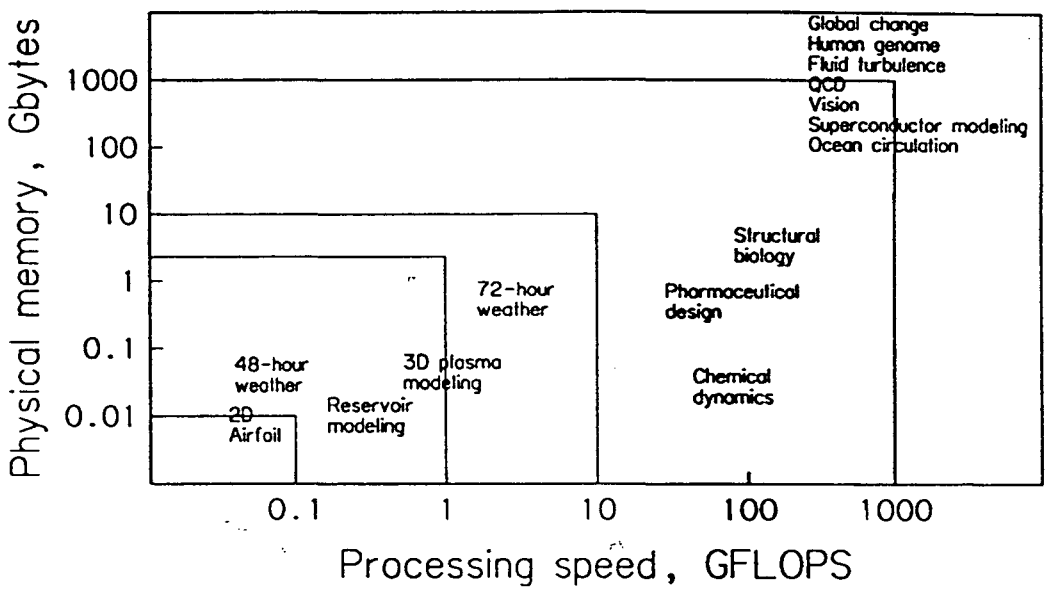

Figure 4: A graphic image of the Grand Challenge problems. Note that they generally require computing resources on the order of a TFLOP with physical memory in the Tbyte range. Today's high-end machines are in the 1 GFLOP and 2 Gbyte range.

performance and a Tbyte of memory. This machine would;

- have 5000 processors,
- require 1.3 Mwatts,
- cost 200 million dollars,
- fill 7000 cubic feet ($8 \times 30 \times 30$) and,
- need one tremendous air conditioner.

This certainly seems like an intractable piece of hardware! As we will illustrate in a subsequent section the *peak* performance of this machine would be a substantially larger number than the *sustainable* performance on a real application. Therefore to get a TFLOPS on a real problem would require an even larger machine.

To get a sanity check on the size of a Tbyte of data consider the familiar 2400 foot 9-track magnetic tape. Written at the highest possible density this tape holds approximately 150 Mbytes of data. To get a Tbyte of data onto magnetic tape would therefore require 6,667 such tapes. This is a pile 486 feet high—nearly the height of the Washington Monument! Granted there are other tapes that have larger capacity and are smaller in size. We aren't discussing tapes here, rather we are simply trying to set the notion of the size of a Tbyte of data into perspective.[11] Do people really envision dealing with Tbytes of data? Yes, in fact they do. It is also true that having a Tbyte of physical memory does not imply that the input and output datasets will be of that size. However, as we go to longer computations and more complicated models we will be producing datasets of increasingly large size.

7 Technology advancement

The rebuttal to the machine described above is that the underlying technology of computer hardware is advancing at a rapid pace. We are the first to acknowledge this. Making some assertions about the rate of advancement is fairly straightforward based on historical data. We choose to first look at what has been happening to memory. Figure 5 illustrates the historical trend

what that would cost. These numbers could be argued about but we need not quibble over even a factor of two here.

[11] For the Unix literate reader; Can you imagine the size of the core file that can be produced by a single floating-point exception on a machine with a Tbyte of memory?

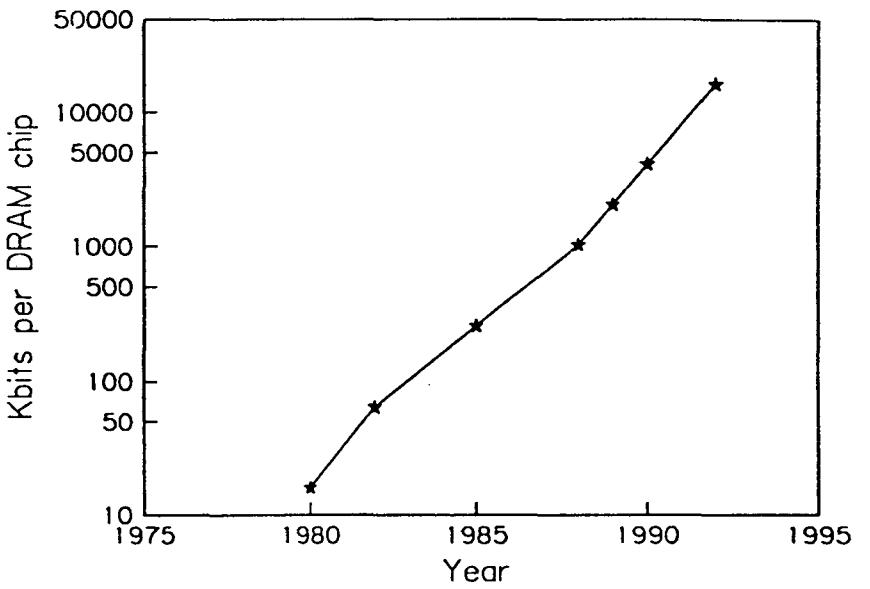

Figure 5: Increase in the per chip density of DRAM over time. Note that the vertical axis is logarithmic.

for the density (bits per memory chip) of DRAM. This shows an impressive factor of 1000 increase over the past 12 years. DRAM chips are, of course, the basic building block of the memory of today's computers.

The increasing density of memory chips is well noted in the press and by computer users. Some consider it the barometer of circuit technology. What we do in Figure 6 is to show, for the same period of time used in Figure 5, the decrease in the access time for DRAM chips. This shows only a factor of 4 improvement (*i.e.*, decrease) in access time over the past 12 years. As we will show later it is the memory access time that is the key to achieving high levels of performance on supercomputers and MPP.

Given the previous two figures one could argue that the good news is that things are getting better. However the spectacular gains in DRAM density are not accompanied by a commensurate drop in access time. A long debate could be developed around the relative importance of density vs. speed for DRAM—we chose not to do that here. The point of these figures is that we should not, *carte blanche*, believe that all the advancing computer technology will directly translate into deliverable performance.

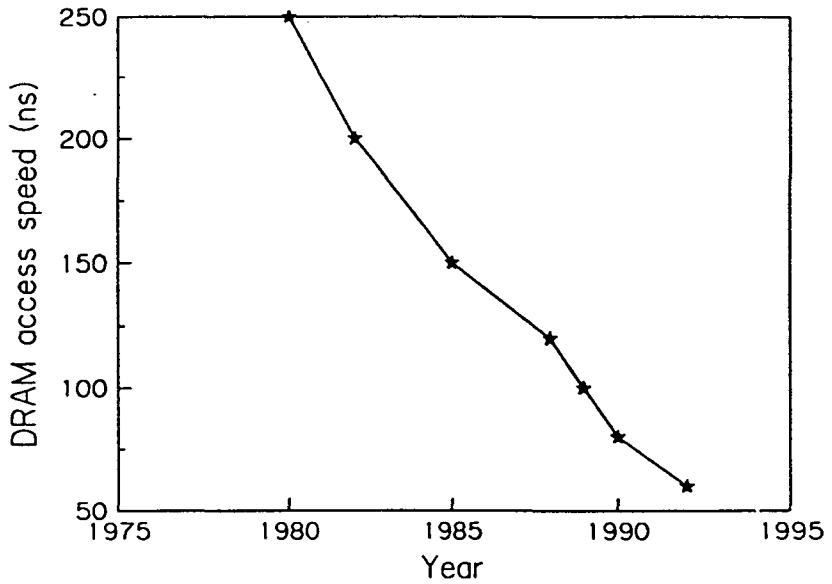

Figure 6: Historical behavior of DRAM access time over time.

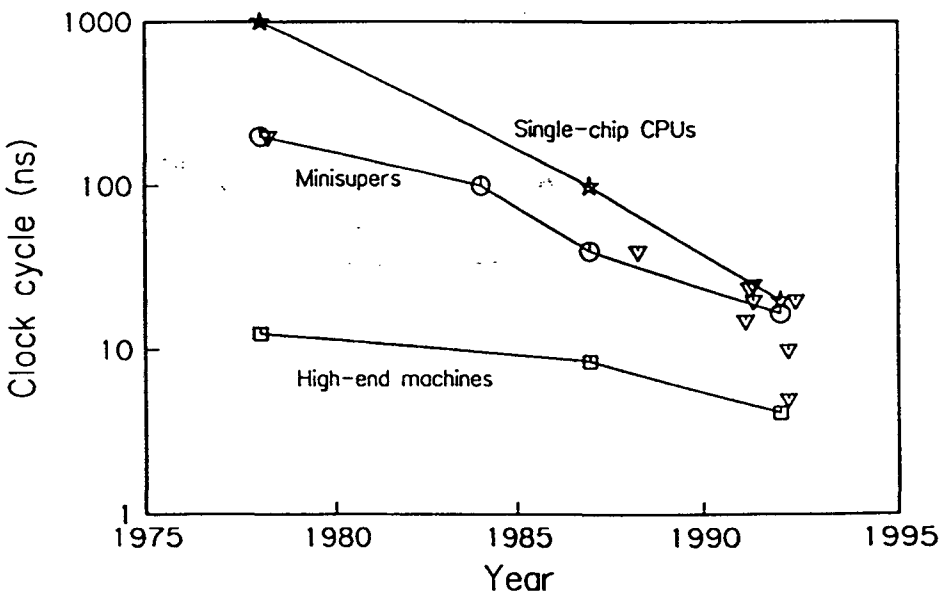

Figure 7: CPU cycle times for several "classes" of computers. We have also added a number of points for various types of processors (as shown by the inverted triangles) to add more substance to the plot.

To further emphasize this point about technology advancement we, in Figure 7, plot the behavior of the clock cycle for three distinct classes of computers over time. We can see that the high-end machine's cycle time is not progressing all that quickly! Note that the cycle times for all three classes appear to be converging to a limit. This is a reinforcement of the point made earlier; physics is limiting high-end processor performance. The spectacular gains in the single chip CPU's cycle times, while dramatic, are rapidly approaching the same physics-based limit that the high-end machines are bumping into. Again the message is that we need to be cognizant of the truth about the "spectacular" gains that computer technology will be offering us in the future.

We now have a sufficient amount of data to make a point about the interaction of memory and processors. Hopefully it is evident that the processors need to get their instructions and data from memory. There needs to be a close interaction between the memory itself, the processor and the connection between them. If we consider the single chip CPU performance growth against the performance growth of memory access time we find that there is a growing disparity. Computer designers go to great lengths to overcome the disparity between processor speed and memory speed in the overall computer design. The supercomputers of today all use highly interleaved memory, a mechanism that "hide's" the latency between memory and processor for certain patterns of memory access. Much of the cost of a high-end machine today is in the memory sub-system. Single chip CPUs will need to face this issue also.

We should know that the peak performance numbers for CPUs are very dependent upon the CPU working under highly favorable conditions. Real application codes seldom offer such an ideal. Most of the single-chip CPUs are based on RISC (Reduced Instruction Set Computer) design principles. They all have a cache that intervenes between the processor and the memory. The cache provides high bandwidth and low latency to keep the processor fed with instructions and data. This is another way of hiding the disparity we mentioned earlier. When a RISC engine is not operating out of its cache its performance is reduced by a substantial factor. We could go on further with this point but the message is that peak performance numbers are always difficult (impossible?) to achieve in real-life codes. Computer technology advancement is always measured by peak performance. Users must judge their actual benefit by sustainable performance on their application; these

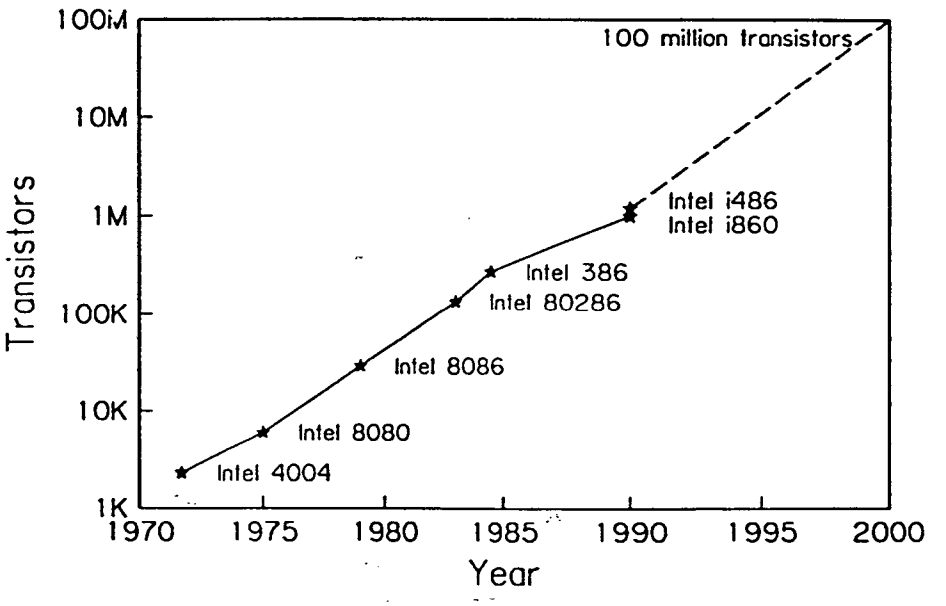

Figure 8: The progression in the number of gates on a single-chip processor over time. The dashed part of the line is (obviously) an extrapolation. The trend is clear however.

are two different numbers and it is only application performance that really matters!

