INTRODUCTION

The processor is the main component of a computer system. It is a logic circuitry that processes instructions. It is also called CPU (Central Processing Unit). It is the brain of the computer system. Processor is mainly responsible to do all the computational calculations, logical decision making and to control different activities of the system. Central Processing Unit is very complicated chip consisting of billions of electronic components. It is fitted on the motherboard with other electronic parts. The main work of the processor is to execute low level instructions loaded into the memory. The processor can be manufactured using different technologies - Single core processor and multicore processor. According to [1] processors can be divided into three types- multiprocessors, multithreaded processors and multicore processors.

There are new trends in the CPU manufacturing industry which are based on the idea that while clock speeds can only be increased to a limit and there is limit to number of electronic components to be used in a core. Many other technologies are there to speed things up and open ways for better and more powerful central processing units [3].

When we are unable to increase the performance of CPU furthermore by modifying its running frequency, then new technology called multicore architecture helps. In multicore architecture we can put more than one core on a single silicon die. This new approach to enhance the speed came with some additional benefits like better performance, better power management and better cooling as the multi core processors run at a lower speed to dissipate less heat. It also has some disadvantages like existing programs need to be rewritten as per new architecture. If we do not write programs with special focus for running on parallel cores, we will not get advantage of multicores. In this paper section II discusses the single core processor while in section III, multicore processors have been discussed in detail. The section IV gives a detailed comparison to two different types of processor and the last section V, concludes this topic.

SINGLE-CORE PROCESSORS

Single core processors have only one processor in die to process instructions. All the processor developed by different manufacturers till 2005 were single core. In today’s computers we use multicore processors but single core processor also perform very well. Single core processors have been discontinued in new computers, so these are available at very cheap rates.

Problems of Single Core Processors:
As we try to increase the clock speed of this processor, the amount of heat produced by the chip also increases. It is a big hindrance in the way of single core processors to continue evolving.

Fig. 1 Single core Processor Architecture
MULTI-CORE PROCESSORS

Multicore processors are the latest processors which became available in the market after 2005. These processors use two or more cores to process instructions at the same time by using hyper threading. The multiple cores are embedded in the same die. The multicore processor may looks like a single processor but actually it contains two (dual-core), three (tri-core), four (quad-core), six (hexa-core), eight (octa-core) or ten (deca-core) cores. Some processor even have 22 or 32 cores. Due to power and temperature constraint, the multicore processors are only practical solution for increasing the speed of future computers.

Problems with multicore processors:
According to Amdahl’s law, the performance of parallel computing is limited by its serial components. So, increasing the number of cores may not be the best solution[2]. There is need to increase the clock speed of individual cores.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Single-Core Processor</th>
<th>Multi-Core Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of cores on a die</td>
<td>Single</td>
<td>Multiple</td>
</tr>
<tr>
<td>Instruction Execution</td>
<td>Can execute Single instruction at a time</td>
<td>Can execute multiple instructions by using multiple cores</td>
</tr>
<tr>
<td>Gain</td>
<td>Speed up every program or software being executed</td>
<td>Speed up the programs which are designed for multi-core processors</td>
</tr>
<tr>
<td>Performance</td>
<td>Dependent on the clock frequency of the core</td>
<td>Dependent on the frequency, number of cores and program to be executed</td>
</tr>
<tr>
<td>Examples</td>
<td>Processor launched before 2005 like 80386,486, AMD 29000, AMD K6, Pentium I,II,III etc.</td>
<td>Processor launched after 2005 like Core-2-Duo,Athlon 64 X2, 13,15 and 17 etc.</td>
</tr>
</tbody>
</table>

One architecture uses single core while the other is using two or more cores on the same die for processing instructions. In todays’ time people use multicore processors but single core processors are also very important as far as further speed up is required. It the single-core processors which are put together to make a multi-core processor.
SIMD AND MIMD SYSTEMS

INTRODUCTION

Computer architecture as “the structure of a computer that a machine language programmer must understand to write a correct program for a machine” [1]. Computer architecture can be classified into four main categories. These categories are defined under the Flynn’s Taxonomy. Computer architecture is classified by the number of instructions that are running in parallel and how its data is managed. The four categories that computer architecture can be classified under are:

1. SISD: Single Instruction, Single Data
2. SIMD: Single Instruction, Multiple Data
3. MISD: Multiple Instruction, Single Data
4. MIMD: Multiple Instruction, Multiple Data

SIMD ARCHITECTURE

Single Instruction stream, Multiple Data stream (SIMD) processors one instruction works on several data items simultaneously by using several processing elements, all of which carried out same operation as illustrated in Fig 2. [4]

SIMD system comprise one of the three most commercially successful classes of parallel computers (the other being vector supercomputer and MIMD systems). A number of factors have contributed to this success including:
A. **Basic Principles:**
- There is a two-dimensional array of processing elements, each connected to its four nearest neighbors.
- All processors execute the same instruction simultaneously.
- Each processor incorporates local memory.
- The processors are programmable, that is, they can perform a variety of functions.
- Data can propagate quickly through the array. [1]

B. **Implementing SIMD Architecture:**
Two types of SIMD architectures exist:
1. True SIMD
2. Pipelined SIMD

- **True SIMD architecture:** True SIMD architectures can be determined by its usage of distributed memory and shared memory. Both true SIMD architectures possess similar implementation as seen on Fig.4, but differ on placement of processor and memory modules. [2]

![Diagram of True SIMD architecture with distributed memory](image1)

**True SIMD architecture with distributed memory:**
- A true SIMD architecture with distributed memory possesses a control unit that interacts with every processing element on the architecture.
- Each processor possesses their own local memory as observe on Fig.5.
- The processor elements are used as an arithmetic unit where the instructions are provided by the controlling unit. In order for one processing element to communicate with another memory on the same architecture, such as for information fetching, it will have to acquire it through the controlling unit. This controlling unit handles the transferring of the information from one processing element to another.
- The main drawback is with the performance time where the controlling unit has to handle the data transfer. [2]
True SIMD architecture with Shared Memory:

- In this architecture, a processing element does not have a local memory but instead it’s connected to a network where it can communicate with a memory component. Fig.6 shows all the processing elements connected to the same network which allows them to share their memory content with others.
- In this architecture, the controlling unit is ignored when it comes to processing elements sharing information. It still, however, provides an instruction to the processing elements for computational reasons.
- The disadvantage in this architecture is that if there is a need to expand this architecture, each module (processing elements and memory) has to be added separately and configured.
- However, this architecture is still beneficial since it improved performance time and the information can be transferred more freely without the controlling unit.

Pipelined SIMD Architecture: This architecture implements the logic behind pipelining an instruction as observe on Fig.7. Each processing element will receive an instruction from the controlling unit, using a shared memory, and will perform computation at multiple stages. The controlling unit provides the parallel processing elements with instructions. The sequential processing element is used to handle other instructions. [2]
MIMD ARCHITECTURE

MIMD stands for Multiple Instruction, Multiple Data. The MIMD class of parallel architecture is the most familiar and possibly most basic form of parallel processor. MIMD architecture consists of a collection of N independent, tightly-coupled processors, each with memory that may be common to all processors, and/or local and not directly accessible by the other processors.

**Fig 8:** MIMD Processor [5]

Two types of MIMD architecture:-
1. Shared Memory MIMD architecture
2. Distributed Memory MIMD architecture

- **Shared Memory MIMD architecture:**
  - Create a set of processors and memory modules.
  - Any processor can directly access any memory module via an interconnection network as observe on Fig.9.
  - The set of memory modules defines a global address space which is shared among the processors. [1]

**Fig 9:** Shared Memory MIMD Architecture [1]

<table>
<thead>
<tr>
<th>TABLE 1</th>
<th>Shared Memory MIMD classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Features</td>
<td>NUMA</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Non-Uniform memory access</td>
</tr>
<tr>
<td>Memory uses</td>
<td>Shared memory is divided into blocks and each block is attached with processor</td>
</tr>
</tbody>
</table>
**Distributed Memory MIMD architecture:**

- It replicates the processor/memory pairs and connects them via an interconnection network. The processor/memory pair is called processing element (PE).
- Each processing element (PE) can interact with each other via sending messages.

![Interconnection Network](image)

**Fig 10:** Distributed Memory MIMD architecture [1]

### COMPARATIVE ANALYSIS OF SIMD AND MIMD

<table>
<thead>
<tr>
<th>Features</th>
<th>SIMD</th>
<th>MIMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abbreviation</td>
<td>Single Instruction Multiple Data</td>
<td>Multiple Instruction Multiple Data</td>
</tr>
<tr>
<td>Ease of programming and debugging</td>
<td>Single program, Processing element(PE) operate synchronously</td>
<td>Multiple communication programs, Processing element(PE) operate asynchronously</td>
</tr>
<tr>
<td>Lower program memory requirements</td>
<td>One copy of the program is stored</td>
<td>Each PE stores its own program</td>
</tr>
<tr>
<td>Lower instruction cost</td>
<td>One decoder in control unit</td>
<td>One decoder in each PE</td>
</tr>
<tr>
<td>Complexity of architectures</td>
<td>Simple</td>
<td>Complex</td>
</tr>
<tr>
<td>Cost</td>
<td>Low</td>
<td>Medium</td>
</tr>
<tr>
<td>Size and Performance</td>
<td>Scalability in size and performance</td>
<td>Complex size and good performance</td>
</tr>
<tr>
<td>Conditional Statements</td>
<td>Conditional statements depend upon data local to processors, all of instructions of then block must broadcast, followed by all else block</td>
<td>The multiple instruction stream of MIMD allow for more efficient execution of conditional statements (e.g.: if then else) because each processor can independently follow either decision path</td>
</tr>
<tr>
<td>Low synchronization overheads</td>
<td>Implicit in program</td>
<td>Explicit data structures and operations needed</td>
</tr>
<tr>
<td>Low PE-to-PE communication overheads</td>
<td>Automatic synchronization of all “send” and “receive” operations</td>
<td>Explicit synchronization and identification protocols needed.</td>
</tr>
<tr>
<td>Efficient execution of variable-time instructions</td>
<td>Total execution time equals the sum of maximal execution times through all processors</td>
<td>Total execution time equals the maximum execution time on a given processor</td>
</tr>
</tbody>
</table>

The purpose is to provide an overview of recent architectural approaches of parallel systems and also comparison between them. Describes the Flynn’s Taxonomy: SIMD and MIMD. SIMD allow for more faster and multiple computation in this field where sacrifice cannot be made on the delay of time. SIMD processing architecture example: a graphic processor processing instructions for translation or rotation or other operations are done on multiple data. MIMD processing architecture example is super computer or distributed computing systems with distributed or single shared memory.
INTRODUCTION

Networking strategy was originally employed in the 1950's by the telephone industry as a means of reducing the time required for a call to go through. Similarly, the computer industry employs networking strategy to provide fast communication between computer subparts, particularly with regard to parallel machines.

The performance requirements of many applications, such as weather prediction, signal processing, radar tracking, and image processing, far exceed the capabilities of single-processor architectures. Parallel machines break a single problem down into parallel tasks that are performed concurrently, reducing significantly the application processing time.

Any parallel system that employs more than one processor per application program must be designed to allow its processors to communicate efficiently; otherwise, the advantages of parallel processing may be negated by inefficient communication. This fact emphasizes the importance of interconnection networks to overall parallel system performance. In many proposed or existing parallel processing architectures, an interconnection network is used to realize transportation of data between processors or between processors and memory modules.

This chapter deals with several aspects of the networks used in modern (and theoretical) computers. After classifying various network structures, some of the most well known networks are discussed, along with a list of advantages and disadvantages associated with their use. Some of the elements of network design are also explored to give the reader an understanding of the complexity of such designs.

NETWORK TOPOLOGY

Network topology refers to the layouts of links and switch boxes that establish interconnections. The links are essentially physical wires (or channels); the switch boxes are devices that connect a set of input links to a set of output links. There are two groups of network topologies: static and dynamic. Static networks provide fixed connections between nodes. (A node can be a processing unit, a memory module, an I/O module, or any combination thereof.) With a static network, links between nodes are unchangeable and cannot be easily reconfigured. Dynamic networks provide reconfigurable connections between nodes. The switch box is the basic component of the dynamic network. With a dynamic network the connections between nodes are established by the setting of a set of interconnected switch boxes.

In the following sections, examples of static and dynamic networks are discussed in detail.

Static Networks

There are various types of static networks, all of which are characterized by their node degree; node degree is the number of links (edges) connected to the node. Some well-known static networks are the following:

- Degree 1: shared bus
- Degree 2: linear array, ring
- Degree 3: binary tree, fat tree, shuffle-exchange
- Degree 4: two-dimensional mesh (Illiac, torus)
- Varying degree: n-cube, n-dimensional mesh, k-ary n-cube
A measurement unit, called *diameter*, can be used to compare the relative performance characteristics of different networks. More specifically, the diameter of a network is defined as the largest minimum distance between any pair of nodes. The minimum distance between a pair of nodes is the minimum number of communication links (hops) that data from one of the nodes must traverse in order to reach the other node.

In the following sections, the listed static networks are discussed in detail.

**Shared bus.** The shared bus, also called *common bus*, is the simplest type of static network. The shared bus has a degree of 1. In a shared bus architecture, all the nodes share a common communication link, as shown in Figure 5.1. The shared bus is the least expensive network to implement. Also, nodes (units) can be easily added or deleted from this network. However, it requires a mechanism for handling conflict when several nodes request the bus simultaneously. This mechanism can be achieved through a bus controller, which gives access to the bus either on a first-come, first-served basis or through a priority scheme. (The structure of a bus controller is explained in the Chapter 6.) The shared bus has a diameter of 1 since each node can access the other nodes through the shared bus.

![Figure 5.1 Shared bus.](image)

**Linear array.** The linear array (degree of 2) has each node connected with two neighbors (except the far-ends nodes). The linear quality of this structure comes from the fact that the first and last nodes are not connected, as illustrated in Figure 5.2. Although the linear array has a simple structure, its design can mean long communication delays, especially between far-end nodes. This is because any data entering the network from one end must pass through a number of nodes in order to reach the other end of the network. A linear array, with N nodes, has a diameter of N-1.

