A Beginner's Guide to High–Performance Computing

1 Module Description

Developer: Rubin H Landau, Oregon State University, `http://science.oregonstate.edu/ rubin/`

Goals: To present some of the general ideas behind and basic principles of high–performance computing (HPC) as performed on a supercomputer. These concepts should remain valid even as the technical specifications of the latest machines continually change. Although this material is aimed at HPC supercomputers, if history be a guide, present HPC hardware and software become desktop machines in less than a decade.

Outcomes: Upon completion of this module and its tutorials, the student should be able to

- Understand in a general sense the architecture of high performance computers.
- Understand how the the architecture of high performance computers affects the speed of programs run on HPCs.
- Understand how memory access affects the speed of HPC programs.
- Understand Amdahl's law for parallel and serial computing.
- Understand the importance of communication overhead in high performance computing.
- Understand some of the general types of parallel computers.
- Understand how different types of problems are best suited for different types of parallel computers.
- Understand some of the practical aspects of message passing on MIMD machines.

Prerequisites:

- Experience with compiled language programming.
- Some familiarity with, or at least access to, Fortran or C compiling.
- Some familiarity with, or at least access to, Python or Java compiling.
- Familiarity with matrix computing and linear algebra.

Intended Audience: Upper–level undergraduates or graduate students.

Teaching Duration: One-two weeks. A real course in parallel computing would of course be at least a full term.

Activities: There is considerable material to be read concerning HPC concepts and terms that are important to understand for a general understanding of HPC. Learning the semantics is an important first step. The tutorial part of the module demonstrates and leads the reader through some techniques for writing programs that are optimized for HPC hardware. The reader experiments with their computer's memory and experiences some of the concerns, techniques, rewards, and shortcomings of HPC.

Student Assessment: The first element to student assessment would be to determine if they understand the semantics. Understanding the importance of memory access on program speed, and particularly the importance of paging and communication would be the next level of assessment. Actually being able to optimize programs for HPC hardware might be beyond the capability of most undergraduates, but important for research–oriented students.

Testing: The materials have been developed and used for teaching in Computational Physics classes and at seminars for many years. They have also been presented at tutorials at SC conferences. Feedback from students involved with research has been positive.

Contents

2 High–Performance Computers

High performance and parallel computing is a broad subject, and our presentation is brief and given from a practitioner's point of view. Much of the material presented here is taken from *A Survey of Computational Physics*, coauthored with Paez and Bordeianu [LPB 08]. More in depth discussions can be found in the text by Quinn [Quinn 04], which surveys parallel computing and MPI from a computer science point of view, and especially the references [Quinn 04, Pan 96, VdeV 94, Fox 94]. More recent developments, such as programming for multicore computers, cell computers, and field–programmable gate accelerators, are discussed in journals and magazines [CiSE], with a short discussion at the end.

By definition, supercomputers are the fastest and most powerful computers available, and at present the term refers to machines with hundreds of thousands of processors. They are the superstars of the high–performance class of computers. Personal computers (PCs) small enough in size and cost to be used by an individual, yet powerful enough for advanced scientific and engineering applications, can also be high–performance computers. We define *high–performance computers* as machines with a good balance among the following major elements:

- Multistaged (pipelined) functional units.
- Multiple central processing units (CPUs) (parallel machines).
- Multiple cores.

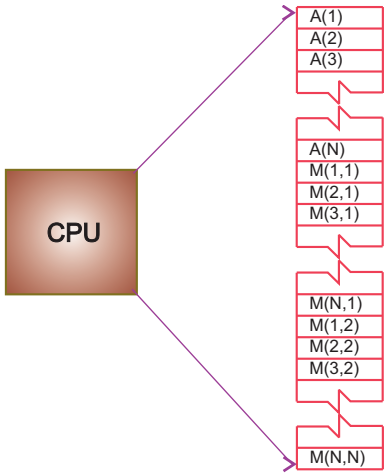


Figure 1: The logical arrangement of the CPU and memory showing a Fortran array $A(N)$ and matrix $M(N,\ N)$ loaded into memory.

- Fast central registers.
- Very large, fast memories.
- Very fast communication among functional units.
- Vector, video, or array processors.
- Software that integrates the above effectively.

As a simple example, it makes little sense to have a CPU of incredibly high speed coupled to a memory system and software that cannot keep up with it.

3 Memory Hierarchy

An idealized model of computer architecture is a CPU sequentially executing a stream of instructions and reading from a continuous block of memory. To illustrate, in Figure 1 we see a vector `A[ ]` and an array `M[ ][ ]` loaded in memory and about to be processed. The real world is more complicated than this. First, matrices are not stored in blocks but rather in linear order. For instance, in Fortran it is in *column–major* order:

$$M(1,1)\,M(2,1)\,M(3,1)\,M(1,2)\,M(2,2)\,M(3,2)\,M(1,3)\,M(2,3)\,M(3,3),$$

while in Python, Java and C it is in *row–major* order:

$$M(0,0)\,M(0,1)\,M(0,2)\,M(1,0)\,M(1,1)\,M(1,2)\,M(2,0)\,M(2,1)\,M(2,2).$$

Second, as illustrated in Figures 2 and 3, the values for the matrix elements may not even be in the same physical place. Some may be in RAM, some on the disk, some in cache, and some in the CPU. To give some of these words more meaning, in Figures 2 and 3 we show simple models of the memory architecture of a high–performance computer. This hierarchical arrangement arises from an effort to balance speed and cost with fast, expensive memory supplemented by slow, less expensive memory. The memory architecture may include the following elements:

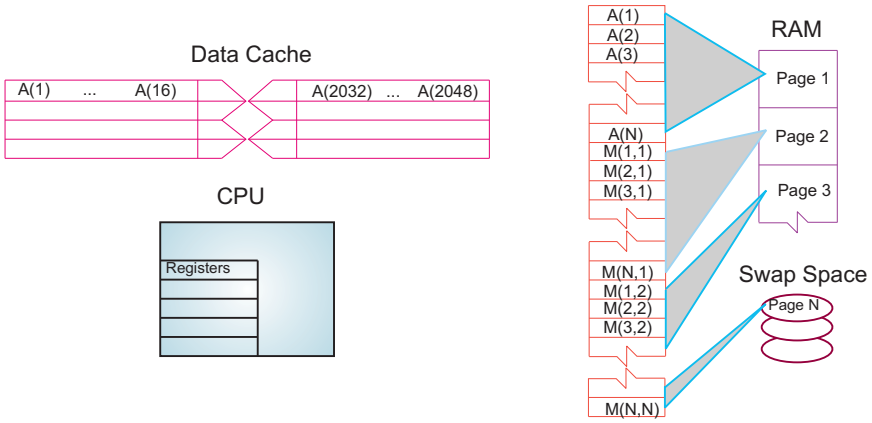


Figure 2: The elements of a computer's memory architecture in the process of handling matrix storage.

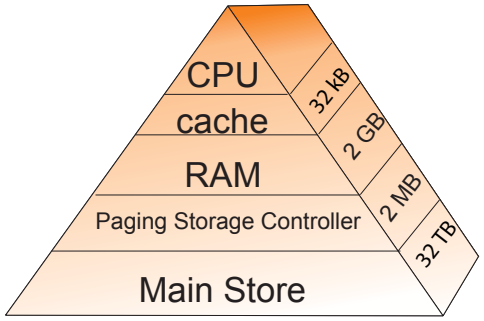


Figure 3: Typical memory hierarchy for a single–processor, high–performance computer (B = bytes, K, M, G, T = kilo, mega, giga, tera).

CPU: Central processing unit, the fastest part of the computer. The CPU consists of a number of very–high–speed memory units called *registers* containing the *instructions* sent to the hardware to do things like fetch, store, and operate on data. There are usually separate registers for instructions, addresses, and *operands* (current data). In many cases the CPU also contains some specialized parts for accelerating the processing of floating–point numbers.

Cache: A small, very fast bit of memory that holds instructions, addresses, and data in their passage between the very fast CPU registers and the slower RAM. (Also called a high–speed buffer.) This is seen in the next level down the pyramid in Figure 3. The main memory is also called *dynamic RAM* (DRAM), while the cache is called *static RAM* (SRAM). If the cache is used properly, it can greatly reduce the time that the CPU waits for data to be fetched from memory.

Cache lines: The data transferred to and from the cache or CPU are grouped into cache or data lines. The time it takes to bring data from memory into the cache is called *latency*.

RAM: Random–access or central memory is in the middle of the memory hierarchy in Figure 3. RAM is fast because its addresses can be accessed directly in random order, and because no mechanical devices are needed to read it.

Pages: Central memory is organized into pages, which are blocks of memory of fixed length. The operating system labels and organizes its memory pages much like we do the pages of a book; they are numbered and kept track of with a *table of contents*. Typical page sizes are from 4 to 16 kB, but on supercomputers may be in the MB range.

Hard disk: Finally, at the bottom of the memory pyramid is permanent storage on magnetic disks or optical devices. Although disks are very slow compared to RAM, they can store vast amounts of data and sometimes compensate for their slower speeds by using a cache of their own, the *paging storage controller*.

Virtual memory: True to its name, this is a part of memory you will not find in our figures because it is *virtual*. It acts like RAM but resides on the disk.

When we speak of fast and slow memory we are using a time scale set by the clock in the CPU. To be specific, if your computer has a clock speed or cycle time of 1 ns, this means that it could perform a billion operations per second if it could get its hands on the needed data quickly enough (typically, more than 10 cycles are needed to execute a single instruction). While it usually takes 1 cycle to transfer data from the cache to the CPU, the other types of memories are much slower. Consequently, you can speed up your program by having all needed data available for the CPU when it tries to execute your instructions; otherwise the CPU may drop your computation and go on to other chores while your data gets transferred from lower memory (we talk more about this soon in the discussion of pipelining or cache reuse). Compilers try to do this for you, but their success is affected by your programming style.

As shown in Figure 2 for our example, virtual memory permits your program to use more pages of memory than can physically fit into RAM at one time. A combination of operating system and hardware *maps* this virtual memory into pages with typical lengths of 4–16 kB. Pages not currently in use are stored in the slower memory on the hard disk and brought into fast memory only when needed. The separate memory location for this switching is known as *swap space* (Figure 2).

Observe that when the application accesses the memory location for `M[i][j]`, the number of the page of memory holding this address is determined by the computer, and the location of `M[i][j]` within this page is also determined. A *page fault* occurs if the needed page resides on disk rather than in RAM. In this case the entire page must be read into memory while the least recently used page in RAM is swapped onto the disk. Thanks to virtual memory, it is possible to run programs on small computers that otherwise would require larger machines (or extensive reprogramming). The price you pay for virtual memory is an order-of-magnitude slowdown of your program's speed when virtual memory is actually invoked. But this may be cheap compared to the time you would have to spend to rewrite your program so it fits into RAM or the money you would have to spend to buy a computer with enough RAM for your problem.

Virtual memory also allows *multitasking*, the simultaneous loading into memory of more programs than can physically fit into RAM (Figure 4). Although the ensuing switching among applications uses computing cycles, by avoiding long waits while an application is loaded into memory, multitasking increases the total throughout and permits an improved computing environment for users. For example, it is multitasking that permits a windowing system, such as Linux, OS X or Windows, to provide us with multiple windows. Even though each window application uses a fair amount of memory, only the single application currently receiving input must actually reside in

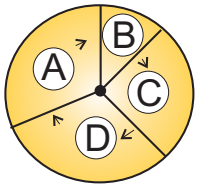


Figure 4: Multitasking of four programs in memory at one time in which the programs are executed in round–robin order.

Table 1: Computation of $c = (a + b)/(d * f)$

Arithmetic Unit	*Step 1*	*Step 2*	*Step 3*	*Step 4*
A1	Fetch a	Fetch b	Add	—
A2	Fetch d	Fetch f	Multiply	—
A3	—	—	—	Divide

memory; the rest are *paged out* to disk. This explains why you may notice a slight delay when switching to an idle window; the pages for the now active program are being placed into RAM and the least used application still in memory is simultaneously being paged out.

4 The Central Processing Unit

How does the CPU get to be so fast? Often, it employs *prefetching* and *pipelining*; that is, it has the ability to prepare for the next instruction before the current one has finished. It is like an assembly line or a bucket brigade in which the person filling the buckets at one end of the line does not wait for each bucket to arrive at the other end before filling another bucket. In the same way a processor fetches, reads, and decodes an instruction while another instruction is executing. Consequently, even though it may take more than one cycle to perform some operations, it is possible for data to be entering and leaving the CPU on each cycle. To illustrate, Table 1 indicates how the operation $c = (a + b)/(d * f)$ is handled. Here the pipelined arithmetic units A1 and A2 are simultaneously doing their jobs of fetching and operating on operands, yet arithmetic unit A3 must wait for the first two units to complete their tasks before it has something to do (during which time the other two sit idle).

5 CPU Design: Reduced Instruction Set Computers

Reduced instruction set computer (RISC) architecture (also called *superscalar*) is a design philosophy for CPUs developed for high-performance computers and now used broadly. It increases the arithmetic speed of the CPU by decreasing the number of instructions the CPU must follow. To understand RISC we contrast it with *complex instruction set computer* (CISC), architecture. In the late 1970s, processor designers began to take advantage of *very-large-scale integration* (VLSI), which allowed the placement of hundreds of thousands of devices on a single CPU chip. Much of the space on these early chips was dedicated to *microcode* programs written by chip designers and containing machine language instructions that set the operating characteristics of the computer. There were more than 1000 instructions available, and many were similar to higher–level programming languages like *Pascal* and *Forth*. The price paid for the large number of complex instructions

was slow speed, with a typical instruction taking more than 10 clock cycles. Furthermore, a 1975 study by Alexander and Wortman of the *XLP* compiler of the IBM System/360 showed that about 30 low–level instructions accounted for 99% of the use with only 10 of these instructions accounting for 80% of the use.

The RISC philosophy is to have just a small number of instructions available at the chip level but to have the regular programmer's high level–language, such as Fortran or C, translate them into efficient machine instructions for a particular computer's architecture. This simpler scheme is cheaper to design and produce, lets the processor run faster, and uses the space saved on the chip by cutting down on microcode to increase arithmetic power. Specifically, RISC increases the number of internal CPU registers, thus making it possible to obtain longer pipelines (cache) for the data flow, a significantly lower probability of memory conflict, and some instruction–level parallelism.