We close this section with this a more favorable piece of news. The chip manufacturers are making dramatic progress with respect to the number of gates that are being placed on a single chip. We look at this for a particular manufacturer, Intel, in Figure 8. The benefit to the user is that the increased gate count on a single chip allows for placing more logic closer together, that is without suffering the large off-chip signal delays. In recent history this has allowed the size of "on-chip" caches to grow larger. In the future the increased gate count might translate into multiple CPUs on a single chip. Both these approaches will help increase the performance that the user can extract from the underlying hardware.

8 The problems with MPP

By anyone's measure MPP is not considered mainstream computing today. This is not due to a lack of technology; there are approximately a dozen vendors who are selling MPPs in various flavors. Why hasn't this technology caught on and become mainstream? The reason is simple—there are difficulties associated with it.

We could easily enumerate a long list of problems related to MPP's acceptance by the computing community but instead let's focus on what we consider as the "top five" reasons. In a bullet list, in no particular order, our choices are;

- third party software packages
- porting effort
- software (in particular, compilers)
- latencies
- data decomposition

Before discussing each of these points in turn we make the observation that if these problems had viable solutions today then MPP would likely be as commonplace as vector processors. We know this since the adoption of vector processing went through a similar cycle and similar problems. Vector supercomputers weren't readily accepted in the mid-seventies for some of the same reasons we listed above.

In the following sections we will be addressing these problems and hopefully shed some light onto the prospect for their resolution. The problems listed *are* solvable; the issue is when, to what degree, and at what cost.

9 Third-party software for MPP

Third-party codes are those codes that are written by some organization, other than the computer manufacturer, and subsequently sold, or leased, to an end-user organization. An organization that requires the functionality of a particular code simply buys, or leases, it and saves the time and effort of writing the equivalent code internally. The third-party vendor has the task of porting his code to several host machines so that it can be sold to many organizations which are undoubtedly using different vendor's hardware.

Many of the users of third-party codes possess little or no knowledge of the host hardware. The computer is simply an engine that runs the code. The end-user need not know, or be concerned with, the questions of architecture, compilation, tuning, optimization, etc. The vendor of the software does, the user does not.

We should not underestimate the importance and wide-spread use of third party software. We at CONVEX know that the majority of the cycles of all of our machines are spent within third-party codes. (We have a belief that the same is true of the other supercomputer vendors—we know that the market segments they sell to are very similar to those we sell to[12].) Hopefully, without general disagreement, we can accept that third-party software is important.

In the context of MPP there is a "Catch–22" taking place. The first point is that if many of the third-party codes in use today were already ported to MPP machines then many more MPP machines would be sold to the end-users of these packages. Remember the end-user of the code only requires that the code run and that its performance is acceptable for the problems to be solved. If this were to happen then MPP would be legitimized in the market place. Has this happened? Definitely not! Without too much exaggeration it is safe to say that none of the really important, widely used, third-party software packages is available on any MPP today.

[12]If we compare a pie chart of the market segments we sell into vs. the overall high-end computing market segments, as reported by DataQuest, they are almost identical.

The other piece of this Catch–22 we are describing involves the third-party vendor. The third-party vendor is a business. They rightly seek to make a profit and be commercially successful. They achieve this by selling their software. The third-party codes are almost always significant pieces of software. This happens since, if the code were trivial, the end-user organizations would write it themselves rather than buy it. To compete with other, similar, third-party vendors each must offer a larger feature set than the competition. The code must be easy to use so this requires user interfaces. The end result is a large package. This has significance as we shall see in a moment.

Adopting the perspective of the third-party vendor and looking at the MPP market we are faced with a dilemma. Relatively few MPPs are being sold today, there are quite a few different varieties, and several of the hardware vendors are in jeopardy of failing. The third-party software vendor needs to decide whether it makes good business sense to port their code to an MPP. This takes a large amount of human resources and expense. Is there the prospect of getting a return on this investment? Not with the number of sales of MPP machines today! Even if there were, what then is the "architecture of choice" in the MPP field? With a number of architecture choices the third-party vendor is faced with a decision about which machine to port to. The changes to the algorithm/code are generally going to be vendor or architecture dependent so the choice must be a wise one. The prudent third-party vendor knows that at some point in the future a port to MPP will be necessary, but for now it is best to wait to see how the technology and marketplace shakes out. By waiting until a future date the ease of porting to the MPP machines is enhanced. This will reduce the time and expense of the port to an MPP when it is actually done.

So there we have it. A classic stand-off with respect to third-party software that is not really anyone's fault. It is very important in closing this section for the reader to know that the third-party code users account for a major portion of the cycles on all high-end machines today. To ignore this group of users will likely cause MPP vendors to fail commercially. Until some event breaks this cycle MPP will not really be accepted as mainstream computing. Given sufficient time the normal evolutionary course of events will resolve this. We, and probably the end-users of application codes and third-party codes, think that is too long of a time to wait.

10 Porting effort

In a subsequent section we will spend a great deal of time discussing latency. For the moment let's simply state that the latency is the time delay between a processor asking for a piece of data and its actually receiving it. With this definition in hand we can (glibly) state a phenomenological law[13].

$$PortingEffort\left[\frac{1}{latency}\right] = constant \tag{1}$$

To make this equation more clear consider that today's vector supercomputers have latencies that are $\mathcal{O}(10)$; any processor can get any piece of data in memory, neglecting second order effects, in approximately ten machine cycles[14]. This plus the effort to port to this machine can approximately define the constant in the equation. Now as we move to machines that have higher latencies we will experience a larger porting effort (as required to maintain the value of the constant in the face of the decreasing value of $1/latency$).

The porting effort we allude to includes several aspects; the application, algorithm selection, coding, tuning and testing, and debugging. Stated another way: as the latency gets smaller the end-user needs to worry less about the architecture of the machine in order to get performance. So what? Well MPP's have, and will continue to have, high inter-processor latencies. After we discuss latency in more detail in a subsequent section we will return to this issue of porting applications. Suffice it to say here that you simply can't take an application from a vector shared-memory multiprocessor and simply recompile on an MPP with good performance as a result.

The MPP machines that are in general use today offer the prospect of a difficult port. As an example, those machines that are strict distributed memory designs present to the user a programming model that requires the explicit use of message passing. This invariably requires a lot of additional and architecture specific code in order to make the port successful. There is another down-side in that the code can no longer be debugged on a workstation or the non-MPP platform; we don't mean the multi-processor debugging

[13]Such a law is one that is based on observation of a phenomenon rather than derived from first principles of physics.

[14]If the operations we are considering are vectorized then the latency is $\mathcal{O}(1)$ machine cycle.

but rather the sequential flow of the program's logic. A question that ought to be faced is: what use is TFLOPS of performance if it is going to require TUCPAs[15] to achieve? Machine cycles are cheap compared to user cycles.

11 Compilation for MPPs

We now turn our discussion to the area of compilation of high-level code. This is a subtle way of saying Fortran, and to a lesser degree, C. Users of high-end machines write Fortran—period! Better languages for parallel coding have been proposed, and some are available, but they are seldom used today. If we are honest about what most users of high-end machines are doing in the course of their work on an application they are thinking, editing code, compiling, running and analyzing results. This process repeats until the code is producing correct answers and is also running at an acceptable level of performance. If you read the press it is easy to believe that users spend a vast amount of time fine-tuning code and developing new algorithms for the host architecture on which they are running. Having the benefit of working for a computer company and interacting with a great many users, and users' codes, we can assert that this is a misconception. The "meat and potatoes" of getting performance from high-end machines is simply compiling Fortran and C and accepting the performance offered by the auto-parallelization from the compiler.

There is a class of users, the "bleeding-edge", who do whatever is necessary to get performance from a new class of machines. They also are among the first to acquire new computing technology when it becomes available. The efforts and findings of the bleeding-edge users are quite important to the evolution of new computer technology. Much of the contributed literature on MPP today comes from this group. They are however a distinct minority in the overall community of users of high-end machines.

The generic user simply writes codes to solve the problem at hand using the same style and methods to which they are accustomed. It is surprising (perhaps even disconcerting) how little regard is given to writing efficient code. This is not a criticism of these users; their interests are in solving problems in their respective fields, which is not necessarily computer science

[15]Another new term! TUCPA is this author's acronym for Tera User Cycles Per Application.

or computer design. What currently makes this less of a problem than it sounds is the ability of today's automatically vectorizing compilers. While these compilers can't optimize everything, they often transform poorly written code into code which achieves an "acceptable" percentage of the peak performance of the machine. This is a result of three factors. First, the scope of the optimizations that the compiler must perform are confined to DO loops and DO loop nests. With this restricted scope the compiler can automatically vectorize a substantial number of loops. Second, the latencies are of $\mathcal{O}(1)$ on the vector machines which allows the user the luxury of not needing to be all that careful in the coding style and algorithm selection. Finally there is the mature and aggressive nature of the auto-vectorizing compilers today.

It is an accepted notion that the transition from sequential to vector computing is much easier then the transition from vector to massively parallel computing. A sense of perspective here is that vector computers first appeared in the mid '70s, and as of today, there are still a large number of published articles that detail better vector algorithms. If this is a base-line case then the transition to MPP codes and algorithms, which is at least twice as difficult and probably *much* more, will be taking place until at least the year 2030!

The literature, the practitioners, and the researchers in compilation know that the MPP compiler, to automatically deliver high performance, has a much more onerous task than the compilers of today. The first thing that we must accept is that optimization(s) based exclusively on DO loops or even DO loop-nests will *not* produce high efficiency on MPPs. Recall that the goal is to spread the computational work across multiple processors. With the attendant higher latencies and coupled with the relatively small granularity of the DO loops this becomes a fruitless path. A fundamental difference is that the compilers for MPP will need to have a much larger scope in their optimizations.

The compilers that are in use today can be characterized as procedure-based compilers. This means that when doing a compilation the unit on which the compiler operates, and all that it can "see", is a single procedure or subroutine at a time. This restrictive view limits the optimization that can be automatically performed to the context that is within this procedure. In Fortran terms this means that only DO loops and DO loop nests, without embedded procedure calls, can be optimized. Here we take optimized to mean parallelized. What is necessary to find the coarse-grain parallelism in a

user's code is a compiler that has a larger scope when doing its optimizations (*i.e.*, parallelizations). In fact this scope should be the entire application code. This means that the compiler must to be able to do interprocedural optimization (IPO).

IPO simply means that the compiler can see across procedure boundaries when it is looking for opportunities for optimization and in particular, parallelism. One of the goals of an interprocedural compiler is to be able to automatically parallelize procedures and procedure call trees. This is the level of granularity that is needed in order to produce good performance on an MPP. An advantage of IPO is that it also holds the prospect of doing automatic data decomposition[16] of arrays within the user application code.

There are several parts to IPO. First is simply the idea of carrying optimizations across procedure boundaries. A simple example of this is shown in the code fragment in Figure 9. A procedure-based vectorizing compiler would always be required to compile the loop in subroutine `foo` as scalar since there is a possible recurrence depending on the value of `i`, one of the arguments and an offset in the array on the left-hand side of the loop body. The procedure based compiler has no knowledge of what the value of `i` could be so it must be conservative to insure correctness. An IPO compiler would see that the first call to `foo` has a recurrence since it also sees the value of `i` as -1. Likewise, in the second call to `foo` it sees that there is no recurrence. An IPO compiler can then perform a technique called procedure-cloning where it generates object code for two versions of subroutine `foo`. One version will have a scalar loop and the other a vector loop. The correct variant is called at run-time to achieve the highest performance.

Another aspect of compilation via IPO for an MPP is called regular section analysis. In this the compiler can quantify a use-def chain or def-use chain for a section of an array in subroutines or subroutine call-trees. Considering a 2-dimensional array, then a regular section could be a set of rows, a set of columns, or a rectangular section of rows and columns. Having this knowledge for various calls to the routine it can then perform Boolean algebra on the array sections. If the intersection of the sections is null then there are no data dependencies on these sections and those subroutine calls can be performed in parallel. This will allow the possibility of parallelizing

[16]We discuss data decomposition in detail in a subsequent section. At this point it is sufficient to note that this *is* important for performance.

```
i = -1
call foo ( x , n , i )
j = 2
call foo ( y , n , j )


subroutine foo ( a , n , i )
real a(n)
do j = 1,40
  a(j) = a(j+i) + ...
enddo
```

Figure 9: Example of code that is amenable to optimization via IPO. Note that the first call to `foo` would require its loop to be run in scalar mode while the second call to `foo` *could* permit a vectorized loop.

subroutines and subroutine call-trees automatically. Note that this is not possible with a procedure based compiler.

It is not the expectation that IPO will always work by automatically delivering perfectly parallelized code to the user. However, it is expected that over time as IPO matures and is enhanced that it could possibly approach the level of automatic vectorization today. Without IPO there is little hope of automatically getting large speed-ups from Fortran on any MPP.

Developing an IPO is not an easy task. The complexities far exceed that of a procedure based compiler. CONVEX has had an IPO based compiler as a product on its C-series machines (shared-memory vector multi-processors) for several years. This compiler is called the APplication Compiler (APC). The APC currently does interprocedural optimizations but does not yet do interprocedural parallelizations. We have at this point approximately 100 person-years of effort in the APC. This has been spent in working up to the point that all of the necessary pieces are resident to enable automatic interprocedural parallelization. What the compiler needs to accomplish is an understanding of the control and data dependencies for the entire application. Data dependencies for scalar variables is being accomplished right now. Data dependencies for arrays will be achieved via regular section analysis in a

subsequent release. The Application Compiler will be one of the compilers on our MPP. We think that IPO will be a requisite technology for all MPP compilers in the future.

12 A model MPP and application

Before moving to a more specific technical discussion of the latency issue on MPP we establish some terminology. Let's define a model MPP that we will use for the sake of the current discussion. We consider this to be a set of computing resources that are connected to each other in some fashion. We refer to the medium that connects them as the interconnect. The exact details of the interconnect are not of consequence, nor is the exact topology of the connected computing resources. We could use Ethernet, a multi-stage switch, a routing backplane, etc. The topology could be as simple as a bus or as complex as a hypercube and it doesn't, in any substantial way, change our argument. We have a generic MPP here; multiple, connected computational elements that can cooperate on the solution of a specific problem.