![Figure 5.2 Linear array.](image)

**Ring.** Another networking configuration with a simple design is the ring structure. A ring network has a degree of 2. Similar to the linear array, each node is connected to two of its neighbors, but in this case the first and last nodes are also connected to form a ring. Figure 5.3 shows a ring network. A ring can be unidirectional or bidirectional. In a unidirectional ring the data can travel in only one direction, clockwise or counterclockwise. Such a ring has a diameter of N-1, like the linear array. However, a bidirectional ring, in which data travel in both directions, reduces the diameter by a factor of 2, or less if N is even. A bidirectional ring with N nodes has a diameter of \( \lceil N / 2 \rceil \). Although this ring's diameter is much better than that of the linear array, its configuration can still cause long communication delays between distant nodes for large N. A bidirectional ring network’s reliability, as compared to the linear array, is also improved. If a node should fail, effectively cutting off the connection in one direction, the other direction can be used to complete a message transmission. Once the connection is lost between any two adjacent nodes, the ring becomes a linear array, however.

![Figure 5.3 Ring.](image)

**Binary tree.** Figure 5.4 represents the structure of a binary tree with seven nodes. The top node is called
the root, the four nodes at the bottom are called leaf (or terminal) nodes, and the rest of the nodes are called intermediate nodes. In such a network, each intermediate node has two children. The root has node address 1. The addresses of the children of a node are obtained by appending 0 and 1 to the node's address that is, the children of node \( x \) are labeled \( 2x \) and \( 2x+1 \). A binary tree with \( N \) nodes has diameter \( 2(h-1) \), where \( h = \left\lceil \log_2 N \right\rceil \) is the height of the tree. The binary tree has the advantages of being expandable and having a simple implementation. Nonetheless, it can still cause long communication delays between faraway leaf nodes. Leaf nodes farthest away from each other must ultimately pass their message through the root. Since traffic increases as the root is approached, leaf nodes farthest away from each other will spend the most amount of time waiting for a message to traverse the tree from source to destination.

One desirable characteristic for an interconnection network is that data can be routed between the nodes in a simple manner (remember, a node may represent a processor). The binary tree has a simple routing algorithm. Let a packet denote a unit of information that a node needs to send to another node. Each packet has a header that contains routing information, such as source address and destination address. A packet is routed upward toward the root node until it reaches a node that is either the destination or ancestor of the destination node. If the current node is an ancestor of the destination node, the packet is routed downward toward the destination.

![Figure 5.4 Binary tree.](image)

**Fat tree.** One problem with the binary tree is that there can be heavy traffic toward the root node. Consider that the root node acts as the single connection point between the left and right subtrees. As can be observed in Figure 5.4, all messages from nodes \( N_2, N_4, \) and \( N_5 \) to nodes \( N_3, N_6, \) and \( N_7 \) have no choice but to pass through the root. To reduce the effect of such a problem, the fat tree was proposed by Leiserson [LEI 85]. Fat trees are more like real trees in which the branches get thicker near the trunk. Proceeding up from the leaf nodes of a fat tree to the root, the number of communication links increases, and therefore the communication bandwidth increases. The communication bandwidth of an interconnection network is the expected number of requests that can be accepted per unit of time.

The structure of the fat tree is based on a binary tree. Each edge of the binary tree corresponds to two channels of the fat tree. One of the channels is from parent to child, and the other is from child to parent. The number of communication links in each channel increases as we go up the tree from the leaves and is determined by the amount of hardware available. For example, Figure 5.5 represents a fat tree in which the number of communication links in each channel is increased by 1 from one level of the tree to the next. The fat tree can be used to interconnect the processors of a general-purpose parallel machine. Since its communication bandwidth can be scaled independently from the number of processors, it provides great flexibility in design.

![Figure 5.5 Fat tree.](image)
Shuffle-exchange. Another method for establishing networks is the shuffle-exchange connection. The shuffle-exchange network is a combination of two functions: shuffle and exchange. Each is a simple bijection function in which each input is mapped onto one and only one output. Let $s_{n-1}s_{n-2} \ldots s_0$ be the binary representation of a node address; then the shuffle function can be described as

$$\text{shuffle}(s_{n-1}s_{n-2} \ldots s_0) = s_{n-2}s_{n-3} \ldots s_0s_{n-1}.$$ 

For example, using the shuffle function for $N=8$ (i.e. $2^3$ nodes) the following connections can be established between the nodes.

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>→ 000</td>
<td>100</td>
<td>→ 001</td>
</tr>
<tr>
<td>001</td>
<td>→ 010</td>
<td>101</td>
<td>→ 011</td>
</tr>
<tr>
<td>010</td>
<td>→ 100</td>
<td>110</td>
<td>→ 101</td>
</tr>
<tr>
<td>011</td>
<td>→ 110</td>
<td>111</td>
<td>→ 111</td>
</tr>
</tbody>
</table>

The reason that the function is called shuffle is that it reflects the process of shuffling cards. Given that there are eight cards, the shuffle function performs a perfect playing card shuffle as follows. First, the deck is cut in half; between cards 3 and 4. Then the two half decks are merged by selecting cards from each half in an alternative order. Figure 5.6 represents how the cards are shuffled.

Another way to define shuffle connection is through the decimal representation of the addresses of the nodes. Let $N=2^n$ be the number of nodes and $i$ represent the decimal address of a node. For $0 \leq i \leq (N/2)-1$, node $i$ is connected to node $2i$. For $N/2 \leq i \leq N-1$, node $i$ is connected to node $2i+1-N$.

The exchange function is also a simple bijection function. It maps a binary address to another binary address that differs only in the rightmost bit. It can be described as

$$\text{exchange}(s_{n-1}s_{n-2} \ldots s_1s_0) = s_{n-1}s_{n-2} \ldots s_1s_0.$$ 

Figure 5.7 shows the shuffle-exchange connections between nodes when $N = 8$.
The shuffle-exchange network provides suitable interconnection patterns for implementing certain parallel algorithms, such as polynomial evaluation, fast Fourier transform (FFT), sorting, and matrix transposition [STO 71]. For example, polynomial evaluation can be easily implemented on a parallel machine in which the nodes (processors) are connected through a shuffle-exchange network.

In general, a polynomial of degree $N$ can be represented as

$$a_0 + a_1 x + a_2 x^2 + \ldots + a_{N-1} x^{N-1} + a_N x^N,$$

where $a_0, a_1, \ldots, a_N$ are the coefficients and $x$ is a variable. As an example, consider the evaluation of a polynomial of degree 7. One way to evaluate such a polynomial is to use the architecture given in Figure 5.7. In this figure, assume that each node represents a processor having three registers: one to hold the coefficient, one to hold the variable $x$, and the third to hold a bit called the mask bit. Figure 5.8 illustrates the three registers of a node.

Figure 5.8 A shuffle-exchange node's registers.

The evaluation of the polynomial is done in two phases. First, each term $a_i x^i$ is computed at node $i$ for $i = 0$ to 7. Then the terms $a_i x^i$, for $i = 1$ to 7, are added to produce the final result.

Figure 5.9 represents the steps involved in the computation of $a_i x^i$. Figure 5.9a shows the initial values of the registers of each node. The coefficient $a_i$, for $i = 0$ to 7, is stored in node $i$. The value of the variable $x$ is stored in each node. The mask register of node $i$, for $i = 1, 3, 5,$ and $7$, is set to 1; others are set to 0. In each step of computation, every node checks the content of its mask register. When the content of the mask register is 1, the content of the coefficient register is multiplied with the content of the variable register, and the result is stored in the coefficient register. When the content of the mask register is zero, the content of the coefficient register remains unchanged. The content of the variable register is multiplied with itself. The contents of the mask registers are shuffled between the nodes using the shuffle network. Figures 5.9b, c, and d show the values of the registers after the first step, second step, and third step, respectively. At the end of the third step, each registers contains $a_i x^i$. 
At this point, the terms $a_i x^i$ for $i=0$ to 7 are added to produce the final result. To perform such a summation, exchange connections are used in addition to shuffle connections. Figure 5.10 shows all the connections and the initial values of the coefficient registers.
In each step of computation the contents of the coefficient registers are shuffled between the nodes using the shuffle connections. Then copies of the contents of the coefficient registers are exchanged between the nodes using the exchange connections. After the exchange is performed, each node adds the content of its coefficient register to the value that the copy of the current content is exchanged with. After three shuffle and exchanges, the content of each coefficient register will be the desired $\sum_{i=0}^{7} a_i x^i$. The following shows the three steps required to obtain result.

As you can see in the chart, after the third step, the value $\sum_{i=0}^{7} a_i x^i$ is stored in each coefficient register.
From this example, it should be apparent that the shuffle-exchange network provides the desired connections for manipulating the values of certain problems efficiently.

**Two-dimensional mesh.** A two-dimensional mesh consists of \( k_1 \times k_0 \) nodes, where \( k_i \geq 2 \) denotes the number of nodes along dimension \( i \). Figure 5.11 represents a two-dimensional mesh for \( k_0=4 \) and \( k_1=2 \). There are four nodes along dimension 0, and two nodes along dimension 1. As shown in Figure 5.11, in a two-dimensional mesh, each node is connected to its north, south, east, and west neighbors. In general, a node at row \( i \) and column \( j \) is connected to the nodes at locations \((i-1, j)\), \((i+1, j)\), \((i, j-1)\), and \((i, j+1)\). The nodes on the edge of the network have only two or three immediate neighbors.

The diameter of a mesh network is equal to the distance between nodes at opposite corners. Thus, a two-dimensional mesh with \( k_1 \times k_0 \) nodes has a diameter \((k_1-1)+ (k_0-1)\).

![Figure 5.11 A two-dimensional mesh with \( k_0=4 \) and \( k_1=2 \).](image)

In practice, two-dimensional meshes with an equal number of nodes along each dimension are often used for connecting a set of processing nodes. For this reason in most literature the notion of two-dimensional mesh is used without indicating the values for \( k_1 \) and \( k_0 \); rather, the total number of nodes is defined. A two-dimensional mesh with \( k_1=k_0=n \) is usually referred to as a *mesh* with \( N \) nodes, where \( N = n^2 \). For example, Figure 5.12 shows a mesh with 16 nodes. From this point forward, the term *mesh* will indicate a two-dimensional mesh with an equal number of nodes along each dimension.

![Figure 5.12 A two-dimensional mesh with \( k_0=k_1=4 \).](image)

The routing of data through a mesh can be accomplished in a straightforward manner. The following simple routing algorithm routes a packet from source \( S \) to destination \( D \) in a mesh with \( n^2 \) nodes.

1. Compute the row distance \( R \) as \( R = \left\lfloor \frac{D}{n} \right\rfloor - \left\lfloor \frac{S}{n} \right\rfloor \).
2. Compute the column distance \( C \) as \( C = (D \mod n) - (S \mod n) \).
3. Add the values \( R \) and \( C \) to the packet header at the source node.
4. Starting from the source, send the packet for \( R \) rows and then for \( C \) columns.

The values \( R \) and \( C \) determine the number of rows and columns that the packet needs to travel. The direction the message takes at each node is determined by the sign of the values \( R \) and \( C \). When \( R \) (\( C \)) is positive, the packet travels downward (right); otherwise, the packet travels upward (left). Each time that the packet travels from one node to the adjacent node downward, the value \( R \) is decremented by 1, and when it travels upward, \( R \) is incremented by 1. Once \( R \) becomes 0, the packet starts traveling in the horizontal
direction. Each time that the packet travels from one node to the adjacent node in the right direction, the
value $C$ is decremented by 1, and when it travels in the left direction, $C$ is incremented by 1. When $C$
becomes 0, the packet has arrived at the destination. For example, to route a packet from node 6 (i.e., $S=6$)
to node 12 (i.e., $D=12$), the packet goes through two paths, as shown in Figure 5.13. In this example,

$$ R = \left\lfloor \frac{12}{4} \right\rfloor - \left\lfloor \frac{6}{4} \right\rfloor = 2, $$

$$ C = 0 - 2 = -2 $$

It should be noted that in the case just described the nodes on the edge of the mesh network have no
connections to their far neighbors. When there are such connections, the network is called a \textit{wraparound}
two-dimensional mesh, or an \textit{Iliiac} network. An Iliiac network is illustrated in Figure 5.14 for $N=16$.

In general, the connections of an Iliiac network can be defined by the following four functions:

$$ \text{Iliiac}_+ (j) = j + 1 \pmod{N}, $$
$$ \text{Iliiac}_- (j) = j - 1 \pmod{N}, $$
$$ \text{Iliiac}_{+n} (j) = j + n \pmod{N}, $$
$$ \text{Iliiac}_{-n} (j) = j - n \pmod{N}, $$

where $N$ is the number of nodes, $0 \leq j < N$, $n$ is the number of nodes along any dimension, and $N=n^2$.
For example, in Figure 5.14, node 4 is connected to nodes 5, 3, 8, and 0, since

$$ \text{Iliiac}_+ (4) = (4 + 1) \pmod{16} = 5, $$
$$ \text{Iliiac}_- (4) = (4 - 1) \pmod{16} = 3, $$
$$ \text{Iliiac}_{+4} (4) = (4 + 4) \pmod{16} = 8, $$
$$ \text{Iliiac}_{-4} (4) = (4 - 4) \pmod{16} = 0. $$

The diameter of an Iliiac with $N=n^2$ nodes is $n-1$, which is shorter than a mesh. Although the extra
wraparound connections in Iliiac allow the diameter to decrease, they increase the complexity of the design.
Figure 5.15 shows the connectivity of the nodes in a different form. This graph shows that four nodes can be reached from any node in one step, seven nodes in two steps, and four nodes in three steps. In general, the number of steps (recirculations) to route data from a node to any other node is upper bounded by the diameter (i.e., $n - 1$).

![Figure 5.15 Alternative representation of a 16-node Illiac network.](image)

To reduce the diameter of a mesh network, another variation of this network, called torus (or two-dimensional tours), has also been proposed. As shown in Figure 5.16a, a torus is a combination of ring and mesh networks. To make the wire length between the adjacent nodes equal, the torus may be folded as shown in Figure 5.16b. In this way the communication delay between the adjacent nodes becomes equal. Note that both Figures 5.16a and b provide the same connections between the nodes; in fact, Figure 5.16b is derived from Figure 5.16a by switching the position of the rightmost two columns and the bottom two rows of nodes. The diameter of a torus with $N = n^2$ nodes is $\lceil n^2 / 2 \rceil$, which is the distance between the corner and the center node. Note that the diameter is further decreased from the mesh network.