The theory behind this philosophy for RISC design is the simple equation describing the execution time of a program:

$$\text{CPU time} = \text{number of instructions} \times \text{cycles/instruction} \times \text{cycle time}. \tag{1}$$

Here "CPU time" is the time required by a program, "number of instructions" is the total number of machine–level instructions the program requires (sometimes called the *path length*), "cycles/instruction" is the number of CPU clock cycles each instruction requires, and "cycle time" is the actual time it takes for one CPU cycle. After viewing (1) we can understand the CISC philosophy, which tries to reduce CPU time by reducing the number of instructions, as well as the RISC philosophy, which tries to reduce the CPU time by reducing cycles/instruction (preferably to one). For RISC to achieve an increase in performance requires a greater decrease in cycle time and cycles/instruction than the increase in the number of instructions.

In summary, the elements of RISC are the following:

Single–cycle execution, for most machine–level instructions.

Small instruction set, of less than 100 instructions.

Register–based instructions, operating on values in registers, with memory access confined to loading from and storing to registers.

Many registers, usually more than 32.

Pipelining, concurrent preparation of several instructions that are then executed successively.

High–level compilers, to improve performance.

6 CPU Design: Multiple–Core Processors

The present time is seeing a rapid increase in the inclusion of dual–core, quad–core, or even sixteen–core chips as the computational engine of computers. As seen in Figure 5, a dual–core chip has two CPUs in one integrated circuit with a shared interconnect and a shared level–2 cache. This type of configuration with two or more identical processors connected to a single shared main memory is called *symmetric multiprocessing*, or SMP. As we write this module, chips with 36 cores are available, and it likely that the number has increased by the time you read this.

Although multicore chips were designed for game playing and single precision, they are finding use in scientific computing as new tools, algorithms, and programming methods are employed. These chips attain more integrated speed with less heat and more energy efficiency than single–core

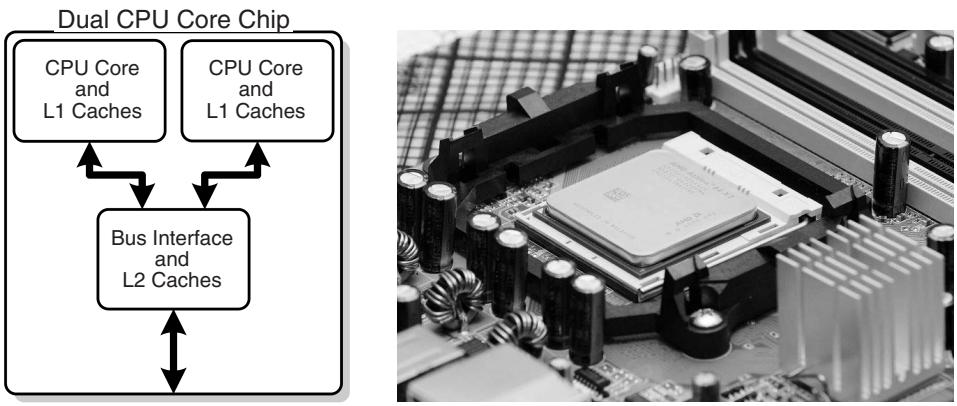


Figure 5: *Left:* A generic view of the Intel core-2 dual–core processor, with CPU-local level-1 caches and a shared, on-die level-2 cache (courtesy of D. Schmitz). *Right:* The AMD Athlon 64 X2 3600 dual-core CPU (Wikimedia Commons).

chips, whose heat generation limits them to clock speeds of less than 4 GHz. In contrast to multiple single–core chips, multicore chips use fewer transistors per CPU and are thus simpler to make and cooler to run.

Parallelism is built into a multicore chip because each core can run a different task. However, since the cores usually share the same communication channel and level-2 cache, there is the possibility of a communication bottleneck if both CPUs use the bus at the same time. Usually the user need not worry about this, but the writers of compilers and software must so that your code will run in parallel. Modern Intel compilers automatically make use of the multiple cores, with MPI even treating each core as a separate processor.

7 CPU Design: Vector Processors

Often the most demanding part of a scientific computation involves matrix operations. On a classic (von Neumann) scalar computer, the addition of two vectors of physical length 99 to form a third ultimately requires 99 sequential additions (Table 2). There is actually much behind–the–scenes work here. For each element i there is the *fetch* of $a(i)$ from its location in memory, the *fetch* of $b(i)$ from its location in memory, the *addition* of the numerical values of these two elements in a CPU register, and the *storage* in memory of the sum in $c(i)$. This fetching uses up time and is wasteful in the sense that the computer is being told again and again to do the same thing.

When we speak of a computer doing *vector processing*, we mean that there are hardware components that perform mathematical operations on entire rows or columns of matrices as opposed to individual elements. (This hardware can also handle single–subscripted matrices, that is, mathematical vectors.) In the vector processing of $[A] + [B] = [C]$, the successive fetching of and addition of the elements A and B are grouped together and overlaid, and $Z \simeq 64$–256 elements (the *section size*) are processed with one command, as seen in Table 3. Depending on the array size, this method may speed up the processing of vectors by a factor of about 10. If all Z elements were truly processed in the same step, then the speedup would be ~ 64–256.

Vector processing probably had its heyday during the time when computer manufacturers produced large mainframe computers designed for the scientific and military communities. These computers had proprietary hardware and software and were often so expensive that only corporate or military laboratories could afford them. While the Unix and then PC revolutions have nearly

Table 2: Computation of Matrix $[C] = [A] + [B]$

Step 1	*Step 2*	$\cdots$	*Step 99*
$c(1) = a(1) + b(1)$	$c(2) = a(2) + b(2)$	$\cdots$	$c(99) = a(99) + b(99)$

Table 3: Vector Processing of Matrix $[A] + [B] = [C]$

Step 1	Step 2	$\cdots$	Step Z
$c(1) = a(1) + b(1)$			
	$c(2) = a(2) + b(2)$		
		$\cdots$	
			$c(Z) = a(Z) + b(Z)$

eliminated these large vector machines, some do exist, as well as PCs that use vector processing in their video cards. Who is to say what the future holds in store?

8 Introduction to Parallel Computing

There is little question that advances in the hardware for parallel computing are impressive. Unfortunately, the software that accompanies the hardware often seems stuck in the 1960s. In our view, message passing has too many details for application scientists to worry about and (unfortunately) requires coding at an elementary level reminiscent of the early days of computing. However, the increasing occurrence of clusters in which the nodes are symmetric multiprocessors has led to the development of sophisticated compilers that follow simpler programming models; for example, *partitioned global address space* compilers such as *Co–Array Fortran*, *Unified Parallel C*, and *Titanium*. In these approaches the programmer views a global array of data and then manipulates these data as if they were contiguous. Of course the data really are distributed, but the software takes care of that outside the programmer's view. Although such a program may make use of processors less efficiently than would a hand–coded program, it is a lot easier than redesigning your program. Whether it is worth *your* time to make a program more efficient depends on the problem at hand, the number of times the program will be run, and the resources available for the task. In any case, if each node of the computer has a number of processors with a shared memory and there are a number of nodes, then some type of a hybrid programming model will be needed.

Exercise: Start with a simple serial program you have already written that is a good candidate for parallelization. Specifically, one that steps through parameter space in order to generate its results is a good candidate because you can have parallel tasks working on different regions of parameter space. Alternatively, a Monte–Carlo calculation that repeats the same step many times is also a good candidate because you can run copies of the program on different processors, and then add up the results at the end. For example, here is a serial calculation of π by Monte–Carlo integration in the C language:

Listing 1: **Serial Calculation of π by Monte–Carlo integration.**

```
// pi.c: *Monte-Carlo integration to determine pi

#include <stdio.h>
#include <stdlib.h>

// if you don't have drand48 uncomment the following two lines
```

```
//    #define drand48 1.0/RAND_MAX*rand
//    #define srand48 srand

#define max 1000                        // number of stones to be thrown
#define seed 68111                          // seed for number generator

main()  {

  int i, pi = 0;
  double x, y, area;
  FILE *output;                                  // save data in pond.dat
  output = fopen("pond.dat","w");
  srand48(seed);                               // seed the number generator
  for (i = 1; i<= max; i++) {
    x = drand48()*2-1;                          // creates floats between
    y = drand48()*2-1;                                    // 1 and -1
    if ((x*x + y*y)<1) pi++;                        // stone hit the pond
    area = 4*(double)pi/i;                              // calculate area
    fprintf(output, "%i\t%f\n", i, area);
  }
  printf("data stored in pond.dat\n");
  fclose(output);
}
```

Modify your serial program so that different processors are used to perform independently, and then have all their results combined. For example, here is a parallel version of `pi.c` that uses the Message Passing Interface (MPI). You may want to concentrate on the arithmetic commands and not the MPI ones at this point.

Listing 2: **MPI.c: Parallel Calculation of π by Monte–Carlo integration using MPI.**

```
   /* MPI.c uses a monte carlo method to compute PI  by Stone Throwing */
   /* Based on http://www.dartmouth.edu/~rc/classes/soft_dev/mpi.html   */
   /* Note: if the sprng library is not available, you may use rnd */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdio.h>
#include <sprng.h>
#include <mpi.h>
#define USE_MPI
#define SEED 35791246

main(int argc, char *argv[])
{
   int niter=0;
   double x,y;
   int i,j,count=0,mycount; /* # of points in the 1st quadrant of unit circle */
   double z;
   double pi;
   int myid,numprocs,proc;
   MPI_Status status;
   int master =0;
   int tag = 123;
   int *stream_id;              /* stream id generated by SPRNGS */

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);

   if (argc <=1) {
     fprintf(stderr,"Usage: monte_pi_mpi number_of_iterations\n");
     MPI_Finalize();
     exit(-1);
   }
```

```
    sscanf(argv[1],"%d",&niter); /* 1st argument is the number of iterations*/

      /* initialize random numbers */
    stream_id = init_sprng(myid,numprocs,SEED,SPRNG_DEFAULT);
    mycount=0;
    for ( i=0; i<niter; i++) {
       x = (double)sprng(stream_id);
       y = (double)sprng(stream_id);
       z = x*x+y*y;
       if (z<=1) mycount++;
    }
    if (myid ==0) { /* if I am the master process gather results from others */
       count = mycount;
       for (proc=1; proc<numprocs; proc++) {
          MPI_Recv(&mycount,1,MPI_REAL,proc,tag,MPI_COMM_WORLD,&status);
          count +=mycount;
       }
       pi=(double)count/(niter*numprocs)*4;
       printf("\n # of trials= %d , estimate of pi is %g \n",niter*numprocs,pi);
    }
    else { /* for all the slave processes send results to the master */
       printf("Processor %d sending results= %d to master process\n",myid,mycount
       );
       MPI_Send(&mycount,1,MPI_REAL,master,tag,MPI_COMM_WORLD);
    }

    MPI_Finalize();              /* let MPI finish up */

}
```

Although this small a problem is not worth your efforts in order to obtain a shorter run time, it is worth investing your time to gain some experience in parallel computing. In general, parallel computing holds the promise of permitting you to obtain faster results, to solve bigger problems, to run simulations at finer resolutions, or to model physical phenomena more realistically; but it takes some work to accomplish this.

9 Parallel Semantics (Theory)

We saw earlier that many of the tasks undertaken by a high–performance computer are run in parallel by making use of internal structures such as pipelined and segmented CPUs, hierarchical memory, and separate I/O processors. While these tasks are run "in parallel," the modern use of *parallel computing* or *parallelism* denotes applying multiple processors to a single problem [Quinn 04]. It is a computing environment in which some number of CPUs are running asynchronously and communicating with each other in order to exchange intermediate results and coordinate their activities.

For an instance, consider the matrix multiplication:

$$[B] = [A][B]. \tag{2}$$

Mathematically, this equation makes no sense unless $[A]$ equals the identity matrix $[I]$. However, it does make sense as an algorithm that produces new value of B on the LHS in terms of old values of B on the RHS:

$$[B^{\text{new}}] = [A][B^{\text{old}}] \tag{3}$$

$$\Rightarrow \; B^{\text{new}}_{i,j} = \sum_{k=1}^{N} A_{i,k} B^{\text{old}}_{k,j}. \tag{4}$$

Because the computation of $B^{\rm new}_{i,j}$ for specific values of i and j is independent of the computation of all the other values of $B^{\rm new}_{i,j}$, each $B^{\rm new}_{i,j}$ can be computed in parallel, or each row or column of $[B^{\rm new}]$ can be computed in parallel. If B were not a matrix, then we could just calculate $B = AB$ with no further ado. However, if we try to perform the calculation using just matrix elements of $[B]$ by replacing the old values with the new values as they are computed, then we must somehow ensure that the $B_{k,j}$ on the RHS of (4) are the values of $[B]$ that existed *before* the matrix multiplication.

This is an example of *data dependency*, in which the data elements used in the computation depend on the order in which they are used. A way to account for this dependency is to use a temporary matrix for $[B^{\rm new}]$, and then to assign $[B]$ to the temporary matrix after all multiplications are complete:

$$[\text{Temp}] = [A][B] \tag{5}$$

$$[B] = [\text{Temp}]. \tag{6}$$

In contrast, the matrix multiplication $[C] = [A][B]$ is a *data parallel* operation in which the data can be used in any order. So already we see the importance of communication, synchronization, and understanding of the mathematics behind an algorithm for parallel computation.

The processors in a parallel computer are placed at the *nodes* of a communication network. Each node may contain one CPU or a small number of CPUs, and the communication network may be internal to or external to the computer. One way of categorizing parallel computers is by the approach they employ in handling instructions and data. From this viewpoint there are three types of machines:

Single–instruction, single–data (SISD): These are the classic (von Neumann) serial computers executing a single instruction on a single data stream before the next instruction and next data stream are encountered.

Single–instruction, multiple–data (SIMD): Here instructions are processed from a single stream, but the instructions act concurrently on multiple data elements. Generally the nodes are simple and relatively slow but are large in number.

Multiple instructions, multiple data (MIMD): In this category each processor runs independently of the others with independent instructions and data. These are the types of machines that employ *message–passing* packages, such as MPI, to communicate among processors. They may be a collection of workstations linked via a network, or more integrated machines with thousands of processors on internal boards, such as the Blue Gene computer described in §16. These computers, which do not have a shared memory space, are also called *multicomputers*. Although these types of computers are some of the most difficult to program, their low cost and effectiveness for certain classes of problems have led to their being the dominant type of parallel computer at present.

The running of independent programs on a parallel computer is similar to the multitasking feature used by Unix and PCs. In multitasking (Figure 6 left) several independent programs reside in the computer's memory simultaneously and share the processing time in a round robin or priority order. On a SISD computer, only one program runs at a single time, but if other programs are in memory, then it does not take long to switch to them. In multiprocessing (Figure 6 right) these jobs may all run at the same time, either in different parts of memory or in the memory of different computers. Clearly, multiprocessing becomes complicated if separate processors are operating on different parts of the *same* program because then synchronization and load balance (keeping all the processors equally busy) are concerns.