Again without being specific we consider that we have some application that we need to run on our model MPP. It is obvious that we want to take our application and have these multiple computing resources operate concurrently on its solution. This implies that at least some of the data will be spread across the various computing resources. Having done this (we don't need to say how we did it), and noting that this is a single application, it is almost certain that the various pieces will need to communicate with each other to share data or to synchronize their efforts. Thus it follows that the interconnect will need to be used by the application; this is obvious we suppose. As we will illustrate, it is this use of the interconnect that is a major problem that inhibits performance on an MPP.

We assumed that we have some "interesting" application. This means that we don't have an embarrassingly parallel application that can be divided among the computing resources and the concurrently executing parts never have a need to communicate. It is true that such applications exist and are not just toy problems. It is our experience that they are a very distinct minority of the problems and algorithms that are in use today. So our model MPP and application will be doing some communicating over its interconnect. This is the prelude to the discussion in the next section.

13 MPP latency

Since this section will be detailing the effect of the interconnect latency define it as;

> the time delay between a computing resource requesting a piece of data and actually receiving it.

Clearly our goal would be to have low latencies; high latency is the bane of performance. It should be clear that latency is not necessarily related to bandwidth. We could have a high latency, high bandwidth connection or a low latency, low bandwidth connection or any other combination. Much of the literature talks about high interconnect, or bi-section, bandwidth, generally measured in Mbytes/sec. We hope to show that it is, for the current state of code/algorithms, the latency that matters most.

We have found the following model of the communication time on an MPP to be useful in our studies of the theoretical performance of algorithms. Consider

$$t_{comm} = t_s + W \cdot t_w + H \cdot t_h \tag{2}$$

Here t_{comm} is the time to move some quantity of data from some portion of the MPP memory to a processor. t_s is the indivisible start-up time which is strictly overhead. t_w is the time it takes to move a floating-point word (64-bits in our discussions) and W is the number of such words being sent. t_h is the "hop" time taken in passing through an intervening node in the routing path between the source and the destination. H is the actual number of hops made in the route. Examining this equation we can see that t_s is a "latency-like" term and t_w is a "bandwidth-like" term. The presence of the t_h term is the motivation for MPP topologies that have a small mesh diameter (see the glossary). With the routing technologies available today the value of t_h is relatively small. Even for large machines H isn't huge. So for simplicity of this exposition we decide to ignore this term in the ensuing discussion.

From the open literature, [14] [21] [23] [46] we construct Table 1 which lists the communication parameters for various machines. We include t_c, which is the cycle time for the processor used in the machine. By examination of the relative values of t_s and t_w in Table 1 we see that $35 \leq t_s/t_w \leq 250$. From the form of the first two right-hand terms in Equation 2 it is evident that

Communication times in μseconds					
Machine	CPUs	t_s	t_w	t_h	t_c
iPSC/860	128	136	1.60	2.00	0.0250
Delta	512	50	0.32	0.05	0.0200
Paragon	2048	10	0.04	0.03	0.0125
nCube-2	8096	60	1.60	2.00	0.0500
CM5	1024	88–350	0.17–0.4	0.50	0.0330
KSR	1088	6.7–30	?	?	0.0500
FastMesh	2048	5	0.04	0.01	0.0030
BigCube	65536	20	0.40	0.60	0.0500

Table 1: Communication times for several existing MPP machines. t_s is the communication start-up time, t_w is the time to move a 64-bit datum, t_h is the time to move a datum through a router, and t_c is the cycle time of the processor. The two machine shown in italic are expected technology for the 1995–1996 time frame.

unless W is large the dominant contribution to t_{comm} is from t_s. This is the latency issue. Our early studies at CONVEX (not yet published) have shown that, for today's codes, W is generally of $\mathcal{O}(1)$. It is the goal of vendors and high-end computer users to make the required movement of data to be small relative to the amount of computation and to get W large when such movement is required.

Our assumption is that a computing resource will need the requested data before continuing its execution, but recalling our definition of latency, it will need to wait for it to arrive. It is true that, while it is waiting, the computing resource *could* (in principle) be off doing other things, but that is a separate discussion which we don't enter into in this paper. To avoid the waiting we would like the communication time to be comparable to the compute time. The truth is we would like to have zero latency; then a processor would never be awaiting required data. The finiteness of our physical world precludes that however.

In Table 1 we have included two hypothetical machines that are expected to be available within the next 3–4 years. The *FastMesh* machine is an ex-

trapolation of the current class of MIMD machines. In *FastMesh* we have a relatively modest number of processors with each being quite powerful. *BigCube* is a hypercube topology that is similar to the architecture of the Intel hypercube but with a much higher dimensionality and faster processors. Relative to *FastMesh* the *BigCube* machine has weaker processors but a substantially larger number of them. Even here we have that t_s/t_w is $\mathcal{O}(100)$.

It is helpful if we cast the latency as the ratio of time it takes to move a single datum (say a 64-bit floating-point number) between computing resources to the time it takes to do a floating-point operation, say addition or multiplication. We denote this ratio the latency ratio, L_r. Note that L_r depends not only on the speed of the interconnect for moving data but also on the processor speed. We claim that, neglecting second-order effects, this ratio is $\mathcal{O}(10)$ for the shared-memory vector supercomputers of today. We state this to show the contrast against what L_r is for the MPP machines of today which from Table 1 is $\mathcal{O}(1000)$.

The obvious question at this point is; what is a good value for L_r? The only correct answer is that the lower the better. The physical problems of the construction of an MPP dictates that it will be larger than the more common tightly coupled shared-memory machines. We take the view that the latency ratio is *the measure* of the overall applicability of an MPP. The lower the latency ratio the more applications, or existing code, that can be profitably mapped to the machine, for a given level of porting effort. Keep in mind the earlier discussion on DRAM speed and processor speed; their respective growth rates are actually exacerbating the latency problem.

To quantify the effect on performance of this latency ratio we will look at two "models" of parallel computing. One is a specific model based on divide-and-conquer with a recombination of data. The other is an empirically derived model that is based on a collection of application codes.

For our first example assume we need to add a very long list of numbers; n of them. We neglect the task of getting the numbers to each of the processors in the first place; we'll assume that they are already in place. We simply take n/p of them on each of the p processors[17] and form the partial sum on each processor. We need to form the entire sum however. This requires that the partial sums computed in each processor will need to be summed. We can

[17]For the sake of simplicity of exposition we assume that p evenly divides n. Assuming otherwise doesn't change the results in any substantial way.

do this by allowing pairs of processors to communicate their partial sum to one of the pair where the partial sum of the pair is then computed. We can do this for each pair of processors simultaneously. In the next step pairs of the processors that are holding the partial sums from the first step communicate with each other to add their contributions. This continues until the entire sum is produced on a single processor. Let's consider the case of $p = 8$. The first step will have the pairs, (p_1, p_2), (p_3, p_4), (p_5, p_6), and (p_7, p_8) communicate their partial sums. Now the odd processors receive the even processor's sum and do the addition. Now in the second step we have the pairs (p_1, p_3), and (p_5, p_7) operate pairwise with, say, p_1 and p_5 receiving the partial sums. Finally we have (p_1, p_5) form their sum in, say, p_1. Note that this took 3 stages. This can be generalized for any number of processors (neglecting some second-order effects) with the result that it requires $log_2(p)$ stages. We can then generalize our specific example with the following equation;

$$T(p,n) = \left\lceil \frac{n}{p} \right\rceil \cdot t_c + log_2(p) \cdot t_{comm} \tag{3}$$

The first term is the time it takes to add the n/p numbers on any one of the p processors (remember this is being done concurrently on all the processors). The second term is the time it takes to communicate the partial sum for a processor to any other, which we also assume happens concurrently. In discussing this equation we will consider that the $log_2(p)$ term applies to the communication of the partial sums between processors. We will simply neglect the addition time associated with each communication since it is, in essence, a single floating-point operation.

To show the effects of L_r on this example we need to define a measure of how well our parallel algorithm is working. A common measure is the speed-up. The speed-up is, in physical terms, how much faster the application runs on p processors when compared to running on a single processor. Letting $T(p,n)$ denote the time an application of size n takes on p processors then the speed-up, S_u, is simply

$$S_u(p,n) = \frac{T(1,n)}{T(p,n)} \tag{4}$$

Another term that we will use later is the efficiency. This can be interpretated as the *average* portion of time that a given processor is doing

useful work. Its definition is simply the speed-up divided by the number of processors.

$$\epsilon(p,n) = \frac{S_u}{p} = \frac{T(1,n)}{p \cdot T(p,n)} \tag{5}$$

Applying Equation 4 to our specific example we have

$$S_u(p,n) = \frac{n \cdot t_c}{\left\lceil \frac{n}{p} \right\rceil \cdot t_c + log_2(p) \cdot t_{comm}} \tag{6}$$

Rewrite this slightly to incorporate L_r

$$S_u(p,n) = \frac{n}{\left\lceil \frac{n}{p} \right\rceil + log_2(p) L_r} \tag{7}$$

If we examine this equation it is clear that if $n \gg p$ then a speed-up of p is *approachable*[18], unless the communication costs are high. Note that the communication costs are only invoked $log_2(p)$ times.

To be specific we show, in Figure 10, the results of this two-parameter model. Letting $n = 10^6$ we show a surface generated as a function of p and L_r. If we look along the back right edge of this surface, $L_r = 0$, we see the expected perfectly linear speed-up in the number of processors. This is obvious from the fact that $L_r = 0$ causes the second term in the denominator of Equation 6 to be identically zero. This is, of course, the best it can be. Now consider the case where $L_r = 1000$ (the front edge). Using 256 processors with this level of L_r produces a speed-up of only 84! We apply 256 processors and this algorithm only runs 84 times faster. Should we be disappointed? Certainly, but this is what life can be like in MPP computing. This case has a very fine-grained parallelism but it also has a low amount of inter-processor communication. Again we caution the reader that this result is for a specific algorithm, with a specific value of n and p. It is not correct to apply this as a general finding regarding speed-up of all applications on any MPP.

[18]This is as good a place as any to talk about ***super-linear speed-up***. A number of papers claim to achieve speed-ups of greater than p on p processors, *i.e.*, super-linear speed-up. When this occurs it is a result of an interaction between the particular algorithm and the fact that it was running on a collection of processors. From a strictly architectural view it is *impossible* to get a speed-up that is larger than the number of processors.

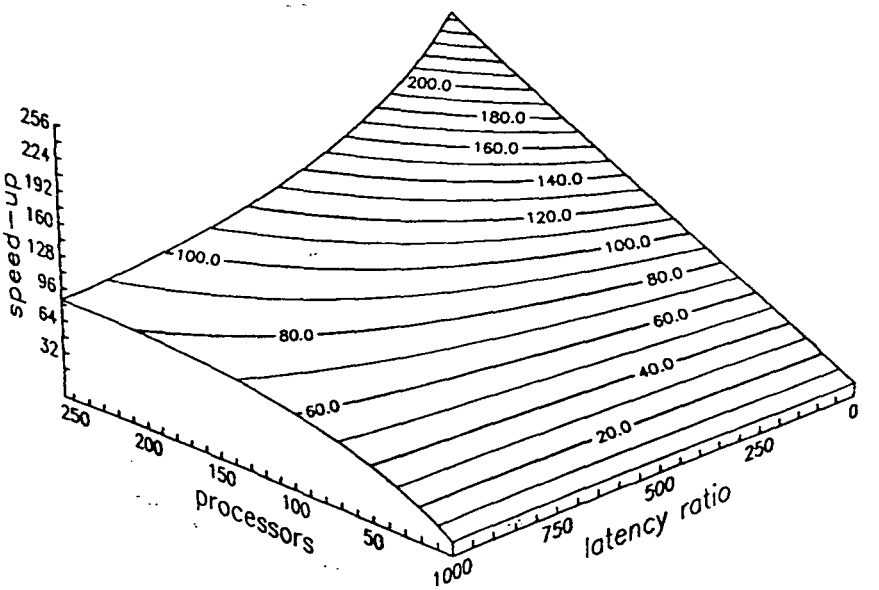

Figure 10: The effects of communication latency on the two-parameter model. We have labeled the axis for L_r as "latency ratio." The isoclines are lines of constant speed-up.

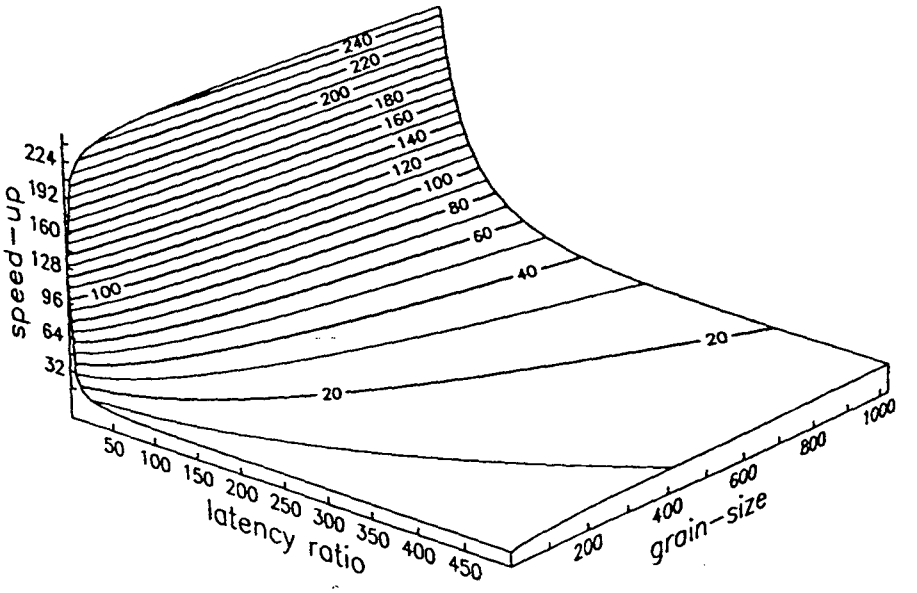

Figure 11: The effects of communication latency and grain-size on parallel speed-up.

