Introduction to
Parallel Programming Concepts

Alan Scheinine, IT Consultant

High Performance Computing
Center for Computational Technology and
Information Technology Services
Louisiana State University

E-mail: scheinin@cct.lsu.edu
Web: http://www.cct.lsu.edu/~scheinin/Parallel/
## Contents

1 Introduction 1

2 Terminology 2

2.1 Hardware Architecture Terminology 2

2.2 HPC Terminology 3

2.3 Bandwidth and Latency Examples 4

2.4 Amdahl’s Law 4

3 Pipelining: From Cray to PC 5

3.1 Hardware Considerations 5

3.2 An Entire Computer Pipelined 6

3.3 The ALU Pipelined 9

3.4 Vectorization Programming Considerations 13

3.5 NEC, a Modern Vector Computer 13

4 Instruction Execution Model 14

4.1 SIMD 14

4.2 SMT 14

4.3 MIMD and SPMD 15

5 Hardware Complexity of Conventional CPUs 16

6 The Cost of a Context Switch 17

7 Tera Computer and Multithreading 18

8 Shared Memory and Distributed Memory 18

9 Beowulf Clusters 19

10 GP-GPU and Multithreading 20

11 IBM-Toshiba-Sony Cell Broadband Engine 23

12 ClearSpeed Coprocessor, Massively SIMD 25

13 Large Embarrassingly Parallel Tasks 26

14 OpenMP 27

14.1 Overview of OpenMP 27

14.2 OpenMP Code Examples 30

15 Message Passing 32
List of Figures

1 Block Diagram of Cray-1 .................................................. 7
2 More Detailed Block Diagram of Cray-1 .................................. 8
3 Manufacturing Pipeline ..................................................... 9
4 Conventional Multiplication .............................................. 10
5 Multiplication Steps ........................................................ 11
6 Pipelined Multiplication ................................................... 12
7 GPU devotes more transistors to data processing ...................... 21
8 CBE Processor ............................................................... 24
9 A Team of Threads ......................................................... 29
10 Domain Decomposition ................................................... 39
11 Ghost Vertices ............................................................. 40
1 Introduction

The speed of CPUs has increased very gradually in the last several years, yet advances in science and engineering require large increases in computational power. As an example, suppose that for a particular simulation based on fluid dynamics an improvement by a factor of ten in spatial resolution would be valuable. A factor of ten in linear resolution requires a factor of 1,000 increase in the number of vertices in a three-dimensional mesh, and in addition, the time step needs to be reduced in order to follow the evolution of smaller structures; hence an overall increase of computational power of 10,000 might be needed in order to finish the calculation without increasing the wallclock time. With regard to fluid dynamics, the wide range of applications is demonstrated by the examples found at the Fluent web site http://www.fluent.com/solutions/index.htm

The required increase in computational power may be much less than the factor of 10,000 given in the above example, nonetheless significant gains clearly cannot be realized by the clock speed increases we’ve seen in the last few years: from roughly 2 GHz to 3 GHz. So we see that for program developers in many fields it is necessary to know how to write efficient parallel programs.

A list of applications areas from the Cray web site http://www.cray.com/industrysolutions.aspx

- Digital Media
- Earth Sciences
- Energy
- Financial Services
- Government and Defense
- Higher Education
- Life Sciences
- Manufacturing

Another view of the range of applications can be found at http://www.top500.org/drilldown by choosing the ‘Statistics Type’ of ‘Application Area’.
This presentation will begin with a description of the hardware of parallel computers. The focus will be on the hardware attributes that have an impact on the style and structure of programs. Then parallelization of serial programs will be described, with emphasis on MPI and OpenMP parallel language extensions. There will be other HPC training sessions discussing MPI and OpenMP in more detail.

2 Terminology

2.1 Hardware Architecture Terminology

Various concepts of computer architecture are defined in the following list. These concepts will be used to describe several parallel computers. Actual computer architectures do not fall into discrete classes because each realization has a mix of attributes. The definitions here are very brief and more detailed descriptions will be given as various subjects are discussed. The list is far from a complete list of architectural attributes, but rather, these are attributes important for understanding parallel computers with regard to efficient programming.

**SIMD** A Single Instruction Multiple Data computer executes the same instruction in parallel on subsets of a collection of data.

**MIMD** A Multiple Instruction Multiple Data computer can execute a different instruction contemporaneously on subsets of a collection of data.

**SPMD** The concept of Single Program Multiple Data is that a MIMD machine runs the same program on each computational element.

**SMT** The same hardware elements support more than one thread simultaneously.

**Cluster** A cluster of PCs uses low-cost PCs or moderate-cost “server” computers available for the mass market that are connected with conventional Ethernet, and in many cases, also connected with a faster, more specialized communication network. One of the original cluster projects was called ‘Beowulf’, a term that continues to be used.

**Shared Memory** If more than one program can change the same memory location directly, typically memory on the same motherboard, then the architecture is called SMP. Initially the acronym meant Symmetric Multi-Processor though now it implies Shared Memory Processor.

**Distributed Memory** Memory is distributed within many nodes (computer cases or blades).
Vector Processor A single instruction is applied to many elements of a dataset in very rapid succession.

Heterogeneous Computer More than one type of computer architecture within the same machine.

Multicore More than one computer within the same chip of silicon.

Cache A copy inside a chip of data in main memory outside the chip. Smaller but faster than main memory.

Cache Coherence Various copies in cache (on different chips or associated with different cores) that correspond to the same main memory address have the same value. Writing to one cache location or main memory results in other corresponding cache locations being marked as out-of-date, done blockwise.

Interleaved Memory Banks In the time interval of one memory cycle, N memory banks provide N times more words than a single bank.

2.2 HPC Terminology

In addition to computer architecture concepts listed above, the following list describes concepts important for describing computational performance.

Bandwidth The amount of data transferred per second. Usually expressed in bytes per second for hard disk and memory, but as bits per second for internode connections.

Latency The time delay beyond what is intrinsic to the bandwidth limit. For example, the time delay from when a CPU instruction requests a word from main memory and when the datum returns from a memory module. Also, the time delay from when a message passing subroutine is called on a node to the time that another node receives the data after having transited the network interface cards on both nodes and the interconnection switches.

FLOPS Floating point operations per second. Care must be taken to specify whether this is single precision (32-bit numbers) or double precision (64-bit numbers).

MIPS Millions of instructions per second.
2.3 Bandwidth and Latency Examples

Since bandwidth and latency apply to all computer architectures, they will be described first. The following table shows why there is an advantage to splitting-up the data among many nodes so that for each node all of the data can reside in main memory without using a local hard disk. It also shows why nodes that communicate with Ethernet often have a specialized interconnection network, such as Infiniband, for MPI communication.

- **Main memory DDR2-800**
  - bandwidth: 6400 MB/s, two interleaved banks give 25-32 GB/sec
  - latency: 15-20 ns DRAM but 200-600 CPU cycles

- **Gigabit Ethernet Interconnection**
  - bandwidth: 1 Gigabits/sec
  - latency: 170 us (1/1000000 sec)

- **Infiniband Interconnection**
  - bandwidth: 10-20 Gigabits/sec
  - latency: 5-10 us

- **Hard Disk Device**
  - bandwidth: 50-125 MB/s sustained data rate
  - latency: 3.5-5.0 ms (1/1000 sec) seek time

2.4 Amdahl’s Law

Another very general concept that applies to all parallel programs is Amdahl’s law, which states: given a speed-up $\sigma$ that affects a portion $P$ of a program, then the overall speedup $S$ will be

$$ S = \frac{1}{(1 - P) + P/\sigma} $$  \hspace{1cm} (1)

To understand the formula we note that when the speedup affects the entire code, $P \equiv 1$ and then $S = \sigma$. Most importantly, if the parallel portion of a program executes on a nearly infinite number of processors so that the speedup of the parallel part is nearly infinite, then

$$ S = \frac{1}{(1 - P)} $$  \hspace{1cm} (2)
The conclusion is that the maximum speedup is limited by the non-parallel portion of the code. For example, if 90% of the program can be efficiently parallelized, the best speedup possible is \( \frac{1}{1 - 0.9} = 10 \). In this example, the ten percent (in terms of execution time) of the program that cannot be parallelized will limit the maximum possible speedup to a factor of ten, even when a large number of concurrent processes are used for the parallel part.

In order to take advantage of parallel computers, the choice of algorithm and method of implementation must strictly minimize the time spent in serial execution. For similar reasons, the communication overhead such as latency will also limit the maximum speedup.

3 Pipelining: From Cray to PC

3.1 Hardware Considerations

The first supercomputers were built before very large-scale integration permitted putting an entire CPU onto a single square of silicon. As an example, we consider the Cray-1, first installed in 1976. These first supercomputers each had a price tag of several million dollars. As an example, Wikipedia says of the first model from Cray

The company expected to sell perhaps a dozen of the machines, but over eighty Cray-1s of all types were sold, priced from $5M to $8M.