![Figure 5.16 Different types of torus network. (a) A 4-by-4 torus network. (b) A 4-by-4 torus network with folded connection.](image)
The mesh network provides suitable interconnection patterns for problems whose solutions require the computation of a set of values on a grid of points, for which the value at each point is determined based on the values of the neighboring points. Here we consider one of these class of problems: the problem of finding a steady-state temperature over the surface of a square slab of material whose four edges are held at different temperatures. This problem requires the solution of the following partial differential equation, known as Laplace's equation:
\[ \frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} = 0, \]
where $U$ is the temperature at a given point specified by the coordinates $x$ and $y$ on the slab.

The following describes a method, given by Slotnick [SLO 71], to solve this problem. Even if unfamiliar with Laplace's equation, the reader should still be able to follow the description. The method is based on the fact that the temperature at any point on the slab tends to become the average of the temperatures of neighboring points.

Assume that the slab is covered with a mesh and that each square of the mesh has $h$ units on each side. Then the temperature of an interior node at coordinates $x$ and $y$ is the average of the temperatures of the four neighbor nodes. That is, the temperature at node $(x, y)$, denoted as $U(x, y)$, equals the sum of the four neighboring temperatures divided by 4. For example, as shown in Figure 5.17, assume that the slab can be covered with a 16-node mesh. Here the value of $U(x, y)$ is expressed as
\[ U(x, y) = \frac{U(x, y+h) + U(x+h, y) + U(x, y-h) + U(x-h, y)}{4}. \]

Figure 5.17 Covering a slab with a 16-node mesh.

Figure 5.18 illustrates an alternative representation of Figure 5.17. Here the position of the nodes is more conveniently indicated by the integers $i$ and $j$. In this case, the temperature equation can be expressed as
\[ U(i,j) = \frac{U(i,j+1) + U(i+1,j) + U(i,j-1) + U(i-1,j)}{4}. \]
Assume that each node represents a processor having one register to hold the node's temperature. The nodes on the boundary are arbitrarily held at certain fixed temperatures. Let the nodes on the bottom of the mesh and on the right edge be held at zero degrees. The nodes along the top and left edges are set according to their positions. The temperatures of these 12 boundary nodes do not change during the computation. The temperatures at the 4 interior nodes are the unknowns. Initially, the temperatures at these 4 nodes are set to zero. In the first iteration of computation, the 4 interior node processors simultaneously calculate the new temperature values using the values initially given.
Figure 5.18 Initial values of the nodes.

Figure 5.19 represents the new values of the interior nodes after the first iteration. These values are calculated as follows:

- $U(1,2) = \frac{U(1,3) + U(2,2) + U(1,1) + U(0,2)}{4} = \frac{8 + 0 + 0 + 8}{4} = 4$;
- $U(2,2) = \frac{U(2,3) + U(3,2) + U(2,1) + U(1,2)}{4} = \frac{4 + 0 + 0 + 0}{4} = 1$;
- $U(1,1) = \frac{U(1,2) + U(2,1) + U(1,0) + U(0,1)}{4} = \frac{0 + 0 + 0 + 4}{4} = 1$;
- $U(2,1) = \frac{U(2,2) + U(3,1) + U(2,0) + U(1,1)}{4} = \frac{0 + 0 + 0 + 4}{4} = 0$.

In the second iteration, the values of $U(1,2)$, $U(2,2)$, $U(1,1)$, and $U(2,1)$ are calculated using the new values just obtained:

- $U(1,2) = \frac{8 + 1 + 1 + 8}{4} = 4.5$;
- $U(2,2) = \frac{4 + 0 + 0 + 4}{4} = 2$;
- $U(1,1) = \frac{4 + 0 + 0 + 4}{4} = 2$;
- $U(2,1) = \frac{1 + 0 + 0 + 1}{4} = 0.5$.

This process continues until a steady-state solution is obtained. As more iterations are performed, the values of the interior nodes converge to the exact solution. When values for two successive iterations are close to each other (within a specified error tolerance), the process can be stopped, and it can be said that a steady-state solution has been reached. Figure 5.20 represents a solution obtained after 11 iterations.
**n-cube or hypercube.** An n-cube network, also called hypercube, consists of $N=2^n$ nodes; $n$ is called the *dimension* of the n-cube network. When the node addresses are considered as the corners of an $n$-dimensional cube, the network connects each node to its $n$ neighbors. In an $n$-cube, individual nodes are uniquely identified by $n$-bit addresses ranging from 0 to $N-1$. Given a node with binary address $d$, this node is connected to all nodes whose binary addresses differ from $d$ in exactly 1 bit. For example, in a 3-cube, in which there are eight nodes, node 7 (111) is connected to nodes 6 (110), 5 (101), and 3 (011). Figure 5.21 demonstrates all the connections between the nodes.

![Figure 5.21 A three-dimensional cube.](image)

As can be seen in the 3-cube, two nodes are directly connected if their binary addresses differ by 1 bit. This method of connection is used to control the routing of data through the network in a simple manner. The following simple routing algorithm routes a packet from its source $S=(s_{n-1}, \ldots, s_0)$ to destination $D=(d_{n-1}, \ldots, d_0)$.

1. Tag $T=S \oplus D = t_{n-1}, \ldots, t_0$ is added to the packet header at the source node ($\oplus$ denotes an XOR gate).
2. If $t_i \neq 0$ for some $0 \leq i \leq n-1$, then use $i$th-dimension link to send the packet to a new node with the same address as the current node except the $i$th bit, and change $t_i$ to 0 in the packet header.
3. Repeat step 2 until $t_i = 0$ for all $0 \leq i \leq n-1$.

For example, as shown in Figure 5.22, to route a packet from node 0 to node 5, the packet could go through two different paths, $P_1$ and $P_2$. Here $T=000 \oplus 101 = 101$. If we first consider the bit $t_0$ and then $t_2$, the packet goes through the path $P_1$. Since $t_0 = 1$, the packet is sent through the 0th-dimension link to node 1.
At node 1, \( t_0 \) is set to 0; thus \( T \) now becomes equal to 100. Now, since \( t_2 = 1 \), the packet is sent through the second-dimension link to node 5. If, instead of \( t_0 \), bit \( t_2 \) is considered first, the packet goes through \( P_2 \).

![Figure 5.22 Different paths for routing a packet from node 0 to node 5.](image)

In the network of Figure 5.22, the maximum distance between nodes is 3. This is because the distance between nodes is equal to the number of bit positions in which their binary addresses differ. Since each address consists of 3 bits, the difference between two addresses can be at most 3 when every bit at the same position differs. In general, in an \( n \)-cube the maximum distance between nodes is \( n \), making the diameter equal to \( n \).

The \( n \)-cube network has several features that make it very attractive for parallel computation. It appears the same from every node, and no node needs special treatment. It also provides \( n \) disjoint paths between a source and a destination. Let the source be represented as \( S = (s_{n-1} s_{n-2} \ldots s_0) \) and the destination by \( D = (d_{n-1} d_{n-2} \ldots d_0) \). The shortest paths can be symbolically represented as

Path 1: \( s_{n-1} s_{n-2} \ldots s_0 \rightarrow s_{n-1} s_{n-2} \ldots d_0 \rightarrow s_{n-1} s_{n-2} \ldots d_1 d_0 \rightarrow d_{n-1} d_{n-2} \ldots d_0 \)
Path 2: \( s_{n-1} s_{n-2} \ldots s_0 \rightarrow s_{n-1} s_{n-2} \ldots d_1 s_0 \rightarrow s_{n-1} s_{n-2} \ldots d_2 d_0 \rightarrow d_{n-1} d_{n-2} \ldots d_1 d_0 \)
Path \( n \): \( s_{n-1} s_{n-2} \ldots s_0 \rightarrow d_{n-1} s_{n-2} \ldots s_0 \rightarrow d_{n-1} s_{n-2} \ldots s_1 d_0 \rightarrow d_{n-1} s_{n-2} \ldots d_0 \rightarrow d_{n-1} d_{n-2} \ldots d_1 d_0 \)

For example, consider the 3-cube of Figure 5.21. Since \( n = 3 \), there are three paths from a source, say 000, to a destination, say 111. The paths are

path 1: 000 → 001 → 011 → 111;
path 2: 000 → 010 → 110 → 111;
path 3: 000 → 100 → 101 → 111.

This ability to have \( n \) alternative paths between any two nodes makes the \( n \)-cube network highly reliable if any one (or more) paths become unusable.

Different networks, such as two-dimensional meshes and trees, can be embedded in an \( n \)-cube in such a way that the connectivity between neighboring nodes remains consistent with their definition. Figure 5.23 shows how a 4-by-4 mesh can be embedded in a 4-cube (four-dimensional hypercube). The 4-cube’s integrity is not compromised and is well-suited for uses like this, where a great deal of flexibility is required. All definitional considerations for both the 4-cube and the 4-by-4 mesh, as stated earlier, are consistent.
The interconnection supported by the $n$-cube provides a natural environment for implementing highly parallel algorithms, such as sorting, merging, fast Fourier transform (FFT), and matrix operations. For example, Batcher's bitonic merge algorithm can easily be implemented on an $n$-cube. This algorithm sorts a bitonic sequence (a bitonic sequence is a sequence of nondecreasing numbers followed by a sequence of nonincreasing numbers). Figure 5.24 presents the steps involved in merging a nondecreasing sequence \([0,4,6,9]\) and a nonincreasing sequence \([8,5,3,1]\). This algorithm performs a sequence of comparisons on pairs of data that are successively $2^2$, $2^1$, and $2^0$ locations apart.

At each stage of the merge each pair of data elements is compared and switched if they are not in ascending order. This rearranging continues until the final merge with a distance of 1 puts the data into ascending order.

Figure 5.24 requires the following connections between nodes:
- Node 0 should be connected to nodes: 1, 2, 4;
- Node 1 should be connected to nodes: 0, 3, 5;
Node 2 should be connected to nodes: 0, 3, 6;  
Node 3 should be connected to nodes: 1, 2, 7;  
Node 4 should be connected to nodes: 0, 5, 6;  
Node 5 should be connected to nodes: 1, 4, 7;  
Node 6 should be connected to nodes: 2, 4, 7;  
Node 7 should be connected to nodes: 3, 5, 6.  
These are exactly the same as 3-cube connections. That is, the $n$-cube provides the necessary connections for the Batcher's algorithm. Thus, applying Batcher's algorithm to an $n$-cube network is straightforward.

In general, the $n$-cube provides the necessary connections for ascending and descending classes of parallel algorithms. To define each of these classes, assume that there are $2^n$ input data items stored in $2^n$ locations (or processors) $0, 1, 2, ..., 2^n-1$. An algorithm is said to be in the descending class if it performs a sequence of basic operations on pairs of data that are successively $2^{n-1}$, $2^{n-2}$, ..., and $2^0$ locations apart. (Therefore, Batcher's algorithm belongs to this class.) In comparison, an ascending algorithm performs successively on pairs that are $2^0$, $2^1$, ..., and $2^{n-1}$ locations apart. When $n=3$ Figures 5.25 and 5.26 show the required connections for each stage of operation in this class of algorithms. As shown, the $n$-cube is able to efficiently implement algorithms in descending or ascending classes.

![Figure 5.25 Descending class.](image)

![Figure 5.26 Ascending class.](image)

Although the $n$-cube can implement this class of algorithms in $n$ parallel steps, it requires $n$ connections for each node, which makes the design and expansion difficult. In other words, the $n$-cube provides poor scalability and has an inefficient structure for packaging and therefore does not facilitate the increasingly important property of modular design.

**$n$-Dimensional mesh.** An $n$-dimensional mesh consists of $k_0, k_2, \ldots, k_0$ nodes, where $k_i \geq 2$ denotes the number of nodes along dimension $i$. Each node $X$ is identified by $n$ coordinates $x_0, x_1, x_2, \ldots, x_n$, where $0 \leq x_i \leq k_i - 1$ for $0 \leq i \leq n-1$. Two nodes $X=(x_0, x_1, x_2, \ldots, x_n)$ and $Y=(y_0, y_1, y_2, \ldots, y_n)$ are said to be neighbors if and only if $y_i = x_i$ for all $i$, $0 \leq i \leq n-1$, except one, $j$, where $y_j = x_j + 1$ or $y_j = x_j - 1$. That is, a node may have from $n$ to $2n$ neighbors, depending on its location in the mesh. The corners of the mesh have $n$ neighbors, and the internal nodes have $2n$ neighbors, while other nodes have $n_b$ neighbors, where $n_b < 2n$. The diameter of an $n$-dimensional mesh is $\sum_{i=0}^{n-1}(k_i - 1)$. An $n$-cube is a special case of $n$-dimensional meshes; it is in fact an $n$-dimensional mesh in which $k_i=2$ for $0 \leq i \leq n-1$. Figure 5.27 represents the structure of two three-
dimensional meshes: one for \(k_2 = k_1 = k_0 = 3\) and the other for \(k_2=4, k_1=3, \) and \(k_0=2\).

(a) \(k_2 = k_1 = k_0 = 3\).

(b) \(k_2=4, k_1=3, \) and \(k_0=2\).

**Figure 5.27** Three-dimensional meshes.

\textbf{k-Ary n-cube.} A k-ary n-cube consists of \(k^n\) nodes such that there are \(k\) nodes along each dimension. Each node \(X\) is identified by \(n\) coordinates, \(x_{n-1},x_{n-2},\ldots,x_0\), where \(0 \leq x_i \leq k-1\) for \(0 \leq i \leq n-1\). Two nodes \(X=(x_{n-1},x_{n-2},\ldots,x_0)\) and \(Y=(y_{n-1},y_{n-2},\ldots,y_0)\) are said to be neighbors if and only if \(y_i=x_i\) for all \(i, 0 \leq i \leq n-1, \) except one, \(j,\) where \(y_j=(x_j + 1) \mod k,\) or \(y_j=(x_j - 1) \mod k.\) That is, in contrast to the \(n\)-dimensional mesh, a \(k\)-ary \(n\)-cube has a symmetrical topology in which each node has an equal number of neighbors. A node has \(n\) neighbors when \(k=2\) and \(2n\) neighbors when \(k>2.\) The \(k\)-ary \(n\)-cube has a diameter of \(\lceil n\lceil k/2 \rceil \rceil.\) An \(n\)-cube is a special case of \(k\)-ary \(n\)-cubes; it is in fact a 2-ary \(n\)-cube. Figure 5.28 represents the structure of two \(k\)-ary \(n\)-cubes: one for \(k=4, n=2\) and the other for \(k=n=3.\) Note that a 4-ary 2-cube is actually a torus network.