Let's take a different look at this same issue of parallel speed-up and latency. In this case we make use of a phenomenological model developed by Fox[24].

$$S_u = \left[\frac{p}{1 + \frac{1}{m^{1-1/d}}}\right] L_r \tag{8}$$

This equation shows the speed-up for p processors with each processor having a grain-size of work whose size is given by m. The variable d is a measure of the "dimensionality" of the problem. This is a small integer generally related to the physical dimensionality of the problem. Finally there is the latency ratio, L_r. We plot a surface of this function in Figure 11. We can see the rather strong dependence of the speed-up on the latency ratio. We should also note a weaker dependence of speed-up on the grain-size. If we look at small granularity, say $m = 50$, we see the very rapid decay in speed-up as L_r becomes larger. For $L_r = 100$ we have a speed-up of only 17! There is some modest improvement if we consider $n = 500$. Here we have, for $L_r = 100$, a speed-up of 46. Still not very impressive for 256 processors.

From Table 1 and its associated discussion and the two previous examples we see that the latency has a strong effect on the achievable performance on large numbers of processors. To offset the damaging effects of high latencies we need to have very large granularity, or low communication, in the algorithm. This can be stated succinctly as;

> The Holy Grail in computing on an MPP is to achieve the highest possible ratio of computation to communication.

By doing this the detrimental effect of the interconnect latency is minimized. The efforts of code tuning and algorithm replacement in today's and tomorrow's codes will be directed to this point. We mention again that in some cases additional computations could be done by the algorithm and still be the "correct" way to proceed so long as the extra computations are done in parallel. Remember it is time-to-solution that matters.

14 Data decomposition

Remembering that latency is a damaging feature of the MPP we can point to an approach that will work to reduce its impact. The way to achieve this is to tune or change the application, algorithm, or code to get the highest possible ratio of computation to communication (recall our earlier statement of MPP's Holy Grail). Generally the amount of computation (the computational complexity) for a particular application is fixed by the nature of the algorithm used. As an example an n-point FFT requires approximately $n \cdot log_2(n)$ operations. Doing the FFT on an MPP doesn't necessarily change that computational complexity. There are instances where porting an algorithm to an MPP results in more computations being performed[19] but they are "masked" by their simultaneous execution. Given that the number of operations is "fixed" by the problem and algorithm one recourse is to minimize the communication. This can be achieved through a combination of the algorithm we choose to use and by placing the data in the "correct" place before the computation begins. The intent is to position the data once, *a priori*, so that a certain amount of computation can be done on the data without its movement during the computation. The notion of data decomposition is

[19]This is acceptable if, and only if, it results in a shorter time-to-solution.

closely related to the idea of divide-and-conquer and domain decomposition. Data decomposition can generally only be done with knowledge of the underlying algorithm or when a compiler having IPO can deterministically detect it.

To make the point more quantitative; we'd like our MPP application to perform, say, several million floating-point operations and then simply send a single floating-point number to some other processor. This would be wonderful! The single "high" latency operation of communication is amortized over several million floating-point operations. At the other extreme if we were truly unlucky then each piece of data needed for any floating-point operation would need to be moved over the interprocessor interconnect. Assuming even a modest latency ratio, then our application would run far slower on the MPP.

Support in existing languages for data decomposition is non-existent. On the MPPs of today it is done by using explicit message passing. Ken Kennedy at Rice University developed a set of extensions to the Fortran language to allow for the *a-priori* placement and alignment of data in the memories of distributed memory machines. This research work was called Fortran-D. The "D" in Fortran-D can be interpretated to mean either "data", "distribution" or "decomposition." The goal was to enrich Fortran with extensions so that the user, having knowledge of the application or algorithm, could instruct the compiler how to decompose and distribute the data arrays for the problem. The decomposition takes place in three steps. First the user defines a template called a `decomposition`. This is much like an array but does not allocate space. The decomposition is an abstraction on which the actual arrays will be placed. The second step is the alignment of an actual data array onto the template, *i.e.*, the decomposition. This is accomplished with the `align` statement. Here is where the actual data array is partitioned. The alignment could be done, as an example, by rows, columns or blocks. The final step that is required is the placement of the data onto the space of processors. This is done with the `distribute` statement.

The use of Fortran-D allows the user the latitude to place the actual data among the processors in the fashion which is best for the calculations that will be performed. It should be obvious that the end result is to get the required communication between processors (*i.e.*, distributed memories) to be as small as the application will allow. Another point to note is that Fortran-D is not an automatic scheme. The effort of applying the Fortran-D

constructs to the application code is performed explicitly by the user.

The wide-spread interest in the functionality that was offered via Fortran-D initiated the formation of the High Performance Fortran Forum (HPFF). HPFF is a group of approximately forty representatives from academia, industry and computer vendors. They are working to define a set of extensions to the Fortran language to define the so-called High Performance Fortran (HPF). Version 1.0 of the draft standard of HPF is, as of the date of this paper, under public review. The draft can be obtained electronically from Rice University by anonymous ftp on `titan.cs.rice.edu`. After the review period it will be a "standard." We need to emphasis that HPF is not a public standard such as the IEEE or Mil-Spec standards. Notwithstanding it will be accepted by the community at large as a de-facto standard.

HPF will be a minimal subset of extensions that will enable users to perform data decomposition and data distribution on high performance architectures and to program within a data-parallel programming model. Since it will be a specification that all the vendors will adhere to the goal would be that HPF would be portable across a multitude of hardware platforms. The focus of these extensions is to give the programmers the ability to position the data of the application across the memory hierarchy of a MPP. It also provides for a richer parallel language than can be obtained from Fortran-77 alone. This will apply regardless of the specifics of the underlying machine. Therefore distributed memory machines and shared-memory machines, as well as SIMD and MIMD machines, could be addressed with a common language.

A full implementation of HPF requires all of Fortran-90. In the short term, several years at least, most implementations will be based upon the official HPF subset. This requires Fortran-77 as the base language with the array syntax of Fortran-90 and the Fortran-90 `interface` blocks.

Many of the extensions to Fortran being developed by the HPFF are in the form of directives. This enables the code to be "clean" in that it could still be run, unaltered, on workstations or machines that don't support the HPFF extensions. HPF requires the use of a `forall` statment to allow for very general loop-independent spreading of work. Some new intrinsics, reduction operators, and prefix and suffix operators are also defined. There is an expectation that HPF will become a de-facto standard in the future.

It should be mentioned that many applications do not run well when there is a single, fixed data decomposition. Rather these applications will

require (perhaps) many redistributions of the data during execution. This greatly complicates performance issues.

It may not be apparent to those not versed in the MPP field but this subject of data decomposition is one of the most important aspects that needs to be dealt with. If there is one thing that a user should wish for it should be the ability of a compiler to do this task robustly and automatically. If the data is positioned, by that we mean distributed, in a nearly optimal fashion by either the user or the compiler then the performance that will be achieved will be dramatic compared to a non-distributed version of the code. Do not dismiss the importance of data decomposition; it will be receiving a lot of attention by the research community and it will be a "buzz-word" for MPP users in much the same fashion that "unit stride" is to the vector machine users of today.

15 Amdahl's Law

We *must* spend some time to elucidate Amdahl's Law[3]. There isn't any new technical innovation in what we present. We simply make some points that are often ignored, overlooked or not understood when Amdahl's Law is cited; and it is cited *often*! We consider a simple example to set the stage for what follows. Consider we have an elaborate application code that models the growth of an elephant's toenails. During the course of its execution 50% of the CPU cycles are spent in data input, preprocessing (setting up the initial geometry of the toenails!), postprocessing and data output. This is very necessary to the application but it is all sequential processing. The remaining 50% of the CPU cycles are in the parallelizable, computational section[20]. To be more specific we state that the run-time on a *single* CPU of our MPP machine is 350 seconds. From our specification it is obvious that 175 seconds are spent in sequential code and 175 seconds are spent in the parallel code. Allowing, for simplicity, that the parallelizable portion is perfectly so, (*i.e.*, it exhibits exactly linear speed-up) we then run on two CPUs. This reduces the 175 seconds in the parallelizable computational portion to 87.5 seconds (175/2). Of course this leaves the sequential portion unchanged. So

[20]This kind of ratio is not at all unusual. It is true that many applications are dominated by the time in the so-called solver but an even division between the computational kernel and the rest of the code is seen quite often.

CPUs	Speed-up	Efficiency
2	1.333	0.6667
4	1.600	0.4000
8	1.778	0.2222
16	1.882	0.1176
32	1.939	0.0606
64	1.969	0.0308
128	1.984	0.0155
256	1.992	0.0078
512	1.996	0.0039
1024	1.998	0.0020
2048	1.999	0.0010

Table 2: Speed-up and efficiency with increasing numbers of processors for an application that is 50% parallelizable according to Amdahl's law.

our speed-up is (recall Equation 4) then 1.33, $(350/(175 + 87.5))$, with an efficiency, Equation 5, of 0.67. Consider now using 4 CPUs, this gives a speed-up of 1.6 and an efficiency of 0.4. If the pattern isn't clear yet, we can go further and place more results for this particular example in Table 2.

Hopefully now the pattern is clear—the largest speed-up we can possibly see for this particular example is going to be nearly 2.0. What is restricting us is the sequential portion of the code. Even if the parallel portion could be done infinitely fast, that is to say performed with a *very* large number of processors, there is still 50% of the time, 175 seconds in our specific example, that can only be done by a single processor. The efficiency is, of course, getting quite low as we add processors. Thus we are under-utilizing the computing resources that are being applied to the problem.

Armed with this simple, but illustrative, example we can now develop the general, and often cited, statement of Amdahl's Law. We consider an application that has a certain fraction of parallelizable code; we denote this fraction, f_p. The remaining portion, $f_s = (1 - f_p)$, is the sequential code. With no loss in generality we can allow the total execution time (sequential and parallel) to be 1. It should be clear that the parallel portion will be run

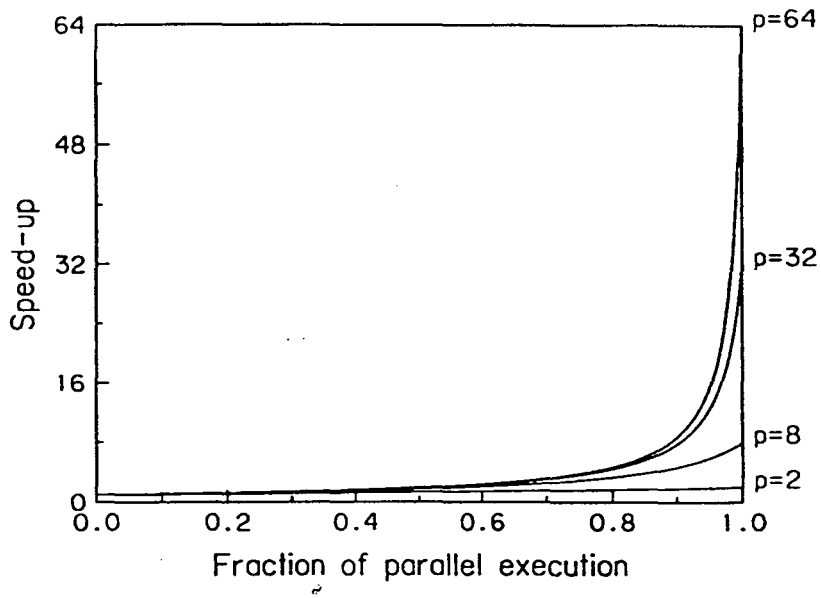

Figure 12: Speed-up predicted by the original version of Amdahl's Law. We show here the behavior for 2, 8, 32, and 64 processors.

on multiple, say p, processors so the parallel time is given as f_p/p[21]. Then applying this to the equation for speed-up yields Amdahl's Law;

$$S_u = \frac{1}{(1-f_p)+\frac{f_p}{p}} = \frac{1}{f_s+\frac{1-f_s}{p}} \tag{9}$$

Now we take a look at a plot of Amdahl's Law. Obviously the value of f_p is bounded as $0 \leq f_p \leq 1$. The lower bound of $f_p = 0$ represents a completely sequential code and the upper limit, $f_p = 1$, represents a perfectly parallel code. In Figure 12 we show the speed-up vs. the parallel fraction as a number of curves representing different values of p, the number of processors.

The obvious features of this plot are that at $f_p = 0$ we have a speed-up of exactly 1, regardless of the number of processors. At $f_p = 1$ we have a completely parallelizable code and the speed-up is identically p. What is troubling about the prediction of Amdahl's Law is that we require a very high fraction of parallel code in order to get close to a speed-up of p, on p processors. This phenomenon gets further exacerbated as we consider larger

[21] Using f_p/p assumes perfectly parallelizable code.

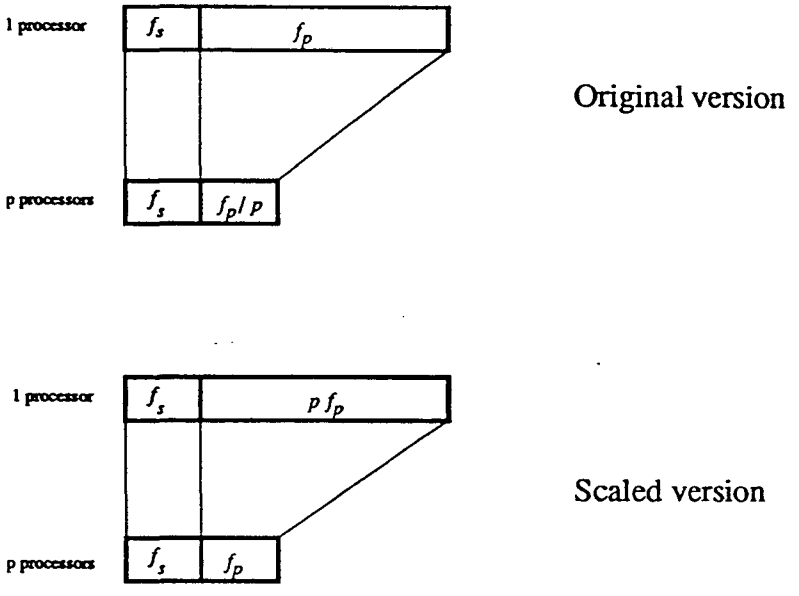

Figure 13: Graphic representation of the two versions of Amdahl's Law. In the first (original) case the problem size and ration of sequential and parallel work is fixed. In the (scaled) version we start with a problem whose parallel fraction is scaled according to the number of processors we will apply.

numbers of processors. For all practical considerations, at anything much greater than about a hundred processors the speed-up curve is so hopelessly steep that the requirement is for 99+% parallel computation. The bottom line is that we would need an unachievably high percentage of parallel execution to get respectable speed-ups. This is the "curse" of Amdahl's Law which we debate shortly.