In contrast to the Cray-1 supercomputer, at the present time, because of the huge market for personal computers, companies such as Intel and AMD are able to invest in state-of-the-art fabrication equipment, hence the speed of transistors in the central processor unit (CPU) of a PC equals that of a supercomputer. Consequently, the low-cost approach to obtaining massive computational power is to connect a group of PCs, that is, a cluster of PCs. Clusters with the highest performance usually use high-end PCs that are called servers, put into thin cases (pizza boxes) that are densely packed in racks. (Or more densely packed as blades.) Though a low-cost cluster can use Ethernet as an interconnect, much faster interconnects (e.g. Myrinet or Infiniband) can be purchased for approximately one-third increase in total cost.

Let us now consider the memory bandwidth. On a PC, the cost of connecting chips on a motherboard is high and the space available on a single motherboard is limited, and as a consequence the bandwidth between CPU and memory is much less than the computational bandwidth within a processor chip. To explain further, because the cost of chip connections are high, the number of wires going between processor and memory is limited to one to two hundred;
moreover, since space is limited on the motherboard, the number of interleaved memory banks is limited to two or four. Modern processors have two levels of onboard cache and will soon have three levels. The level of cache closest to the computational units, L1, is smallest and the fastest, whereas, L2 is larger but has less bandwidth.

Often, present-day supercomputers have a memory bandwidth that is superior to the bandwidth of a PC. Since the processor speed of PCs is similar to supercomputers, what distinguishes the high-cost supercomputers is the higher memory bandwidth.

Returning to the Cray-1, it had no cache. The bandwidth of main memory was sufficient to fully feed the ALU. The Cray-1 was a vector processor. An arithmetic operation was done in many pipelined stages, with data flowing to/from memory, to/from registers then to/from the arithmetic units. No cache was used so in order to obtain full bandwidth 16 banks of memory were used, each bank four words wide. The vector registers were 64-words deep. For best efficiency, the data being used would need to be contiguous in memory. A culture of vector programming developed that favored deep loops with simple operations that had no tests or dependencies between iterations.

3.2 An Entire Computer Pipelined

Fig. 1 shows a block diagram of the Cray-1. Dataflow was between main memory — composed of many banks in order to provide a dense stream of data — and registers. Simultaneously, there was a dataflow between registers and arithmetic logic units (ALU).

Fig. 2 emphasizes that a vector was composed of 64 double precision words. The ALUs were highly pipelined, so the source and destinations (the registers and main memory) needed to be highly pipelined. In a usual block diagram of a processor each register holds a word and the movement between the functional units would be consist of the movement of single words. In contrast, for the Cray-1 the connections between the functional units should be draw as train tracks with trains of 64 carriages. For \( C_i = a * A_i + B_i \), before the last car enters the multiplier, the first car has exited and is entering the adder.
Figure 1: Dataflow is between main memory and registers and between registers and ALUs. No cache was used. The numbers in parenthesis are the pipeline depths.
Figure 2: Various functional units could be chained.

Figure 3-1. Computation section
3.3 The ALU Pipelined

Though vector computers are less common these days, modern computers still have deep pipelines for instruction decoding and in the ALUs. So what is a pipeline? Fig. 3 shows a pipeline as an assembly line. Figs. 4, 5 and 6 show a pipeline for an ALU. Fig. 4 shows the conventional technique used by a human for multiplication. Labels are used to indicate the sequence of steps. Fig. 5 shows the same conventional multiplication as a sequence of stages. Each stage should be considered to be an area of transistors on the processor chip. Fig. 6 shows how seven different calculations can be done contemporaneously using a pipeline.

Pipelining and Systolic

Figure 3: Illustration of a manufacturing pipeline, from [1].
Figure 4: Multiplication as a human would do it.
Figure 5: Multiplication as a series of steps.
Figure 6: Different products being calculated simultaneously in a pipeline.
3.4 Vectorization Programming Considerations

Vectorization considerations also apply to conventional CPUs such as Xeon or Opteron, that are used in PCs. Unlike a vector machine, for a PC the instruction can change each cycle. On the other hand, for a PC the arithmetic units have deep pipelines. Searching on the web I found that the Intel Xeon used around 12 stages for integer arithmetic and 17 stages for floating-point arithmetic. If the logic of a program is to do one floating-point computation and then test the result, at least 17 cycles of waiting would be needed to have a result to test. The deep pipelining of all ALUs motivates the same consideration as that of programming the Cray-1, in particular, avoid data dependencies between loop iterations. I am not certain of the typical pipeline depth, an optimization guide for AMD says that floating-point addition uses a four-stage pipeline for a recent processor. The Intel compiler will show which loops have been vectorized so the programmer can see which loops should be studied to see what is impeding vectorization. This cycle of improvement is fast because it is not necessary to run to code, just look at the compiler remarks.

Moreover, while L1 cache in a modern CPU is random access, access to the main memory (DRAM) is fastest when sequential. Morever, on a PC access to cache is much, much faster than access to main memory, even when main memory is on the same circuit board. Consequently, a program developer should attempt to partition calculations into blocks of data that will fit into cache. This is done automatically for some optimized libraries such as ATLAS, used for linear algebra.

3.5 NEC, a Modern Vector Computer

Vector computers are still being sold, for example, the NEC SX-9 [2]. Each node can have as many of 16 CPUs with as much as a TeraByte of shared memory. The memory bandwidth for one node has a peak value of 4 TeraBytes/sec and the peak performance of a node is 1.6 TeraFLOPS. A cluster can have as many as 512 nodes with a distributed memory architecture. They describe their largest market segment (41 \%) as being meteorology, climate and environment.

Until recently, the highest PC memory bandwidth was the AMD Opteron with a bit more than 10 GigaByte/sec. Recently the Intel Core i7 (Nehalem) has shown a bandwidth of 21 GigaByte on a test application. For the Intel Core i7, the peak memory bandwidth is 32 GigaBytes/sec, so for a dual-socket motherboard with 8 cores and 16 processes using SMT, the peak memory bandwidth would be 64 GigaBytes/sec. The peak speed of the Core i7 is difficult to find. I’ve found dozens of detailed reviews that do not even mention the floating-point performance. Even the Intel literature on the Core i7 does not mention FLOPS. In practice,
the best benchmark score for double-precision F.P. of one socket (four cores) was reported to be around 18 GFLOPS. On the other hand, the Core i7 is said to be capable of 4 D.P. F.P. operations per cycle, which would amount to 48.0 GFLOPS at 3 GHz. Consider a very large dataset for which the a binary double-precision operation was executed using data from main memory; that is, the dataset is too large to fit into cache. The NEC ALU is 9.6 times faster than its memory subsystem; in contrast, using a peak of 48.0 GFLOPS per socket, the Core i7 ALUs are 36.0 times faster than its memory subsystem. We see why PCs have two or three levels of cache: the bandwidth to the main memory is a bottleneck.

For PCs, in general, access to main memory is a bottleneck for many scientific applications that must treat large datasets. In many cases, an algorithm must apply arithmetic operations over a large mesh of data values, then repeat the same operations or a different set of operations on an equally large set of data values. In this case, cache is not useful because the total size of the dataset is larger than cache. For example, in a simulation of a gas, one sweep over all the data models the diffusion of heat, one sweep models the diffusion of chemical species, one sweep models the propagation of pressure, one sweep models the mass flow, then the cycle repeats.

4 Instruction Execution Model

4.1 SIMD

The Single Instruction Multiple Data (SIMD) architecture can reduce the amount of on-chip memory needed to store instructions. In later sections several SIMD architectures will be described. Even though the typical CPUs for a PC (Intel Xeon, Core 2, Core i7 and AMD Opteron, Phenom) are for the most part not SIMD, these CPUs do have a set of wide SIMD registers that are very efficient for processing of streams of data using the SSE instruction set extensions (SSE, SSE2, SSE3 and recently SSE4). When a high level of optimization is specified among the compiler options, the SSE instructions will be used where applicable.

4.2 SMT

The term SMT (a processor with SMT) is not important for the programmer, but is mentioned here because it can easily be confused with SMP. A processor with SMT can have the resources of a CPU (instruction decoding, branching, etc.) appear to behave as two or even four processors. This is different from multicore in which there really are two or four processors on the chip. For a single-socket, single-core PC that used a Xeon with SMT, the hardware mon-
itoring and the program execution would simply behave as if the board had two CPUs. For high-performance computing (HPC) applications, better performance has often been obtained when SMT was disabled. It remains to be determined the efficiency of SMT on the Intel Core i7 processor introduced at the end of 2008.

4.3 MIMD and SPMD

A cluster is primarily a MIMD machine that can run different jobs contemporaneously, on the other hand, the programming model of a parallel program is usually SPMD. To be more precise, an application could have each process of a set of parallel processes run a different, yet related algorithm for which the various algorithms need to communicate information. (Without the communication, these processes would be, trivially, independent programs.) But for the sake of conceptual simplicity, nearly always each process uses the same algorithm but with different data.