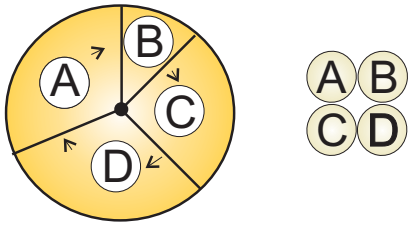


Figure 6: *Left:* Multitasking of four programs in memory at one time. On a SISD computer the programs are executed in round robin order. *Right:* Four programs in the four separate memories of a MIMD computer.

In addition to instructions and data streams, another way of categorizing parallel computation is by *granularity*. A *grain* is defined as a measure of the computational work to be done, more specifically, the ratio of computation work to communication work.

Coarse–grain parallel: Separate programs running on separate computer systems with the systems coupled via a conventional communication network. An illustration is six Linux PCs sharing the same files across a network but with a different central memory system for each PC. Each computer can be operating on a different, independent part of one problem at the same time.

Medium–grain parallel: Several processors executing (possibly different) programs simultaneously while accessing a common memory. The processors are usually placed on a common *bus* (communication channel) and communicate with each other through the memory system. Medium–grain programs have different, independent, *parallel subroutines* running on different processors. Because the compilers are seldom smart enough to figure out which parts of the program to run where, the user must include the multitasking routines in the program.[1]

Fine–grain parallel: As the granularity decreases and the number of nodes increases, there is an increased requirement for fast communication among the nodes. For this reason fine–grain systems tend to be custom–designed machines. The communication may be via a central bus or via shared memory for a small number of nodes, or through some form of high–speed network for massively parallel machines. In the latter case, the user typically divides the work via certain coding constructs, and the compiler just compiles the program. The program then runs concurrently on a user–specified number of nodes. For example, different `for` loops of a program may be run on different nodes.

10 Distributed Memory Programming

An approach to concurrent processing that, because it is built from commodity PCs, has gained dominant acceptance for coarse- and medium–grain systems is *distributed memory*. In it, each processor has its own memory and the processors exchange data among themselves through a high–speed switch and network. The data exchanged or *passed* among processors have encoded *to* and *from* addresses and are called *messages*. The *clusters* of PCs or workstations that constitute a

[1]Some experts define our medium grain as coarse grain yet this distinction changes with time.

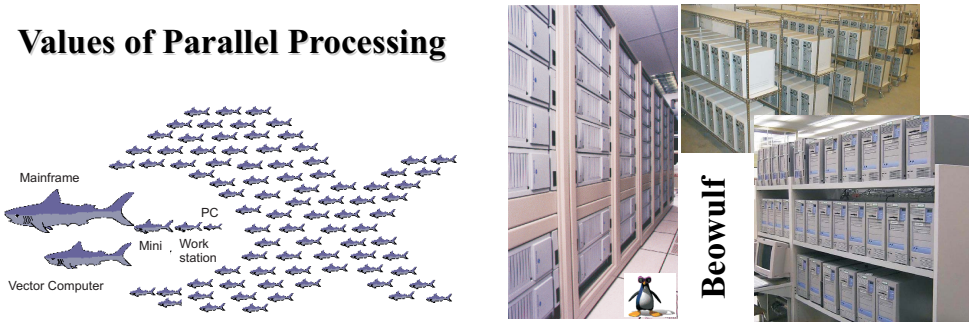


Figure 7: Two views of modern parallel computing (courtesy of Yuefan Deng).

Beowulf[2] are examples of distributed memory computers (Figure 7). The unifying characteristic of a cluster is the integration of highly replicated compute and communication components into a single system, with each node still able to operate independently. In a Beowulf cluster, the components are commodity ones designed for a general market, as are the communication network and its high–speed switch (special interconnects are used by major commercial manufacturers, but they do not come cheaply). *Note*: A group of computers connected by a network may also be called a cluster but unless they are designed for parallel processing, with the same type of processor used repeatedly and with only a limited number of processors (the *front end*) onto which users may log in, they are not usually called a Beowulf.

The literature contains frequent arguments concerning the differences among clusters, commodity clusters, Beowulfs, constellations, massively parallel systems, and so forth [Dong 05]. Even though we recognize that there are major differences between the clusters on the top 500 list of computers and the ones that a university researcher may set up in his or her lab, we will not distinguish these fine points in the introductory materials we present here.

For a message–passing program to be successful, the data must be divided among nodes so that, at least for a while, each node has all the data it needs to run an independent subtask. When a program begins execution, data are sent to all the nodes. When all the nodes have completed their subtasks, they exchange data again in order for each node to have a complete new set of data to perform the next subtask. This repeated cycle of data exchange followed by processing continues until the full task is completed. Message–passing MIMD programs are also *single–program, multiple–data* programs, which means that the programmer writes a single program that is executed on all the nodes. Often a separate host program, which starts the programs on the nodes, reads the input files and organizes the output.

11 Parallel Performance

Imagine a cafeteria line in which all the servers appear to be working hard and fast yet the ketchup dispenser has some relish partially blocking its output and so everyone in line must wait for the ketchup lovers up front to ruin their food before moving on. This is an example of the slowest step in a complex process determining the overall rate. An analogous situation holds for parallel processing, where the ketchup dispenser may be a relatively small part of the program that can

[2]Presumably there is an analogy between the heroic exploits of the son of Ecgtheow and the nephew of Hygelac in the 1000 C.E. poem *Beowulf* and the adventures of us common folk assembling parallel computers from common elements that have surpassed the performance of major corporations and their proprietary, multi–million–dollar supercomputers.

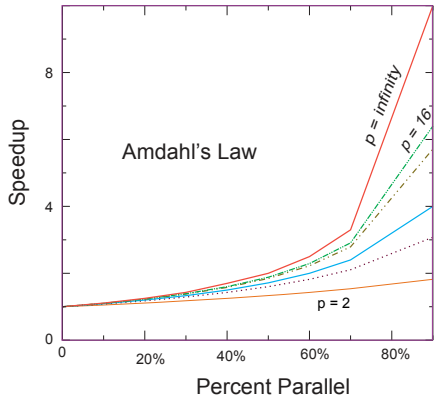


Figure 8: The theoretical maximum speedup of a program as a function of the fraction of the program that potentially may be run in parallel. The different curves correspond to different numbers of processors.

be executed only as a series of serial steps. Because the computation cannot advance until these serial steps are completed, this small part of the program may end up being the bottleneck of the program.

As we soon will demonstrate, the speedup of a program will not be significant unless you can get ∼90% of it to run in parallel, and even then most of the speedup will probably be obtained with only a small number of processors. This means that you need to have a computationally intense problem to make parallelization worthwhile, and that is one of the reasons why some proponents of parallel computers with thousands of processors suggest that you not apply the new machines to old problems but rather look for new problems that are both big enough and well–suited for massively parallel processing to make the effort worthwhile.

The equation describing the effect on speedup of the balance between serial and parallel parts of a program is known as Amdahl's law [Amd 67, Quinn 04]. Let

$$p = \text{number of CPUs}, \quad T_1 = \text{time to run on 1 CPU}, \quad T_p = \text{time to run on } p \text{ CPUs}. \tag{7}$$

The maximum speedup S_p attainable with parallel processing is thus

$$S_p = \frac{T_1}{T_p} \to p. \tag{8}$$

In practise this limit is never met for a number of reasons: some of the program is serial, data and memory conflicts occur, communication and synchronization of the processors take time, and it is rare to attain a perfect load balance among all the processors. For the moment we ignore these complications and concentrate on how the *serial* part of the code affects the speedup. Let f be the fraction of the program that potentially may run on multiple processors. The fraction $1 - f$ of the code that cannot be run in parallel must be run via serial processing and thus takes time:

$$T_s = (1 - f)T_1 \quad \text{(serial time)}. \tag{9}$$

The time T_p spent on the p parallel processors is related to T_s by

$$T_p = f\,\frac{T_1}{p}. \tag{10}$$

That being so, the maximum speedup as a function of f and the number of processors is

$$S_p = \frac{T_1}{T_s + T_p} = \frac{1}{1 - f + f/p} \quad \text{(Amdahl's law)}. \tag{11}$$

Some theoretical speedups are shown in Figure 8 for different numbers of processors p. Clearly the speedup will not be significant enough to be worth the trouble unless most of the code is run in parallel (this is where the 90% of the in-parallel figure comes from). Even an infinite number of processors cannot increase the speed of running the serial parts of the code, and so it runs at one processor speed. In practice this means many problems are limited to a small number of processors, and that only 10%–20% of the computer's peak performance is often all that is obtained for realistic applications.

11.1 Communication Overhead

As discouraging as Amdahl's law may seem, it actually *overestimates* speedup because it ignores the *overhead* for parallel computation. Here we look at communication overhead. Assume a completely parallel code so that its speedup is

$$S_p = \frac{T_1}{T_p} = \frac{T_1}{T_1/p} = p. \tag{12}$$

The denominator is based on the assumption that it takes no time for the processors to communicate. However, in reality it takes a finite time, called *latency*, to get data out of memory and into the cache or onto the communication network. In addition, a communication channel also has a finite *bandwidth*, that is, a maximum rate at which data can be transferred, and this too will increase the *communication time* as large amounts of data are transferred. When we include communication time T_c, the speedup decreases to

$$S_p \simeq \frac{T_1}{T_1/p + T_c} < p \quad \text{(with communication time)}. \tag{13}$$

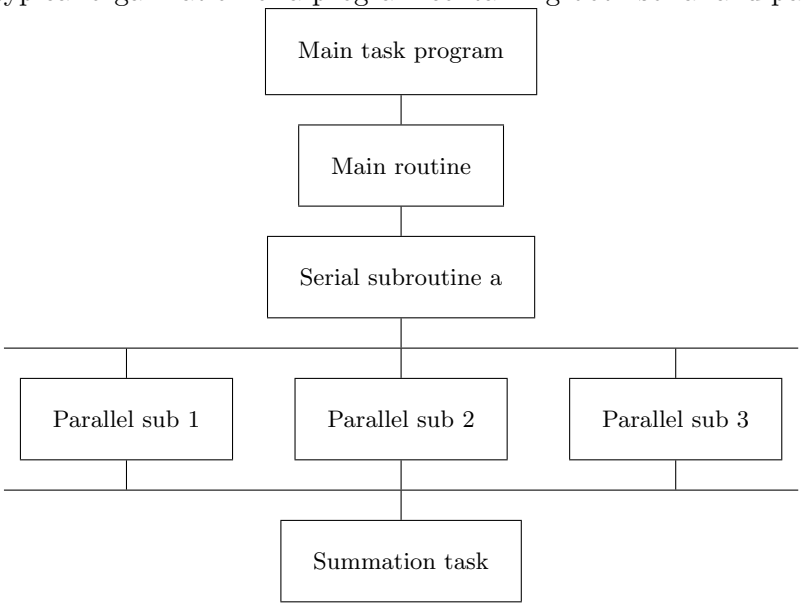
For the speedup to be unaffected by communication time, we need to have

$$\frac{T_1}{p} \gg T_c \;\; \Rightarrow \;\; p \ll \frac{T_1}{T_c}. \tag{14}$$

This means that as you keep increasing the number of processors p, at some point the time spent on computation T_1/p must equal the time T_c needed for communication, and adding more processors leads to greater execution time as the processors wait around more to communicate. This is another limit, then, on the maximum number of processors that may be used on any one problem, as well as on the effectiveness of increasing processor speed without a commensurate increase in communication speed.

The continual and dramatic increase in the number of processors being used in computations is leading to a changing view as to how to judge the speed of an algorithm. Specifically, the slowest step in a process is usually the rate–determining step, yet with the increasing availability of CPU power the slowest step is more often the access to or communication among processors. Such being the case, while the number of computational steps is still important for determining an algorithm's speed, the number and amount of memory access and interprocessor communication must also be mixed into the formula. This is currently an active area of research in algorithm development.

Table 4: A typical organization of a program containing both serial and parallel tasks.



12 Parallelization Strategies

A typical organization of a program containing both serial and parallel tasks is given in Table 4. The user organizes the work into units called *tasks*, with each task assigning work (*threads*) to a processor. The main task controls the overall execution as well as the subtasks that run independent parts of the program (called *parallel subroutines*, *slaves*, *guests*, or *subtasks*). These parallel subroutines can be distinctive subprograms, multiple copies of the same subprogram, or even `for` loops.

It is the programmer's responsibility to ensure that the breakup of a code into parallel subroutines is mathematically and scientifically valid and is an equivalent formulation of the original program. As a case in point, if the most intensive part of a program is the evaluation of a large Hamiltonian matrix, you may want to evaluate each row on a different processor. Consequently, the key to parallel programming is to identify the parts of the program that may benefit from parallel execution. To do that the programmer should understand the program's data structures (see below), know in what order the steps in the computation must be performed, and know how to coordinate the results generated by different processors.

The programmer helps speed up the execution by keeping many processors simultaneously busy and by avoiding storage conflicts among different parallel subprograms. You do this *load balancing* by dividing your program into subtasks of approximately equal numerical intensity that will run simultaneously on different processors. The rule of thumb is to make the task with the largest granularity (workload) dominant by forcing it to execute first and to keep all the processors busy by having the number of tasks an integer multiple of the number of processors. This is not always possible.

The individual parallel threads can have *shared* or *local* data. The shared data may be used by all the machines, while the local data are private to only one thread. To avoid storage conflicts, design your program so that parallel subtasks use data that are independent of the data in the main task and in other parallel tasks. This means that these data should not be modified *or even examined* by different tasks simultaneously. In organizing these multiple tasks, reduce communication *overhead costs* by limiting communication and synchronization. These costs tend to be high for fine–grain

programming where much coordination is necessary. However, *do not* eliminate communications that are necessary to ensure the scientific or mathematical validity of the results; bad science can do harm!

13 Practical Aspects of MIMD Message Passing

It makes sense to run only the most numerically intensive codes on parallel machines. Frequently these are very large programs assembled over a number of years or decades by a number of people. It should come as no surprise, then, that the programming languages for parallel machines are primarily Fortran, which has explicit structures for the compiler to parallelize, and C. [In the past we have not obtained good speedup with Java and MPI, yet we are told that *F-MPJ* [FMPF] and *MPJ Express* [MPJ] have fixed the problems.]

Effective parallel programming becomes more challenging as the number of processors increases. Computer scientists suggest that it is best *not* to attempt to modify a serial code but instead to rewrite it from scratch using algorithms and subroutine libraries best suited to parallel architecture. However, this may involve months or years of work, and surveys find that ∼70% of computational scientists revise existing codes instead [Pan 96].