Recent work has resulted in another view of Amdahl's Law[30]. A graphic depiction of the two versions is shown in Figure 13. The newer version is often referred to as the scaled version. What is different here is that as more processors are added we apply it to larger problems. In the original version the amount of parallelizable work remained constant in size as we added processors. In the scaled version if we double the number of processors we double the problem size. The *assumption* of the scaled version is that the sequential portion stays constant when the problem size is increased. The mathematical statement of the scaled version of Amdahl's Law is,

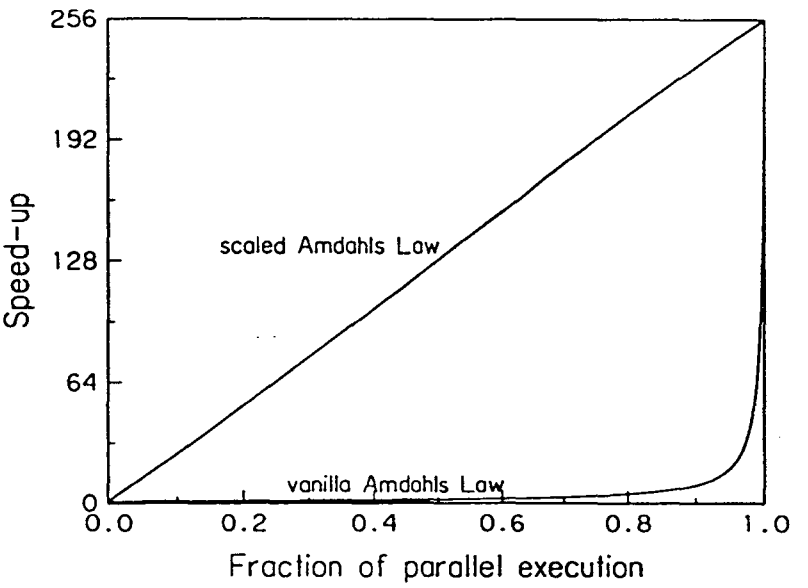

Figure 14: A representation of the difference between the original version of Amdahl's Law and the scaled version. This is for the specific case of 256 processors. Note the much more favorable behavior of the scaled version.

$$S_u = f_s + p \cdot f_p \tag{10}$$

Let's see what this looks like compared to the original version of Amdahl's Law. We do this, for 256 CPUs, in Figure 14. Note that the scaled version of Amdahl's Law is nearly the same as a linear speed-up. Is this real? Well if we accept things at face value it is. What we need is a detailed discussion of the issues surrounding both versions.

From Figure 14 it is clear that the two versions are substantially different and give us a completely opposite perspective on the prospect for efficient MPP computing. We talk first about the original version. It is not apparent that we would leave the parallel version of the code at the same size as we added processors. More likely is that the problem size would be made larger when more processors are applied. Further, it is unlikely that our parallelizable work will speed up perfectly. This ideal speed-up would be so seldom encountered that it is not worth consideration. Therefore the original version is a bit unrealistic to the pessimistic side.

For the scaled version it isn't clear that we could arbitrarily increase the size of the problems in proportion to the number of processors we add. For the scaled version of Amdahl's Law, as well as the original, the constant size of the sequential portion is in general an idealization.

In our studies we have found that neither version applies perfectly in practice. Very often the sequential portion's computational complexity grows in relation to the problem size. It is true that it almost always grows at a slower rate than the parallel portion. The rate of growth of the sequential code is very problem dependent and often a result of how much effort is put in tuning the code or replacing the algorithm. The growth rate of the parallel portion of an application is the aspect of the scaled version of Amdahl's Law that is most troubling. Much of the computational complexity in the core of the computational portion scales as some power of the problem size. The addition of processors only helps linearly at reducing the solution time so that at some increased problem size the time-to-solution becomes intractable. Different applications will have different scalings for the sequential and parallel sections of the algorithm. For this reason either form of Amdahl's Law must be considered an ideal and not necessarily a harbinger of what you can expect from your application.

Having the benefit of working with and talking to practicing end-users of high-end machines we have come to realize that there are two "biological" time-constants in computing. The first is an amount of CPU time equal to "overnight." This is a function of when the person goes home, the load on the target machine and the time the user gets to work. Loosely speaking it is 12–16 hours. Few people are willing to suffer under the burden of only getting a result every few days. The second constant is something like an hour or two, *i.e.*, an extended coffee-break. This allows for doing several computations during the course of the day. We see users adjusting the problem size to fit within these time-frames.

We have the view that the original version of Amdahl's Law serves as a lower bound and the scaled version serves as an upper bound. As the EPA always says, "...your mileage may vary."

16 MPP performance

Reflecting back to the first example in the latency section it could be argued that it was far too simplistic a model and that the second was far too general. We now address a very specific "application."[22] The most popular and often cited benchmark is the LINPACK benchmark[19]. The algorithmic operation in this benchmark is the LU factorization of a dense, unsymmetric matrix. The computational complexity of this is $\frac{2}{3}n^3 - \frac{1}{2}n^2 - \frac{1}{6}n$, where n is the order of the matrix. To avoid confusion we point out that there are several variants of the LINPACK benchmark. The most common is the so-called 100×100. This does an LU factorization of a matrix whose order is 100. In this test the furnished Fortran source code can not be touched by the benchmarker. The achieved performance is due only to the hardware/compiler combination. In the 1000×1000 version of the LINPACK benchmark obviously the order of the matrix is 1000. More importantly however is that the organization running the test is free to; (a) write the code in any language, and (b) use any algorithm so long as the matrix is factored and solved. It is almost always the case that the LINPACK benchmark numbers cited in the literature were produced by the vendors of the cited hardware. For the 1000×1000 case we can then be assured that the hardware vendor will have tuned this benchmark to the best extent possible. We could fairly think of the numbers published in the report as the LINPACK 1000×1000 "speed-of-light"; it is unlikely that a customer would ever be able to get better performance for this particular problem!

Before getting to the results of this benchmark we need to point out that this is a "synthetic" code. That is, it does not do any input nor does it do any output. Also the small amount of preprocessing, the matrix generation, is not timed in the benchmark. What this means is that this is *not* at all like a real application code. It is also a relatively small sized problem. We must be careful in projecting these results onto a real-world application code. An application code that includes the LINPACK factorization will not do as well in overall time-to-solution (which is the important measure) owing to the omissions mentioned above. To be truly fair we also need to

[22]We would like to use real application codes here but the truth is that there aren't any to use. For the few that are running on MPPs there would only be a limited amount of data. This dictated our "choice" of LINPACK.

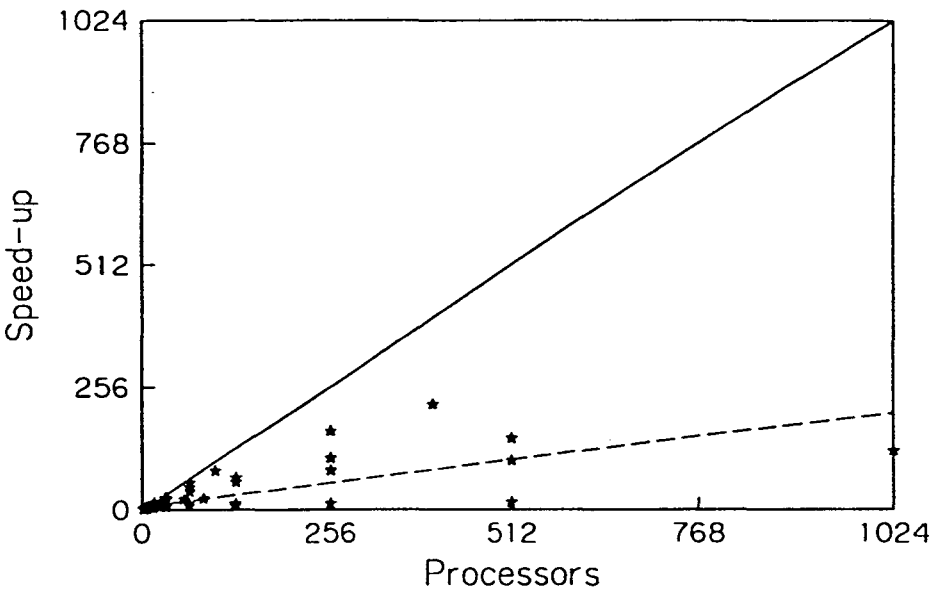

Figure 15: Speedups for 129 computers on the LINPACK 1000×1000 benchmark. The solid line is linear speed-up. The broken line is a least-square fit to the actual data.

emphasize the point that when discussing performance on MPPs we need to give consideration to; (a) the application, (b) the algorithm, (c) the coding, (d) the problem size and, (e) the hardware. In spite of this, the results from the LINPACK 1000×1000 benchmark will tell us something, but we should not view the results as an overall testimonial on MPP performance.

What we have done in Figure 15 is to plot, as discrete points, the speed-up achieved on 129 (different) computers. We say different parenthetically since many are the same underlying hardware with only a different number of CPUs being used. Further, the data includes both shared-memory and distributed memory machines. On the horizontal axis we represent the number of CPUs used. The vertical axis is the speed-up. The solid line indicates linear speed-up; recall that we cannot exceed this. The broken line represents a least-square fit to all the data points; the slope of which is 0.19. Thus for the case of 1024 processors, the speed-up is 194. Is this disappointingly low? The answer to that really depends on your perspective of MPP.

Another point we can take from Figure 15 is that there are no data points

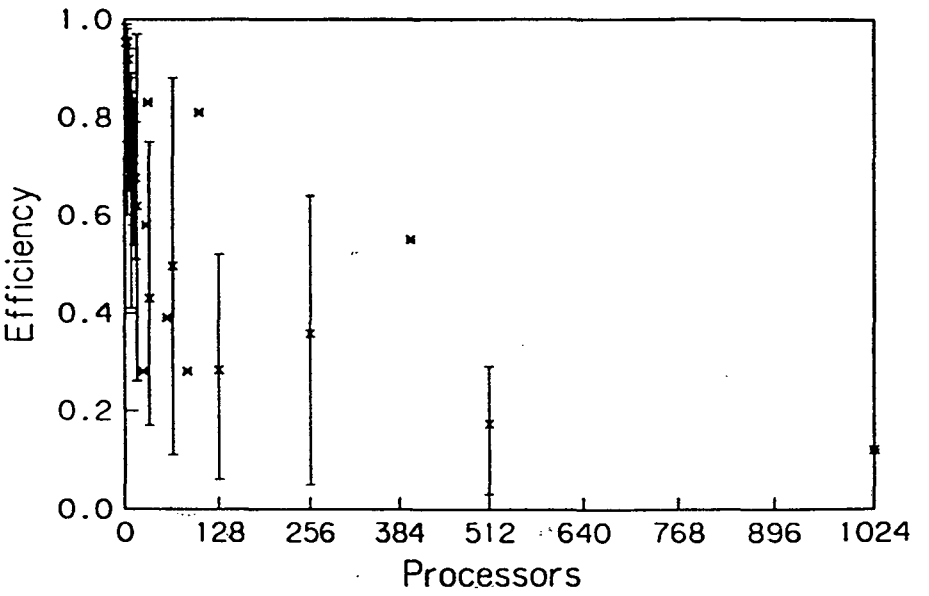

Figure 16: The *average* efficiency for 129 computers on the LINPACK 1000 × 1000 benchmark. For all the computers having the same number of processors we averaged their respective efficiencies. The average is plotted as an "x" and there are vertical bars from the minimum to maximum values.

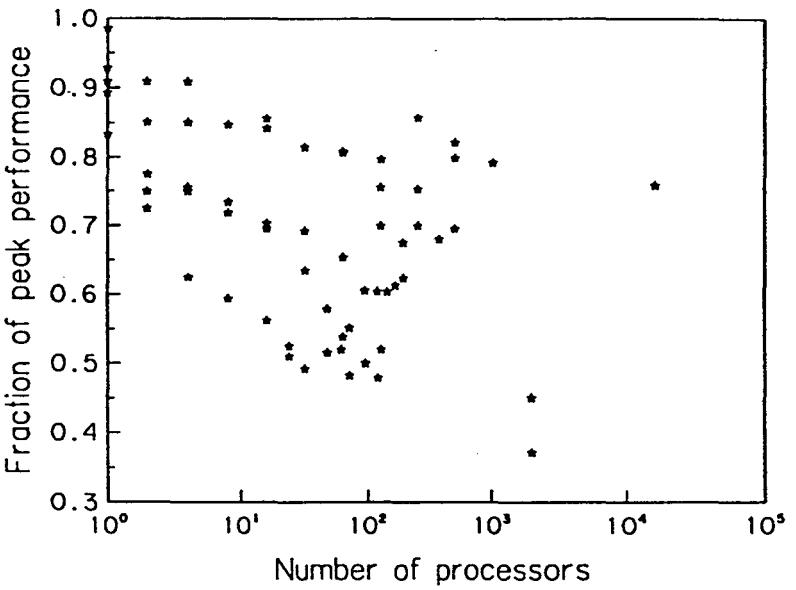

Figure 17: Percentage of peak performance for 64 cases of the LINPACK benchmark. In this instance the problem size is made as large as can be made to fit on the computer in order to allow the machine ample opportunity to get close to the highest achievable performance.

for machines with greater than 1024 processors. Remember that earlier we pointed out that it will take several thousand processors to achieve a peak TFLOP. Where are such machines today? To further exacerbate this let's assume that the 0.19 speed-up continues to apply for an increased number of processors. That means we would need a machine with 5.3 TFLOPS of peak performance to get real sustained TFLOPS of performance.