As an example, consider a numerical simulation of heat flow that uses domain decomposition. The outer loop is over the subdomains.

// Serial version.
for( idomain = 0; idomain < ndomain; idomain++ ) {
    for( iy = 0; iy < ny; iy++ ) {
        for( ix = 0; ix < nx; ix++ ) {
            v[ix + iy*nx + idomain*nx*ny] =
                stencil(ix, iy);
        }
    }
    // More code to deal with edges of the subdomains.
}

Here stencil() is a function that does arithmetic involving nearest neighbors of point (ix, iy, idomain). Using a highly schematic pseudo-code in order to describe the general idea of SPMD, consider the following parallel version.

// Parallel version.
for( idomain = 0; idomain < ndomain; idomain++ ) {
    if( processID == idomain ) {
        for( iy = 0; iy < ny; iy++ ) {
            for( ix = 0; ix < nx; ix++ ) {


15
In the parallel version, each process has a unique `processID` and the array `v` has different values in each process. The processes do not run synchronously, that is, the value of `(ix)` in one process may be different from the value in another process. The processes become synchronized when they wait to receive from other processes the data from other domains along the edges. Though this is MIMD, the terminology SPMD is used to emphasize that the code of the algorithm is nearly identical for each process. In practice, usually the size and shape of the subdomains are different for each process. On the other hand, for the sake of load balancing the computation time should be nearly equal for every process.

Because of the latency of communication when using MPI, the parallelism is usually `coarse-grained` rather than `fine-grained`. When programs are written as nested loops, the outer loop is distributed onto different processors.

## 5 Hardware Complexity of Conventional CPUs

Modern CPUs allow out-of-order execution when there is no conflict in the use of registers. In addition, in order to avoid dependencies the registers actually used can be different from the ones specified in the machine code by using the technique of `renaming` of registers. Another optimization technique is speculative execution before the result of a conditional test is known. Another cause for complexity is that the caches need to communicate at various levels in order to assure consistency of shared memory. The use of virtual memory also increases the complexity of a CPU. In general, the approach is to have a dynamic response to the pending and active instructions in order to maximize the use of multiple ALU resources. This complexity uses a substantial amount of chip real estate and a substantial amount of electrical power.

One solution is the Itanium architecture, which has a simple control structure in hardware and relies upon optimization performed by the compiler. The principal of the Itanium is that the space saved by not using a complex control structure can be used to increase the number of ALU components. The Itanium architecture uses speculative execution, branch prediction and
register renaming, all under control of the compiler; each instruction word includes extra bits for this. The Itanium 2 processor series began in 2002 and continues to be available, however, the fraction of the HPC market remains modest. No more will be said about the Itanium.

At present, on the LONI network you will only encounter Beowulf clusters and IBM clusters. The descriptions in this paper of other architectures serve to emphasize how the limitations encountered in the design of parallel computers have an impact on programming style.

Though not on LONI clusters, it would not be unusual for a scientific programmer to have access to computers with advanced GPU. An advanced, programmable GPU is called a General-Purpose GPU (GP-GPU). The processing power of GPUs are superior to the processing power of CPUs for single-precision floating-point arithmetic.

Another type of non-conventional processor is the IBM/Sony/Toshiba Cell Broadband Engine, (CBE, also known as Cell). Each chip has a convention CPU (Power architecture) and eight highly specialized processing elements. Anyone can make a cluster of CBEs using the Sony Playstation-3.

For both the GP-GPU and CBE architectures, more transistors on the chip are devoted to calculations and fewer transistors (relative to a conventional PC) are dedicated to the efficiency tricks described in the first part of this section.

6 The Cost of a Context Switch

Switching between threads of execution (which instruction stream controls the CPU) is called a ‘context switch’. A programmer usually does not take into consideration the slowdown due to a context switch because for most part it is not under the control of the programmer. Nonetheless, the overhead of context switches is usually high and should be avoided for a typical CPU. In contrast, the utilization (as multithreading) is encouraged when programming a GP-GPU.

A context switch can be described as follows.

For a multitasking system, context switch refers to the switching of the CPU from one process or thread to another. Context switch makes multitasking possible. At the same time, it causes unavoidable system overhead. The cost of context switch may come from several aspects. The processor registers need to be saved and restored, the OS kernel code (scheduler) must execute, the TLB entries need to be reloaded, and processor pipeline must be flushed. These costs are directly
associated with almost every context switch in a multitasking system. We call them direct costs. In addition, context switch leads to cache sharing between multiple processes, which may result in performance degradation. [3]

7 Tera Computer and Multithreading

There are a few computer architectures that have solved in hardware the problem of context switches. Of historical importance was the Tera Computer architecture, though it was not a commercial success. Tera Computer Company was a manufacturer of high-performance computing software and hardware, founded in 1987 in Seattle, Washington by James Rottsolk and Burton Smith. Their idea of interleaved multi-threading is noteworthy. For the Tera architecture the memory was shared, with memory on some communication nodes and processors on other communication nodes. Even with speed of light communication between memory and processors, there is a significant latency in accessing memory when the components are distributed in a supercomputer the size of a large room. Moreover, the pipelined instructions had an average execution time of 70 clock cycles (ticks) so if an instruction depended on the result of the previous instruction, the average wait would be 70 ticks. The solution was to have 128 active threads that could switch context in a single clock cycle. If one thread needed to wait, another thread would immediately start using the CPU. There was a minimum of 21 active threads in order to keep the processor busy because of the latency of instruction decode. So the Tera architecture was efficient if a problem could be divided into many threads. Take the case of the general registers of the Tera architecture as an example: each thread had 32 registers and the processor hardware had 4096 registers. Hence it was not necessary to dump the register state to cache in order to change context among the 128 threads. More details can be found in [4], [5], [6].

Most modern processors do not have hardware support for storing more than one context per core; or at most, a few contexts can be stored. (The widely used Intel x86 series has hardware support for context switching but not all context is saved. Moreover, in practice both the Microsoft Windows and Linux operating systems use software context switching on the Intel x86 CPUs.)

8 Shared Memory and Distributed Memory

The use of cache means that a change of a word in memory might only be affected on the word in cache without changing the main memory. With two or four processors (in the sense of chips
that are inserted into sockets), each processor must tell the other processors when a word in cache has changed. This is especially important for the programming model called *threads*, also implemented in OpenMP. Each thread is an independent sequence of instructions but the various threads of a program share the same memory. A PC can be purchased with as many as four sockets on the motherboard and four cores in each processor. A *core* is effectively a complete CPU. Though one could have each of the 16 cores running different programs, the interface logic on the motherboard implements *cache coherence* between all the cores on the motherboard, as if data was being changed in main memory even if it is only changed in cache. It should be mentioned that cache coherence is not word for word, but rather, in blocks so that changing one word in a cache may result in a block of words (a *cache line*) in another processor being marked invalid.

There is no cache coherence between motherboards for a typical cluster. Each computer case or blade, called a *node*, handles the main memory independently. Communication between nodes is done by sending messages, similar to the packets used on the Internet. For some interconnect hardware the communication can be done between the memory of the two nodes (remote direct memory access) nonetheless cache coherence is not maintained between the nodes so that the communication paradigm is message passing. In other words, the term *shared memory* refers to memory shared by all cores on the motherboard and the term *distributed memory* refers to the memory of an entire cluster for which different nodes send messages through an interconnection fabric.

9 Beowulf Clusters

Because of the huge market for personal computers, companies such Intel and AMD can purchase state of the art integrated circuit fabrication equipment. The speed of the transistors in PCs is state of the art. So a low-cost solution for obtaining massive computer power is to buy a large number of PCs. In order to reduce floor space, most clusters use cases somewhat larger than a pizza box that are stacked into racks. Many of the machines in the list of the fastest computers
http://www.top500.org/
are Beowulf clusters. Peter H. Beckman and Thomas Sterling had leading roles in the development of this approach to supercomputing. (The name “Beowulf” was a rather arbitrary choice.)

Any collection of PCs can become a Beowulf cluster by simply starting a parallel program on many PCs and having the program communicate using Ethernet. When a large amount of communication is needed, the PCs should all be in the same room to reduce latency. Often, a
spatially high-speed interconnection is added that has lower latency and higher bandwidth than conventional Ethernet.

Nearly always, a Beowulf cluster runs the Linux operating system. Any group of PCs can be a parallel computer simply running a programming that communicates with other PCs, typically by exchanging messages using MPI. A cluster is more than a collection of PCs by having some administrative software such as a batch queue system. Also, usually users can only log onto a few nodes, called “head nodes” while the nodes doing computation are not interrupted by servicing user requests such as editing a file.

The machines available on LONI and other machines at LSU are Beowulf clusters and IBM machines that, in effect, present the user with the same programming environment as the Beowulf clusters.