Most parallel computations at present are done on multiple–instruction, multiple–data computers via message passing using MPI. Next we outline some practical concerns based on user experience [Dong 05, Pan 96].

Parallelism carries a price tag: There is a steep learning curve requiring intensive effort. Failures may occur for a variety of reasons, especially because parallel environments tend to change often and get "locked up" by a programming error. In addition, with multiple computers and multiple operating systems involved, the familiar techniques for debugging may not be effective.

Preconditions for parallelism: If your program is run thousands of times between changes, with execution time in days, and you must significantly increase the resolution of the output or study more complex systems, then parallelism is worth considering. Otherwise, and to the extent of the difference, parallelizing a code may not be worth the time investment.

The problem affects parallelism: You must analyze your problem in terms of how and when data are used, how much computation is required for each use, and the type of problem architecture.

Perfectly parallel: The same application is run simultaneously on different data sets, with the calculation for each data set independent (e.g., running multiple versions of a Monte Carlo simulation, each with different seeds, or analyzing data from independent detectors). In this case it would be straightforward to parallelize with a respectable performance to be expected.

Fully synchronous: The same operation applied in parallel to multiple parts of the same data set, with some waiting necessary (e.g., determining positions and velocities of particles simultaneously in a molecular dynamics simulation). Significant effort is required, and unless you balance the computational intensity, the speedup may not be worth the effort.

Loosely synchronous: Different processors do small pieces of the computation but with intermittent data sharing (e.g., diffusion of groundwater from one location to another). In this case it would be difficult to parallelize and probably not worth the effort.

Pipeline parallel: Data from earlier steps processed by later steps, with some overlapping of processing possible (e.g., processing data into images and then into animations). Much work may be involved, and unless you balance the computational intensity, the speedup may not be worth the effort.

13.1 High–Level View of Message Passing

Although it is true that parallel computing programs may become very complicated, the basic ideas are quite simple. All you need is a regular programming language like C or Fortran, plus four communication statements:[3]

- `send`**:** One processor sends a message to the network.
- `receive`**:** One processor receives a message from the network.
- `myid`**:** An integer that uniquely identifies each processor.
- `numnodes`**:** An integer giving the total number of nodes in the system.

Once you have made the decision to run your program on a computer cluster, you will have to learn the specifics of a message–passing system such as MPI. Here we give a broader view. When you write a message–passing program, you intersperse calls to the message–passing library with your regular Fortran or C program. The basic steps are

1. Submit your job from the command line or a job control system.
2. Have your job start additional processes.
3. Have these processes exchange data and coordinate their activities.
4. Collect these data and have the processes stop themselves.

We show this graphically in Figure 9 where at the top we see a *master* process create two *slave* processes and then assign work for them to do (arrows). The processes then communicate with each other via message passing, output their data to files, and finally terminate.

What can go wrong: Figure 9 also illustrates some of the difficulties:

- The programmer is responsible for getting the processes to cooperate and for dividing the work correctly.
- The programmer is responsible for ensuring that the processes have the correct data to process and that the data are distributed equitably.
- The commands are at a lower level than those of a compiled language, and this introduces more details for you to worry about.
- Because multiple computers and multiple operating systems are involved, the user may not receive or understand the error messages produced.
- It is possible for messages to be sent or received not in the planned order.

[3]Personal communication, Yuefan Deng.

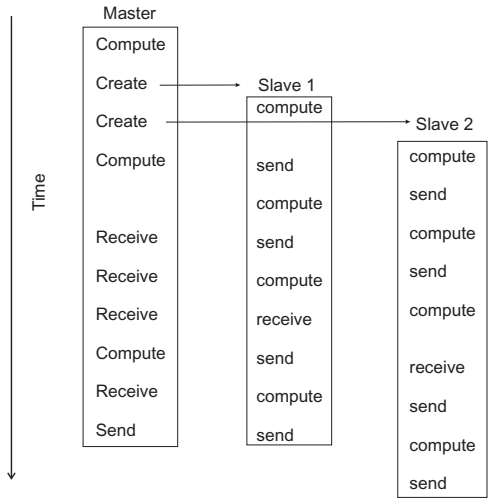


Figure 9: A master process and two slave processes passing messages. Notice how in this not-well-designed program there are more sends than receives, and consequently the results may depend upon the order of execution, or the program may even lock up.

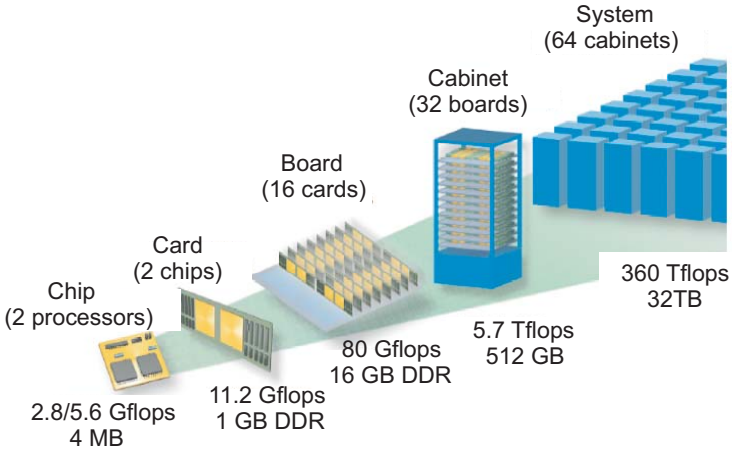


Figure 10: The building blocks of Blue Gene (from [Gara 05]).

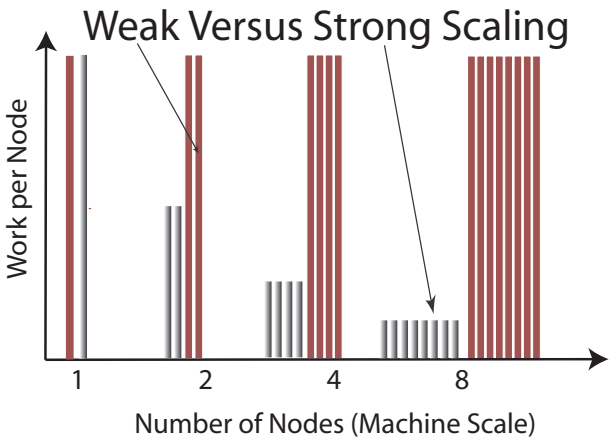


Figure 11: A graphical representation of weak versus strong scaling. Weak scaling keeps each node doing the same amount of work as the problem is made bigger. Strong scaling has each node doing less work (running for less time) as the number of nodes is made bigger. (From [Sterl 11].)

- A *race condition* may occur in which the program results depend upon the specific ordering of messages. There is no guarantee that slave 1 will get its work done before slave 2, even though slave 1 may have started working earlier (Figure 9).
- Note in Figure 9 how different processors must wait for signals from other processors; this is clearly a waste of time and has potential for deadlock.
- Processes may *deadlock*, that is, wait for a message that never arrives.

14 Scalability

A common discussion at HPC and Supercomputing conferences of the past heard application scientists get up, after hearing about the latest machine with what seemed like an incredible number of processors, and ask "But how can I use such a machine on my problem, which takes hours to run, but is not trivially parallel like your example?". The response from the computer scientist was often something like "You just need to think up some new problems that are more appropriate to the machines being built. Why use a supercomputer for a problem you can solve on a modern laptop?" It seems that these anecdotal exchanges have now been incorporated into the fabric of parallel computing under the title of *scalability*. In the most general sense, *scalability is defined as the ability to handle more work as the size of the computer or application grows.*

As we have already indicated, the primary challenge of parallel computing is deciding how best to break up a problem into individual pieces that can each be computed separately. In an ideal world a problem would *scale* in a linear fashion, that is, the program would speed up by a factor of N when it runs on a machine having N nodes. (Of course, as $N \to \infty$ the proportionality cannot hold because communication time must then dominate). In present day terminology, this type of

scaling is called **strong scaling**, and refers to a situation in which the problem size remains fixed while the number of number of nodes (the *machine scale*) increases. Clearly then, the goal when solving a problem that scales strongly is to decrease the amount of time it takes to solve the problem by using a more powerful computer. These are typically CPU-bound problems and are the hardest ones to yield something close to a linear speedup.

In contrast to strong scaling in which the problem size remains fixed, in **weak scaling** we have applications of the type our CS colleagues referred to above; namely, ones in which we make the problem bigger and bigger as the number of processors increases. So here, we would have linear or perfect scaling if we could increase the size of the problem solved in proportion to the number N of nodes.

To illustrate the difference between strong and weak scaling, consider Figure 11 (based on a lecture by Thomas Sterling). We see that for an application that scales perfectly strongly, the work done on each node decreases as the scale of the machine increases, which of course means that the time it takes to complete the problem decreases linearly. In contrast, we see that for an application that scales perfectly weakly, the work done by each node remains the same as the scale of the machine increases, which means that we are solving progressively larger problems in the same time as it takes to solve smaller ones on a smaller machine.

The concepts of weak and strong scaling are ideals that tend not to be achieved in practice, with real world applications having some of each present. Furthermore, it is the combination of application and computer architecture that determine the type of scaling that occurs. For example, shared memory systems and distributed-memory, message passing systems scale differently. Furthermore, a *data parallel* application (one in which each node can work on its own separate data set) will by its very nature scale weakly.

Before we go on and set you working on some examples of scaling, we should introduce a note of caution. Realistic applications tend to have various levels of complexity and so it may not be obvious just how to measure the increase in "size" of a problem. As an instance, it is known that the solution of a set of N linear equations via Gaussian elimination requires $\mathrm{O}(N^3)$ floating-point operations (flops). This means that doubling the number of equations does not make the "problem" twice as large, but rather eight times as large! Likewise, if we are solving partial differential equations on a three-dimensional spatial grid and a 1-D time grid, then the problem size would scale like N^4. In this case doubling the problem size would mean increasing N by only $2^{1/4} \simeq 1.19$.

14.1 Scalability Exercises

We have given above, and included in the Codes directory, a serial code `pi.c` that computes π by Monte-Carlo integration of a quarter of a unit circle (we know that area is $\pi/4$). We have also given the code `MPIpi.c` that computes π by the same algorithm using MPI to compute the algorithm in parallel. Your exercise is to see how well this application scales. You can modify the codes we have given, or you can write your own.

1. Determine the CPU time required to calculate π with the serial calculation using 1000 iterations (stone throws). Make sure that this is the actual run time and does not include any system time. (You can get this type of information, depending upon the operating system, by inserting `timer` calls in your program.)

2. Get the MPI code running for the same number (1000) of iterations.

3. First let's do some running that constitutes a **strong scaling test**. This means keeping the problem size constant, or in other words, keeping $N_{iter} = 1000$. Start by running the MPI

code with only one processor doing the numerical computation. A comparison of this to the serial calculation gives you some idea of the overhead associated with MPI.

4. Again keeping $N_{iter} = 1000$, run the MPI code for 2, 4, 8 and 16 computing nodes. In any case, make sure to go up to enough nodes so that the system no longer scales. Record the run time for each number of nodes.

5. Make a plot of run time versus number of nodes from the data you have collected.

6. Strong scalability here would yield a straight line graph. Comment on your results.

7. Now let's do some running that constitutes a **weak scaling test**. This means increasing the problem size simultaneously with the number of nodes being used. In the present case, increasing the number of iterations, N_{iter}.

8. Run the MPI code for 2, 4, 8 and 16 computing nodes, with proportionally larger values for N_{iter} in each case (2000, 4000, 8000, 16,000 *etc.*). In any case, make sure to go up to enough nodes so that the system no longer scales.

9. Record the run time for each number of nodes and make a plot of the run time versus number of computing nodes.

10. Weak scaling would imply that the runtime remains constant as the problem size and the number of compute nodes increase in proportion. Comment on your results.

11. Is this problem more appropriate for weak or strong scaling?

15 Data Parallelism and Domain Decomposition

As you have probably realized by this point, there are two basic, but quite different, approaches to creating a program that runs in parallel. In **task parallelism** you decompose your program by tasks, with differing tasks assigned to different processors, and with great care taken to maintain *load balance*, that is, to keep all processors equally busy. Clearly you must understand the internal workings of your program in order to do this, and must also have made an accurate *profile* of your program so that you know how much time is being spent in various parts.

In **data parallelism** you decompose your program based on the data being created or acted upon, with differing data spaces (**domains**) assigned to different processors. In data parallelism, there often must be data shared at the boundaries of the data spaces, and synchronization among the data spaces. Data parallelism is the most common approach and is well suited to message-passing machines in which each node has its own private data space, although this may lead to a large amount of data transfer at times.

When planning how to decompose global data into subspaces suitable for parallel processing, it is important divide the data into contiguous blocks in order to minimize the time spent on moving data through the different stages of memory (page faults). Some compilers and operating systems help you in this regard by exploiting **spatial locality**, that is, by assuming that if you are using a data element at one location in data space, then it is likely that you may use some nearby ones as well, and so they are also made readily available. Some compilers and operating systems also exploit **temporal locality**, that is, by assuming that if you are using a data element at one time, then there is an increased probability that you may use it again in the near future, and so it too is kept handy. You can help optimize your programs by taking advantage of these localities while programming.

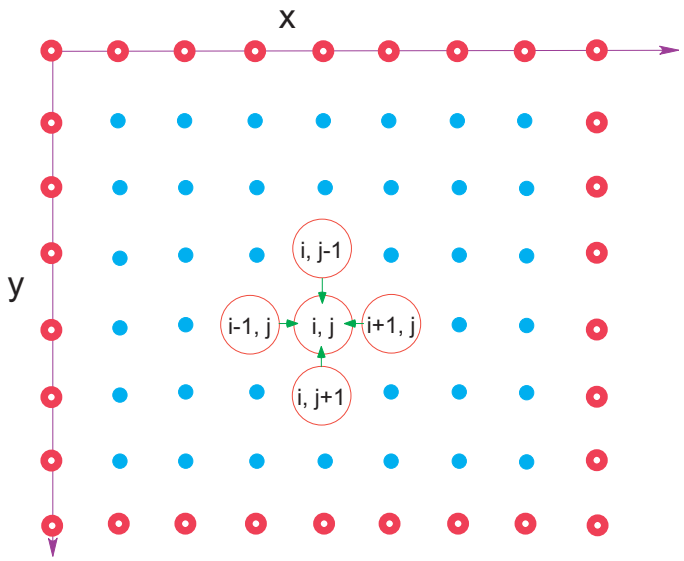


Figure 12: A representation of the lattice in a 2-D rectangular space upon which Laplace's equation is solved using a finite difference approach (a realistic lattice contains hundreds or thousands of sites). The lattice sites with white centers correspond to the boundary of the physical system, upon which boundary conditions must be imposed for a unique solution. The large circles in the middle of the lattice represent the algorithm used to solve Laplace's equation in which the potential at the point $(x, y) = (i, j)\Delta$ is set to the average of the potential values at the four nearest-neighbor points.