We can still do more with the data we just used. We now show this as efficiency vs. processors. Recall that the definition of efficiency is simply the speed-up divided by the number of processors. In Figure 16 we do just this. Note that this is the *average* efficiency. That is to say that we averaged all the data we had for each number of processors and plotted it. Some discussion needs to be made regarding this plot. Note that, with the exception of a few data points, this is a decreasing function of p. What is really disconcerting is the rapid decline in efficiency as we move up to only say 100 processors.

It is easy for criticism to be leveled against what we have shown in this

section (this author can think of a number of criticisms!). This is a circumstance that is representative of the original version of Amdahl's Law (we detailed this in a previous section); a fixed problem size being solved on an increased number of processors. It would be more accurate to discuss real bona-fide applications and how well they map onto existing MPPs. The only problem is doing that. There is a critical lack of real applications to choose from and finding a set that have been ported to a number of MPPs is impossible. Given this we, out of necessity, resorted to a simple synthetic benchmark. On balance we believe it makes a fair point about the issue of MPP performance.

There is yet another method by which the LINPACK benchmark could be run. That is to make the problem as large as possible in order to achieve the highest level of performance. This is sometimes called the "LINPEAK" benchmark. The approach is somewhat akin to the approach proposed by the scaled version of Amdahl's Law. In Figure 17 we show 64 data points for this variant of the LINPACK benchmark. Since we don't have enough data for this case we can't show speed-up or efficiency; this would require that we have the time for the same sized problem on one CPU. What we do show is the percentage of the peak performance of the machine vs. the number of processors.

We again see that as we add more and more processors we are getting less and less of the maximum possible performance. This is occurring in spite of the ability to make the problem as large as possible. What is more illuminating is that some of the data points are down in the 50% range in spite of the fact that we are allowing the problem size to be the most favorable.

The scaled version of Amdahl's Law would have us increase the problem size in proportion to the number of processors. This was not done in all of the data points shown in Figure 17. Thus, with reference to the scaled version of Amdahl's Law we are being conservative with respect to the scaling of the problem size. Even with this relaxed approach we should look at the time-to-solution for some of these data points. We do this in Figure 18.

The largest problem is running for 56 days on a dedicated machine with 256 processors. If this were a real application in a commercial organization it would need to be a mighty important one to deserve such a resource for such a long time. Keep in mind that additional processors reduce the time-to-solution, at best, linearly. Most scientific algorithms scale as some power of the problem size.

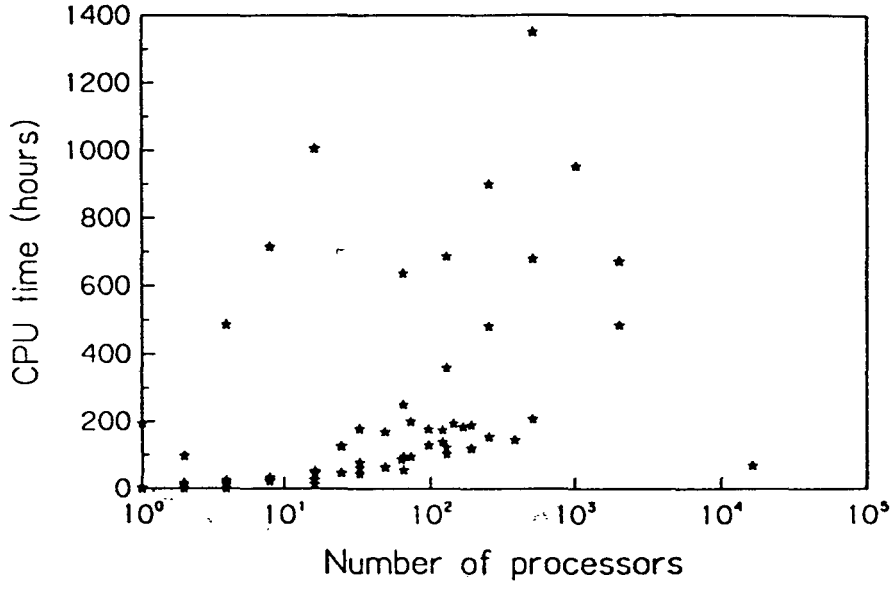

Figure 18: The time to solution for the problems shown in the previous figure. Note that the scale is in CPU hours so the longest running problem is 56 CPU days!

To diverge from LINPACK for a moment we examine, in Figure 19, the results from studies done at NASA Ames and Lawrence Livermore Laboratories. In this case the codes studied were actual codes [9] [10] [39] rather than synthetic benchmarks. It is apparent that again a small percentage of the peak performance of the machines is achieved. The vertical axis is "Y-MP equivalents." Thus for the code denoted BT the 8 processor Y-MP (2.6 GFLOPS peak performance) achieved nearly 7 times the performance of a single CPU of the Y-MP. The CM-200 with 32K processors (7 GFLOPS peak performance) achieved approximately 0.4 times the performance of a single Y-MP processor. The iPSC/860 with 128 processors (5.1 GFLOPS peak performance) achieved the performance equivalent of 1.9 Y-MP processors. Examining the other codes along the horizontal axis shows that the MPP machines are not able to achieve performance levels that are as impressive as those that are possible on a shared-memory vector supercomputer[23].

To close this section we make the observation, from the data presented, that using a small problem will obviously not permit high levels of performance. Further, simply allowing the problem to grow to an arbitrarily large size can make the computation time quite unacceptable. The data presented might well serve as a qualitative guide to what real applications will experience. This section should make us view claims of easy achievement of high performance on true MPPs with healthy skepticism.

17 Further reading

We have no illusion that this paper can even begin to tell the whole story about MPPs and all the related issues. What we can do however is to provide a reasonable bibliography of reading that we consider good places to start for developing your own perspective on MPP computing. Some of the references support our claims. To be fair some of them take contrary views. We can only hope that the reader is now sufficiently armed to read on and form their own conclusions, whether or not they agree with ours. The bias in the bibliography is that many of the citations are for articles that are of a general nature. This is in line with the theme, and level, of this paper. We couldn't even begin to entertain the idea of a "complete" bibliography on the subject.

[23]We want to emphasize (again) that we are not trying to compare vendors. Such data is scarce and we use it to illustrate a point about MPP in general.

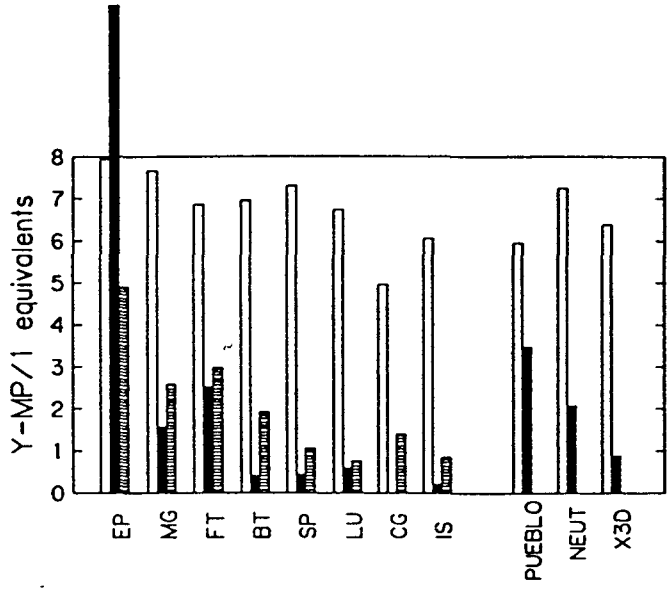

Figure 19: Results from studies done at NASA Ames and Lawrence Livermore Laboratories on actual codes. The horizontal axis represents the different codes that were studied. The studies were done on three very different machines; the Cray Y-MP (shared-memory), the CM-200 (SIMD), and the Intel iPSC/860 (MIMD). The white bar represents the Cray Y-MP, the "grey" bar the Intel iPSC/860 and the black bar the CM-200. Not all instances of the CM-200 and iPSC/860 were obtained with the same number of processors.

In what follows we make some general comments about each of the citations to narrow down the selection process based on the readers interests.

17.1 General

The often asked question is, "What is a good book to read on MPP?" This author enjoys the book by Almasi and Gottlieb [2]. It covers a broad range including hardware, software, architectures and applications. There is a sprinkling of humor throughout that makes the reading more enjoyable. If you only want to look at one book and your interests are not really specific then this is a good place to start.

For the non-specialist that wants a general introduction to supercomputing see [38] and [47].

For a user-friendly introduction that is not so specific on the strictly parallel machines but more for supercomputers in general see [36]. The nice thing about this paper is that it also covers languages, it has a synopsis of many of the machines in the market-place, there is a glossary, and it has a lot of references and a separate bibliography. Worth the reading.

There are several articles that engage in multi-faceted discussions about parallel processing. For example, see [11], [12], [18], and [28]. They are worth reading since they illuminate a number of widely disparate views of the subject.

One author's view of the prospect for MPP to be our general-purpose computing engine is offered in [31]. A brief synopsis of a panel discussion at a recent conference on massively parallel processing can be seen in [48].

A perspective on the impact of MPP within a single field, seismic processing, is offered by French in [26].

A sense of caution regarding blind acceptance of MPP is imparted to the reader by Worlton [52]. A similar theme is presented, not quite so strongly, in [5].

To understand the scope and direction of the US government's efforts to push computing technology the HPCC should be understood The official government booklet on this is [29].

17.2 Hardware

If you want to get detailed information about the hardware aspects of supercomputers and parallel computers then [49] is a book to see.

17.3 Software

A general view of software directions and software needs for MPP is discussed in Pancake [45]. A favorite article that illustrates the differences in programming style for several MPP variants is by Karp [35]. There was a special issue of the *IEEE Spectrum* journal [33] devoted to TFLOPS level of computing. It touches on a lot of issues, one of which is software.

17.4 Languages

Pancake and Bergmark [44] discuss the state of the languages that programmers use or could choose to use to write their MPP applications.

Fortran-90 is considered by many to be the obvious candidate as the programming language for MPP. To grasp all of the features of this language and to see just how different it is from Fortran-77 then you should consult [1].

Fortran-D and HPF are currently topics of intense interest to the MPP community. Remembering that HPF is a superset of Fortran-D descriptions of Fortran-D are available [25] [32].

The version 1.0 draft of HPF is available at the time of this paper. For those with access to Internet it is available via anonymous ftp from `titan.cs.rice.edu` in the `/public/HPFF/draft` directory. If you want a paper copy then you can contact

HPF Comments
c/o Theresa Chatman
CITI/CRPC, Box 1892
Rice University
Houston, TX 77251

A related discussion would be on IPO. While CONVEX has not published much of the detail regarding this effort, [40] contains some of the early and fundamental things that need to be addressed in order to evolve a compiler to enable IPO.

17.5 Benchmarks

The performance of a machine is of course always an issue that people want to develop a feel for. The most often cited measure of performance is the LINPACK benchmark. Discussion continues to this day about the relevance of this, but one fact that can not be denied is the popularity of this code. A large number of machines, something like 130, are listed in [19].

For an amusing and depressingly realistic perspective on the reporting of performance numbers don't miss [8].

Amdahl's Law almost always evokes lots of emotion from people. The original is described in [3]. The newer version, the so-called scaled version, is discussed in [30]. For a more detailed and hopefully a more practical look at both of these variants along with some others see [6].

The "mother of all benchmarks" is detailed in [51]. Please note the date of this reference.

17.6 Algorithms

An article that is worth looking at is [41]. This isn't as much for its content (which is also good) but rather for the time in which it was written; 1971. While MPP is certainly a contemporary topic it is also not a new topic. Over two decades ago consideration was given to parallel computing.

The development and understanding of algorithms for MPP is probably one of the deepest subjects in the entire field. We would suggest that [13] and [37] as two books that cover a lot of material.

We mentioned domain decomposition several times throughout this paper. This is a very broad and important topic that will be utilized extensively in the field of MPP and differential equations. You can get started by consulting [16].

For coverage that extends beyond just algorithms but discusses performance of algorithms, programming them and other good information then [4] and [24] would be good reading.

To sense the breadth of the subject the "single" topic of linear algebra on dense systems is covered in very informative detail in [27].

Partial differential equations are an important consumer of the power of MPPs. A somewhat dated but still valuable survey of this topic, which is somewhat biased to vector machines, is [42].

For an extensive bibliography on vector and parallel algorithms related to numerical analysis then you should get [43].

17.7 Architectures

To get a review of the parallel machines that are, were and will be (?) see [50]. This is a good source of information about the machine architectures and capabilities. Two additional places to review the architecture of machines are [7] and [20]. Brooks discusses the problems with shared-memory machines in [15].

18 Glossary

To help the reader in deciphering the literature on MPP, and in fact some of what was written here, we provide this glossary of terms that are often seen in the literature. Of course this can't do justice to all of the terminology that is encountered in the MPP literature but this should be of some help.

Algorithm: A statement of the steps that are taken in order to complete a task. A recipe could be considered an algorithm for making a cake. In the scientific computing community an algorithm can be considered to lie between mathematics and computer code (say Fortran).

Amdahl's Law: This is a statement that the ultimate performance of a computer is limited by the slowest component. In the context of MPPs this is interpretated to mean that the sequential component of the application code will restrict the maximum speed-up that is achievable.

Bandwidth: A measure of the rate at which a medium can move data. This is almost always measured in Mbytes per second but occasionally reported in Mbits per second.

Barrier: This is a type of synchronization used in parallel programs. When a barrier is coded into the application, using whatever exact mechanism is provided by the vendor, it will cause all the threads (*i.e.*, parallel tasks) to pause at the barrier until *all* the threads arrive. After that all the threads can resume their individual executions.