\textbf{Figure 5.28} (a) 4-Ary 2-cube and (b) 3-ary 3-cube networks.
Parallel and distributed processing did not lose their allure since their inception in 1960’s; the allure is in terms of their ability to meet a wide range of price and performance. However, in many cases these advantages were not realized due to longer design times, limited scalability, lack of OS and programming support, and the ever increasing performance/cost ratio of uniprocessors [Bell 99]. Historically, parallel processing systems were classified as SIMD (Single Instruction Multiple Data) or MIMD (Multiple Instructions Multiple Data). SIMD systems involved the use of a single control processor and a number of arithmetic processors. The array of arithmetic units executed identical instruction streams, but on different data items, in a lock-step fashion under the control of the control unit. Such systems were deemed to be ideal for data-parallel applications. Their appeal waned quickly since very few applications could garner the performance to justify their cost. Multicomputers or multiprocessors fall under the MIMD classification, relying on independent processors. They were able to address a broader range of applications than SIMD systems. One of the earliest MIMD system was the C.mmp built at CMU that included 16 modified PDP-11/20 processors connected to 16 memory modules via a crossbar switch. This can be viewed as a Symmetric Multiprocessor (SMP) or a shared memory system. The next version of a multiprocessor system at CMU was known as Cm* and can be deemed as the first hardware implemented distributed shared memory system. It consisted of an hierarchy of processing nodes; LSI-11 processors comprising clusters where processors of a single cluster were connected by a bus and clusters were connected by inter-cluster connections using specialized controllers to handle accesses to remote memory.

The next wave of multiprocessors relied on distributed memory, where processing nodes have access only to their local memory, and access to remote data was accomplished by request and reply messages. Numerous designs on how to interconnect the processing nodes and memory modules were published in the literature. Examples of such message-based systems included Intel Paragon, N-Cube, IBM' SP systems. As compared to shared memory systems, distributed memory (or message passing) systems can accommodate larger number of computing nodes. This scalability was expected to increase the utilization of message-passing architectures.

The Changing Nature Of Parallel Processing

Although parallel processing systems, particularly those based on message-passing (or distributed memory) model, have been researched for decades leading to the implementation of several large scale computing systems and specialized
supercomputers, their use has been limited for very specialized applications. One of the major reason for this is that most programmers find message passing very hard to do - especially when they want to maintain a sequential version of program (during development and debugging stages) as well as the message passing version. Programmers often have to approach the two versions completely independently. They in general feel more comfortable in viewing the data in a common global memory, hence programming on a shared memory multiprocessor system (or SMP) is considered easier. In a shared memory paradigm, all processes (or threads of computation) share the same logical address space and access directly any part of the data structure in a parallel computation. A single address space enhances the programmability of a parallel machine by reducing the problems of data partitioning, migration and load balancing. The shared memory also improves the ability of parallelizing compilers, standard operating systems, resource management and incremental performance tuning of applications.

The trend even in commercial parallel processing applications has been leaning towards the use of small clusters of SMP systems, often interconnected to address the needs of complex problems requiring the use of large number of processing nodes. Even when working with a networked resources, programmers are relying on messaging standards such as MPI (and PVM) or relying on systems software to automatically generate message passing code from user defined shared memory programs. The reliance on software support to provide a shared memory programming model (i.e., distributed shared memory systems, DSMs) can be viewed as a logical evolution in parallel processing. Distributed Shared Memory (DSM) systems aim to unify parallel processing systems that rely on message passing with the shared memory systems. The use of distributed memory systems as (logically) shared memory systems addresses the major limitation of SMP’s; namely scalability.

The growing interest in multithreading programming and the availability of systems supporting multithreading (Pthreads, NT-Threads, Linux Threads, Java) further emphasizes the trend towards shared memory programming model. Finally, the formation of OpenMP group and the specification of OpenMP Fortran in October 1997, can only be viewed as a trend that drives the final nail into the coffin of message passing paradigm. The new standard is designed for computers that use either the Unix or Windows NT operating systems and employ multiple microprocessors for parallel computing. OpenMP Fortran is designed for the development of portable parallel programs on shared memory parallel computer systems. One effect of the OpenMP standard will be to increase the shift of complex scientific and engineering software development from the supercomputer world to high-end desktop work stations.

**Programming Example**

In order to appreciate the differences between the shared memory and message passing paradigms, consider the following code segments to compute inner products. The first program was written using Pthreads (on shared memory), while the second using MPI (for message passing systems). Both are assume a master process and multiple worker processes, and each worker process is allocated equal amounts of work by the master.
There are two major differences in the two implementation, related to how the work is distributed and how the worker processes access the needed data. The Pthread version shows that each worker process is given the address of the data they need for their work. In the MPI version, the actual data is sent to the worker processors. The worker processes of the Pthread version access the needed data directly as if the data is local. It can also be seen that the worker processors directly accumulate their partial results in a single global variable (using mutual exclusion). The worker processes of the MPI program are supplied the actual data via messages, and they send their partial results back to the master for the purpose of accumulation.

In addition to the aforementioned differences, there is another important difference. In the Pthread implementation, the user is not concerned with the data distribution, while in the MPI version, the programmer must explicitly specify where the data should be sent.

**Code for shared memory using Pthreads.**

```c
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
double sum = 0.0;

typedef struct {
    /* data structure used for work distribution */
    int n;
    /* number of elements */
    double *x;
    /* address of the first element of the 1st array */
    double *y;
    /* address of the first element of the 2nd array */
} arg_t;

static void *worker(arg_t *arg) /* worker process begins here */
{
    int i;
    int n = arg->n;
    double *x = arg->x; /* takes data location */
    double *y = arg->y; /* takes data location */
    double partial = 0.0;

    for (i = 0; i < n; ++i) /* calculation */
        partial += x[i]*y[i];

    pthread_mutex_lock(&lock); /* lock */
    sum += partial; /* accumulate the partial sums */
    pthread_mutex_unlock(&lock); /* unlock */

    return NULL;
}

main(int argc, char *argv[])
{
    double x[N_VEC], y[N_VEC]; /* input vectors */
```
/* generates two vectors */

generate(x, y);

for (i = 0; i < N_PROC; i++) {
    arg[i].n = SZ_WORK;
    arg[i].x = x + i*SZ_WORK;
    arg[i].y = y + i*SZ_WORK;
    status = pthread_create(&thread[i], NULL, (void*) worker, &arg[i]);
    if (status) exit(1);
}

for (i = 0; i < N_PROC; i++) {
    status = pthread_join(thread[i], NULL);
    if (status) exit(1);
}

printf("Inner product is %fn", sum);

# define MASTER       0
# define FROM_MASTER  1
# define TO_MASTER    2

double sum = 0.0;

static double worker(int n, double *x, double *y)
{
    double partial = 0.0;
    int i;

    for (i = 0; i < n; ++i)
        partial += x[i]*y[i];

    return partial;
}

main(int argc, char *argv[])
{
    double x[N_VEC], y[N_VEC];          /* input vectors */
    double x_partial[SZ_WORK], y_partial[SZ_WORK]; /* a chunk of each vector */
    double partial;
int i, self, mtype, offset;
MPI_Status status;

MPI_Init(&argc, &argv); /* initialize MPI */
MPI_Comm_rank(MPI_COMM_WORLD, &self);

if (self == MASTER) { /* master process */
    generate(x, y); /* generates two vectors */

    mtype = FROM_MASTER;
    offset = 0;
    for (i = 1; i < N_PROC; i++) { /* send messages to workers */
        MPI_Send(&x[offset], SZ_WORK, MPI_DOUBLE, i, mtype,
            MPI_COMM_WORLD);
        MPI_Send(&y[offset], SZ_WORK, MPI_DOUBLE, i, mtype,
            MPI_COMM_WORLD);
        offset += SZ_WORK;
    }
    mtype = TO_MASTER;
    for (i = 1; i < N_PROC; i++) { /* receive messages from workers */
        MPI_Recv(&partial, 1, MPI_DOUBLE, i, mtype,
            MPI_COMM_WORLD, &status);
        sum += partial;
    }
    printf("Inner product is %f\n", sum);
}
else { /* worker process */
    mtype = FROM_MASTER; /* receive a message from master */
    MPI_Recv(x_partial, SZ_WORK, MPI_DOUBLE, MASTER, mtype,
        MPI_COMM_WORLD, &status);
    MPI_Recv(y_partial, SZ_WORK, MPI_DOUBLE, MASTER, mtype,
        MPI_COMM_WORLD, &status);
    partial = worker(SZ_WORK, x_partial, y_partial);
    mtype = TO_MASTER;
    MPI_Send(&partial, 1, MPI_DOUBLE, MASTER, mtype,
        MPI_COMM_WORLD);
    /* send result back to master */
}

MPI_Finalize();

Distributed Shared Memory Systems.

As mentioned previously, distributed shared memory systems attempt to unify the message passing and shared memory programming models. Since DSM’s span both
physically shared and physically distributed memory systems, DSM’s are also concerned with the interconnection network that provide the data to the requesting processor in an efficient and timely fashion. Both the bandwidth (amount of data that can be supplied in a unit time) and latency (the time it takes to receive the first piece of requested data from the time the request is issued) are important to the design of DSM’s. Precisely because of the generally longer latencies encountered in large scale DSM’s, multithreading has received considerable attention; multithreading model can be utilized to tolerate (or mask) memory latencies. In this paper we will not address issues related to interconnection networks or latency tolerance techniques.

Given that the overall objective of DSM’s is to provide cost-effective mechanisms to manage the extended memory space across multiple levels of hierarchy, the design space is huge. In this tutorial will address some of the more important classes in the design space. In this section we will overview the major issues that face the design of DSM’s. To start, the management of large logical memory space involves moving data dynamically across the memory layers of a distributed system. This includes the mapping of the user data to the various memory modules. The data may be uniquely mapped to a physical address (as done in cache coherent systems) or replicating the data to several physical addresses (as done in reflective memory systems and, to some extent, in cache-only systems). Even in uniquely mapped systems, data may be replicated in lower levels of memories (i.e., caches). Replication, in turn requires means for maintaining consistency. Directories are often the key to tracking the multiple copies and play a key role in coherency of replicated data. The coherency can be maintained by hardware or software.

The granularity of the data that is shared and moved across memory hierarchies is another design consideration. The granularity can be based on objects without semantic meaning, based purely on a sequence of bytes (e.g., a memory word, a cache block, a page) or it can be based on objects with semantic basis (e.g., variables, data structures or objects in the sense of object-oriented programming model). Hardware solutions often use finer grained objects (often without semantic meaning) while software implementations rely on coarser grained objects.

The design trade-offs in managing shared memory, be it in hardware or software, depend on the freedom (or limitations) provided by the semantics of the programming model. This leads to issues such as the order in which memory accesses performed by individual processors to individual memory modules and their interactions with memory accesses performed by other processors. The most restrictive programming semantics (known as Sequential Consistency) requires that (1) the memory accesses of each individual processor be executed in the order defined by the sequence defined by the program executed on the processor and (2) memory accesses of different processors be executed in some interleaved fashion. In order to utilize modern processors as the building blocks in a DSM, sequential consistency model is often not guaranteed by the underlying system, placing the onus on the programmer in maintaining the necessary order of memory accesses. There has been a considerable research into the various memory consistency
semantics and the trade-offs they present in terms of performance and the programming complexity.

Synchronization and coordination among concurrent computations (i.e., processes or threads) in shared memory programming relies on the use of mutual exclusion, critical sections and barriers. Implementation of the mechanisms needed for mutual exclusion (often based on the use of locks) and barriers explore the design space spanning hardware instructions (such as test&set, Load-Linked and Store-Conditional instructions, fetch&operate, combining networks), spin locks (including the use of shadow locks), Queue or Array locks, sense-reversing barriers and tree-barriers. These solutions explore trade-offs between the performance (in terms of latency, serialization, network traffic) and scalability.

Related to the above set of issues, resource management, particularly in terms of minimizing data migration, thread migration, messages exchanged are all significant in achieving commercial cost-performance ratios in making DSM’s as contenders. In section II, we will detail issues related to data-coherency, memory consistency models, and the implementation of locks and barriers. In section III, we will case-study 3 hardware DSM implementations, representing cache-coherent architectures, cache-only architectures and reflective memories. In section IV, we will describe software DSM implementations covering page-based, variable-based and object-based data models. In section V, we will describe how DSM’s can be implemented using both hardware and software techniques. In section VI, we conclude the survey with a set of open issues, and a prognosis on the DSM’s commercial viability, and alternatives to DSMs.

**Issues in Shared Memory Systems**

In this section we will concentrate on 3 main issues in the design of hardware or software based distributed shared memory system. They are related to data coherence, memory consistency and synchronization.

**Data Coherency.**

The use of cache memories is so pervasive in today’s computer systems it is difficult to imagine processors without them. Cache memories, along with virtual memories and processor registers form a continuum of memory hierarchies that rely on the principle of locality of reference. Most applications exhibit temporal and spatial localities among instructions and data. Spatial locality implies that memory locations that are spatially (address-wise) near the currently referenced address will likely be referenced. Temporal locality implies that the currently referenced address will likely be referenced in the near future (time-wise). Memory hierarchies are designed to keep most likely referenced items in the fastest devices. This results in an effective reduction in access time.

Cache Coherency.
The inconsistency that exists between main memory and write-back caches does not cause any problems in uniprocessor systems. But techniques are needed to ensure that consistent data is available to all processors in a multiprocessor system. Cache coherency can be maintained either by hardware techniques or software techniques. We will first introduce hardware solutions.

Snoopy Protocols.

These protocols are applicable for small scale multiprocessors systems where the processors are connected to memory via a common bus, making the shared memory equally accessible to all processors (also known as Symmetric Multiprocessor systems SMP, Uniform Memory Access systems, UMA). In addition to the shared memory, each processor contains a local cache memory (or multi-level caches). Since all processors and their cache memories (or the controller hardware) are connected to a common bus, the cache memories can snoop on the bus for maintaining coherent data. Each cache line is associated with a state, and the cache controller will modify the states to track changes to cache lines made either locally or remotely. A hit on a read implies that the cache data is consistent with that in main memory and copies that may exist in other processors' caches. A read miss leads to a request for the data. This request can be satisfied by either the main memory (if no other cache has a copy of the data), or by another cache which has a (possibly newer) copy of the data. Initially, when only one cache has a copy, the cache line is set to Exclusive state. However, when other caches request for a read copy, the state of the cache line (in all processors) is set to Shared.