To build a Beowulf cluster it is necessary to understand some basics concerning the Linux operating system. Useful web sites for information are:

http://www.beowulf.org/
http://www.clustermonkey.net/
http://svn.oscar.openclustergroup.org/trac/oscar
http://www.phy.duke.edu/~rgb/brahma/Resources/links.php

10 GP-GPU and Multithreading

Rapid context switching re-emerged in the architecture of General Purpose Graphics Processor Unit (GP-GPU). To place GP-GPUs within the context of HPC, it should be noted that GP-GPUs have very high performance for single-precision floating point arithmetic but are much less efficient for double-precision floating point (D.P. F.P.) arithmetic, whereas a large amount of numerical calculations for scientific computing require D.P. F.P. arithmetic.

This presentation will focus on the NVIDIA product, but it should be noted the AMD/ATI has a similar GP-GPU. The architecture implemented by NVIDIA is single-instruction, multiple-thread (SIMT). A program is expected to have many threads divided into blocks of threads. The programmer can choose the number of threads in each block but should keep in mind that blocks execute 32 (or fewer) threads at a time. These 32 threads are called a warp. The threads of a warp are execute concurrently when the instruction stream of all threads in a warp are identical; though the instructions are applied to different items of data, of course. Each thread can have a different instruction stream, but in that case the various instruction streams are
processed sequentially. In other words, if the instructions of all threads in a warp are equal then the GP-GPU operates as a SIMD computer. On the other extreme, if all threads in a warp execute different instructions then the thread execution is completely sequential: one thread then the next thread, etc. As an example of a mixed case, threads can execute as SIMD until they arrive at a branch then threads that take one particular branch are executed while threads for the other branches wait their turn. If all branches converge later in the code, the execution can once again become fully SIMD. I’ve not had experience programming a GP-GPU, my information comes from [7]. The CUDA manual makes the following architectural comparison:

For the purposes of correctness, the programmer can essentially ignore the SIMT behavior; however, substantial performance improvements can be realized by taking care that the code seldom requires threads in a warp to diverge. In practice, this is analogous to the role of cache lines in traditional code: Cache line size can be safely ignored when designing for correctness but must be considered in the code structure when designing for peak performance. Vector architectures, on the other hand, require the software to coalesce loads into vectors and manage divergence manually.

Fig. 7 compares the silicon real estate of a CPU and a GPU. The block labeled DRAM is on the GPU circuit board but is external to the chip, it contains both local memory and global memory. On-chip memory is called shared memory.

![Image of CPU and GPU comparison](image)

Figure 7: The GPU’s design allows more transistors to be devoted to data processing rather than data caching and flow control. The block labeled DRAM is outside of the GPU device.
For the NVIDIA product line, each GP-GPU device is composed of between 1 to 120 multiprocessors, depending on the model. Each multiprocessor has 8 processors, so that the 32 threads of a warp can be processed in 4 cycles on one multiprocessor.

With regard to memory, each thread has private memory and each block of threads has shared memory. The access to shared memory is interesting because the use of memory banks inside the chip is similar to the a vector machine where the banks consist of memory modules, though outside the chip.

Because it is on-chip, the shared memory space is much faster than the local and global memory spaces. In fact, for all threads of a warp, accessing the shared memory is as fast as accessing a register as long as there are no bank conflicts between the threads, as detailed below. To achieve high memory bandwidth, shared memory is divided into equally-sized memory modules, called banks, which can be accessed simultaneously. So, any memory read or write request made of n addresses that fall in n distinct memory banks can be serviced simultaneously, yielding an effective bandwidth that is n times as high as the bandwidth of a single module. However, if two addresses of a memory request fall in the same memory bank, there is a bank conflict and the access has to be serialized. The hardware splits a memory request with bank conflicts into as many separate conflict-free requests as necessary, decreasing the effective bandwidth by a factor equal to the number of separate memory requests. If the number of separate memory requests is n, the initial memory request is said to cause n-way bank conflicts. To get maximum performance, it is therefore important to understand how memory addresses map to memory banks in order to schedule the memory requests so as to minimize bank conflicts. In the case of the shared memory space, the banks are organized such that successive 32-bit words are assigned to successive banks and each bank has a bandwidth of 32 bits per two clock cycles. [7]

The memory outside the GP-GPU chip that is on the same circuit board is called global memory. The global memory is not cached inside the GP-GPU, moreover the access time to/from global memory is between 400 and 600 cycles. The key point that motivates a comparison to the Tera architecture is the following.

Much of this global memory latency can be hidden by the thread scheduler if there are sufficient independent arithmetic instructions that can be issued while waiting for the global memory access to complete. […] Given a total number of threads per grid, the number of threads per block, or equivalently the number of blocks, should be chosen to maximize the utilization of the available computing resources. This
means that there should be at least as many blocks as there are multiprocessors
in the device. Furthermore, running only one block per multiprocessor will force
the multiprocessor to idle during thread synchronization and also during device
memory reads if there are not enough threads per block to cover the load latency.
It is therefore usually better to allow for two or more blocks to be active on each
multiprocessor to allow overlap between blocks that wait and blocks that can run.
For this to happen, not only should there be at least twice as many blocks as there
are multiprocessors in the device, but also the amount of allocated shared memory
per block should be at most half the total amount of shared memory available per
multiprocessor. More thread blocks stream in pipeline fashion through the device
and amortize overhead even more. [...] Allocating more threads per block is better
for efficient time slicing, but the more threads per block, the fewer registers are
available per thread. [...] The number of registers per multiprocessor is 8192. The
amount of shared memory available per multiprocessor is 16 KB organized into 16
banks. [7]

An NVIDIA GP-GPU does not have cache, though it does have on-chip memory (called
shared memory). Cache reflects the contents of main memory in a way that is transparent
to the programmer whereas the GP-GPU shared memory must be loaded explicitly. As a
consequence, on a GP-GPU a context switch does not result in the new thread stomping over
the cache of the previous thread.

11 IBM-Toshiba-Sony Cell Broadband Engine

The NVIDIA GP-GPU is an example of a processor that is not strictly SIMD but is more
efficient when all threads in a warp execute the same instruction. Another example of an
architecture that is partially SIMD is the IBM/Sony/Toshiba Cell Broadband Engine (CBE).
The CBE is used in the Sony Playstation 3 (PS3) and and it is possible to build a cluster of
PS3s running Linux. (When gaming, the PS3 does not use the Linux OS.) The CBE has one
'Power' type of processor that is able to run Linux, in addition, the CBE has eight Synergistic
Processing Elements (SPE) that provide the high-speed processing. On the PS3 when running
Linux, six SPEs are available. It is possible to purchase a server blade with two CBEs on a
circuit board, for which all eight SPEs are available on each CBE. The server blade versions
include a model that has higher performance for double-precision floating-point arithmetic
than is available on the PS3. As is the case for GP-GPU, since the mass market for the PS3 is
visualization, it is optimized for single-precision floating-point arithmetic. The layout in silicon
of the CBE is shown in Fig. [8]
IBM has built a supercomputer with the CBE. The computer nicknamed "Roadrunner" was built for the Department of Energy’s National Nuclear Security Administration and will be housed at Los Alamos National Laboratory in New Mexico. It is capable of a PetaFLOPS, a 1,000-trillion operations per second.

The Power processor on the CBE has cache and uses virtual memory, though it is a rather slow implementation. The eight, very fast SPEs on the same chip do not use cache and data must be explicitly moved to the local memories of the SPEs. More details of the architecture can be found in [8], [9]. The regard to SIMD, the SPE registers are 128 bits wide, which corresponds to four single-precision words. An instruction can operate simultaneous on four 32-bit words, eight 16-bit words or 16 bytes, and somewhat slowly on two 64-bit words. The results of operating on four 32-bit words from a pair of registers and writing to a third register are independent; for example, arithmetic overflow from the first pair does not effect the result from the same operation on the second pair. Optimizations for vector architecture also apply because the operations are most efficient when words are aligned along 128-bit boundaries and...
because the complex operations such as floating-point are pipelined, so conditional tests should
not share the same loop as arithmetic. Unlike the GP-GPU, the CBE does not have special
hardware support for multithreading. Support for prefetching is available in software.

12 ClearSpeed Coprocessor, Massively SIMD

The ClearSpeed coprocessor \[10\] provides a large number of double precision floating point
processors on a single chip. For the model CSX-600 chip, ClearSpeed makes a circuit board
with two such chips plus memory that can be inserted into a PCI-X or PCIe slot of a PC, like
any other add-on card such as WiFi or graphics. (They also manufacture a board with one
CSX-700 chip which is similar to a pair of CSX-600’s on the same chip \[11\] \[12\].)