Cache: A fast separate memory that intervenes between the processor and the main memory of the computer. Cache is used to "hide" the higher latency of main memory access with a smaller but faster image of memory. In most cases data that is resident in the cache can be loaded into a register in a single CPU cycle. By comparison, on a contemporary RISC processor it takes something like 20 CPU cycles to get data from main memory. The size of the cache is the single most important measure. Cache sizes in the the 1/8 to 1/2 Mbyte range (128 Kbyte and 512 Kbyte, respectively) are commonplace today. Additional complexity is introduced into this definition by virtue of the fact that some RISC processors have a multi-level cache structure. This generally appears as a small, on-chip single cycle access cache and a secondary, off-chip cache. The secondary cache has an access penalty greater than that of the primary, on-chip cache.

Code: This is often used to refer to a program, either in source code form or in the form of an executable image on a machine. We often say things like, "the code is executing on several CPUs" or "the code is a mixture of Fortran and C."

Coherency: This is most often applied to caches. In a machine with more than one processor if a data item is referenced by a particular processor then a copy of the data from the main memory will be made into that processor's cache. If some other processor subsequently references that data there then needs to be a mechanism by which the subsequent reference to the data item will get the latest and greatest value from the cache of the original processor that copied it from main memory (this assumes that the value was changed by that processor). If this is insured then the caches are coherent.

COTS: Commercial Off The Shelf components. These are parts that are not specially designed by a vendor for their particular machine. Rather they are available as a generic item that is widely used by many people. An example would be the DRAMs that are used in the memory of most computers. Also the disk drives used in computers would fall into this category.

Commodity processors: In the MPP world when people make reference to commodity processors they mean that the processors are purchased from a chip manufacturer. This is in contrast to the computer vendor designing and developing their own processor chips.

Communication: In the context of this paper we refer to communication as the moving or sharing of data among processors. Communication is also required in order to synchronize several processors. The classic example is for a distributed memory computer to communicate via a message containing data to another processor.

Concurrent: Taking place at the same time. In the context of this paper we mean that some computing task has several parts and they are being computed at the same time by different processors within the MPP.

Cross-bar: For the shared-memory computers this device acts to connect the multiple CPUs with the various banks of the memory system. Another view of a cross-bar is as a switch that can on any given cycle connect some CPU to any memory bank.

Critical section: A portion of a parallel code that can be entered by only one thread at a time. Note that this does not exclude all threads from executing the code within the critical section but only that one, and only one, can execute it at a time.

Computational complexity: This is a measure of the amount of work an algorithm is expected to do in order to complete its task. As it is usually applied the work is given by an order of magnitude. In other words it is not exactly precise. As a specific example the LU factorization of a matrix of order n requires exactly $\frac{2}{3}n^3 - \frac{1}{2}n^2 - \frac{1}{6}n$ floating-point operations (this neglects pivoting). We say that its computational complexity is $\mathcal{O}(n^3)$. Note the lack of a constant in stating the computational complexity. These order of magnitude estimates are generally more accurate as n gets large.

Control parallelism: The assigning of different functional sections of the code to different processors for simultaneous execution.

Data parallel: Data parallel is one of the two programming models applied to parallel programming. In the data parallel approach a thread of execution is applied to a set or subset of the data for the application.

Def-use chain: A def-use chain begins at the point of definition of a variable, or subsection of an array, and tracks forward through the subroutine call-tree to all possible sites that could use the defined data. Therefore the def-use chain knows about every instance in every routine that could use

the definition, relative to the point where the def-use chain starts. Note that def-use chains are not just defined for a variable but it also depends on where the variable's definition occurs in the call-tree of the code.

Distributed memory: An MPP where each computing resource is a processor and memory with any processor being unable to address the memory of another processor has distributed memory. The aggregate memory of the machine is all of the memories on all of the processors. Each memory is distinct from the others.

Domain decomposition: A technique for breaking a problem into pieces for placement on the separate processors of an MPP. This involves breaking the computational domain into pieces and sending the pieces to the processors. There are restrictions on the separate domains and special algorithmic efforts are needed to knit the solutions for each domain together to form the global solution. This is often utilized in the context of the discrete solution of partial differential equations, especially via the finite element method (FEM).

DRAM: Dynamic Random Access Memory. These are the chips that are the fundamental building blocks of the memory of today's computers. The reason that they are denoted "dynamic" is that they must have their contents refreshed or restored periodically otherwise the charge (*i.e.*, information) is lost. The DRAMs come in a variety of densities. Commonplace today is 4 Mbits on a single chip.

Efficiency: This is used in specifying the average amount of time that a "generic" processor in an MPP machine is doing useful work. The precise definition is the speed-up (see below) divided by the number of processors being used.

Gbytes: This is an amount of data equal to 10^9 bytes. Spoken another way it is one thousand million bytes.

GFLOPS: Acronym for Giga FLoating-point Operations per Second. This is 1000 MFLOPS.

Grain-size: A measure of the size of a parallel chore. Often referenced as "fine-grain" or "coarse-grain." There is no specific number of instructions which denotes the difference. On shared-memory machines fine-grain could

be as low as several hundred instructions, on a true SIMD machine fine-grain could be a single instruction.

Grand Challenges: A class of problems that are used to characterize the goal of the HPCC. All of these problems are of such a size and complexity that the required computing resource to practically solve them will require a TFLOPS of performance and a Tbytes of physical memory.

GSDVM: This is somewhat Convex specific. In our vernacular it stands for Globally Shared Distributed Virtual Memory. This is a memory scheme in which the memory is physically distributed among the processors in an MPP but it can be uniformily addressed by a processor which does not directly own the data location (by physical proximity). Additionally it can support virtual memory. The literature also refers to this as Shared Virtual Memory or Global Virtual Memory.

HPCC: The is an acronym for the High Performance Computing and Communication initiative that was established, and is funded, by the US government. The goal is to develop the hardware, software, networks, and education to realize the solution to the "Grand Challenge" problems by the end of this decade.

Hypercube: A topology used on some MPPs. In this, each processor is connected to its binary neighbors. The number of processors is always a power of two. The dimension of the hypercube is the power. Thus a hypercube with 1024 processors is a 10-dimensional ($2^{10} = 1024$) hypercube.

Interleaved memory: This is a method of building the physical memory of a computer, generally a vector supercomputer, so that the data is laid out in round-robin fashion across several independent "banks." The memory banks can service a request for data and then while that bank is recycling, the next several data items requested will be taken from the other (presumably) refreshed banks. By the time a piece of data is needed from the first bank it will have refreshed and can then service another request for data. This is a reason why algorithms that access data on unit stride intervals are best; they are making references across the banks of the memory.

Interprocedural: This references a technique used in compilation whereby the compiler can see across the boundaries of the subroutines and functions (*i.e.*, procedures) when looking for optimizations to perform. This is distinct

from the procedure-based compilers that are in common use by all vendors.

LAN: Local Area Network. This is a network that connects computers with some physical media so that they can communicate data between them. The classic example of this is an Ethernet connection between a number of machines within a building. LANs are now also being used to tie machines, typically workstations, into some semblance of an MPP.

LAPACK: Linear Algebra PACKage. LAPACK is a mathematical software library that addresses problems in linear algebra. The package grew out of the earlier forerunners such as the level-1,2,3 BLAS, LINPACK and EISPACK. The algorithms in LAPACK are more contemporary and the code is designed for contemporary architectures.

Latency: The time delay between a processor requesting a piece of data and its actual receipt of the data. A more precise view is that it is the overhead of getting the data separated out from the time it takes to move the data.

LINPACK: LINPACK was, and still is, a library of routines for doing linear algebra. This includes things such as matrix factorizations and solutions. This original LINPACK has recently been superseded by the LAPACK library. Also LINPACK is often used to refer to the computer benchmark of the same name. This is often cited as a performance measure for computers that range in power from a PC to the high-end supercomputers.

Louse: Any of various small wingless, usually flattened, insects (orders Anoplura and Mallophaga) parasitic on warm-blooded animals. These things have *nothing* to do with parallel processing; we just wanted to see if you were actually reading this paper.

Mbits: Mega bits. A million bits. A bit is the smallest quanta of information that computers deal with. It is a single binary value, say 0 or 1.

Mbytes: Mega bytes. A million bytes. A byte is made up of 8 bits.

Memory: That portion of the computer's hardware where the data and instructions for the application are stored. All information processed by the computer is accessed from and stored in the memory.

Mesh diameter: This is a measure of the maximum distance between any pair of processors in an MPP. The metric of this distance is processors. For

example, if the topological connection is a ring then the mesh diameter is $n/2$ processors. In a hypercube the diameter is given as n, where n is the dimension of the hypercube.

Message passing: This refers to a programming style and to a machine type. When an MPP has physically distributed memory then any processor must pass a message, consisting of the data, to any other processor in order for them to interact. As a programming style message passing requires the explicit addition of subroutine calls to the users' code. These calls are to system libraries to package and transmit or receive data from other specifically identified processors.

MFLOPS: The is an acronym for Million **FL**oating-point **OP**erations per Second. Currently this is the most common way to measure the "power" of high-end computers.

Microkernel: This is a recent approach in the Unix operating system arena. Rather than have the entire Unix kernel resident on each processor of a machine you only need to have a minimal subset of it present. This is the microkernel. If your process requires any O/S functionality that is outside the scope of the microkernel, it will cause the appropriate server to be invoked.

MIMD: Acronym for **M**ultiple **I**nstruction streams, **M**ultiple **D**ata streams. A classification of a parallel architecture that has multiple processors, each of which can operate on a sequence of instructions which are separate from the other processor's instructions. Additionally the processors can be operating on separate data.

Nanosecond: 1×10^{-9} seconds. The cycle time of today's computers are generally measured in nanoseconds. As some specific examples, the Hewlett-Packard 735 workstation has a 10 nanosecond clock cycle, the Cray Y-MP has a 6 nanosecond cycle and the CONVEX C3800 has a 16.67 nanosecond cycle.

Network: This is the medium, or more generally, the generic connection that ties a number of computers together so they can communicate with each other. The most common form of network medium is Ethernet that ties machines that are relatively close, geographically. The Internet is likely the largest example of a network. This network spans the entire globe and connects thousands of machines together.

Peak performance: The theoretical maximum rate at which a computer can operate. For scientific computing this is generally cited in MFLOPS. There are several views of peak performance; (a) the speed of the computer with no software on it, (b) the "guaranteed not to exceed speed", or (c) bull-stuff! Except for contrived problems peak performance is never realized.

Pipeline: A hardware feature of some RISC computers (and vector computers also) which is a number of independent stages of a particular operation. Each stage can be active and operating on a separate piece of data. Consider that a floating-point addition requires a number of stages. Under pipelined execution subsequent floating-point adds can have their operands in each of the stages simultaneously.

PRAM: Parallel Random Access Machine. An abstraction of a parallel computer that has the properties of; (a) p serial processors, (b) a single, shared global memory for all of the p processors, (c) all p processors can read from or write to the global memory in parallel, and (d) the processors can (obviously) perform arithmetic and logic operations in parallel with each other. This is often used in computational complexity studies. It is not achievable in real-life. The major divergence between a PRAM model and a real machine is that the latter will have different access times to memory depending upon the access pattern(s) made by the processor's execution.

RISC: For Reduced Instruction Set Computer. The instruction set is generally more simple and smaller in scope than a complex instruction set computer (somewhat circular statement isn't it?). In the purest interpretation all instructions execute in one cycle except those that reference memory. The simpler design and only having a memory to register load and store allows for faster cycle times. There is constant debate about the exact attributes that constitute a RISC processor. Some of the more commonly accepted ones are; (a) all arithmetic/logic operations are performed from registers (b) the only way to reference memory is via load or store instructions, and (c) most instructions will complete in a few CPU cycles. Compiler writers sometime refer to RISC as meaning, Relegate all the Interesting Stuff to the Compiler. If you don't appreciate this comment then talk to your friendly neighborhood compiler guru.

Router: A piece of hardware that is part of each MPP node whose sole function is to route data either from, to or through the node.

Scalable: This is the property of a computer system that allows it to be increased in size without, (a) significant redesign, and (b) introducing any bottleneck to the performance.

Sequential section: Within a parallel code there are often (always?) some sections of the code that must be executed by a single processor, this is referred to as a sequential section (see critical section).

SIMD: Acronym for Single Instruction stream, Multiple Data streams. A classification of a parallel architecture that has multiple processors each of which is doing the same instruction but operating on different pieces of data. All processors are executing the stream of instructions in lock-step fashion. Roughly speaking there is a single program counter for the code, in spite of the presence of many processors.

SPMD: For Single Program Multiple Data. A single code executing on all processors simultaneously. This is usually taken to mean there is redundant execution of sequential scalar code by all processors.

Shared-memory: A type of multi-processor in which all of the processors see the same physical memory.

Speed-up: A measure of the performance of a program on an MPP. The precise statement is $S_u = \frac{T(1,n)}{T(p,n)}$. Here p is the number of processors and n is some measure of the problem size. Note that the speed-up is bounded as $1 \leq S_u \leq p$.

Superscalar: This is a class of RISC computers that allows multiple instructions to be issued in every clock period.

Synchronization: The communication between two or more processors to ascertain that they are at a specified place in their execution.

Thread: This is a stream of execution that is being performed by some processor while, possibly, another thread is being executed on another processor. A parallel application will start with a single thread and then at some point, and for some period of time, have multiple threads operating concurrently. More simply, a sequence of instructions that is independent of another sequence of instructions both of which might be executing concurrently.

TFLOPS: This is an acronym for Trillion **FL**oating-point **OP**erations per Second. Note that this is the same as a million MFLOPS (or a thousand GFLOPS).

Time-to-solution: The actual elapsed time, as would be measured by a clock, to complete a task. With sequential computers the measure of the performance was simply the CPU time that was used. On an MPP this is an inadequate measure. Consider that the CPU time used by a code executing across multiple processors is the sum total of the CPU times used on each of the processors. This is not a useful measure since those CPU times might have been consumed concurrently across the processors. To achieve a correct measure of the performance of a parallel algorithm or machine you need to consider how long it takes in elapsed time from saying "go" until it stops. This places other restrictions on performance measuring such as; what about the "interference" of another user's codes? The penalty is that a dedicated machine, or a least sub-complex of the machine, is required in order to get a precise measurement of the time-to-solution.