Consider what happens when a processor attempts to write to a (local) cache line. On a hit, if the state of the local cache line is Exclusive (or Modified), the write can proceed without any delay, and state is changed to Modified. This is because, Exclusive or Modified state with the data guarantees that no copies of the data exist in other caches. If the local state is Shared (which implies the existence of copies of the data item in other processors) then an invalidation signal must be broadcast on the common bus, so that all other caches will set their cache lines to Invalid state. Following the invalidation, the write can be completed in local cache, changing the state to Modified.

On a write-miss request is placed on the common bus. If no other cache contains a copy, the data comes from memory, the write can be completed by the processor and the cache line is set to Modified. If other caches have the requested data in Shared state, the copies are invalidated, the write can complete with a single Modified copy. If a different processor has a Modified copy, the data is written back to main memory and the processor invalidates its copy. The write can now be completed, leading to a Modified line at the requesting processor. Such snoopy protocols are sometimes called MESI, standing for the names of states associated with cache lines: Modified, Exclusive, Shared or Invalid. Many variations of the MESI protocol have been reported [Baer]. In general the performance of a cache coherency protocol depends on the amount of sharing (i.e., number of shared cache blocks), number of copies, number of writers and granularity of sharing.
Instead of invalidating shared copies on a write, it may be possible to provide updated copies. It may be possible with appropriate hardware to detect when a cache line is no longer shared by other processors, eliminating update messages. The major trade-off between update and invalidate technique lies in the amount of bus traffic resulting from the update messages that include data as compared to the cache misses subsequent to invalidation messages. Update protocols are better suited for applications with a single writer and multiple readers, while invalidation protocols are favored when multiple writers exist.

Directory Protocols

Snoopy protocols rely on the ability to listen to and broadcast invalidations on a common bus. However, the common bus places a limit on the number of processing nodes in an SMP system. Large scale multiprocessor and distributed systems must use more complex interconnection mechanisms, such as multiple buses, N-dimensional grids, Crossbar switches and multistage interconnection networks. New techniques are needed to assure that invalidation messages are received (and acknowledged) by all caches with copies of the shared data. This is normally achieved by keeping a directory with main memory units. There exists one directory entry corresponding to each cache block, and the entry keeps track of shared copies, or the identification of the processor that contains modified data. On a read miss, a processor requests the memory unit for data. The request may go to a remote memory unit depending on the address. If the data is not modified, a copy is sent to the requesting cache, and the directory entry is modified to reflect the existence of a shared copy. If a modified copy exists at another cache, the new data is written back to the memory, a copy of the data is provided to the requesting cache, and the directory is marked to reflect the existence of two shared copies.

In order to handle writes, it is necessary to maintain state information with each cache block at local caches, somewhat similar to the Snoopy protocols. On a write hit, the write can proceed immediately if the state of the cache line is Modified. Otherwise (the state is Shared), Invalidation message is communicated to the memory unit, which in turn sends invalidation signals to all caches with shared copies (and receive acknowledgements). Only after the completion of this process can the processor proceed with a write. The directory is marked to reflect the existence of a modified copy. A write miss is handled as a combination of read-miss and write-hit.

Notice that in the approach outlined here, the directory associated with each memory unit is responsible for tracking the shared copies and for sending invalidation signals. This is sometimes known as p+1 directory to reflect the fact that each directory entry may need p+1 bits to track the existence of up to p read copies and one write copy. The memory requirements imposed by such directory methods can be alleviated by allowing fewer copies (less than p), and the copies are tracked using "pointers" to the processors containing copies. Whether using p+1 bits or fixed number of pointers, copies of data items is maintained by (centralized) directories associated with memory modules. We can consider distributing each directory entry as follows. On the first request for data, the memory unit (or the home memory unit) supplies the requested data, and marks the
directory with a "pointer" to the requesting processor. Future read requests will be forwarded to the processor which has the copy, and the requesting processors are linked together. In other words, the processors with copies of the data are thus linked, and track all shared copies. On a write request, an invalidation signal is sent along the linked list to all shared copies. The home memory unit can wait until invalidations are acknowledged before permitting the writer to proceed. The home memory unit can also send the identification of the writer so that acknowledgements to invalidations can be sent directly to the writer. Scalable Coherence Interface (SCI) standard uses a doubly linked list of shared copies. This permits a processor to remove itself from the linked list when it no longer contains a copy of the shared cache line.

Numerous variations have been proposed and implemented to improve the performance of the directory based protocols. Hybrid techniques that combine Snoopy protocols with Directory based protocols have also been investigated in Stanford DASH system. Such systems can be viewed as networks of clusters, where each cluster relies on bus snooping and use directories across clusters (see Cluster Computing).

The performance of directory based techniques depend on the number of shared blocks, number of copies of individual shared blocks, if multicasting is available, number of writers. The amount of memory needed for directories depend on the granularity of sharing, the number of processors (in p+1 directory), number of shared copies (in pointer based methods).

Software Based Coherency Techniques.

Using large cache blocks can reduce certain types of overheads in maintaining coherence as well as reduce the overall cache miss rates. However, larger cache blocks will increase the possibility of false-sharing. False sharing refers to the situation when 2 or more processors which do not really share any specific memory address, however they appear to share a cache line, since the variables (or addresses) accessed by the different processors fall to the same cache line. Compile time analysis can detect and eliminate unnecessary invalidations in some false sharing cases.

Software can also help in improving the performance of hardware based coherency techniques described above. It is possible to detect when a processor no longer accesses a cache line (or variable), and “self-invalidation” can be used to eliminate unnecessary invalidation signals. In the simplest method (known as Indiscriminate Invalidation), consider an indexed variable X being modified inside a loop. If we do not know how the loop iterations will be allocated to processors, we may require each processor to read the variable X at the start of each iteration and flush the variable back to memory at the end of the loop iteration (that is invalidated). However, this is unnecessary since not all values of X are accessed in each iteration, and it is also possible that several, contiguous iterations may be assigned to the same processor.

In Selective invalidation technique, if static analysis reveals that a specific variable may be modified in a loop iteration, the variable will be marked with a "Change Bit". This
implies that at the end of the loop iteration, the variable may have been modified by some processor. The processor which actually modifies the variable resets the Change Bit since the processor already has an updated copy. All other processors will invalidate their copies of the variable.

A more complex technique involves the use of Version numbers with individual variables. Each variable is associated with a Current Version Number (CVN). If static analysis determines that a variable may be modified (by any processor), all processors will be required to increment the CVN associated with the variable. In addition, when a processor acquires a new copy of a variable, the variable will be associated with a Birth Version Number (BVN) which is set to the current CVN. When a processor actually modifies a variable, the processor will set the BVN to the CVN+1. If the BVN of a variable in a processor is greater than the CVN of that variable, the processor has the updated value; otherwise, the processor invalidates its copy of the variable.

Migration of processes or threads from one node to another can lead to poor cache performances since the migration can cause “false” sharing: the original node where the thread resided may falsely assume that cache lines are shared with the new node to where the thread migrated. Some software techniques to selectively invalidate cache lines when threads migrate have been proposed.

Software aided prefetching of cache lines is often used to reduce cache misses. In shared memory systems, prefetching may actually increase misses, unless it is possible to predict if a prefetched cache line will be invalidated before its use.

**Memory Consistency**

While data coherency (or cache coherency) techniques aim at assuring that copies individual data items (or cache blocks) will be up to date (or copies will be invalid), a consistent memory implies that view of the entire shared memory presented to all processors will be identical. This requirement can also be stated in terms of the order in which operations performed on shared memory will be made visible to individual processors in a multiprocessor system. In order to understand the relationship between the ordering of memory operations and memory consistency, consider the following example with two processors P1 and P2. P1 performs a write to a shared variable X (operation-1) followed by a read of variable Y (operation-2); P2 performs a read of variable X (operation-3) followed by a write to shared variable Y (operation-4). For each processor, we can potentially consider 4! different orders for the four operations. However, we expect that the order in which each processor executes the operations (i.e., program order) be preserved. This requires that operation-1 always be executed before operation-2; operation-3 before operation-4. Now we have only 6 possible orders in which the operations can appear.
While it is possible for the two processors to see the operations in different order, intuition tells us that “correct” program behavior requires that all processors see the memory operations performed in the same order. These two requirements (Program Order be preserved, and all processors see the memory operations in the same order) are used to define a correct behavior of concurrent programs, and is termed as Sequential Consistency of memory.

<table>
<thead>
<tr>
<th>Ordering-1</th>
<th>Ordering-2</th>
<th>Ordering-3</th>
<th>Ordering-4</th>
<th>Ordering-5</th>
<th>Ordering-6</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: Write X</td>
<td>P1: Write X</td>
<td>P1: Write X</td>
<td>P2: Read X</td>
<td>P2: Read X</td>
<td>P2: Read X</td>
</tr>
<tr>
<td>P1: Read Y</td>
<td>P2: Read X</td>
<td>P2: Read X</td>
<td>P1: Write X</td>
<td>P2: Write Y</td>
<td>P1: Write X</td>
</tr>
<tr>
<td>P2: Read X</td>
<td>P1: Read Y</td>
<td>P2: Write Y</td>
<td>P1: Read Y</td>
<td>P1: Write X</td>
<td>P2: Write Y</td>
</tr>
<tr>
<td>P2: Write Y</td>
<td>P2: Write Y</td>
<td>P1: Read Y</td>
<td>P2: Write Y</td>
<td>P1: Read Y</td>
<td>P1: Read Y</td>
</tr>
</tbody>
</table>

Sequential Consistency.

Sequential consistency may also be described as follows. In addition to preserving program order of each concurrent program, the behavior should be the same as if the program is executed on a single processor by interleaving the operations of the concurrent program segments. The order in which all memory operations appear to all processors is then given by the interleaved execution.

Although appealing conceptually, Sequential consistency can result in very poor performance on modern processing systems. Some cause can be understood by examining modern architectural features that must be thwarted to guarantee sequential consistency. Consider processors with Cache memories. In such systems, it is not only sufficient to make sure a write is completed (written to local cache and even the main memory), but we need make sure that this write is “seen” by all processors and they either update or invalidate their caches. To guarantee this, we need to wait for an acknowledgement from all cache copies before assuming that a write was made visible to all processors. Such a requirement will significantly affect the performance of write operations. Likewise, the use of buffers for writes cannot be utilized if Sequential Consistency must be guaranteed.

In order to improve the performance of concurrent processing systems, we may not require that the hardware guarantee the second requirement of maintaining a common ordering among memory operations that is seen by all processors, but assure sequential consistency by other means (e.g., programming constructs). Consider one such relaxation on hardware requirements. In Process consistency, different processors of a parallel processing system may see different ordering among memory operations; however, the program order is still preserved. In our previous example, it is now possible for processor P1 to see a different ordering among the four memory operations (say Ordering 1) while P2 sees a different ordering (say Ordering 5). This is troubling because, now P1 and P2 will obtain different values for X and Y. In order to assure a correct program (as per Sequential Consistency), it is now necessary to use programming constructs such as locks (or synchronization) before allowing a processor to modify shared memory. In other words, it is now the responsibility of the programmer to define a sequential order among
memory accesses for shared variables. This is normally done by defining critical sections, and the use of mutual-exclusion to enter critical sections.

Weak Ordering.

Once we accept that it is programmer responsibility to specify “serialization” of accesses shared variables, we can further reduce the restrictions on what the hardware must preserve. That is the basis of all Weak consistency model.

1. Accesses to synchronization variables is **sequentially consistent**

2. No access to a synchronization variable is allowed to be performed until all previous writes have been completed everywhere.

3. No data access (read or write) is allowed to be performed until all previous accesses to synchronization variables have been performed.

The first condition forces a global order on all synchronization variables. Since ordinary variables are accessed only in critical sections (after accessing synchronization variables), Sequential consistency is assured even on ordinary variables.

The second condition implies that before a synchronization variable is released (and subsequently obtained by a different processor), all accesses made to ordinary variables are made globally visible to all processors. Likewise, the third condition requires that before a processor can access ordinary variables inside a critical section, accesses to synchronization variables must be globally visible. This forces mutual exclusion on synchronization variables and makes changes made in previous critical sections are globally visible.

Release Consistency.

While using synchronization variables (in Weak ordering), we normally associate locks with synchronization variables. We use lock and unlock (or acquire and release) operations on these locks. When you acquire, you have not yet made any changes to shared variables (other processes may have). When you release a lock, you may have updated variables and these variables must be made available to other processors. So, we need to maintain consistent data only on a lock release. How does the performance improve? You are not guaranteeing consistency memory until a lock is released

1. Before an ordinary variable is accessed, all previous acquires of synchronization variables performed by the processor must have completed.

2. Before a release on a synchronization variable is allowed, all previous reads and writes on ordinary variables performed by the processor must have completed.

3. The acquire and release accesses must be processor consistent.
Notice that since we assume mutual exclusion on acquires, we do not require Sequential consistency on acquires.

Program correctness is assured by the programmer by using acquires and releases in proper order and accessing ordinary variables in critical section.

The performance of Release consistency can be improved using "Lazy" release whereby, the shared memory is made consistent only on acquire by a different processor.

Entry and Scope Consistency

In both Weak ordering and Release consistency models, the shared memory is made consistent when any synchronization variable is released (or accessed). However, if we can associate a set of shared variable with each synchronization variable, then we only need to maintain consistency on these variables when the associated synchronization variable is released (or accessed). Entry consistency requires that the program specify the association between shared memory and synchronization variables. Scope consistency is similar to Entry consistency, but the associations between shared variables and synchronization variables is implicitly extracted. Consider the following example where lock-1 and lock-2 are synchronization variables while A,B,C, and D are ordinary shared variables.

```
P1
Lock lock-1
  A =1
  Lock lock-2
  B =1
Unlock lock-2
Unlock l2
P2
  Lock lock-2
  C = A  ----- may not see A=1
  D = B
Unlock lock-2
```

A similar effect will be achieved in Entry consistency by associating lock-1 with variable A and lock-2 with variables A, B, C and D.

Hardware Prefetch and Speculative Loads.