The CSX-600 96 has poly execution (PE) cores. Each PE core has a register file (a
small, multi-port memory of 128 bytes), a small memory (6 KBytes) with fewer ports, an
integer arithmetic unit and two floating-point units. All 96 cores execute the same instruction,
therefore the architecture is massively SIMD. The CSX-600 has one master control unit that is
similar to a conventional processor. The control unit is efficient in multithreading allowing fast
swapping between multiple threads. Like the GP-GPU, multithreading is used to overlap I/O
and computation. The threads are intended primarily to support efficient overlap of I/O and
compute in order to hide latency of external data accesses. To quote the product literature

In the simplest case of multi-threaded code, a program would have two threads:
one for I/O and one for compute. By pre-fetching data in the I/O thread, the
programmer (or the compiler) can ensure the data is available when it is required
by the execution units and the processor can run without stalling.

Unlike a GP-GPU, the control unit executes only one thread at a time, the PE’s do not each
have their own thread.

Programs typically have ‘if’ statements that control program execution. The control unit
can branch based on logical or arithmetic conditions. In contrast, the PE’s cannot branch
(they all receive the same instruction) but they can each individually disable execution of an
instruction based on a logical or arithmetic condition. On each PE, the result of one cycle can
be used to disable the execution of an instruction on the next cycle.

ClearSpeed provides a compiler which uses extensions to the C language to identify data
that can be processed in parallel. The ClearSpeed libraries allow the code on the coprocessor to
be called from Fortran, C or C++.
Another feature is the ‘swazzle path’ that connects the register file of each PE core with the register files of its left and right neighbors. This communication between register files is contemporaneous with the ALU execution.

By way of comparison to the ClearSpeed swazzle path, the CBE has a ‘racetrack’ for high-speed communication between the eight SPEs that provides a much higher internal bandwidth than the I/O to cachable memory (of the Power processor or external memory).

The product has a large FLOPS/watt though the clock speed of 250 MHz leaves a lot to be desired; an Intel or AMD processor runs at 2-3 GHz. As is the case with a GPU or Cell, the computational capacity far exceeds the bandwidth to external memory.

13 Large Embarrassingly Parallel Tasks

The term *embarrassingly parallel* refers to programs that can be partitioned into parallel tasks having very little communication between the tasks. When the number of contemporaneous tasks is in the hundreds or thousands, then it is not unusual for an error to occur in some task during the course of several days. For MPI, an error in just one processor may result in the failure of the entire program. Rather than running a single program such as MPI, for large, embarrassingly parallel programs, independent processes can be coordinated by a master program. The configuration is like a queue, though not the PBS/MOAB or LoadLeveler job queue system of the LSU and LONI clusters. The independent contemporaneous processes distributed on the computational nodes receive task specifications from the master. When a task has been completed, the process does not halt but rather requests a new task from the master. If any process halts before completing a task, the master can choose another process to redo the task. This master-slave paradigm could even be used on a single computational node with threads and shared memory, however, for the cluster-wide or even multi-cluster implementation each slave can be an independent process.

A well-documented example is the MapReduce programming abstraction developed by Google, see [13]. The abstraction starts with a set of key-value pairs, which map to an intermediate set of key-value pairs, then for each intermediate key the list of values is reduced. In order to be more concrete, consider the example of counting the number of occurrences of each word in a large collection of documents. This example is from [13].

```java
map(String key, String value) {
    // key: document name
    // value: document contents
```
for each word w in value:
    EmitIntermediate(w, "1");
}
reduce(String key, Iterator values) {
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
}

The map function emits each word plus an associated count of occurrences (just 1 in this simple example). The reduce function sums together all counts emitted for a particular word.

The MapReduce abstraction fits a variety of different kinds of computations performed by Google. More algorithmic details will not be given in this presentation, a Google search on “mapreduce” finds articles from labs.google.com, Wikipedia and even an article that says “MapReduce: A major step backwards ... As both educators and researchers, we are amazed at the hype that the MapReduce proponents have spread about how it represents a paradigm shift in the development of scalable, data-intensive applications.” The key point for our purposes is to understand how the contemporaneous processes are managed. The text to be processed has already been broken into blocks and distributed on worker machines, moreover, the blocks are replicated. A master process distributes the tasks of a query onto worker machines that have the relevant blocks of text, or at least on worker machines near the relevant blocks, where near means nodes that share the same interconnect switch. For fault tolerance and speed, as results come back and a query is close to completion, portions that have not been complete are assigned to other worker machines.

14 OpenMP

14.1 Overview of OpenMP

The descriptions and examples of this section are taken from several sources [14] [15] [16] [17]. The primary resources on the Web are:

OpenMP is a set of extensions of Fortran, C and C++ for parallel execution of processes in a shared memory address space. For a cluster of PCs, the processors that share memory address space are the processors of one node, that is, one motherboard. Some specialized computers can have shared memory for a number of processors far greater than those on a PC motherboard. Processes communicate by reading and writing the shared memory. Though the OpenMP processes are threads, the advantage of OpenMP in contrast to programming directly with threads is that OpenMP adds a syntactic structure that avoids the complexity of multithreading. The OpenMP extensions take the form of C/C++ pragmas or Fortran directives that are ignored by compilers that do not support OpenMP. In many cases, the code will be valid as serial code if the OpenMP pragmas are ignored.

Another training session will be dedicated to teaching OpenMP, so this will be just a brief overview. The OpenMP model consists of one master thread and regions of code in which a team of threads run in parallel, while sharing data through the shared memory, as shown in Fig. 9. As well as shared data, each thread can also have private data. Though threads could be ‘forked’ as needed, in practice the worker threads persist but are inactive. Usually the most efficient usage of a node is one thread per core. The number of threads can be specified by the environment variable OMP_NUM_THREADS and it can be specified at runtime using omp_set_num_threads().
OpenMP: Programming Model

- Fork-Join Parallelism:
  - Master thread spawns a team of threads as needed.
  - Parallelism is added incrementally: i.e. the sequential program evolves into a parallel program.

Figure 9: The programming model consists of one master thread and the use of multiple threads in parallel regions of the program.
14.2 OpenMP Code Examples

The following example simply has the same code execute in parallel with a different value for the first parameter of the function call.

```c
double a[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int id = omp_get_thread_num();
    myfunction(id, a);
}
```

In the above, each thread calls `myfunction(id, a)` for id from 0 to 3.

There is a pragma tells the compiler to split up loop iterations among the threads. The following code would call the function n times whether or not the compiler and runtime environment implemented OpenMP.

```c
#pragma omp parallel
#pragma omp for
for(i = 0; i < n; i++) {
    somefunction(i);
}
```

A variable can be declared as either shared or private to each thread. Most variables are shared by default, such as Fortran COMMON blocks and C file scope variables. On the other hand, ‘stack’ variables used in subroutine calls are private, as are automatic variable declared within a statement block.

A private copy is not storage associated with the original.

```c
// Wrong.
int sum = 0;
#pragma parallel for private(sum)
for(int j = 1; j < 1000; j++) {
    sum += j;
}
```
It is possible to write an OpenMP program that runs correctly as a serial program if the OpenMP pragmas are ignored. It is usually desirable to avoid OpenMP syntax that gives different results depending on the number of parallel processes. One can expect some small differences between a parallel version in comparison to a serial version because operations such as floating point addition give results that depend upon the order of the operations. Specifically, a serial loop would have a fixed order whereas the parallel version of a loop or a reduction operation may have a different order.

OpenMP with MPI can be combined. If the MPI calls are not thread safe, then the MPI calls must occur in the serial part of the OpenMP program. If MPI is not thread safe then the various threads would share the same housekeeping variables internal to the message passing, the buffer space and the variables related to the driver of the interconnection hardware, making a mess. It would seem that a hybrid of OpenMP and MPI would be most efficient because the OpenMP part has fast communication. However, in practice the hybrid approach is complicated to program.

Only one thread at a time can enter a ‘critical’ section.

```
C$OMP PARALLEL DO PRIVATE(B), SHARED(RES)
  DO I = 1, N
    B = DOIT(I)
C$OMP CRITICAL
    CALL UTILIZE(B, RES)
C$OMP END CRITICAL
END DO
```

OpenMP has synchronization pragmas such as ‘barrier’ and it has a pragma to specify that only the master process execute a specific block of code.

Example of computing pi.

```
use omp_lib
pi=0.0
w=1.0/n
!$OMP parallel private(x,sum)
sum=0.0
!$OMP do
  DO i=1,n
    x=w*(i-0.5)
```

\[
\text{sum}=\text{sum}+f(x)
\]
\text{ENDDO}
\!\$\text{OMP end do}
\!\$\text{OMP critical}
\pi=\pi+w*\text{sum}
\!\$\text{OMP end critical}
\!\$\text{OMP end parallel}

Where \( f(a) = \frac{4.0}{1.0 + a^2} \).

The example shows the use of private variables, which are undefined at the start of a parallel region and undefined upon exit of a parallel region. There are, however, directives to change this behavior. A general rule of multithreading is that read-modify-write of a shared variable must be atomic, otherwise two threads might get the same value and one of the modifications would be lost. As well as the use of ‘critical’ in the example, there are other solutions such as the pragma ‘atomic’ and a reduction pragma.