Topology: This term describes the geometric connection of the processors within an MPP. The pattern of the connections that connect the various processors together define the topology. Common topologies in the MPP arena today are the hypercube, the mesh, the ring or the tree.

Use-def chain: A use-def chain begins at the point of usage of a variable, or subsection of an array, and tracks backward through the subroutine call-tree to all possible sites that could define the used data. Therefore the use-def chain knows about every instance in every routine that could impact on the usage at the point where the use-def chain starts. Note that use-def chains are not just defined for a variable but it also depends on where the variable's usage occurs in the call-tree of the code.

Vector: In physics a quantity that has a magnitude and a direction. In our context here is a sequence, possibly not contiguous, of data that are operated on by a single vector instruction. Most current supercomputers are vector processors.

19 CONVEX's MPP directions

Having spent this entire paper detailing the short-comings of MPP computing the question can be fairly asked what are the characteristics of a good MPP machine. Rather than present an abstract machine we will mention a few attributes of the machine that CONVEX is designing. The details are intentionally vague for competitive reasons but we hope they convey some sense of the approach that we are pursuing. Obviously we think our machine is representative of a "good", if not perfect, MPP! A short itemized list of our MPP's attributes, with brief discussion, follows.

- MIMD
  We have decided against the SIMD architecture. There is a growing body of evidence that the future "architecture of choice" will be MIMD. This is not a global condemnation of SIMD, we just feel it will become more of a niche machine in the future.

- Globally Shared Distributed Virtual Memory
  This feature allows the machine to have physically distributed memory so the machine is scalable but presents to the user a logical programming model that is reminiscent of the shared-memory machines that are used today.

- OSF/1 AD operating system
  This offers to the user a Unix "look and feel" but being microkernel based allows the operating system to be scalable across many processors with physically distributed memory.

- Stand-alone and air-cooled
  Machines that do not require a front-end machine are attractive since the front-end invariably becomes a bottleneck in the performance of the machine. Additionally it is an added cost. Air-cooled machines are also less costly and require less expensive machine room facilities.

- Hewlett-Packard PA-RISC processors
  The Hewlett-Packard RA-RISC processors [17] we are using are 99 MHz, 198 (peak) MFLOP CPUs. This is the seventh generation of this chip family. Using processors of significant power is important so that fewer processors are required to achieve high levels of performance.

Having a powerful processor to execute the sequential code aids in getting reduced time-to-solution.

- Gbytes of physical memory
  The applications that our users want to address, now and most certainly in the future, require very large amounts of memory. The machine we are building allows for *many* Gbytes of physical memory.

- Fortran-77 (with Fortran-90 array notation), C, and C++
  Support for these common languages allows users to code a language that is familiar to them. These languages coupled with the interprocedural optimization capabilities in our Application Compiler offers the prospect of achieving some degree of parallel performance via compilation. The native language will be supplemented with a rich suite of directives to make explicit parallel programming visible to the interested user. Also there will be a message passing library consisting of PVM and other syntax forms.

- Emphasis on ease of use
  Philosophically we have the attitude that people time is more expensive than machine time. Having a very powerful machine that requires very large amounts of programming time to get a certain level of performance is overall cost-inefficient to an organization.

We freely admit that this list doesn't reveal too much about the specifics of our approach but that is not the intent of this paper.

20 Conclusions

Well it seems like we have said a lot in this paper. It is worth trying to summarize it a bit, hopefully without losing too much context. Mindful of the caveat of using too much brevity, here follows a bullet-list of the messages we detailed above.

- MPP is going to happen to us
- high-end computing will be done on MPP
- the exact number of processors that constitutes massive is elusive
- there are different classes of MPP architectures
- the MIMD class offers the most generality

- TFLOPS and Tbytes are laudable goals but won't be seen in the near-term
- MPP's acceptance as main-stream computing hinges crucially on the availability of third-party software
- porting application codes to MPP is hard work
- automatic parallelization of existing and future codes via compilation is necessary
- the important hardware metric for an MPP is the inter-processor latency
- a tremendous amount of algorithmic work needs to be done for MPP
- the performance achieved on an MPP is likely going to be a smaller percentage of peak than we experience on today's hardware
- Amdahl's Law, either form, is generally extreme so it must be understood and applied correctly

We remain overall confident that the future of high-end computing will be done on MPP machines. We are realistic in expecting the transition to MPP to take longer than much of the popular press indicates it will. Does this deter us from proceeding? No. There is tremendous work to be done on virtually all fronts related to MPP; hardware, software, operating systems, debuggers, profilers, compilers, languages, coding techniques, algorithms, teaching, etc., etc. It *will* be an interesting decade to be involved in supercomputing.

21 Epilogue

No paper of finite length can tell the entire story. Our aim was to bring a number of issues into the light so that the readers will develop a bit of healthy respect, and perhaps cynicism, about much of what has been published, in the popular press, regarding MPP computing. If there are dissenting opinions or data this author would welcome an open discussion or rebuttal.

22 Acknowledgments

The truth is that I need to acknowledge *many* people at CONVEX for illuminating discussions about MPP. This is especially true of the hardware and software engineers in "Building D." Unfortunately they are too numer-

ous to explicitly mention; my apologies to them for that. Particular thanks go to my colleagues Aaron Potler, Murray Smigel, and Joel Williamson for their meticulously careful reading of this document. It is much better owing to their effort. Finally, thanks to Dean Clark of *The Leading Edge* for approaching me with the suggestion to write this.

23 References

1. J.C. ADAMS, W.S. BRAINERD, J.T. MARTIN, B.T. SMITH, AND J.L. WAGENER, *Fortran 90 Handbook*, McGraw Hill, New York, 1992.

2. G.S. ALMASI AND A. GOTTLIEB, *Highly Parallel Computing*, Benjamin/Cummings, Redwood City, 1989.

3. G. AMDAHL, *Validity of the single-processor approach to achieving large scale computing capabilities*, in Proceedings of the AFIPS Conference, Atlantic City, NJ, AFIPS Press, Reston, 1967, pp. 483–485.

4. I. ANGUS, G. FOX, J. KIM, AND D. WALKER, *Solving Problems on Concurrent Processors, Vol.II*, Prentice Hall, Englewood Cliffs, 1990.

5. G. ASTFALK, *Convex's view of TFLOPS computing*, in Parallel Computing and Transputer Applications, M. Valero, E. Oñate, M. Jane, J.L. Larriba, and B. Suárez, ed., IOS Press, Amesterdam, 1992, pp. 51–61.

6. G. ASTFALK, *Revisiting several variants Amdahl's Law*, in preparation, 1993.

7. J. BAER, *Computer architecture*, Computer, 17(1984), pp. 77–87.

8. D.H. BAILEY, *Twelve ways to fool the masses*, Supercomputing Review, 4(1991), pp. 54–55.

9. D. BAILEY, J. BARTON, T. LASINSKI, AND H. SIMON, *The NAS parallel benchmarks*, NASA Ames Research Center, Moffett Field, CA, RNR–91–002, January, 1991.

10. D.H. BAILEY, E. BARSZCZ, L. DAGUM, AND H.D. SIMON, *NAS parallel benchmark results*, in Proceedings of Supercomputing '92, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 386–393.

11. G. BELL, *Ultracomputers: A Teraflops before its time*, Communications of the ACM, 35(1992), pp. 27–47.

12. G. BELL, *Massively parallel computers: Why not parallel computers for the masses?*, in Fourth Symposium on the Frontiers of Massively Parallel Computation, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 292–297.

13. D.P. BERTSEKAS AND J.N. TSITSIKLIS, *Parallel and Distributed Computation*, Prentice Hall, Engelwood Cliffs, 1989.

14. Z. BOZKUS, S. RANKA, AND G.C. FOX, *Benchmarking the CM-5 multicomputer*, in Fourth Symposium on the Frontiers of Massively Parallel Computation, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 100–107.

15. E.D. BROOKS, *Massive parallelism overcomes shared-memory limitations*, Computers in Physics, 6(1992), pp. 139–145.

16. T.F. CHAN, R. GLOWINSKI, J. PERIAUX, AND O.B. WIDLUND, ed., *Domain Decomposition Methods*, SIAM, Philadelphia, 1989.

17. E. DELANO, W. WALKER, J. YETTER, AND M. FORSYTH, *A high speed superscalar PA-RISC processor*, in Proceedings of IEEE COMPCON '92, February, 1992.

18. Y. DENG, J. GLIMM, AND D.H. SHARP, *Perspectives on parallel computing*, DÆDALUS, Winter(1992), pp. 31–52.

19. J.J. DONGARRA, *Performance of various computers using standard linear equations software*, CS-89–85, University of Tennessee, Knoxville, TN 1992.

20. R. DUNCAN, *A survey of parallel computer architectures*, IEEE Computer, 23(1990), pp. 5–16.

21. T.H. DUNIGAN, *Kendall Square multiprocessor: Early experiences and performance*, ORNL/TM–12065, Oak Ridge National Laboratory, Oak Ridge, TN, April, 1992.

22. M.J. FLYNN *Some computer organizations and their effectiveness*, IEEE Transactions on Computers, C-**21**(1972), pp. 948–960.

23. I. FOSTER, W. GROPP, AND R. STEVENS, *The parallel scalability of the spectral transform method*, Preprint MCS-P215-0291, Argonne National Laboratory, Argonne, IL, 1991.

24. G. FOX, M. JOHNSON, G. LYZENGA, S. OTTO, J. SALMON, AND D. WALKER, *Solving Problems on Concurrent Processors, Vol. I*, Prentice Hall, Englewood Cliffs, 1988.

25. G. FOX, S. HIRANANDANI, K. KENNEDY, C. KOELBEL, U. KREMER, C-H. TSENG, AND M-Y. WU, *Fortran D language specification*, TR90–141, Rice University, Department of Computer Science, Houston, TX, 1990.

26. W.S. FRENCH, *Implications of parallel computation in seismic data processing*, Geophysics: The Leading Edge of Exploration, 11(1992), pp. 22–25.

27. K.A. GALLIVAN, R.J. PLEMMONS, AND A.H. SAMEH, *Parallel algorithms for dense linear algebra computations*, SIAM Review, 32(1990), pp. 54–135.

28. G. GILDER, *Hillis versus the Law of the Microcosm*, Upside, January(1992), pp. 24–42.

29. *Grand Challenges: High Performance Computing and Communications*, available from the Office of Science and Technology Policy, National Science Foundation, 1800 G Street, NW, Washington, DC 20550.

30. J.L. GUSTAFSON, *The scaled-size model: A revision of Amdahl's Law*, in Proceedings of the Third International Conference on Supercomputing, L.P. Kartashev and S.I. Kartashev, ed., Boston, 1988, pp. 130–133.

31. J.J. HACK, *On the promise of general-purpose parallel computing*, Parallel Computing, 10(1989), pp. 261–275.

32. S. HIRANANDANI, K. KENNEDY, C. KOELBEL, U. KREMER, AND C-W. TSENG, *An overview of the Fortran D programming system*, COMP TR91–154, Rice University, Department of Computer Science, Houston, TX, 1991.

33. IEEE Spectrum, 29(1992).

34. D.J. KUCK, *High-speed machines and their compilers*, in Parallel Processing Systems, D. Evans, ed., Cambridge University Press, 1982.

35. A.H. KARP, *Programming for parallelism*, Computer, 20(1987), pp. 43–57.

36. E.L. LAFFERTY, M.C. MICHAUD, M.J. PRELLE, AND J.B. GOETHERT, *Introduction to parallel processing*, MTR-92B0000103, MITRE, Bedford, MA, 1992.

37. S. LAKSHMIVARAHAN AND S.K. DHALL, *Analysis and Design of Parallel Algorithms*, McGraw Hill, New York, 1990.

38. R.D. LEVINE, *Supercomputers*, Scientific American, 246(1982), pp. 118–135.

39. O.M. LUBECK, M.L. SIMMONS, AND H.J. WASSERMAN, *The performance realities of massively parallel processors: A case study*, in Proceedings of Supercomputing '92, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 403–412.

40. R. METZGER, *Pointer target tracking*, The C Users Journal, 10(1992), pp. 85–92.

41. W.L. MIRANKER, *A survey of parallelism in numerical analysis*, SIAM Review, 13(1971), pp. 524–547.

42. J.M. ORTEGA AND R.G. VOIGT, *Solution of partial differential equations on vector and parallel computers*, SIAM Review, 27(1985), pp. 149–240.

43. J.M. ORTEGA, R.G. VOIGT, AND C.H. ROMINE, *A bibliography on parallel and vector numerical algorithms*, Interim Report #6, ICASE, Hampton, VA, 1988.

44. C.M. PANCAKE AND D. BERGMARK, *Do parallel languages meet the needs of the scientific programmer*, IEEE Computer, 23(1990), pp. 13–23.

45. C.M. PANCAKE, *Software support for parallel computing: Where are we headed?*, Communications of the ACM, 34(1991), pp. 52–64.

46. R. PONNUSAMY, A. CHOUDHARY, AND G. FOX, *Communication overhead on CM5: An experimental performance evaluation*, in Fourth Symposium on the Frontiers of Massively Parallel Computation, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 108–115.

47. Scientific American, August, 1987.

48. H.J. SIEGEL, *What are the two most important issues facing the design and use of massively parallel computers?*, in Third Symposium on the Frontiers of Massively Parallel Computation, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 526–529.

49. H.S. STONE, *High-Performance Computer Architectures*, Addison-Wesley, Reading, 1990.

50. A. TREW AND G. WILSON, *Past, Present, Parallel*, Springer-Verlag, London, UK, 1991.

51. J.H. WHARTON, *New benchmark puts RISCs/CISCs on even footing*, Computer, 22(1989), pp. 72–73.

52. J. WORLTON, *Be sure the MPP bandwagon is going somewhere before you jump aboard*, High-performance Computing Review, 1(1992), pp. 41–42.