Relaxed memory consistency models discussed here increase the programming complexity, either by requiring the programmer to carefully and correctly use synchronization among the processors, or by requiring the compiler to insert necessary barrier instructions (e.g., Memory Barrier and Write Barrier) to assure program correctness. A recent study indicated that while it is difficult to outperform relaxed
memory models, two mechanisms can substantially improve the performance of Sequential consistency hardware [Pai and Adve's paper].

a). Hardware prefetch for write. Here, a request for Exclusive access of a cache block is issued even before the processor is ready to do a write, overlapping the time for invalidations and acknowledgements needed to implement sequential consistency correctly, with other computations. However, improper use of the prefetch may unnecessarily invalidate cache copies at other processors and this may lead to increased cache misses.

b). Speculative loads. Here cache blocks for load are prefetched, without changing any exclusive accesses that are currently held by other processors. If the data is modified before it is actually used, the prefetched copies are invalidated. Otherwise, one can realize performance gains from the prefetch.

Summary. It appears that weak ordering is the dominating memory consistency that is supported by current processors that are using in a multiprocessor systems. Sequential consistency is assured implicitly and transparently by software layers or by the programmer. In addition to improved performance, weak ordering may also have benefits in supporting fault-tolerance to applications using DSM systems, in terms of the amount state information that must be checkpointed [Hecht 99].

Support For Synchronization.

In the previous section we assumed that the programmer relies on synchronization among the processors (or processes) while accessing shared memory in order to assure program correctness as defined by the Sequential consistency model. The two fundamental synchronization constructs used by programmers are mutual exclusion locks and barriers. Mutual exclusion locks can be acquired by only one processor at a time, forcing a global sequential order on the locks. When barriers are used a processor is forced to wait for its partners and proceed beyond the barrier only when all partners reach the barrier. In this section we describe how mutual exclusion locks and barriers can be supported in hardware or software, and discuss performance of various implementations.

Mutual Exclusion Locks.

In the simplest implementation, mutual exclusion can be achieved using an atomic instruction that sets a flag if it is currently reset, otherwise, fails. Such Test-and-Set instructions were implemented in many older machines that supported multi-tasking (or time-shared) operating systems. With the advent of cache memories (and associated re-ordering of memory accesses), it become extremely inefficient to implement atomic Test-and-Set instructions. Modern processors actually use two separate instructions to achieve the atomicity. Consider the following example that uses Load Linked (LL) and Store Conditional (SC) instructions. The SC to the same memory location as the LL will be successful only if there are no intervening memory accesses to the same memory location. Otherwise, the SC fails to modify the memory location.
Try:  
Move R3, R4 ; Move value to be exchanged
LL R2, 0(R1) ; load linked to memory
SC R3, 0(R1) ; Store conditional to the same memory location
BEQZ R3, Try ; if unsuccessful, try again
Move R4, R3

In this example, the value in R3 will be stored only if SC is successful and the value in R3 will be non-zero. Otherwise, SC will fail to change the value in memory and R3 will be set to zero.

Typically, LL stores the memory address in a special Link Register which is compared with the memory address of a subsequent SC instruction. The Link Register is reset by any memory accesses to the same address by other processors or on a context switch of the current process. In a multiprocessor system, LL and SC can be implemented using cache coherency techniques previously discussed. For example, in snoopy systems, the Link Register is reset by snooping on the shared bus.

Notice that in this example we are actually using "Spin Locks" where a processor is not blocked on an unsuccessful attempt to acquire a lock. Instead, the unsuccessful processor will repeat its attempt to acquire the lock.

Shadow locks.

So far we have been assuming that the coherency of mutual exclusion lock variable is guaranteed, which can significantly reduce the performance of shared memory systems. Consider how the repeated attempts to acquire a spin-lock can lead to repeated invalidations of the lock variable. In order to improve the performance, processors are required to spin on a "shadow" lock. All spinning processors try to acquire the lock by accessing the lock variable in common memory. Unsuccessful processor will cache the "locked-value" in local caches and spin on local copies. The local copies are invalidated when the lock becomes available.

Consider the following code segment for an implementation of the spin locks

```
Lockit:  
LL R2, 0(R1) ; Load Linked
BNEZ R2, Lockit ; not available, spin
Load Immediate R2, #1 ; locked value
SC R2, 0(R1) ; store
BEQZ R2, Lockit ; branch if unsuccessful
```

The first branch (BNEZ) does the spin. The second branch is needed for atomicity.

Other Variations.
Spinning on a single variable causes excessive network traffic since all unsuccessful processors attempt acquire the lock as soon as it becomes available. We can consider implementation of "exponential-back-off" techniques whereby a processor uses different amounts of delays while attempting to acquire a lock. Alternatively, we can associate an "Array" for lock variables so that each processor spins on a different array element. A similar effect can be achieved using "Ticket" locks. Here, an integer (or ticket number) is assigned to each unsuccessful processor. The value of the ticket being serviced is incremented on each release of the lock, and a processor with a matching ticket number will then be allowed to acquire the lock. This technique is similar to how customers are assigned a service number and they are serviced when the current number serviced matches their number.

In Queue locks and Granting Locks, the unsuccessful processors are linked together so that the current lock holder can release the lock to the next processor in the queue. Notice that these techniques are similar to the techniques used in older systems where an unsuccessful process is blocked and queued on the lock by the operating system.

**Barrier Synchronization.**

In addition to mutual exclusion, shared memory parallel programming requires coordination among the current processes (or threads), whereby processes are required to rendezvous periodically before proceeding with computation. Such coordination is traditionally achieved using Fork and Join constructs. Before forking parallel computations, an integer value indicating the number of concurrent processes is set in a join-variable. As processes complete their assigned work, they execute the Join operation which atomically decrements the current value of the join-variable. A value of zero indicates that all concurrent processes have completed their tasks. In order to implement such barriers (i.e., Join construct), we need to use two mutual exclusion locks - one to decrement the counter and one to force processes to wait at the barrier.

How would we implement the join (barrier) using the atomic instructions we have seen so far. We need two locks: first acquire the join variable -- and increment; and one to make sure all processes wait until the last process arrives at the barrier. Consider the following implementation.

```plaintext
Lock (counter_lock);  /*to make sure updates are atomic
If (counter == p) release = 0; /* first reset release
    /* this is to make sure that processes are forced to wait
count = count-1;
unlock(counter_lock);
If (count==0) {
    count = p; /* after all arrive, reset the counter
    release = 1; }
else spin (release = 1); /* otherwise spin until release
```
The goal is to make sure that READ(X) returns the most recent value of the shared variable X, i.e. all valid copies of a shared variable are identical.

1. Software solutions
2. Hardware solutions

- **Snooping Cache Protocol**
  (for bus-based machines)
- **Directory Based Solutions**
  (for NUMA machines using a scalable switch)
Software Solutions

Compiler tags data as cacheable and non-cacheable. Only read-only data is considered cachable and put in private cache. All other data are non-cachable, and can be put in a global cache, if available.
Hardware Solution: Snooping Cache

Widely used in bus-based multiprocessors. The cache controller constantly watches the bus.

Write Invalidate

When a processor writes into \( C \), all copies of it in other processors are invalidated. These processors have to read a valid copy either from \( M \), or from the processor that modified the variable.

Write Broadcast

Instead of invalidating, why not broadcast the updated value to the other processors sharing that copy? This will act as write through for shared data, and write back for private data.

\textit{Write broadcast} consumes more bus bandwidth compared to \textit{write invalidate}. Why?
MESI Protocol (Papamarcos & Patel 1984)

It is a version of the snooping cache protocol. Each cache block can be in one of four states:

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INVALID</td>
<td>Not valid</td>
</tr>
<tr>
<td>SHARED</td>
<td>Multiple caches may hold valid copies.</td>
</tr>
<tr>
<td>EXCLUSIVE</td>
<td>No other cache has this block, M-block is valid</td>
</tr>
<tr>
<td>MODIFIED</td>
<td>Valid block, but copy in M-block is not valid</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Event</th>
<th>Local</th>
<th>Remote</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read hit</td>
<td>Use local copy</td>
<td>No action</td>
</tr>
<tr>
<td>Read miss</td>
<td>I to S, or I to E</td>
<td>(S,E,M) to S</td>
</tr>
<tr>
<td>Write hit</td>
<td>(S,E) to M</td>
<td>(S,E,M) to I</td>
</tr>
<tr>
<td>Write miss</td>
<td>I to M</td>
<td>(S,E,M) to I</td>
</tr>
</tbody>
</table>

When a cache block changes its status from M, it first updates the main memory.
Examples of state transitions under MESI protocol

A. Read Miss
Following the read miss, the holder of the *modified copy* signals the initiator to *try again*. Meanwhile, it seizes the bus, and write the updated copy into the main memory.
C. Write Miss

```
x=9  x=2  x=4  I  I  I  I  x=5
M

x=9  x=2  x=4  I  I  I  I  x=9
writeback M

x=9  x=2  x=3  x=9  x=9
allocate X

x=9  x=2  x=3  x=9  x=9
now modify

x=9  x=2  x=3  x=10  x=9
I  I  I  M
```
Directory-based cache coherence

The snooping cache protocol does not work if there is no bus. Large-scale shared memory multiprocessors may connect processors with memories through switches.

A directory has to keep track of the states of the shared variables, and oversee that they are modified in a consistent way. Maintenance of the directory in a distributed environment is another issue.

Naïve solutions may lead to deadlock. Consider this:

P1 has a read miss for x2 (local to P2)
P2 has a read miss for x1 (local to P1)

Each will block and expect the other process to send the correct value of x: deadlock (!)
Memory Consistency Models

Multiple copies of a shared variable can be found at different locations, and any write operation must update *all of them*.

**Coherence vs. consistency**

Cache coherence protocols guarantee that *eventually* all copies are updated. Depending on how and when these updates are performed, a read operation may sometimes return unexpected values.

**Consistency** deals with *what values can be returned to the user* by a read operation (may return unexpected values if the update is not complete). **Consistency model** is a contract that defines what a programmer can expect from the machine.
Sequential Consistency

Program 1.

<table>
<thead>
<tr>
<th>process 0</th>
<th>process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>{initially, x=0 and y=0}</td>
<td></td>
</tr>
<tr>
<td>x:=1; y:=1;</td>
<td></td>
</tr>
<tr>
<td>if (y=0) then x:=2; if (x=0) then y:=2;</td>
<td></td>
</tr>
<tr>
<td>print x; print y;</td>
<td></td>
</tr>
</tbody>
</table>

If both processes run concurrently, then can we see a printout (x=2, y=2)?

A key question is: Did process 0 read y before process 1 modified it? One possible scenario is:

x=0 \[w(x,1)\] read y \[process 0\]

y=0 \[w(y,1)\] read x \[process 1\]

Here, the final values are: (x=1, y=1)
Here, the final values are: (x=2, y=1)

**Properties of SC.**

**SC1.** All operations in a single process are executed in program order.

**SC2.** The result of any execution is the same as if a single sequential order has been maintained among all operations.
Implementing SC

Case 1
Consider a switch-based multiprocessor. Assume there is no cache.

Process 0 executes: \((x:=1; \ y:=2)\)
To prevent p1 or p2 from seeing these in a different order, p0 must receive an acknowledgement after every write operation.

Case 2
In a multiprocessor where processors have private cache, all invalidate signals must be acknowledged.
Write-buffers and New problems

Go back to the first program now.

Let both \( x:=1 \) and \( y:=1 \) be written into the write buffers, but before the memory is updated, let the two if statements be evaluated.

Both can be true, and \( (x:=2, y:=2) \) are possible!

*This violates sequential consistency.*
Our Pattern Language (OPL), described in the following paper A Design Pattern Language for Engineering (Parallel) Software, represents the confluence of three different directions of research. The first direction of research was aimed at analyzing the varieties of computing, with a goal of identifying key influences on computer architecture. This research led to the identification of “13 dwarfs” of computing, which in turn were became “13 computational patterns” of OPL (See Figure 1 of the following paper on OPL in this chapter). The second direction of research arose from generations of research on architecting large pieces of software. This research led to the identification of a series of “architectural styles” [6] that were incorporated into OPL as “structural patterns.” These computational and structural patterns sit, side by side, at the top of OPL in Figure 1 of the following paper. The overriding vision of OPL, a unified pattern language for designing and implementing parallel software, came from Mattson’s book Patterns for Parallel Programming [3]. A revised of version of Mattson’s pattern languages constitutes the three lower levels of OPL.

### Introduction

Our Pattern Language (OPL), described in the following paper A Design Pattern Language for Engineering (Parallel) Software, represents the confluence of three different directions of research. The first direction of research was aimed at analyzing the varieties of computing, with a goal of identifying key influences on computer architecture. This research led to the identification of “13 dwarfs” of computing, which in turn were became “13 computational patterns” of OPL (See Figure 1 of the following paper on OPL in this chapter). The second direction of research arose from generations of research on architecting large pieces of software. This research led to the identification of a series of “architectural styles” [6] that were incorporated into OPL as “structural patterns.” These computational and structural patterns sit, side by side, at the top of OPL in Figure 1 of the following paper. The overriding vision of OPL, a unified pattern language for designing and implementing parallel software, came from Mattson’s book Patterns for Parallel Programming [3]. A revised of version of Mattson’s pattern languages constitutes the three lower levels of OPL.

### Quick Introduction to Patterns

A design pattern is simply a generalizable solution to a recurring problem that occurs within a well-defined context. Patterns are written down in a highly structured format to capture the most essential elements of a problem’s solution in such a way that a software designer can quickly find what he or she needs to solve a problem. The concept of design patterns originated in the work of Christopher Alexander. Alexander gathered his ideas in the book A Pattern Language: Towns, Buildings, Construction [2]. His goal was to identify and codify common solutions to recurrent problems ranging from the arrangement of towns and cities to the construction of a single building. Alexander’s ideas on civil architecture are still debated, but the impact of his ideas on software design has been substantial.