15 Message Passing

15.1 Overview

Message passing does not require shared memory and so can be used for the communication between hundreds or even thousands of parallel processes.

The most widely used protocol is Message Passing Interface (MPI). Conventionally, both the sender and receiver must call the appropriate routine, though MPI-2 also implements one-sided communication.

Because of the overhead of communication, MPI is more appropriate for coarse-grained parallelism rather than fine-grained parallelism. MPI can implement Multiple Instruction Multiple Data (MIMD) though the most commonly used paradigm is Single Program Multiple Data (SPMD).

Initially MPI launches copies of an executable. Each copy could branch to execute a different part of the code (different yet related algorithms) but it is much more common for each process to execute the same code with differences in the data for the inner-most loops. For example, domain decomposition with one domain per process. Actually, each process can
handle more than one domain, this is especially relevant when domains have a wide range of sizes for which the goal is load-balancing so that every process finishes an iteration at the same time.

MPI works better for coarse-grained parallelism, as opposed to fine-grained parallelism. Each copy of the executable runs at its own rate and is not synchronized with other copies unless the synchronization is done explicitly with a “barrier” or with a blocking data receive. As an aside, the LAM version of MPI permits heterogeneous executables.

You can install public-domain MPI on your home computer, though it is oriented towards Unix-like operating systems. You can run MPI on a single personal computer, even off-line using “localhost” as the node name. So one can test parallel programs on a single laptop, afterall the laptop might have two or even four cores. The number of processes can exceed the number of cores, though efficiency will be reduced.

Sources of Documentation and Software

- **http://www.mpi-forum.org/docs/docs.html**
  Contains the specifications for MPI-1 and MPI-2.

- **http://www.redbooks.ibm.com/redbooks/pdfs/sg245380.pdf**
  RS/6000 SP: Practical MPI Programming
  Yukiya Aoyama and Jun Nakano

- **http://www-unix.mcs.anl.gov/mpi/**
  Links to tutorials, papers and books on MPI.

- **http://www-unix.mcs.anl.gov/mpi/tutorial/index.html**
  A list of tutorials.

- **Books**
  - Using MPI, by W. Gropp, E. Lusk and A. Skjellum [18]
  - Using MPI-2, by W. Gropp, E. Lusk and R. Thakur [19]
  - MPI: The Complete Reference, Volume 2 - The MPI-2 Extensions,
    William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg,
    William Saphir and Marc Snir [20]
  - Parallel Programming With MPI, by Peter S. Pacheco [21]

- **Software**
  - MPICH2
    http://www.mcs.anl.gov/research/projects/mpich2/
15.2 Examples

The overall, outermost structure of an MPI program is shown below.

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv) {
    int rank;
    MPI_Init( argc, argv );
    // From here on, N processes are running, where N is
    // specified on the command line.
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    // Can use the rank to determine algorithm behavior.
    if( rank == 0 ) fprintf(stdout, "Starting program\n");
    fprintf(stdout, "Process %d running\n", rank );
    // At end of program, finalize.
    MPI_Finalize();
    return 0;
}
```

All code before `MPI_Init()` runs as a single executable but typically `MPI_Init()` is the first executable line. Since MPI is based on distributed memory, aside from MPI communications, the variables of each process are independent.

The C++ bindings of MPI will not be discussed. For other languages, the headers for each program unit are the following.

- For C,
  ```c
  #include <mpi.h>
  ```
- For Fortran 77,
  ```
  include 'mpif.h'
  ```
- For Fortran 95,
  ```
  use mpi
  ```
The Fortran subroutines of MPI use the last argument to supply an error status whereas C functions of MPI use the return value for the error status. Success corresponds to the value of MPI_SUCCESS.

The symmetry of the copies of the algorithm must be broken for communication. In particular, the following example will deadlock.

Process 0:
    send(message, proc1); // Wrong, for blocking send, will deadlock.
    recv(message, proc1);

Process 1:
    send(message, proc0);
    recv(message, proc0);

The send() and recv() in this example are more simple than the actual MPI routines. The point here is to emphasize that often the sequence of instructions of an MPI program are nearly identical for every process, but additional attention is needed to avoid communication deadlock.

15.3 Point-to-Point Communication

15.4 Blocking Send

int MPI_Send(void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm)

The program can change the contents of buf when the function returns. When the function returns either a local copy has been made or the message has already been sent to the destination, depending on implementation and message size. The count is the number of elements, not the number of bytes. The most common case is that “comm” is MPI_COMM_WORLD.

15.5 Blocking Receive

int MPI_Recv(void *buf, int count, MPI_Datatype type, int source, int tag, MPI_Comm comm, MPI_Status *status)
The size of count must be at least as large as the message size. Array buf has the complete message when the function returns.

To discover the actual size of the message use

\[
\text{int MPI\_Get\_count(MPI\_Status \*status, MPI\_Datatype type, int \*count);} \\
\]

Rather than specifying a specific source process, any source can be specified by the argument MPI\_ANY\_SOURCE. Messages can be categorized by an integer-valued “tag.” A recv() can be restricted to a specific tag, or to receive a message with any tag the argument MPI\_ANY\_TAG can be used. Messages are non-overtaking between matching send() and recv() for a single-threaded application. On the other hand, MPI does not guarantee fairness for wildcard receives, that is, all receives could come from the same source while ignoring messages from other sources.

Other send types are “buffered”, “synchronous” and “ready.” Only buffered send will be described.

15.6 Buffered Communication

A local buffer is used to hold a copy of the message. The function returns as soon as the message has been copied to the local buffer. Advantage: prevents deadlocks and quickly returns. Disadvantage: latency introduced by having an additional copy operation to the local buffer. The buffered send routine has the same arguments as blocking send and is called MPI\_Bsend.

Buffer Allocation

The program code for allocating and releasing buffer space is shown below.

```
// Allocate buffer space
int size = 10000;
int stat;
char* buf = (char*) malloc(size);
// Make buffer available to MPI.
stat = MPI\_Buffer\_attach(buf, size);
// When done release the buffer.
```
stat = MPI_Buffer_detach(buf, &size);
free(buf);

If the program has nearly constant behavior during a run, then the buffer allocation need only be done once at the beginning of the program (after calling MPI_Init).

### 15.7 Non-blocking Send and Recv

For non-blocking send and recv, when a function returns, the program should not modify the contents of the buffer because the call only initiates the communication. The functions MPI_Wait() or MPI_Test() must be used to know whether a buffer is available for reuse. Analogously two pending MPI_Isend(), MPI_Ibsend() or MPI_Irecv() cannot share a buffer.

Without user-defined buffer

```c
int MPI_Isend(void* buf, int count, MPI_Datatype type, int dest,
              int tag, MPI_Comm comm, MPI_Request *request);
```

With user-defined buffer

```c
int MPI_Ibsend(void* buf, int count, MPI_Datatype type, int dest,
               int tag, MPI_Comm comm, MPI_Request *request);
```

The receive command is

```c
int MPI_Irecv(void* buf, int count, MPI_Datatype type, int source,
              int tag, MPI_Comm comm, MPI_Request *request);
```

The following function is used to test whether an operation is complete. Operation is complete if “flag” is true.