As an example of **domain decomposition** consider the solution of a partial differential equation by a finite difference method. (Derivations, as well as discussions of the specific techniques used for various types of partial differential equations can be found in [LPB 08].) It is known from classical electrodynamics that the electric potential $U(\mathbf{x})$ in a charge free region of 2-D space satisfies *Laplace's equation* (here in 2-D rectangular coordinates):

$$\frac{\partial^2 U(x,y)}{\partial x^2} + \frac{\partial^2 U(x,y)}{\partial y^2} = 0. \tag{15}$$

We see that the potential depends simultaneously on x and y, which is what makes it a partial differential equation. The charges, which are the source of the field, enter indirectly by specifying the potential values on some boundaries or charged objects.

The finite difference algorithm for the numeric solution of (15) is based on the forward-difference algorithm for the second derivatives, for example,

$$\frac{\partial^2 U(x,y)}{\partial x^2} \simeq \frac{U(x+\Delta x, y) + U(x-\Delta x, y) - 2U(x,y)}{(\Delta x)^2}. \tag{16}$$

As shown in Figure 12, we look for a solution on a lattice of (x, y) values separated by finite difference Δ in each dimension and specified by discrete locations on the lattice:

$$x = x_0 + i\Delta, \quad y = y_0 + j\Delta, \quad i, j = 0, \ldots, N_{max-1}. \tag{17}$$

When the finite difference expressions for the derivatives are substituted into (15), and the equation is rearranged, we obtain the finite-difference algorithm for the solution of Laplace's equation:

$$U_{i,j} = \frac{1}{4}\left[U_{i+1,j} + U_{i-1,j} + U_{i,j+1} + U_{i,j-1}\right]. \tag{18}$$

This equation says that when we have a proper solution, it will be the average of the potential at the four nearest neighbors (Figure 12). As an algorithm, (18) does not provide a direct solution to Laplaces's equation but rather must be repeated many times to converge upon the solution. We start with an initial guess for the potential, improve it by sweeping through all space taking the average over nearest neighbors at each node, and keep repeating the process until the solution no longer changes to some level of precision or until failure is evident. When converged, the initial guess is said to have *relaxed* into the solution.

Below we list a serial code `laplace.c` that solves the Laplace equation in two dimensions for a straight wire kept at 100 V in a grounded box, using the relaxation algorithm (18). There are five basic elements of the code:

1. Initialize the potential values on the lattice.
2. Provide an initial guess for the potential, in this case $U = 0$ except for the wire at 100V.
3. Maintain the boundary values and the source term values of the potential at all times.
4. Iterate the solution until convergence [(18) being satisfied to some level of precision] is obtained.
5. Output the potential in a form appropriate for 3-D plotting with gnuplot.

As you can see, the code is a simple pedagogical example with its essential structure being the array `p[40][40]` representing a (small) regular lattice. Industrial strength applications might use much larger lattices as well as adaptive meshes and hierarchical multigrid techniques.

When thinking of parallelizing this program, we note an algorithm being applied to a space of data points, in which case we can divide the domain into subspaces and assign each subspace to a different processor. This is domain decomposition or data parallelism. In essence, we have divided a large boundary-value problem into an equivalent set of smaller boundary-value problems that eventually get fit back together. Often extra storage is allocated on each processor to hold the data values that get communicated from neighboring processors. These storage locations are often referred to as *ghost cells, ghost rows, ghost columns, halo cells,* or *overlap areas.*

Two points are essential in domain decomposition: 1) Load balancing is critical, and is obtained here by having each domain contain the same number of lattice points. 2) Communication among the processors should be minimized because this is a slow process. Clearly the processors must communicate to agree on the potential values at the domain boundaries, except for those boundaries on the edge of the box that remain grounded at 0 V. But since there are many more lattice sites that need computing than there are boundaries, communication should not slow down the computation severely for large lattices.

To see an example of how this is done, in the Codes directory we have included the serial code `poisson_1d.c` that solves Laplace's equation in 1-D, and `poisson_parallel_1d.c` that solves the same 1-D equation in parallel (codes are courtesy of Michel Vallieres). This code uses an accelerated version of the iteration algorithm using the parameter Ω [LPB 08], a separate method for domain decomposition, as well as ghost cells to communicate the boundaries.

Listing 3: **laplace.c Serial solution of Laplace's equation using a finite difference technique (from [LPB 08]).**

```
   /*
*********************************************************************
*  laplace.c:  Solution of Laplace equation with finite differences    *
*                    *
```

```
*  From: "A SURVEY OF COMPUTATIONAL PHYSICS"
   by RH Landau, MJ Paez, and CC BORDEIANU
   Copyright Princeton University Press, Princeton, 2008.
   Electronic Materials copyright: R Landau, Oregon State Univ, 2008;
   MJ Paez, Univ Antioquia, 2008; & CC BORDEIANU, Univ Bucharest, 2008
   Support by National Science Foundation
*                                   *
*  comment: Output data is saved in 3D grid format used by gnuplot     *
*************************************************************************
*/
#include <stdio.h>

#define max 40                          /* number of grid points */

main()
{
   double x, p[max][max];
   int i, j, iter, y;
   FILE *output;       /* save data in laplace.dat */
   output = fopen("laplace.dat","w");
   for(i=0; i<max; i++)                  /* clear the array  */
   {    for (j=0; j<max; j++) p[i][j] = 0;}
   for(i=0; i<max; i++) p[i][0] = 100.0;        /* p[i][0] = 100 V */
   for(iter=0; iter<1000; iter++)               /* iterations */
   {    for(i=1; i<(max-1); i++)                  /* x-direction */
      { for(j=1; j<(max-1); j++)                /* y-direction */
        {  p[i][j] = 0.25*(p[i+1][j]+p[i-1][j]+p[i][j+1]+p[i][j-1]);   }
      }
   }
   for (i=0; i<max ; i++)          /* write data gnuplot 3D format */
   {    for (j=0; j<max; j++)
      { fprintf(output, "%f\n",p[i][j]);    }
      fprintf(output, "\n");    /* empty line for gnuplot */
   }
   printf("data stored in laplace.dat\n");
   fclose(output);
}
```

15.1 Domain Decomposition Exercises

1. Get the serial version of either `laplace.c` or `laplace.f` running.

2. Increase the lattice size to 1000 and determine the CPU time required for convergence to six places. Make sure that this is the actual run time and does not include any system time. (You can get this type of information, depending upon the operating system, by inserting `timer` calls in your program.)

3. Decompose the domain into four subdomains and get an MPI version of the code running on four compute nodes. [Recall, we give an example of how to do this in the Codes directory with the serial code `poisson_1d.c` and its MPI implementation, `poisson_parallel_1d.c` (courtesy of Michel Vallieres).]

4. Convert the serial code to three dimensions. This makes the application more realistic, but also more complicated. Describe the changes you have had to make.

5. Decompose the 3-D domain into four subdomains and get an MPI version of the code running on four compute nodes. This can be quite a bit more complicated than the 2-D problem.

6. *Conduct a weak scaling test for the 2-D or 3-D code.

7. *Conduct a strong scaling test for the 2-D or 3-D code.

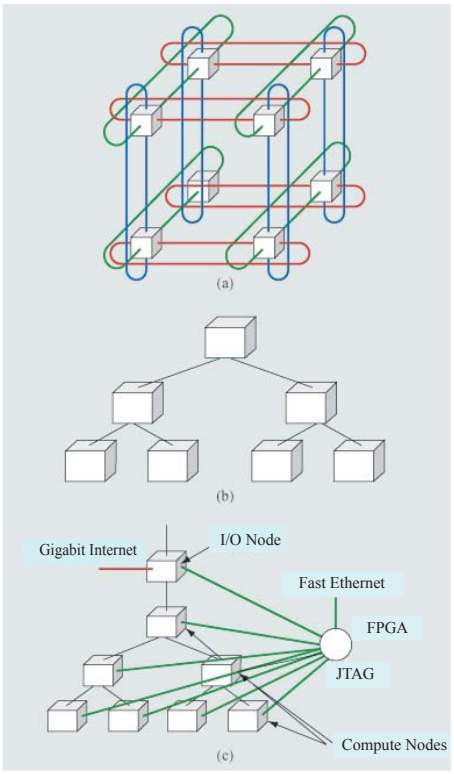


Figure 13: *(a)* A 3-D torus connecting $2 \times 2 \times 2$ compute nodes. *(b)* The global collective memory system. *(c)* The control and Gb-Ethernet memory system (from [Gara 05]).

16 The IBM Blue Gene Supercomputers

Whatever figures we give to describe the latest supercomputer will be obsolete by the time you read them. Nevertheless, for the sake of completeness and to set the present scale, we do it anyway. At the time of this writing the fastest computer is the IBM Blue Gene/Q member of the Blue Gene series. In its largest version, its 96 racks contains 98,304 compute nodes with 1.6 million processor cores and 1.6 PB of memory [Top 500]. In June 2012 it reached a peak speed of 20.1 PFLOPS.

The name Blue Gene reflects the computer's origin in gene research, although Blue Genes are now general–purpose supercomputers. In many ways these are computers built by committee, with compromises made in order to balance cost, cooling, computing speed, use of existing technologies, communication speed, and so forth. As a case in point, the compute chip has 18 cores, with 16 for computing, one to assist the operating system with communication, and one as a redundant spare in case one of the others was damaged. Having communication on the chip reflects the importance of communication for distributed–memory computing (there are both on– and off–chip distributed memories). And while the CPU is fast with 204.8 GFLOPS at 1.6 GHz, there are faster ones made, but they would generate so much heat that it would not be possible to obtain the extreme scalability up to 98,304 compute nodes. So with the high efficiency figure of 2.1 GFLOPS/watt, Blue Gene is considered a "green"computer.

We look more closely now at one of the original Blue Genes, for which we were able to obtain illuminating figures [Gara 05]. Consider the building–block view in Figure 10. We see multiple cores on a chip, multiple chips on a card, multiple cards on a board, multiple boards in a cabinet, and multiple cabinets in an installation. Each processor runs a Linux operating system (imagine what

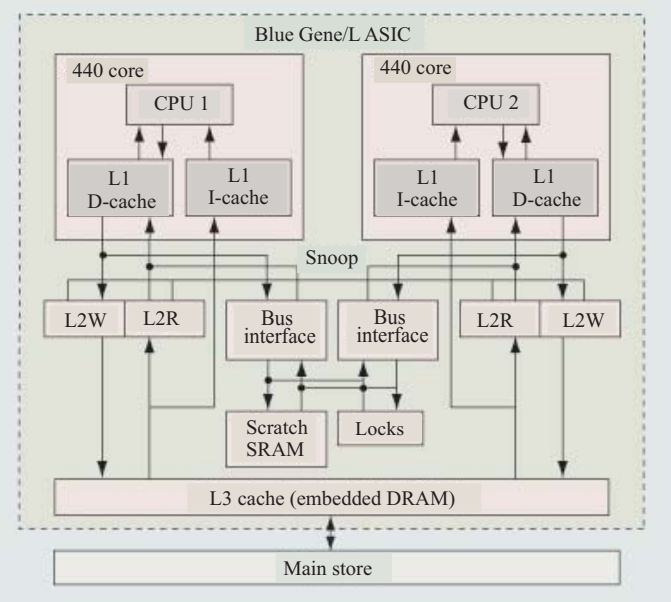


Figure 14: The single–node memory system (as presented by [Gara 05]).

the cost in both time and money would be for Windows!) and utilizes the hardware by running a distributed memory MPI with C, C++, and Fortran90 compilers.

Blue Gene has three separate communication networks (Figure 13). At the heart of the network is a 3-D torus that connects all the nodes; Figure 13a shows a sample torus of $2 \times 2 \times 2$ nodes. The links are made by special link chips that also compute; they provide both direct neighbor-neighbor communications as well as cut–through communication across the network. The result of this sophisticated network is that there is approximately the same effective bandwidth and latencies (response times) between all nodes. However, the bandwidth may be affected by interference among messages, with the actual latency also depending on the number of *hops* to get from one node to another[4]. A rate of 1.4 Gb/s is obtained for node–to–node communication. Finally, the collective network in Figure 13b is used for collective communications among processors, such as a *broadcast*. Finally, the control and gigabit ethernet network (Figure 13c) is used for I/O to communicate with the switch (the hardware communication center) and with ethernet devices.

The computing heart of Blue Gene is its integrated circuit and the associated memory system (Figure 14). This is essentially an entire computer system on a chip, with the exact specifications depending upon the model and changing with time.

17 Towards Exascale Computing via Multinode–Multicore–GPU Computers

The current architecture of top–end supercomputers uses a very large numbers of nodes, with each node containing a chip set that includes multiple cores (up to 32 at present) as well as a graphical processing unit (GPU) attached to the chip set (Figure 15). In the near future we expect to see laptop computers capable of teraflops (10^{12} floating point operations per second), deskside computers capable of petaflops, and supercomputers at the exascale in terms of both flops and memory.

[4]The number of hops is the number of devices a given data packet passes through.

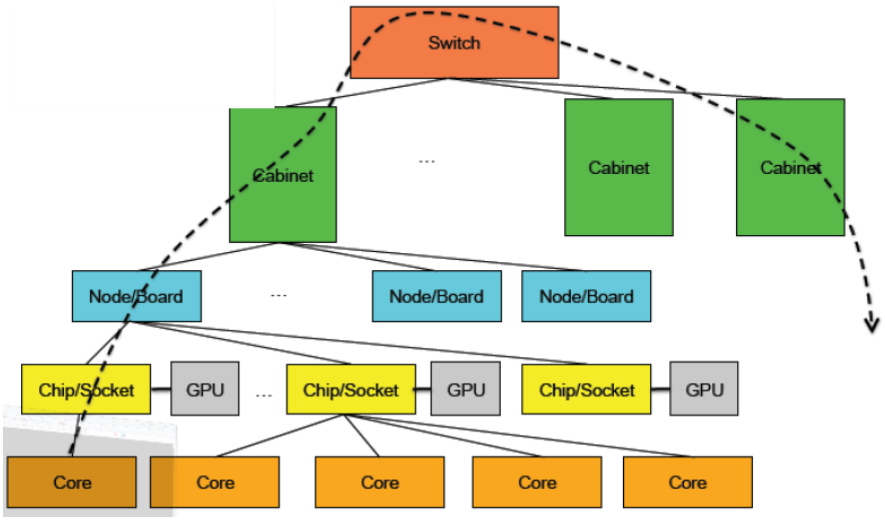


Figure 15: A schematic of an exascale computer in which, in addition to each chip having multiple cores, a graphical processing unit is attached to each chip. (Adapted from [Dong 11].)