Design patterns help software designers in a number of ways. First, they give a name to solutions to common problems in software design. Further, they define a palette of solution alternatives to these problems. This palette serves to educate novice programmers as to design alternatives, and it also serves to comfort more mature programmers that there is such a fixed set of alternatives. At a deeper level work by cognitive psychologists on the nature of programming has shown that patterns (which they called “plans”) are at the core of the thought processes used by expert programmers [5]. Thus, a design pattern encodes expertise. In fact, we can view the difference between a novice and an expert in any field in terms of patterns; i.e., an expert has a larger library of design patterns to draw on.
Hence, in our own work to bring order and discipline to the practice of engineering parallel software, we have found design patterns to be a valuable tool. It is the connections between patterns that matter in a high-quality software design, so a catalog of isolated design patterns will not be sufficient. Instead, we need a web of interconnected patterns. A hierarchical layered construction of design patterns works together to define a language for patterns oriented-design. Following Alexander, we call this interconnected family of patterns a “Design Pattern Language.”

**Dwarfs, Motifs, and Computational Patterns**

As discussed in the Introduction to this entire volume, efforts to define the 13 dwarfs predate the efforts on developing OPL, and OPL is not discussed at all in the original Berkeley View paper [1]. It was Tim Mattson who suggested that while the dwarfs did not indicate a definite micro-architecture or software implementation, they did provide a nice bridge between the computationally focused thinking of application developers and particular approaches to parallelizing computations that he had expressed in his pattern language for parallel programming [3]. Mattson made another important suggestion: these dwarfs could be understood as computational patterns. In other words, dwarfs were also generalizable solutions to recurrent problems of computation, and could form an integral part of a pattern language for parallel computing.

**Architectural Styles, Design Patterns, and Computational Patterns**

For some time OPL was envisioned as a set of computational patterns (the former dwarfs) sitting on top of Mattson’s pattern language for parallel programming [3]. The result was still less than fully satisfying because few applications could be naturally decomposed into a single More generally, this generation of OPL didn’t help at all with the important problem of how to compose computations. It was Koushik Sen who first suggested that a small set of techniques could be used to compose all the diverse sets of computations in parallel programs.

Approaches to composing software had received considerable attention in software engineering of sequential programs, and particularly in object-oriented programming. Initially this trend of research seemed unimportant and perhaps totally irrelevant for parallel software. Examining the software architectures developed in an object-oriented style gave little insight into how to structure an efficient parallel program. Over time, Kurt Keutzer began to champion the notion that while software architecture did not offer immediate insights into how to parallelize software, the need for software modularity was as great or greater in parallel software. Shaw and Garland had already codified a list of architectural styles [6] for composing software and creating software architectures. This list of architectural styles seemed immediately useful for parallel software as well; however, four structural patterns particularly useful for parallel computing were added - Map Reduce, Iterative Refinement, and Puppeteer, and Arbitrary Static Task Graph replaced the common Call-and-Return architecture of object-oriented programming.

**Patterns for parallel programming and OPL**

The structural and computational patterns are composed to define software architecture. A software architecture defines the components that make up an application, the communication among components, and the fundamental computations that occur inside components. The software architecture, however, has little to say about how the software architecture is mapped onto the hardware of a parallel computer. To address parallelism, we combined the computational and structural patterns with the lower level, parallel programming design pattern languages from [3]. These parallel programming patterns define a distinct pattern language for parallel programming (PLPP).

The original version of PLPP dates back to the late 1990’s and early 2000’s. At that time, data parallel algorithms had fallen into disrepute. The use of graphics processing units (GPUs) for general-purpose programming was considered more of a gimmick than a serious option for computing. Data parallel algorithms popularized by the older SIMD machines of the late 1980’s to mid-1990’s had failed spectacularly. The result is that the original version of PLPP deemphasized data parallel patterns and instead emphasized patterns relevant to cluster and shared-memory multiprocessor computers.

The work with the Par Lab, however, forced a reconsideration of PLPP. Projects such as the MRI image processing and Content Based Image Recognition applications described elsewhere in this book demonstrated the importance of
GPGPU programming and the data parallel algorithms behind them. Furthermore, the wider vector units on modern microprocessors showed the importance of data parallel algorithms mapped onto vector computing. The result is a complete redesign of PLPP. A book describing this new version of PLPP is being prepared. A preview of the new PLPP, however, can be found in the paper on OPL, A Design Pattern Language for Engineering (Parallel) Software, elsewhere in this chapter.

Patterns in Par Lab Applications and SEJITS

Patterns from OPL were used to guide the design of a number of applications described elsewhere in this book. A similar OPL-based methodology was used for the magnetic resonance image reconstruction application, the content-based image reconstruction application, the speech recognition application and a number of other variants of audio-processing described in the paper “PyCASP: A Pattern-Oriented Framework for Parallel Programming” found elsewhere in this chapter. In this methodology a high-level software architecture was proposed using structural and computational patterns, a parallel algorithm strategy was chosen, then an implementation strategy, and finally an execution strategy. In particular, the development of the high-level software architecture for content-based image retrieval is used as a model in the paper “A Design Pattern Language for Engineering (Parallel) Software,” found elsewhere in this chapter. In truth, the design approach was not entirely top-down: often the sequence bounced back between explorations of high-level software architectures and final execution strategies with the parallel algorithm strategy following suit. In fact, it was the observation that after the software architecture and execution strategy were chosen, the parallel algorithm and implementation strategy seemed obvious and led to a pattern-oriented approach to SEJITS.

(Referring to Figure 1 in “A Design Pattern Language for Engineering (Parallel) Software” will help in following this discussion.) In a number of Par Lab applications we discovered that dense-linear algebra, structured grid, and other computational patterns offered ample opportunities for a data parallel algorithm strategy. With stream processing on a GPGPU processor as the execution target, an SPMD implementation strategy is a natural choice for exploiting the data parallelism. Thus we saw many applications follow a dense linear algebra, data parallelism, SPMD, stream processing flow through OPL. SEJITS was an ideal mechanism for building a single “stovepipe” flow to hardware rather than use a traditional layered compiler architecture with a number of intermediate representations. The idea of using SEJITS as an implementation medium for pattern-oriented frameworks is demonstrated in the paper “PyCASP: A Pattern-oriented Framework for Parallel Programming” later in this chapter and also in the paper “Copperhead: Compiling an Embedded Data Parallel Language” found elsewhere in this volume.

Evolving and Evangelizing OPL

The pattern language was evolved and evangelized using a number of mechanisms. Three Parallel Patterns Workshops were held from 2009 to 2011. Each workshop drew over thirty participants from around the world. Textual descriptions of the structural and computational patterns were developed in these workshops and also in three graduate seminars taught at Berkeley. A course entitled Engineering Parallel Software, using OPL’s approach, was taught to undergraduate audiences three times. Average class size in each instance was twenty-five. In each course projects demonstrated that students could learn in a single semester how to architect code and efficiently implement it using the design patterns in OPL. More on educational efforts within Par Lab is discussed in the education chapter.
UNIT 2 - PARALLEL PROGRAM CHALLENGES

The past few decades have seen large fluctuations in the perceived value of parallel computing. At times, parallel computation has optimistically been viewed as the solution to all of our computational limitations. At other times, many have argued that it is a waste of effort given the rate at which processor speeds and memory prices continue to improve. Perceptions continue to vacillate between these two extremes due to a number of factors, among them: the continual changes in the “hot” problems being solved, the programming environments available to users, the supercomputing market, the vendors involved in building these supercomputers, and the academic community’s focus at any given point and time. The result is a somewhat muddied picture from which it is difficult to objectively judge the value and promise of parallel computing.

In spite of the rapid advances in sequential computing technology, the promise of parallel computing is the same now as it was at its inception. Namely, if users can buy fast sequential computers with gigabytes of memory, imagine how much faster their programs could run if $p$ of these machines were working in cooperation! Or, imagine how much larger a problem they could solve if the memories of $p$ of these machines were used cooperatively!

The challenges to realizing this potential can be grouped into two main problems: the hardware problem and the software problem. The former asks, “how do I build a parallel machine that will allow these $p$ processors and memories to cooperate efficiently?” The software problem asks, “given such a platform, how do I express my computation such that it will utilize these $p$ processors and memories effectively?”
In recent years, there has been a growing awareness that while the parallel community can build machines that are reasonably efficient and/or cheap, most programmers and scientists are incapable of programming them effectively. Moreover, even the best parallel programmers cannot do so without significant effort. The implication is that the software problem is currently lacking in satisfactory solutions. This dissertation focuses on one approach designed to solve that problem.

In particular, this work describes an effort to improve a programmer’s ability to utilize parallel computers effectively using the ZPL parallel programming language. ZPL is a language whose parallelism stems from operations applied to its arrays’ elements. ZPL derives from the description of Orca C in Calvin Lin’s dissertation of 1992 [Lin92]. Since that time, Orca C has evolved to the point that it is hardly recognizable, although the foundational ideas have remained intact. ZPL has proven to be successful in that it allows parallel programs to be written at a high level, without sacrificing portability or performance. This dissertation will also describe aspects of Advanced ZPL (A-ZPL), ZPL’s successor language which is currently under development.

One of the fundamental concepts that was introduced to Orca C during ZPL’s inception was the concept of the region. A region is simply a user-specified set of indices, a concept which may seem trivially uninteresting at first glance. However, the use of regions in ZPL has had a pervasive effect on the language’s appearance, semantics, compilation, and runtime management, resulting in much of ZPL’s success. This dissertation defines the region in greater depth and documents its role in defining and implementing the ZPL language.

This dissertation’s study of regions begins in the next chapter. The rest of this chapter provides a general overview of parallel programming, summarizing the challenges inherent in writing parallel programs, the techniques that can be used to create them, and the metrics used to evaluate these techniques. The next section begins by providing a rough overview of parallel architectures.
**Parallel Architectures**

**Parallel Architecture Classifications**

This dissertation categorizes parallel platforms as being one of three rough types: *distributed memory*, *shared memory*, or *shared address space*. This taxonomy is somewhat coarse given the wide variety of parallel architectures that have been developed, but it provides a useful characterization of current architectures for the purposes of this dissertation.

Distributed memory machines are considered to be those in which each processor has a local memory with its own address space. A processor’s memory cannot be accessed directly by another processor, requiring both processors to be involved when communicating values from one memory to another. Examples of distributed memory machines include commodity Linux clusters.

Shared memory machines are those in which a single address space and global memory are shared between multiple processors. Each processor owns a local cache, and its values are kept coherent with the global memory by the operating system. Data can be exchanged between processors simply by placing the values, or pointers to values, in a predefined location and synchronizing appropriately. Examples of shared memory machines include the SGI Origin series and the Sun Enterprise.

Shared address space architectures are those in which each processor has its own local memory, but a single shared address space is mapped across the distinct memories. Such architectures allow a processor to access the memories of other processors without their direct involvement, but they differ from shared memory machines in that there is no implicit caching of values located on remote machines. The primary example of a shared address machine is Cray’s T3D/T3E line.

Many modern machines are also built using a combination of these technologies in a hierarchical fashion, known as a *cluster*. Most clusters consist of a number of shared memory machines connected by a network, resulting in a hybrid of shared and distributed memory characteristics. IBM’s large-scale SP machines are an example of this design.
ZPL supports compilation and execution on these diverse architectures by describing them using a single machine model known as the Candidate Type Architecture (CTA) [Sny86]. The CTA is a reasonably vague model, and deliberately so. It characterizes parallel machines as a group of von Neumann processors, connected by a sparse network of unspecified topology. Each processor has a local memory that it can access at unit cost. Processors can also access other processors’ values at a cost significantly higher than unit cost by communicating over the network. The CTA also specifies a controller used for global communications and synchronization, though that will not be of concern in this discussion. See Figure 1.1 for a simple diagram of the CTA.

Why use such an abstract model? The reason is that parallel machines vary so widely in design that it is difficult to develop a more specific model that describes them all. The CTA successfully abstracts the vast majority of parallel machines by emphasizing the importance of locality and the relatively high cost of interprocessor communication. This is in direct contrast to the overly idealized PRAM [FW78] model or the extremely parameterized LogP model [CKP+93], neither of which form a useful foundation for a compiler concerned with portable performance. For more details on the CTA, please refer to the literature [Sny86, Sny95, Lin92].
Writing parallel programs is strictly more difficult than writing sequential ones. In sequential programming, the programmer must design an algorithm and then express it to the computer in some manner that is correct, clear, and efficient to execute. Parallel programming involves these same issues, but also adds a number of additional challenges that complicate development and have no counterpart in the sequential realm. These challenges include: finding and expressing concurrency, managing data distributions, managing interprocessor communication, balancing the computational load, and simply implementing the parallel algorithm correctly. This section considers each of these challenges in turn.

Concurrency

Concurrency is crucial if a parallel computer’s resources are to be used effectively. If an algorithm cannot be divided into groups of operations that can execute concurrently, performance improvements due to parallelism cannot be achieved, and any processors after the first will be of limited use in accelerating the algorithm. To a large extent, different problems inherently have differing amounts of concurrency. For most problems, developing an algorithm that achieves its maximal concurrency requires a combination of cleverness and experience from the programmer.

As motivating examples, consider matrix addition and matrix multiplication. Mathematically, we might express these operations as follows:

**Matrix addition:** Given matrices $A$ and $B$ $(m \times n)$,
\[
A + B = C \ (m \times n), \text{ where } C_{i,j} = A_{i,j} + B_{i,j}.
\]

**Matrix multiplication:** Given matrices $A$ $(m \times n)$ and $B$ $(n \times o)$,
\[
A \times B = C \ (m \times o), \text{ where } C_{i,k} = \sum_{j=1}^{n} A_{i,j} \cdot B_{j,k}
\]

Consider the component operations that are required to implement these definitions. Matrix addition requires $m \cdot n$ pairwise sums to be computed. Matrix multiplication requires
the evaluation of $m \cdot n \cdot o$ pairwise products and $\Omega(m \cdot \log n \cdot o)$ pairwise sums. In considering the parallel implementation of either of these algorithms, programmers must ask themselves, “can all of the component operations be performed simultaneously?” Looking at matrix addition, a wise parallel programmer would conclude that they can be computed concurrently—each sum is independent from the others, and therefore they can all be computed simultaneously. For matrix multiplication, the programmer would similarly conclude that all of the products could be computed simultaneously. However, each sum is dependent on values obtained from previous computations, and therefore they cannot be computed completely in parallel with the products or one another.

As a result of this analysis, a programmer might conclude that matrix addition is inherently more concurrent than matrix multiplication. As a second observation, the programmer should note that for matrices of a given size, matrix multiplication tends to involve more operations than matrix addition.