```c
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);
```

The following can be called to wait until the operation associated with request has completed.
int MPI_Wait(MPI_Request *request, MPI_Status *status);

Often the simplicity of blocking send and recv is preferred.

There are more types of point-to-point communication such as “persistent communication requests”, send-receive combined, and derived data types including structures; all of these will not be covered in this tutorial.

15.8 Summary of Collective Communication

The various categories are the following.

- Barrier synchronization blocks the caller until all group members have called it.
- Broadcast from one member to all members of a group.
- Gather data from all group members to one member.
- Scatter data from one member to all members of a group.
- Allgather, a gather where all members of the group receive the result.
- Alltoall, every process in a group sends a message to every other process.
- Global reduction such as sum, max, min and others. The result can be returned to all group members. The reduction can be a user-defined function.
- Reduction and scatter where the result is a vector that is split and scattered.
- Scan, which is a reduction from all a each processes with all lower ranked processes. No further description of scan will be given in this tutorial.

For collective routines that have a single originating or receiving process, that special process is called the “root”. MPI routines have the same set of arguments on all processes, though for some collective routines some arguments are ignored for all processes except the root. When a caller returns, the caller is free to access the communication buffers. But it is not necessarily the case that the collective communication is complete on other processes, so the call (aside from MPI_Barrier) does not synchronize the processes. The processes involved are the group associated with the MPI_COMM argument. MPI guarantees that messages generated on behalf of collective communication calls will not be confused with point-to-point communications. The amount of data sent must be equal to the amount of data received, pairwise between each process and the root.
15.9 Domain Decomposition and Ghost Vertices

A widely used technique for parallelizing a simulation of a physical process on a grid is domain decomposition. Very often physical processes are modelled by nearest neighbor interactions, in which case when subdomains are put onto different processes, communication is needed along the edges. Because of the communication overhead of MPI, the subdomains should not be too small.

Figure 10: Domain decomposition is a widely used method of parallelization. Values (data) are at the intersection of the lines. Three subdomains are shown.
Figure 11: Put each subdomain onto a different process. Each subdomain is expanded to include ghost vertices. The extra vertices shown are appropriate when the interactions are between nearest neighbors. Arrows indicate the communication using MPI.
16 Other Parallel Languages

16.1 Unified Parallel C (UPC)

To quote the UPC web page:

The language provides a uniform programming model for both shared and distributed memory hardware. The programmer is presented with a single shared, partitioned address space, where variables may be directly read and written by any processor, but each variable is physically associated with a single processor. UPC uses a Single Program Multiple Data (SPMD) model of computation in which the amount of parallelism is fixed at program startup time, typically with a single thread of execution per processor.

In short, UPC both

- extends C to an SPMD parallel programming model, and
- shares data based on a parallel global address space.

UPC uses the GASNet networking layer to implement the global address space. GASNet has a reference implementation that can easily be ported to new architectures, and in addition, “implementers can choose to bypass certain functions in the reference implementation and implement them directly on the hardware to improve performance of specific operations when hardware support is available (e.g. special network support for puts/gets or hardware-assisted broadcast).” GASNet is available for third-party networks such as Infiniband/Mellanox VAPI, Quadrics QSNet1/2, and Myrinet GM; as well as the proprietary networks of Cray and SGI. Moreover, there is an implementation for MPI and Ethernet IP.

16.2 Co-Array Fortran

Co-Array Fortran, an extension of Fortran 95, allows arrays to be distributed among processes. It is most efficient for shared memory architectures but can also be implemented with MPI-2 on distributed memory architectures in which the interconnection supports remote direct memory access. For distributed memory, Co-Array Fortran has been ported to two other one-sided communication protocols: ARMCI and GASNet.
16.3 High Performance Fortran (HPF)

In the late 1990’s the parallel extension to the Fortran 90 language, HPF, became widely used. In this language array indices could describe the distribution of data on a group of nodes, as well as the local memory array dimensions. These arrays could be on distributed memory for which the exchange of data would require internode communications, usually message-passing. Arithmetic expressions involving array syntax, as is common for Fortran 90, would be converted by the compiler into internode communication where necessary. In practice, the efficiencies of HPF programs were often much less than equivalent programs written using explicit message passing. At the present time HPF is rarely used.

By hiding the communication the programmer would tend to not write the most efficient code. The same may be true of the parallel global address space implemented with the GASNet networking layer; that is, it simplifies the writing of a parallel programs but the author may ignore the very large latencies when the access requires communication between nodes.

17 Program Optimization

The advice and code examples for program optimization described herein are from several sources: IBM XLF Optimization Guide [22], Software Optimization Guide for AMD Family 10h Processors [23] and IBM pSeries Program Optimization [24]. The HPC@LSU and LONI Linux clusters have Intel processors for which the Intel Performance Libraries can be used. An overview is given in the white paper: Boosting Application Performance using Intel Performance Libraries [25].

The advice in this section applies to serial programs. Nonetheless, the discussion of limited cache size, memory latency and ALU pipelining sets the stage for understanding the following suggestions.

17.1 Increase Operations per Memory Access

To increase operations per memory access, reuse a variable by unrolling the outer loop. (Compiler might do this.)

\[
\begin{align*}
\text{C 2 FLOPs / 2 loads} & \quad \text{C 8 FLOPs / 5 loads} \\
\text{DO i = 1, n} & \quad \text{DO i = 1, n, 4}
\end{align*}
\]
Increasing the number of ALU operations compared to the number of memory reads and writes has been emphasized by the CUDA programming manual for the NVIDIA GP-GPU, since a single GP-GPU can have hundreds of ALUs. This would also apply to the ATI GP-GPU. More generally, as the number of cores in a typical PC processor goes to four and more, the overall speedup is limited by the memory bandwidth.

An example of reducing by half the data dependence and eliminating some loads and stores. (Compiler might do this.) Note that in the second line inside the do-loop on the right, calculating $a(i+2)$ needs to wait for $a(i+1)$, but at least there is a reuse of $a(i)$, and also $a(i+1)$ can be used without being stored then loaded again.

Unrolling outer loop exposes prefetch stream. IBM P-Series can have up to 8 prefetch streams. Note that the distance in memory between $a(i,j)$ and $a(i,j+1)$ is the inner dimension of array $a$.

Various CPUs and operating systems have different strategies of prefetching from main memory to cache. Moreover, control of prefetching is not part of typical high-level programming...
languages. The AMD optimization manual discusses prefetching in detail but the examples are in assembly language. The IBM XLF optimization guide refers to compiler options related to prefetching.

The following prefetch information is specific to the IBM P-Series. Other processors will have prefetching but the rules would be different.

- Memory bandwidth is proportional to the number of streams.
- Streams are roughly the number of right hand side arrays.
- Up to eight streams.

17.2 Explicit Parallelism in Code

The example shown here has four independent parallel chains, but the hardware implementation is not parallel. Instead, the parallelism allows more efficient use of the pipeline of the floating point adder.

Avoid

double a[100], sum;
int i;
sum = 0.0;
for (i = 0; i < 100; i++) {
    sum += a[i];
}

Preferred

double a[100], sum1, sum2, sum3, sum4, sum;
int i;
sum1 = 0.0; sum2 = 0.0; sum3 = 0.0; sum4 = 0.0;
for (i = 0; i < 100; i + 4) {
    sum1 += a[i];
    sum2 += a[i+1];
    sum3 += a[i+2];
    sum4 += a[i+3];
The dependency may not be obvious, it is in the operation ‘+=’; the previous sum must exit from the ALU pipeline to be used in the next iteration of the loop. While waiting, three other independent accumulations can be done. The splitting into four independent streams is similar to OpenMP, but in this case the four streams are executed in one core in order to keep the ALU pipeline full. The compiler should be able to find this simple transformation but similar yet more complicated cases may need to be done by hand.

17.3 Remove Store-to-Load Dependencies

To paraphrase the AMD Software Optimization Guide: If the dependency occurs while operating on arrays, many compilers are unable to optimize the code in a way that avoids the store-to-load dependency. The dependency shown in the following code does not avoid the delay due to waiting for the previous result to exit from the floating point pipeline. The use of a temporary variable that can be stored in a register does avoid the storing to memory of the previous result.

Avoid

double x[VECLEN], y[VECLEN];
unsigned int k;
for (k = 1; k < VECLEN; k++) {
    x[k] = x[k-1] + y[k];
}

Preferred

double x[VECLEN], y[VECLEN];
unsigned int k;
double t;
t = x[0];
for (k = 1; k < VECLEN; k++) {
    t = t + y[k];
    x[k] = t;
}
17.4 Memory Access and Cache

A stride will reduce the memory bandwidth since sequential access is most efficient. On IBM P-Series a stride of 2 cuts in half the memory bandwidth and a stride of 4 reduces the memory bandwidth by a quarter.

```c
// Inefficient due to stride of 4.
for( i = 0; i < n; i += 4 ) {
    sum += a[i];
}
```

Use blocks of data that fit into cache. For example, matrix transpose.

```c
DO j1 = 1, n - nb + 1, nb
    j2 = MIN(j1 + nb - 1, n)
    DO i1 = 1, m - nb + 1, nb
        DO i = 1, n
            i2 = MIN(i1 + nb - 1, m)
            DO j = 1, m
                B(j,i) = A(i,j)
            END DO
            B(j,i) = A(i,j)
        END DO
    END DO
END DO
```

In practice, rather than writing complicated code, basic matrix operations have been optimized in various libraries such as ATLAS.

There are two goals that can be obtained with blocking data:

- The inner loops should work on a dataset that can fit into cache.
- The number of transfers to and from main memory should be minimized.

17.5 Aliasing

To quote from the IBM XLF optimization guide:
When aliasing occurs there is less opportunity for optimization transformations to occur on and around aliased variables than variables where no aliasing has taken place. For example, if variables A, B, and C are all aliased, any optimization must assume that a store into or a use of A is also a store or a use of B and C, even if that is not the case.

The following situations can lead to pessimistic aliasing:

- When you assign a pointer the address of any variable, the pointer can be aliased with globally visible variables and with static variables visible in the pointer's scope.
- When you call a procedure that has dummy arguments passed by reference, aliasing occurs for variables used as actual arguments, and for global variables.
- The compiler will make several worst-case aliasing assumptions concerning variables in common blocks and modules. These assumptions can inhibit optimization.

Example of aliasing in Fortran.

```fortran
C Example of dependency that
C inhibits optimization.
DO i = 2, n
   a(i) = a(i - 1)*c(i)
ENDDO
C Apparently, no dependency here.
DO i = 2, n
   a(i) = b(i - 1)*c(i)
ENDDO
```

Example of aliasing in Fortran, continued.