Look again at the schematic in Figure 15. As in Blue Gene, there really are large numbers of chip boards and large numbers of cabinets. Here we show just one node and one cabinet and not the full number of cores. The dashed line in Figure 15 represents communications, and it is seen to permeate all components of the computer. Indeed, communications have become such an essential part of modern supercomputers, which may contain 100's of 1000's of CPUs, that the network interface "card" may be directly on the chip board. Because a computer of this sort contains shared memory at the node level and distributed memory at the cabinet or higher levels, programming for the requisite data transfer among the multiple elements is the essential challenge.

The GPU component in Figure 15 extends this type of supercomputer's architecture beyond that of the previously–discussed IBM Blue Gene. The GPU is an electronic device designed to accelerate the creation of visual images as they are displayed on a graphical output device. A GPU's efficiency arises from its ability to create many different parts of an image in parallel, an important ability since there are millions of pixels to display simultaneously. Indeed, these units can process 100s of millions of polygons in a second. Because GPUs are designed to assist the video processing on commodity devices such as personal computers, game machines, and mobile phones, they have become inexpensive, high performance, parallel computers in their own right. Yet because GPUs are designed to assist in video processing, their architecture and their programming are different from that of the general purpose CPUs usually used for scientific algorithms, and it takes some work to use them for scientific computing.

Programming of GPUs requires specialized tools specific to the GPU architecture being used, and are beyond the discussion of this module. For example, CUDA programming refers to programming for the architecture developed by Nvidia, and is probably the most popular approach at present. Although possible to program at the basic level, there are now extensions and wrappers developed for popular programming languages such as C, Fortran, Python, Java and Perl. However, the general principles involved are just an extension of those used already discussed, and after we have you work through some examples of those general principles, in §21 we give some practical tips for programming multinode–multicore–GPU computers.

18 General HPC Program Optimization

The type of optimization often associated with *high–performance* or *numerically intensive* computing is one in which sections of a program are rewritten and reorganized in order to increase the program's speed. The overall value of doing this, especially as computers have become so fast and so available, is often a subject of controversy between computer scientists and computational scientists. Both camps agree that using the optimization options of compilers is a good idea. However, computational scientists tend to run large codes with large amounts of data in order to solve real–world problems and often believe that you cannot rely on the compiler to do all the optimization, especially when you end up with time on your hands while waiting for the computer to finish executing your program. Here are some entertaining, yet insightful, views [Har 96] reflecting the conclusions of those who have reflected upon this issue:

More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason—including blind stupidity.

— W.A. Wulf

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.

— Donald Knuth

The best is the enemy of the good.

— Voltaire

Rule 1: Do not do it.
Rule 2 (for experts only): Do not do it yet.
Do not optimize as you go: Write your program without regard to possible optimizations, concentrating instead on making sure that the code is clean, correct, and understandable. If it's too big or too slow when you've finished, then you can consider optimizing it.
Remember the 80/20 rule: In many fields you can get 80% of the result with 20% of the effort (also called the 90/10 rule—it depends on who you talk to). Whenever you're about to optimize code, use profiling to find out where that 80% of execution time is going, so you know where to concentrate your effort.
Always run "before" and "after" benchmarks: How else will you know that your optimizations actually made a difference? If your optimized code turns out to be only slightly faster or smaller than the original version, undo your changes and go back to the original, clear code.
Use the right algorithms and data structures: Do not use an $\mathcal{O}(n^2)$ DFT algorithm to do a Fourier transform of a thousand elements when there's an $\mathcal{O}(n \log n)$ FFT available. Similarly, do not store a thousand items in an array that requires an $\mathcal{O}(n)$ search when you could use an $\mathcal{O}(\log n)$ binary tree or an $\mathcal{O}(1)$ hash table.

18.1 Programming for Virtual Memory (Method)

While paging makes little appear big, you pay a price because your program's run time increases with each page fault. If your program does not fit into RAM all at once, it will run significantly slower. If virtual memory is shared among multiple programs that run simultaneously, they all can't have the entire RAM at once, and so there will be memory access *conflicts*, in which case the performance of all the programs will suffer. The basic rules for programming for virtual memory are:

1. Do not waste your time worrying about reducing the amount of memory used (the *working set size*) unless your program is large. In that case, take a global view of your entire program

and optimize those parts that contain the largest arrays.

2. Avoid page faults by organizing your programs to successively perform their calculations on subsets of data, each fitting completely into RAM.

3. Avoid simultaneous calculations in the same program to avoid competition for memory and consequent page faults. Complete each major calculation before starting another.

4. Group data elements close together in memory blocks if they are going to be used together in calculations.

18.2 Optimizing Programs; Python *versus* Fortran/C

Many of the optimization techniques developed for Fortran and C are also relevant for Python applications. Yet while Python is a good language for scientific programming and is as universal and portable as Java, at present Python code runs slower than Fortran, C or even Java code. In part, this is a consequence of the Fortran and C compilers having been around longer and thereby having been better refined to get the most out of a computer's hardware, and in part this is also a consequence of Python not being designed for speed. Since modern computers are so fast, whether a program takes 1s or 3s usually does not matter much, especially in comparison to the hours or days of *your* time that it might take to modify a program for different computers. However, you may want to convert the code to C (whose command structure is similar to that of Python) if you are running a computation that takes hours or days to complete and will be doing it many times.

Especially when asked to, compilers may look at your entire code as a single entity and rewrite it in an attempt to make it run faster. In particular, Fortran and C compilers often speed up programs by being careful in how they load arrays into memory. They also are careful in keeping the cache lines full so as not to keep the CPU waiting or having it move on to some other chore. That being said, compilers still cannot optimize a program as well as can a skilled and careful programmer who understands the order and structure of the code.

There is no fundamental reason why a program written in Java or Python cannot be compiled to produce a highly efficient code, and indeed such compilers are being developed and becoming available. However, such code is optimized for a particular computer architecture and so is not portable. In contrast, the byte code (`.class` file and `.pyc` in Python) produced is designed to be interpreted or recompiled by the *Java or Python Virtual Machine* (just another program). When you change from Unix to Windows, for example, the Java Virtual Machine program changes, but the byte code is the same. This is the essence of Java's portability.

In order to improve the performance of Java and Python, many computers and browsers run a *Just-in-Time* (JIT) compilers. If a JIT is present, the Virtual Machine feeds your byte code `Prog.class, Prog.pyc` to the JIT so that it can be recompiled into native code explicitly tailored to the machine you are using. Although there is an extra step involved here, the total time it takes to run your program is usually 10–30 times faster with a JIT as compared to line-by-line interpretation. Because the JIT is an integral part of the Virtual Machine on each operating system, this usually happens automatically.

In the experiments below you will investigate techniques to optimize both Fortran and Java or Python programs and to compare the speeds of both languages for the same computation. If you run your Java or Python code on a variety of machines (easy to do with themor), you should also be able to compare the speed of one computer to that of another. Note that a knowledge of Fortran is not required for these exercises; you should be able to look at the code and figure out what changes may be needed.

18.2.1 Good and Bad Virtual Memory Use (Experiment)

To see the effect of using virtual memory, convert these simple pseudocode examples (Listings 3 and 4) into actual code in your favorite language, and then run them on your computer. Use a command such as `time` to measure the time used for each example. These examples call functions `force12` and `force21`. You should write these functions and make them have significant memory requirements.

Listing 4: **BAD program, too simultaneous.**

```
for j = 1, n; {  for i = 1, n; {
    f12(i,j) = force12(pion(i), pion(j))    // Fill f12
    f21(i,j) = force21(pion(i), pion(j))    // Fill f21
    ftot = f12(i,j) + f21(i,j) }}           // Fill ftot
```

You see (Listing 3) that each iteration of the `for` loop requires the data and code for all the functions as well as access to all the elements of the matrices and arrays. The working set size of this calculation is the sum of the sizes of the arrays `f12(N,N)`, `f21(N,N)`, and `pion(N)` plus the sums of the sizes of the functions `force12` and `force21`.

A better way to perform the same calculation is to break it into separate components (Listing 4):

Listing 5: **GOOD program, separate loops.**

```
for j = 1, n;
      { for i = 1, n;  f12(i,j) = force12(pion(i), pion(j)) }
for j = 1, n;
      { for i = 1, n;  f21(i,j) = force21(pion(i), pion(j)) }
for j = 1, n;
      { for i = 1, n;  ftot = f12(i,j) + f21(i,j) }
```

Here the separate calculations are independent and the *working set size*, that is, the amount of memory used, is reduced. However, you do pay the additional overhead costs associated with creating extra `for` loops. Because the working set size of the first `for` loop is the sum of the sizes of the arrays `f12(N, N)` and `pion(N)`, and of the function `force12`, we have approximately half the previous size. The size of the last `for` loop is the sum of the sizes for the two arrays. The working set size of the entire program is the larger of the working set sizes for the different `for` loops.

As an example of the need to group data elements close together in memory or common blocks if they are going to be used together in calculations, consider the following code (Listing 5):

Listing 6: **BAD Program, discontinuous memory.**

```
Common zed, ylt(9), part(9), zpart1(50000), zpart2(50000), med2(9)
  for j = 1, n;
  ylt(j) = zed * part(j)/med2(9)          // Discontinuous variables
```

Here the variables `zed`, `ylt`, and `part` are used in the same calculations and are adjacent in memory because the programmer grouped them together in `Common` (global variables). Later, when the programmer realized that the array `med2` was needed, it was tacked onto the end of `Common`. All the data comprising the variables `zed`, `ylt`, and `part` fit onto one page, but the `med2` variable is on a different page because the large array `zpart2(50000)` separates it from the other variables. In fact, the system may be forced to make the entire 4-kB page available in order to fetch the 72 B of data in `med2`. While it is difficult for a Fortran or C programmer to ensure the placement of variables within page boundaries, you will improve your chances by grouping data elements together (Listing 6):

Listing 7: **GOOD program, continuous memory.**

```
Common zed, ylt(9), part(9), med2(9), zpart1(50000), zpart2(50000)
       for j = 1, n;
           ylt(j) = zed*part(j)/med2(J)            // Continuous
```

19 Empirical Performance of Hardware

In this section you conduct an experiment in which you run a complete program in several languages and on as many computers as are available to you. In this way you will explore how a computer's architecture and software affect a program's performance.

Nothing can surpass the optimization that a skillful and careful programmer can obtain when working on their program. Assuming that you have already done that, you might be able to improve the performance of your program further by asking the compiler to optimize your program. You control how completely the compiler tries to do this by adding *optimization options* to the compile command:

```
> f90 --O tune.f90
```

Here `--O` turns on optimization (`O` is the capital letter "oh," not zero). The actual optimization that is turned on differs from compiler to compiler. Fortran and C compilers have a bevy of such options and directives that let you truly customize the resulting compiled code. Sometimes optimization options make the code run faster, sometimes not, and sometimes the faster-running code gives the wrong answers (but does so quickly).

Because computational scientists may spend a good fraction of their time running compiled codes, the compiler options tend to become quite specialized. As a case in point, most compilers provide a number of levels of optimization for the compiler to attempt (there are no guarantees with these things). Although the speedup obtained depends upon the details of the program, higher levels may give greater speedup. However, we have had the experience of higher levels of optimization sometimes giving wrong answers (presumably this may be related to our programs not following the rules of grammar perfectly).

Some typical Fortran compiler options include

`--O`	Use the default optimization level (`--O3`)
`--O1`	Provide minimum statement-level optimizations
`--O2`	Enable basic block-level optimizations
`--O3`	Add loop unrolling and global optimizations
`--O4`	Add automatic inlining of routines from the same source file
`--O5`	Attempt aggressive optimizations (with profile feedback)

The **gnu compilers** `gcc, g77, g90` accept `--O` options as well as

`--malign--double`	Align doubles on 64-bit boundaries
`--ffloat--store`	For codes using IEEE-854 extended precision
`--fforce--mem, --fforce--addr`	Improves loop optimization
`--fno--inline`	Do not compile statement functions inline
`--ffast--math`	Try non-IEEE handling of floats
`--funsafe--math--optimizations`	Speeds up float operations; incorrect results possible
`--fno--trapping--math`	Assume no floating-point traps generated
`--fstrength--reduce`	Makes some loops faster
`--frerun--cse--after--loop`	
`--fexpensive--optimizations`	
`--fdelayed--branch`	
`--fschedule--insns`	
`--fschedule--insns2`	
`--fcaller--saves`	
`--funroll--loops`	Unrolls iterative `DO` loops
`--funroll--all--loops`	Unrolls `DO WHILE` loops

19.1 Python *versus* Fortran/C

The various versions of the program `tune` that are given in the Codes directory solve the matrix eigenvalue problem

$$\mathbf{Hc} = E\mathbf{c} \tag{19}$$

for the eigenvalues E and eigenvectors $\mathbf{c}$ of a Hamiltonian matrix $\mathbf{H}$. Here the individual Hamiltonian matrix elements are assigned the values

$$H_{i,j} = i, \quad \text{for} \quad i = j, 0.3^{|i-j|}, \quad \text{for} \quad i \neq j, = \begin{bmatrix} 1 & 0.3 & 0.14 & 0.027 & \dots \\ 0.3 & 2 & 0.3 & 0.9 & \dots \\ 0.14 & 0.3 & 3 & 0.3 & \dots \\ \ddots & & & & \end{bmatrix}. \tag{20}$$

Listing 8: **Tune.f90** is meant to be numerically intensive enough to show the results of various types of optimizations. The program solves the eigenvalue problem iteratively for a nearly diagonal Hamiltonian matrix using a variation of the power method.

```
!    tune.f90: matrix algebra program to be tuned for performance

Program  tune

  parameter (ldim = 2050)
  Implicit Double precision (a - h, o - z)
  dimension ham(ldim, ldim), coef(ldim), sigma(ldim)
                                    ! set up H and starting vector
  Do i = 1, ldim
    Do j = 1, ldim
      If ( abs(j - i)  >  10) then
        ham(j, i) = 0.
      else
        ham(j, i) = 0.3**Abs(j - i)
      EndIf
    End Do
    ham(i, i) = i
    coef(i) = 0.
  End Do
```

```
  coef(1) = 1.
                                                    ! start iterating
  err = 1.
  iter = 0
 20   If (iter< 15 .and. err >1.e-6) then
  iter = iter + 1
                                  ! compute current energy & normalize
  ener = 0.
  ovlp = 0.
  Do    i = 1, ldim
    ovlp = ovlp + coef(i)*coef(i)
    sigma(i) = 0.
    Do     j = 1, ldim
      sigma(i) = sigma(i) + coef(j)*ham(j, i)
    End Do
    ener = ener + coef(i)*sigma(i)
  End Do
  ener = ener/ovlp
  Do    I = 1, ldim
    coef(i) = coef(i)/Sqrt(ovlp)
    sigma(i) = sigma(i)/Sqrt(ovlp)
  End Do
                                       ! compute update and error norm
  err = 0.
  Do  i = 1, ldim
    If (i == 1) goto 22
    step = (sigma(i) - ener*coef(i))/(ener - ham(i, i))
    coef(i) = coef(i) + step
    err = err + step**2
 22 Continue
 23 End Do
   err = sqrt(err)
     write(*, '(1x, i2, 7f10.5)') iter, ener, err, coef(1)
     goto 20
    Endif
  Stop
End Program tune
```

Because the Hamiltonian is almost diagonal, the eigenvalues should be close to the values of the diagonal elements, and the eigenvectors should be close to the set of N-dimensional unit vectors. For example, let's say that H has dimensions of $N \times N$ with $N = 2000$ The number of elements in the matrix is then $2000 \times 2000 = 4,000,000$, and so it will take 4 million $\times$ 8 B = 32 MB to store this many double precision numbers. Consequently, if your computer's RAM is not this large, then memory access issues may slow down your program. Because modern PCs often have 256 MB of RAM, and often 2 GB or more if built for video application, this small a matrix should not have memory issues. Accordingly, *determine the size of RAM on your computer and increase the dimension of the H matrix until it surpasses that size.* (On Windows, this will be indicated as one of the "Properties" of "Computer" or in the information about "System" in the Control Panel.)