```fortran
C Inhibited optimization because of
C potential aliasing.
SUBROUTINE abc(a, b, c)
   DO i = 2, n
      a(i) = b(i - 1)*c(i)
   ENDDO
END
C The problematic subroutine call.
CALL abc(a,a,c)
```
With regard to C and C++, use the array operator rather than pointer. For example, *p and *q can refer to the same memory location, while x[0] and x[2] cannot.

17.6 Inlining

Avoid many calls to small procedures in order to avoid the function call overhead. When you do have small procedures, use the inlining compiler option that places the code for the procedure where the call would be made. However, excessive inlining can lead to a decrease in performance because running larger programs can cause more cache misses and page faults.

17.7 Avoid Small Loops

Unroll small loops that have a small fixed loop count and a small loop body in order to avoid the overhead of the loop.

```c
// 3D-transform: Multiply vector V by // 4x4 transform matrix M.
for (i = 0; i < 4; i++) {
    r[i] = 0;
    for (j = 0; j < 4; j++) {
        r[i] += m[j][i] * v[j];
    }
}

r[0] = m[0][0] * v[0] + m[1][0] * v[1] + m[2][0] * v[2] + m[3][0] * v[3];
```
17.8 Loop Hoisting

The programmer should try to remove branches from inner loop.

Avoid

```c
for (i = 0; i < n; i++) {
    if (CONSTANT0) {
        DoWork0(i); // Does not affect CONSTANT0.
    } else {
        DoWork1(i); // Does not affect CONSTANT0.
    }
}
```

Preferred

```c
if (CONSTANT0) {
    for (i = 0; i < n; i++) {
        DoWork0(i);
    }
} else {
    for (i = 0; i < n; i++) {
        DoWork1(i);
    }
}
```

17.9 Extracting Common Subexpressions

Try to minimize the number of division operations because division is slow.

Avoid

```c
double a, b, c, e, f;
e = a / c;
f = b / c;
```
Preferred

double a, b, c, e, f, t;
t = 1 / c;
e = a * t
f = b * t;

The compiler should be able to optimize the simple example shown in the previous slide. But for more complicated code the general rule is to move calculations as far as possible away from the inner loop.

Here is another example of placing the calculation of values as far outside of a loop as possible. The most simple case shown here should be done automatically by any decent compiler, though less obvious cases may need to be hand coded.

for( k = 0; k < nz; k++) {
    q = k*nx*ny;
    for( j = 0; j < ny; j++) {
        m = q + j*nx;
        for( i = 0; i < nx; i++) {
            c[i + m] = a[i + m] * b[i + m];
        }
    }
}

17.10 Loop Unroll and Jam

Several decades ago, Prof. Ken Kennedy or Rice University (who died in 2007) claimed that commercial compiler optimization techniques were always ten years behind what was being developed and documented by academic research. He said that his preferred compiler test was to compare the execution time of a simple set of nested loops against the execution time of the same loops after hand-coding an unroll and jam transformation. This is a well-known transformation that a compiler should do automatically but he found that most new compilers did not do. A ‘new’ compiler is an optimized compiler for new hardware, such as IBM Power, DEC Alpha, MIPS, Itanium or HP PA-RISC when each of these architectures were first introduced. Those who have started programming recently may not be aware of the diversity of architectures that once existed. After hearing his presentation I tried it on a new compiler for a workstation by Apollo Computer Corp. and indeed I found that hand-coded unroll and jam resulted in
a program that ran much faster than the best compiler optimization of the original nested loops. In recent years when I have mentioned this to people who follow compiler development, I’ve been told that the time lag between academic research and commercial implementation of compiler techniques is now much less than ten years. The moral of this story is: don’t assume that a compiler will make the obvious optimizations.

With regard to unroll and jam, I found in an old (1998) HP Parallel Programming Guide [26] a compiler directive to force this transformation. Also, Rice University has a page on an application called Loop Tool for controlled loop fusion, but the page was last updated in 2003.

http://www.hipersoft.rice.edu/looptool/

An example of “unroll and jam” is shown below, taken from [26]. The original loop could use as many as 3 registers whereas the final version can use as many as 14 registers.

Original loop.

DO I = 1, N
    DO J = 1, N
        DO K = 1, N
            A(I,J) = A(I,J) + B(I,K) * C(K,J)
        ENDDO
    ENDDO
ENDDO
ENDDO

Unrolled loop.

DO I = 1, N, 2
    DO J = 1, N
        DO K = 1, N
            A(I,J) = A(I,J) + B(I,K) * C(K,J)
        ENDDO
    ENDDO
    DO J = 1, N
        DO K = 1, N
            A(I+1,J) = A(I+1,J) + B(I+1,K) * C(K,J)
        ENDDO
    ENDDO
ENDDO
ENDDO
Jamming (fusing).

DO I = 1, N, 2
   DO J = 1, N
      DO K = 1, N
         A(I,J) = A(I,J) + B(I,K) * C(K,J)
         A(I+1,J) = A(I+1,J) + B(I+1,K) * C(K,J)
      ENDDO
   ENDDO
ENDDO

DO I = 1, N, 2
   DO J = 1, N, 4
      DO K = 1, N
         A(I,J) = A(I,J) + B(I,K) * C(K,J)
         A(I+1,J) = A(I+1,J) + B(I+1,K) * C(K,J)
      ENDDO
      DO K = 1, N
         A(I,J+1) = A(I,J+1) + B(I,K) * C(K,J+1)
         A(I+1,J+1) = A(I+1,J+1) + B(I+1,K) * C(K,J+1)
      ENDDO
      DO K = 1, N
         A(I,J+2) = A(I,J+2) + B(I,K) * C(K,J+2)
         A(I+1,J+2) = A(I+1,J+2) + B(I+1,K) * C(K,J+2)
      ENDDO
      DO K = 1, N
         A(I,J+3) = A(I,J+3) + B(I,K) * C(K,J+3)
         A(I+1,J+3) = A(I+1,J+3) + B(I+1,K) * C(K,J+3)
      ENDDO
   ENDDO
ENDDO

Final jamming (fusing).

DO I = 1, N, 2
   DO J = 1, N, 4
      DO K = 1, N
         A(I,J) = A(I,J) + B(I,K) * C(K,J)
17.11 Self-Tuning Code

Self-adapting Numerical Software (SANS) is software that can adapt itself to the hardware architecture, either during installation onto a specific hardware or during runtime [27] [28] [29]. Two well-known examples are Fastest Fourier Transform in the West (FFTW) [30] and Atlas (which provides a complete BLAS API and a very small subset of the LAPACK API) [31] [32].

To paraphrase Ref. [27], the challenges of self-adapting software are

- Due to the complexity of modern computers and compilers, few people can predict the performance based on an algorithm expressed in a high-level language
- The speed of innovation in hardware and compilers is so great that a specific choice of tuning soon becomes obsolete
- The number of algorithmic kernels is large and growing rapidly
- Tuning depends on the specific instance, for example, solving a sparse matrix requires information about the matrix structure.

One approach actively being researched is the “Self-Adapting System for Linear Solver Selection” (SALSA) [33], which selects iterative methods for linear systems. Very briefly, the procedure is the following:

- A large collection of test problems are solved by every available method.
- Each problem is assigned a class corresponding to the best solution method.
- For each problem a list of characteristics is created.
• The choice of algorithm for a new problem depends on the characteristics of the new problem.

## 17.12 Note on efficiency of all-to-all

Several times there has been a debate on the Beowulf mailing list about the efficiency of MPI\_Alltoall. One would expect that MPI developers (MPICH, OpenMPI and Cray, for example) would spend time optimizing this function. Mailing list contributors who have worked on MPI professionally have claimed that MPI\_Alltoall is often slow because of the huge number of combinations that must be covered. The number of processors can vary from tens to thousands. Short messages often need to be handled differently from long messages. Moreover, some processes may be on the same node. Even if MPI\_Alltoall is optimized at one point in time, the attributes of the communication fabric will change over time. Though I recall this being a perennial subject, debated on the mail list every few years, I am only able to find a 2000 thread named “Beowulf and FFT” in http://www.beowulf.org/archive/2000-July/thread.html

The claim made by some, for which there is no universal agreement, is that the number of possible cases exceeds the capacity of even the brightest programming team.

As described in Ref. [27], J. Dongarra et al found that the efficiency of MPI collective communications depends on

• the underlying hardware architecture

• the virtual topology used to implement the communication (flat-tree, binary tree, k-chain, etc.)

• and the segment size used to breakdown the data into individual messages.

This is another area for which a method similar to SALSA would be valuable.

To summarize, though self-tuning is a large undertaking for any code and would only be practical for important applications, even more modest programs should be written in a style that can easily be modified. This is especially true for applications that should be portable to many different types of supercomputers.
Acknowledgements

I want to thank my colleagues for suggestions and corrections. In particular, Brett Estrade had many helpful suggestions.
References


57