We find the solution to (19) via a variation of the *power* or *Davidson method*. We start with an arbitrary first guess for the eigenvector $\mathbf{c}$ and use it to calculate the energy corresponding to this eigenvector,[5]

$$\mathbf{c}_0 \simeq (\,1\,)\,0\dot{:}0, \qquad E \simeq \frac{\mathbf{c}_0^\dagger \mathbf{H} \mathbf{c_0}}{\mathbf{c}_0^\dagger \mathbf{c}_0}, \tag{21}$$

where $\mathbf{c}_0^\dagger$ is the row vector adjoint of $\mathbf{c}_0$. Because $\mathbf{H}$ is nearly diagonal with diagonal elements that increase as we move along the diagonal, this guess should be close to the eigenvector with the smallest eigenvalue. The heart of the algorithm is the guess that an improved eigenvector has the

[5]Note that the codes refer to the eigenvector c_0 as `coef`.

kth component

$$\mathbf{c}_1|_k \;\simeq\; \mathbf{c}_0|_k + \frac{[\mathbf{H} - E\mathbf{I}]\mathbf{c}_0|_k}{E - H_{k,k}}, \tag{22}$$

where k ranges over the length of the eigenvector. If repeated, this method converges to the eigenvector with the smallest eigenvalue. It will be the smallest eigenvalue since it gets the largest weight (smallest denominator) in (22) each time. For the present case, six places of precision in the eigenvalue are usually obtained after 11 iterations. Here are the steps to follow:

1. Vary the value of `err` in `tune` that controls precision and note how it affects the number of iterations required.

2. Try some variations on the initial guess for the eigenvector (22) and see if you can get the algorithm to converge to some of the other eigenvalues.

3. Keep a table of your execution times *versus* technique.

4. Compile and execute `tune.f90` and record the run time. On Unix systems, the compiled program will be placed in the file `a.out`. From a Unix shell, the compilation, timing, and execution can all be done with the commands

   ```
   > f90 tune.f90                    Fortran compilation
   > cc --lm tune.c                  C compilation, gcc also likely
   > time a.out                      Execution
   ```

   Here the compiled Fortran program is given the (default) name `a.out`, and the `time` command gives you the execution (`user`) time and `system` time in seconds to execute `a.out`.

5. As indicated in §19, you can ask the compiler to produce a version of your program optimized for speed by including the appropriate compiler option:

   ```
   > f90 --O tune.f90
   ```

   Execute and time the optimized code, checking that it still gives the same answer, and note any speedup in your journal.

6. Try out optimization options up to the highest levels and note the run time and accuracy obtained. Usually `--O3` is pretty good, especially for as simple a program as `tune` with only a main method. With only one program unit we would not expect `--O4` or `--O5` to be an improvement over `--O3`. However, we do expect `--O3`, with its loop unrolling, to be an improvement over `--O2`.

7. The program `tune4` does some *loop unrolling* (we will explore that soon). To see the best we can do with Fortran, record the time for the most optimized version of `tune4.f95`.

8. The program `Tune.py` in Listing 9 is the Python equivalent of the Fortran program `tune.f90`.

9. To get an idea of what `Tune.py` does (and give you a feel for how hard life is for the poor computer), assume `ldim =2` and work through one iteration of `Tune` *by hand*. Assume that the iteration loop has converged, follow the code to completion, and write down the values assigned to the variables.

10. Compile and execute `Tune.py`. You do not have to issue the `time` command since we built a timer into the Python program. Check that you still get the same answer as you did with Fortran and note how much longer it takes with Python. You might be surprised how much slower Python is than Fortran.

11. We now want to perform a little experiment in which we see what happens to performance as we fill up the computer's memory. In order for this experiment to be reliable, it is best for you *not* to be sharing the computer with any other users. On Unix systems, the `who --a` command shows you the other users (we leave it up to you to figure out how to negotiate with them).

12. To get some idea of what aspect of our little program is making it so slow, compile and run `Tune.py` for the series of matrix sizes `ldim` = 10, 100, 250, 500, 750, 1025, 2500, and 3000. You may get an error message that Python is out of memory at 3000. This is because you have not turned on the use of virtual memory.

    Listing 9: **Tune.py** is meant to be numerically intensive enough to show the results of various types of optimizations. The program solves the eigenvalue problem iteratively for a nearly diagonal Hamiltonian matrix using a variation of the power method.

```
# Tune.py Basic tuning program showing memory allocation

import datetime; from numpy import zeros; from math import (sqrt, pow)

Ldim = 251;          iter = 0;           step = 0.
diag  = zeros((Ldim, Ldim), float);     coef = zeros( (Ldim), float)
sigma = zeros((Ldim), float);       ham = zeros( (Ldim, Ldim), float)
t0 = datetime.datetime.now()             # Initialize time
for i in range(1, Ldim):                 # Set up Hamiltonian
    for j in range(1, Ldim):
        if (abs(j - i) >10): ham[j, i] = 0.
        else : ham[j, i] = pow(0.3, abs(j - i) )
    ham[i,i] = i ;              coef[i] = 0.;
coef[1] = 1.;                   err = 1.;          iter = 0 ;
print("iter  ener      err ")
while (iter  < 15 and err > 1.e-6):  # Compute current E,   normalize
    iter = iter + 1; ener = 0. ; ovlp = 0.;
    for i in range(1, Ldim):
        ovlp = ovlp + coef[i]*coef[i]
        sigma[i] = 0.
        for j in range(1, Ldim): sigma[i] = sigma[i] + coef[j]*ham[j][i]
        ener = ener + coef[i]*sigma[i]
    ener = ener/ovlp
    for i in range(1, Ldim):
        coef[i] = coef[i]/sqrt(ovlp)
        sigma[i] = sigma[i]/sqrt(ovlp)
    err = 0.;
    for i in range(2, Ldim):                              # Update
        step = (sigma[i]  -  ener*coef[i])/(ener - ham[i, i])
        coef[i] = coef[i] + step
        err = err +  step*step
    err = sqrt(err)
    print(" %2d  %9.7f  %9.7f "%(iter, ener, err))
delta_t = datetime.datetime.now()  -  t0             # Elapsed time
print(" time = ", delta_t)
```

13. Make a graph of run time *versus* matrix size. It should be similar to Figure 16, although if there is more than one user on your computer while you run, you may get erratic results. Note that as our matrix becomes larger than $\sim 1000 \times 1000$ in size, the curve sharply increases in slope with execution time, in our case increasing like the *third* power of the dimension. Since

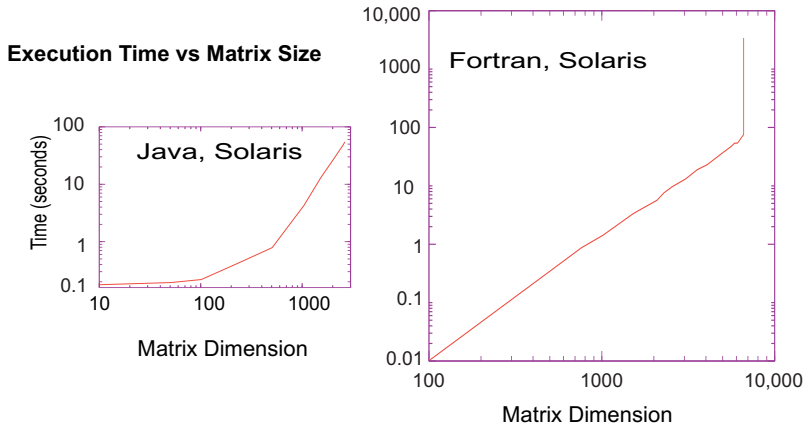


Figure 16: Running time *versus* dimension for an eigenvalue search using `Tune.py` and `tune.f90`.

the number of elements to compute increases as the *second* power of the dimension, something else is happening here. It is a good guess that the additional slowdown is due to page faults in accessing memory. In particular, accessing 2-D arrays, with their elements scattered all through memory, can be very slow.

Listing 10: **Tune4.py** does some loop unrolling by explicitly writing out two steps of a `for` loop (steps of 2). This results in better memory access and faster execution.

```
# Tune4.py  Model tuning program

import datetime
from numpy import zeros
from math import (sqrt,pow)
from math import pi
from sys import version

if int(version[0])>2: raw_input=input   # raw_input deprecated Python3

Ldim = 200;          iter1 = 0;          step = 0.
ham = zeros( (Ldim, Ldim), float);       diag = zeros( (Ldim), float)
coef = zeros( (Ldim), float);            sigma = zeros( (Ldim), float)
t0 = datetime.datetime.now()             # Initialize time

for i in range(1, Ldim):                # Set up Hamiltonian
        for j in range(1, Ldim):
           if abs(j - i) >10: ham[j, i] = 0.
           else : ham[j, i] = pow(0.3, abs(j - i) )
for i in range(1, Ldim):
    ham[i, i] = i
    coef[i]   = 0.
    diag[i]   = ham[i, i]
coef[1] = 1.;        err = 1.;           iter = 0 ;
print("iter      ener           err ")

while (iter1  < 15 and err > 1.e-6): # Compute current E, normalize
    iter1 = iter1 + 1
    ener  = 0.
    ovlp1 = 0.
    ovlp2 = 0.
    for i in range(1, Ldim - 1, 2):
        ovlp1 = ovlp1 + coef[i]   * coef[i]
        ovlp2 = ovlp2 + coef[i+1] * coef[i+1]
        t1 = 0.
        t2 = 0.
        for j in range(1, Ldim):
```

```
            t1 = t1 + coef[j] * ham[j, i]
            t2 = t2 + coef[j] * ham[j, i+1]
        sigma[i]   = t1
        sigma[i+1] = t2
        ener       = ener + coef[i] * t1 + coef[i+1] * t2
    ovlp = ovlp1 + ovlp2
    ener = ener/ovlp
    fact = 1./sqrt(ovlp)
    coef[1] = fact*coef[1]
    err = 0.                              # Update & error norm
    for i in range(2, Ldim):
        t       = fact*coef[i]
        u       = fact*sigma[i] - ener*t
        step    = u/(ener - diag[i])
        coef[i] = t + step
        err     = err +  step*step
    err = sqrt(err)
    print (" %2d  %15.13f  %15.13f "%(iter1, ener, err))
delta_t = datetime.datetime.now()  -  t0  # Elapsed time
print (" time = ",  delta_t)
print("press a key to finish")
s = raw_input()
```

Listing 11: The program **tune4.f95** does some loop unrolling by explicitly writing out two steps of a `Do` loop (steps of 2). This results in better memory access and faster execution.

```
!  tune4.f95: matrix algebra with RISC tuning                    !

      Program  tune4
      PARAMETER (ldim = 2050)
      Implicit Double Precision (a-h,o-z)
      Dimension ham(ldim,ldim),coef(ldim),sigma(ldim),diag(ldim)
!       set up Hamiltonian and starting vector
      Do  i = 1,ldim
         Do  j = 1,ldim
             If( Abs(j-i) > 10) Then
                ham(j,i) = 0.0
             Else
                ham(j,i) = 0.3**Abs(j-i)
             EndIf
         End Do
      End Do

!      start iterating towards the solution
      Do  i = 1,ldim
         ham(i,i) = i
         coef(i)  = 0.0
         diag(i)  = ham(i,i)
      End Do
      coef(1) = 1.0
      err = 1.0
      iter = 0
 20   If(iter<15 .and. err>1.0e-6) Then
        iter = iter+1
      ener = 0.0
      ovlp1 = 0.0
      ovlp2 = 0.0
      Do    i = 1,ldim-1,2
         ovlp1 = ovlp1+coef(i)*coef(i)
         ovlp2 = ovlp2+coef(i+1)*coef(i+1)
         t1    = 0.0
         t2    = 0.0
         Do    j = 1,ldim
            t1 = t1 + coef(j)*ham(j,i)
            t2 = t2 + coef(j)*ham(j,i+1)
         End Do
      sigma(i)   = t1
```

```
      sigma(i+1) = t2
      ener       = ener + coef(i)*t1 + coef(i)*t2
      End Do
      ovlp = ovlp1 + ovlp2
      ener = ener/ovlp
      fact  = 1.0/Sqrt(ovlp)
      coef(1)  =  fact*coef(1)
      err  =  0.0
      Do    i = 2,ldim
         t         =   fact*coef(i)
         u         =   fact*sigma(i) - ener*t
         step      =   u/(ener - diag(i))
         coef(i)   =   t + step
         err       =   err + step*step
      End Do
      err = Sqrt(err)
      Write(*,'(1x,i2,7f10.5)') iter,ener,err,coef(1)
      GoTo 20
      EndIf
      Stop
      End  Program  tune4
```

14. Repeat the previous experiment with `tune.f90` that gauges the effect of increasing the `ham` matrix size, only now do it for `ldim` = 10, 100, 250, 500, 1025, 3000, 4000, 6000,.... You should get a graph like ours. Although our implementation of Fortran has automatic virtual memory, its use will be exceedingly slow, especially for this problem (possibly a 50-fold increase in time!). So if you submit your program and you get nothing on the screen (though you can hear the disk spin or see it flash busy), then you are probably in the virtual memory regime. If you can, let the program run for one or two iterations, kill it, and then scale your run time to the time it would have taken for a full computation.

15. To test our hypothesis that the access of the elements in our 2-D array `ham [i][j]` is slowing down the program, we have modified `Tune.py` into `Tune4.py` in Listing 10 (and similar modification with the Fortran versions).

16. Look at `Tune4` and note where the nested `for` loop over `i` and `j` now takes step of $\Delta i = 2$ rather the unit steps in `Tune.py`. If things work as expected, the better memory access of `Tune4.py` should cut the run time nearly in half. Compile and execute `Tune4.py`. Record the answer in your table.

17. In order to cut the number of calls to the 2-D array in half, we employed a technique know as *loop unrolling* in which we explicitly wrote out some of the lines of code that otherwise would be executed implicitly as the `for` loop went through all the values for its counters. This is not as clear a piece of code as before, but it evidently, permits the compiler to produce a faster executable. To check that `Tune` and `Tune4` actually do the same thing, assume `ldim =4` and run through one iteration of `Tune4` *by hand*. Hand in your manual trial.

20 Programming for the Data Cache (Method)

Data caches are small, very fast memory used as temporary storage between the ultrafast CPU registers and the fast main memory. They have grown in importance as high-performance computers have become more prevalent. For systems that use a data cache, this may well be the single most important programming consideration; continually referencing data that are not in the cache (*cache misses*) may lead to an order-of-magnitude increase in CPU time.

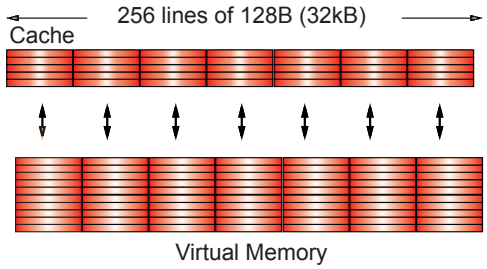


Figure 17: The cache manager's view of RAM. Each 128-B cache line is read into one of four lines in cache.

As indicated in Figures 2 and 17, the data cache holds a copy of some of the data in memory. The basics are the same for all caches, but the sizes are manufacturer-dependent. When the CPU tries to address a memory location, the *cache manager* checks to see if the data are in the cache. If they are not, the manager reads the data from memory into the cache, and then the CPU deals with the data directly in the cache. The cache manager's view of RAM is shown in Figure 17.

When considering how a matrix operation uses memory, it is important to consider the *stride* of that operation, that is, the number of array elements that are stepped through as the operation repeats. For instance, summing the diagonal elements of a matrix to form the trace

$$\mathrm{Tr}\, A = \sum_{i=1}^{N} a(i,i) \tag{23}$$

involves a large stride because the diagonal elements are stored far apart for large N. However, the sum

$$c(i) = x(i) + x(i+1) \tag{24}$$

has stride 1 because adjacent elements of x are involved. The basic rule in programming for a cache is

- Keep the stride low, preferably at 1, which in practice means:
- Vary the leftmost index first on Fortran arrays.
- Vary the rightmost index first on Python and C arrays.

20.1 Exercise 1: Cache Misses

We have said a number of times that your program will be slowed down if the data it needs are in virtual memory and not in RAM. Likewise, your program will also be slowed down if the data required by the CPU are not in the cache. For high-performance computing, you should write programs that keep as much of the data being processed as possible in the cache. To do this you should recall that Fortran matrices are stored in successive memory locations with the row index varying most rapidly (column-major order), while Python and C matrices are stored in successive memory locations with the column index varying most rapidly (row-major order). While it is difficult to isolate the effects of the cache from other elements of the computer's architecture, you should now estimate its importance by comparing the time it takes to step through the matrix elements row by row to the time it takes to step through the matrix elements column by column.

Run on machines available to you a version of each of the two simple codes listed below. Check that even though each has the same number of arithmetic operations, one takes significantly more

time to execute because it makes large jumps through memory, with some of the memory locations addressed not yet read into the cache:

Listing 12: **Sequential column and row references.**

```
for j = 1, 999999;
    x(j) = m(1,j)                    // Sequential column reference
```

Listing 13: **Sequential column and row references.**

```
for j = 1, 999999;
    x(j) = m(j,1)                    // Sequential row reference
```

20.2 Exercise 2: Cache Flow

Below in Listings 14 and 15 we give two simple code fragments that you should place into full programs in whatever computer language you are using. Test the importance of cache flow on your machine by comparing the time it takes to run these two programs. Run for increasing column size `idim` and compare the times for loop A *versus* those for loop B. A computer with very small caches may be most sensitive to stride.

Listing 14: **GOOD f90, BAD Python/C Program; minimum, maximum stride.**

```
Dimension Vec(idim,jdim)                    // Stride 1 fetch (f90)
        for j = 1, jdim; { for i=1, idim;     Ans = Ans + Vec(i,j)*Vec(i,j)}
```

Listing 15: **BAD f90, GOOD Python/C Program; maximum, minimum stride.**

```
Dimension Vec(idim, jdim)                   // Stride jdim fetch (f90)
        for i = 1, idim; {for j=1, jdim;   Ans = Ans + Vec(i,j)*Vec(i,j)}
```

Loop A steps through the matrix `Vec` in column order. Loop B steps through in row order. By changing the size of the columns (the rightmost Fortran index), we change the step size (*stride*) taken through memory. Both loops take us through all the elements of the matrix, but the stride is different. By increasing the stride in any language, we use fewer elements already present in the cache, require additional swapping and loading of the cache, and thereby slow down the whole process.

20.3 Exercise 3: Large-Matrix Multiplication

As you increase the dimensions of the arrays in your program, memory use increases geometrically, and at some point you should be concerned about efficient memory use. The penultimate example of memory usage is large-matrix multiplication:

$$[C] = [A] \times [B], \tag{25}$$

$$c_{ij} = \sum_{k=1}^{N} a_{ik} \times b_{kj}. \tag{26}$$

Listing 16: **BAD f90, GOOD Python/C Program; maximum, minimum stride.**

```
for i = 1, N; {                                          // Row
    for j = 1, N; {                                    // Column
        c(i,j) = 0.0                                // Initialize
    for k = 1, N; {
        c(i,j) = c(i,j) + a(i,k)*b(k,j) }}}          // Accumulate
```

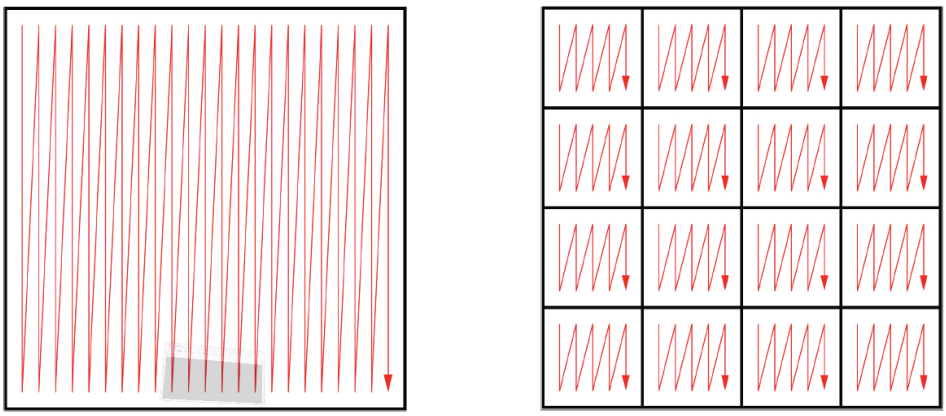


Figure 18: A schematic of how the contiguous elements of a matrix must be tiled for parallel processing (from [Dong 11]).

This involves all the concerns with different kinds of memory. The natural way to code (25) follows from the definition of matrix multiplication (26), that is, as a sum over a row of A times a column of B. Try out the two codes in Listings 16 and 17 on your computer. In Fortran, the first code has B with stride 1, but C with stride N. This is corrected in the second code by performing the initialization in another loop. In Python and C, the problems are reversed. On one of our machines, we found a factor of 100 difference in CPU times even though the number of operations is the same!

Listing 17: **GOOD f90, BAD Python/C Program; minimum, maximum stride.**

```
for j = 1, N; {                          // Initialization
    for  i = 1, N; {
        c(i,j) = 0.0 }
    for k = 1, N; {
        for  i = 1, N; {c(i,j) = c(i,j) + a(i,k)*b(k,j) }}}
```

21 Practical Tips for Multicore, GPU Programming

We have already described some basic elements of exascale computing in §17. Some practical tips for programming multinode-multicore-GPU computers follow along the same lines as we have been discussing, but with an even greater emphasis on minimizing communication costs[6]. This means that the "faster" of two algorithms may be the one that takes more steps, but requires less communications. Because the effort in programming the GPU directly can be quite high, many application programmers prefer to let compiler extensions and wrappers deal with the GPU.

Exascale computers and computing are expected to be *"distruptive technologies"* in that they lead to drastic changes from previous models for computers and computing. This is in contrast to the more evolving technology of continually increasing the clock speed of the CPU, which was the practice until the power consumption and associated heat production imposed a roadblock. Accordingly we should expect that software and algorithms will be changing (and we will have to rewrite our applications), much as it did when supercomputers changed from large vector machines with proprietary CPUs to cluster computing using commodity CPUs and message passing. Here are some of the major points to consider:

[6]Much of the material in this section comes from talks we have heard by John Dongarra [Dong 11].

Exacascale data movement is expensive The time for a floating point operation and for a data transfer can be similar, although if the transfer is not local, as Figures 13 and 15 show happens quite often, then communication will be the rate-limiting step.

Exacascale flop/s are cheap and abundant GPUs and local multicore chips provide many very fast flops for us to use, and we can expect even more of these elements to be included in future computers. So do not worry about flops as much as communication.

Synchronization-reducing algorithms are essential Having many processors stop in order to synchronize with each other, while essential in ensuring that the proper calculation is being performed, can slow down processing to a halt (literally). It is better to find or derive an algorithm that reduces the need for synchronization.

Break the fork-join model This model for parallel computing takes a queue of incoming jobs, divides them into subjobs for service on a variety of servers, and then combines them to produce the final result. Clearly, this type of model can lead to completed subjobs on various parts of the computer that must wait for other subjobs to complete before recombination. A big waste of resources.

Communication-reducing algorithms As already discussed, it is best to use methods that lessen the need for communication among processors, even if more flops are needed.

Use mixed precision methods Most GPUs do not have native double precision (or even full single precision) and correspondingly slow down by a factor-of-two or more when made to perform double precision calculation, or when made to move double-precision data. The preferred approach then is to use single precision. One way to do this is to employ perturbation theory in which the calculation focuses on the small (single precision) correction to a known, or separately calculated, large (double precision) basic solution. The rule-of-thumb then is to use the lowest precision required to achieve the required accuracy.

Push for and use autotuning The computer manufactures have advanced the hardware to incredible speeds, but have not produced concordant advances in the software that scientists need to employ in order to make good use of these machines. It often takes people-years to rewrite a program for these new machines, and that is a tremendous investment that not many scientists can afford. We need smarter software to deal with such complicated machines, and tools that permit us to optimize experimentally our programs for these machines.

Fault resilient algorithms Computers containing millions or billions of components do make mistakes at times. It does not make sense to have to start a calculation over, or hold a calculation up, when some minor failure such as a bit flip occurs. The algorithms should be able to recover from these types of failures.

Reproducibility of results Science is at its heart the search for scientific truth, and there should be only one answer when solving a problem whose solution is mathematically unique. However, approximations in the name of speed are sometimes made and this can lead to results whose exact reproducibility cannot be guaranteed. (Of course exact reproducibility is not to be expected for Monte Carlo calculations involving chance.)

Data layout is critical As we discussed with Figures 17, 2 and 18, much of HPC deals with matrices and their arrangements in memory. With parallel computing we must arrange the data into tiles such that each data tile is contiguous in memory. Then the tiles can be processed

in a fine-grained computation. As we have seen in the exercises, the best size for the tiles depends upon the size of the caches being used, and these are generally small.

References

[Amd 67] Amdahl, G., *Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities*, Proc. AFIPS., 483 (1967).

[CiSE] *Computing in Science & Engineering*, `http://www.computer.org/portal/web/cise/home`.

[Dong 05] Dongarra, J., T. Sterling, H. Simon, and E. Strohmaier (2005), *High-Performance Computing,* Comp. in Science & Engr. **7**, 51.

[Dong 11] Dongarra, J., *On the Future of High Performance Coputing: How to Think for Peta and Exascale Computing*, Conference on Computational Physics 2011, Gatlinburg, 2011; *Emerging Technologies for High Performance Computing*, GPU Club presentation, 2011, University of Manchester, `http://www.netlib.org/utk/people/JackDongarra/SLIDES/gpu-0711.pdf`.

[FMPF] *The Fast MPJ Project (associated with the Java Fast Sockets Project)*, `http://jfs.des.udc.es/`.

[Fox 94] Fox, G., *Parallel Computing Works!*, (1994) Morgan Kaufmann, San Diego.

[Gara 05] Gara, A., M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas, *Overview of the Blue Gene/L system architecure,* (2005) IBM J. Res & Dev **49**, 195.

[Har 96] Hardwich, J., *Rules for Optimization,* `www.cs.cmu.edu/~jch/java`.

[LPB 08] Landau, R. H, M. J. Paez, and C. Bordeianu, *A Survey of Computational Physics, introductory computational science,* (2008) Princeton University Press, Princeton.

[MPJ] *MPJ Express*, an open source Java message passing library that allows application developers to write and execute parallel applications for multicore processors and compute clusters/-clouds. `http://mpj-express.org/`.

[Pan 96] Pancake, C. M., (1996), *Is Parallelism for You?*, IEEE Computational Sci. & Engr, **3**, 18.

[Quinn 04] Quinn, M. J. (2004), *Parallel Programming in C with MPI and OpenMP*, McGraw Hill Higher Education, New York, NY.

[Sterl 11] Sterling, T. (2011), *CSC 7600 Lecture Notes*, Spring 2011, Louisiana State University.

[Top 500] *Lawrence Livermores Sequoia Supercomputer Towers above the Rest in Latest TOP 500 List*, http://www.top500.org/.

[VdeV 94] van de Velde, E. F. (1994), *Concurrent Scientific Computing*, Springer-Verlag, New York.

Index