If the programmer is designing an algorithm to run on $p$ processors where $p \ll m, n, o$, a related question is “are there better and worse ways to divide the component operations into $p$ distinct sets?” It seems likely that there are, although the relevant factors may not be immediately obvious. The rest of this section describe some of the most important ones.

**Data Distribution**

Another challenge in parallel programming is the distribution of a problem’s data. Most conventional parallel computers have a notion of *data locality*. This implies that some data will be stored in memory that is “closer” to a particular processor and can therefore be accessed much more quickly. Data locality may occur due to each processor having its own distinct local memory—as in a distributed memory machine—or due to processor-specific caches as in a shared memory system.

Due to the impact of data locality, a parallel programmer must pay attention to where data is stored in relation to the processors that will be accessing it. The more local the values are, the quicker the processor will be able to access them and complete its work. It
should be evident that distributing work and distributing data are tightly coupled, and that an optimal design will consider both aspects together.

For example, assuming that the $m \times n$ sums in a matrix addition have been divided between a set of $p$ processors, it would be ideal if the values of $A$, $B$, and $C$ were distributed in a corresponding manner so that each processor’s sums could be computed using local values. Since there is a one-to-one correspondence between sums and matrix values, this can easily be achieved. For example, each processor $p_k$ could be assigned matrix values $A_{i,j}$, $B_{i,j}$, and $C_{i,j}$, $\forall i \equiv k \pmod{p}$, $\forall j \in 1 \ldots n$.

Similarly, the implementor of a parallel matrix multiplication algorithm would like to distribute the matrix values, sums, and products among the processors such that each node only needs to access local data. Unfortunately, due to the data interactions inherently required by matrix multiplication, this turns out to be possible only when matrix values are explicitly replicated on multiple processors. While this replication may be an option for certain applications, it runs counter to the general goal of running problems that are $p$ times bigger than their sequential counterparts. Such algorithms that rely on replication to avoid communication are not considered scalable. Furthermore, replication does not solve the problem since matrix products are often used in subsequent multiplications and would therefore require communication to replicate their values across the processor set after being computed.

To create a scalable matrix multiplication algorithm, there is no choice but to transfer data values between the local memories of the processors. Unfortunately, the reality is that most interesting parallel algorithms require such communication, making it the next parallel programming challenge.

**Communication**

Assuming that all the data that a processor needs to access cannot be made exclusively local to that processor, some form of data transfer must be used to move remote values to a processor’s local memory or cache. On distributed memory machines, this commun-
cation typically takes the form of explicit calls to a library designed to move values from
one processor’s memory to another. For shared memory machines, communication in-
volves cache coherence protocols to ensure that a processor’s locally cached values are
kept consistent with the main memory. In either case, communication constitutes work that
is time-consuming and which was not present in the sequential implementation. There-
fore, communication overheads must be minimized in order to maximize the benefits of
parallelism.

Over time, a number of algorithms have been developed for parallel matrix multipli-
cation, each of which has unique concurrency, data distribution, and communication charac-
teristics. A few of these algorithms will be introduced and analyzed during the course of
the next few chapters. For now, we return to our final parallel computing challenges.

Load Balancing

The execution time of a parallel algorithm on a given processor is determined by the time
required to perform its portion of the computation plus the overhead of any time spent per-
forming communication or waiting for remote data values to arrive. The execution time of
the algorithm as a whole is determined by the longest execution time of any of the proces-
sors. For this reason, it is desirable to balance the total computation and communication
between processors in such a way that the maximum per-processor execution time is mini-
mized. This is referred to as load balancing, since the conventional wisdom is that dividing
work between the processors as evenly as possible will minimize idle time on each proces-
sor, thereby reducing the total execution time.

Load balancing a matrix addition algorithm is fairly simple due to the fact that it can
be implemented without communication. The key is simply to give each processor approxi-
mately the same number of matrix values. Similarly, matrix multiplication algorithms are
typically load balanced by dividing the elements of $C$ among the processors as evenly as
possible and trying to minimize the communication overheads required to bring remote $A$
and $B$ values into the processors’ local memories.
Once all of the parallel design decisions above have been made, the nontrivial matter of implementing and debugging the parallel program still remains. Programmers often implement parallel algorithms by creating a single executable that will execute on each processor. The program is designed to perform different computations and communications based on the processor’s unique ID to ensure that the work is divided between instances of the executable. This is referred to as the Single Program, Multiple Data (SPMD) model, and its attractiveness stems from the fact that only one program must be written (albeit a nontrivial one). The alternative is to use the Multiple Program, Multiple Data (MPMD) model, in which several cooperating programs are created for execution on the processor set. In either case, the executables must be written to cooperatively perform the computation while managing data locality and communication. They must also maintain a reasonably balanced load across the processor set. It should be clear that implementing such a program will inherently require greater programmer effort than writing the equivalent sequential program.

As with any program, bugs are likely to creep into the implementation, and the effects of these bugs can be disastrous. A simple off-by-one error can cause data to be exchanged with the wrong processor, or for a program to deadlock, waiting for a message that was never sent. Incorrect synchronization can result in data values being accessed prematurely, or for race conditions to occur. Bugs related to parallel issues can be nondeterministic and show up infrequently. Or, they may occur only when using large processor sets, forcing the programmer to sift through a large number of execution contexts to determine the cause. In short, parallel debugging involves issues not present in the sequential world, and it can often be a huge headache.
Computing effectively with a single processor is a challenging task. The programmer must be concerned with creating programs that perform correctly and well. Computing with multiple processors involves the same effort, yet adds a number of new challenges related to the cooperation of multiple processors. None of these new factors are trivial, giving a good indication of why programmers and scientists find parallel computing so challenging.

The design of the ZPL language strives to relieve programmers from most of the burdens of correctly implementing a parallel program. Yet, rather than making them blind to these details, ZPL’s regions expose the crucial parallel issues of concurrency, data distribution, communication, and load balancing to programmers, should they care to reason about such issues. These benefits of regions will be described in subsequent chapters. For now, we shift our attention to the spectrum of techniques that one might consider when approaching the task of parallel programming.

Approaches to Parallel Programming

Techniques for programming parallel computers can be divided into three rough categories: parallelizing compilers, parallel programming languages, and parallel libraries. This section considers each approach in turn.

Parallelizing Compilers

The concept of a parallelizing compiler is an attractive one. The idea is that programmers will write their programs using a traditional language such as C or Fortran, and the compiler will be responsible for managing the parallel programming challenges described in the previous section. Such a tool is ideal because it allows programmers to express code in a familiar, traditional manner, leaving the challenges related to parallelism to the compiler. Examples of parallelizing compilers include SUIF, KAP, and the Cray MTA compiler [HAA+96, KLS94, Ter99].
The primary challenge to automatic parallelization is that converting sequential programs to parallel ones is an entirely non-trivial task. As motivation, let us return to the example of matrix multiplication. Written in C, a sequential version of this computation might appear as in Listing 1.1.

Well-designed parallel implementations of matrix multiplication tend to appear very different than this sequential algorithm, in order to maximize data locality and minimize communication. For example, one of the most scalable algorithms, the SUMMA algorithm [vdGW95], bears little resemblance to the sequential triply nested loop. SUMMA consists of $n$ iterations. On the $i^{th}$ iteration, A’s $i^{th}$ column is broadcast across the processor columns and B’s $i^{th}$ row is broadcast across processor rows. Each processor then calculates the cross product of its local portion of these values, producing the $i^{th}$ term in the sum for each of C’s elements. Figure 1.2 shows an illustration of the SUMMA algorithm.

The point here is that effective parallel algorithms often differ significantly from their sequential counterparts. While having an effective parallel compiler would be a godsend, expecting a compiler to automatically understand an arbitrary sequential algorithm well enough to create an efficient parallel equivalent seems a bit naive. The continuing lack of such a compiler serves as evidence to reinforce this claim.
Figure 1.2: The SUMMA Algorithm For Matrix Multiplication
Many parallelizing compilers, including those named above, take an intermediate approach in which programmers add directives to their codes to provide the compiler with information to aid it in the task of parallelizing the code. The more of these that need to be relied upon, the more this approach resembles a new programming language rather than a parallelizing compiler, so further discussion of this approach is deferred to the next section.

*Parallel Programming Languages*

A second approach to parallel programming is the design and implementation of parallel programming languages. These are languages designed to support parallel computing better than sequential languages, though many of them are based on traditional languages in the hope that existing code bases can be reused. This dissertation categorizes parallel languages as being either *global-view* or *local-view*.

*Global-view Languages*

Global-view languages are those in which the programmer specifies the behavior of their algorithm as a whole, largely ignoring the fact that multiple processors will be used to implement the program. The compiler is therefore responsible for managing all of the parallel implementation details, including data distribution and communication.

Many global-view languages are rather unique, providing language-level concepts that are tailored specifically for parallel computing. The ZPL language and its regions form one such example. Other global-view languages include the directive-based variations of traditional programming languages used by parallelizing compilers, since the annotated sequential programs are global descriptions of the algorithm with no reference to individual processors. As a simple example of a directive-based global-view language, consider the pseudocode implementation of the SUMMA algorithm in Listing 1.2. This is essentially a sequential description of the SUMMA algorithm with some comments (directives) that indicate how each array should be distributed between processors.
Listing 1.2: Pseudo-Code for SUMMA Using a Global View

```c
double A[m][n];
double B[n][o];
double C[m][o];
double ColA[m];
double RowB[o];

// distribute C [block,block]
// align A[::] with C[::]
// align B[::] with C[::]
// align ColA[:] with C[*,*]
// align RowB[:] with C[*,:]

for (i=0; i<m ; i++) {
    for (k=0; k<o; k++) {
        C[i][k] = 0;
    }
}

for (j=0; j<n; j++) {
    for (i=0; i<m; i++) {
        ColA[i] = A[i][j];
    }
    for (k=0; k<o; k++) {
        RowB[k] = B[j][k];
    }
    for (i=0; i<m ;i++) {
        for (k=0; k<o; k++) {
            C[i][k] += ColA[i] * RowB[k];
        }
    }
}
```
The primary advantage to global-view languages is that they allow the programmer to focus on the algorithm at hand rather than the details of the parallel implementation. For example, in the code above, the programmer writes the loops using the array’s global bounds. The task of transforming them into loops that will cause each processor to iterate over its local data is left to the compiler.

This convenience can also be a liability for global-view languages. If a language or compiler does not provide sufficient feedback about how programs will be implemented, knowledgeable programmers may be unable to achieve the parallel implementations that they desire. For example, in the SUMMA code of Listing 1.2, programmers might like to be assured that an efficient broadcast mechanism will be used to implement the assignments to ColA and RowB, so that the assignment to C will be completely local. Whether or not they have such assurance depends on the definition of the global language being used.

*Local-view Languages*

Local-view languages are those in which the implementor is responsible for specifying the program’s behavior on a per-processor basis. Thus, details such as communication, data distribution, and load balancing must be handled explicitly by the programmer. A local-view implementation of the SUMMA algorithm might appear as shown in Listing 1.3.

The chief advantage of local-view languages is that users have complete control over the parallel implementation of their programs, allowing them to implement any parallel algorithm that they can imagine. The drawback to these approaches is that managing the details of a parallel program can become a painstaking venture very quickly. This contrast can be seen even in short programs such as the implementation of SUMMA in Listing 1.3, especially considering that the implementations of its Broadcast...(), IOwn...(), and GlobToLoc...() routines have been omitted for brevity. The magnitude of these details are such that they tend to make programs written in local-view languages much more difficult to maintain and debug.
Listing 1.3: Pseudo-Code for SUMMA Using a Local View

```c
int m_loc = m/proc_rows;
int o_loc = o/proc_cols;
int n_loc_col = n/proc_cols;
int n_loc_row = n/proc_rows;

double A[m_loc][n_loc_col];
double B[n_loc_row][o_loc];
double C[m_loc][o_loc];
double ColA[m_loc];
double RowB[o_loc];

for (i=0; i<m_loc; i++) {
    for (k=0; k<o_loc; k++) {
        C[i][k] = 0;
    }
}

for (j=0; j<n; j++) {
    if (IOwnCol(j)) {
        BroadcastColSend(A, GlobToLocCol(j));
        for (i=0; i<m_loc; i++) {
            ColA[i] = A[i][j];
        }
    } else {
        BroadcastColRecv(ColA);
    }
    if (IOwnRow(j)) {
        BroadcastRowSend(B, GlobToLocRow(j));
        for (k=0; k<o_loc; k++) {
            RowB[k] = B[j][k];
        }
    } else {
        BroadcastRowRecv(RowB);
    }
    for (i=0; i<m_loc; i++) {
        for (k=0; k<o_loc; k++) {
            C[i][k] += ColA[i] * RowB[k];
        }
    }
}
```
Parallel Libraries

Parallel libraries are the third approach to parallel computing considered here. These are simply libraries designed to ease the task of utilizing a parallel computer. Once again, we categorize these as global-view or local-view approaches.

Global-view Libraries

Global-view libraries, like their language counterparts, are those in which the programmer is largely kept blissfully unaware of the fact that multiple processors are involved. As a result, the vast majority of these libraries tend to support high-level numerical operations such as matrix multiplications or solving linear equations. The number of these libraries is overwhelming, but a few notable examples include the NAG Parallel Library, ScaLAPACK, and PLAPACK [NAG00, BCC+97, vdG97].

The advantage to using a global-view library is that the supported routines are typically well-tuned to take full advantage of a parallel machine’s processing power. To achieve similar performance using a parallel language tends to require more effort than most programmers are willing to make.

The disadvantages to global-view libraries are standard ones for any library-based approach to computation. Libraries support a fixed interface, limiting their generality as compared to programming languages. Libraries can either be small and extremely special-purpose or they can be wide, either in terms of the number of routines exported or the number of parameters passed to each routine [GL00]. For these reasons, libraries are a useful tool, but often not as satisfying for expressing general computation as a programming language.

Local-view Libraries

Like languages, libraries may also be local-view. For our purposes, local-view libraries are those that aid in the support of processor-level operations such as communication between
processors. Local-view libraries can be evaluated much like local-view languages: they give the programmer a great deal of explicit low-level control over a parallel machine, but by nature this requires the explicit management of many painstaking details. Notable examples include the MPI and SHMEM libraries [Mes94, BK94].

Summary

This section has described a number of different ways of programming parallel computers. To summarize, general parallelizing compilers seem fairly intractable, leaving languages and libraries as the most attractive alternatives. In each of these approaches, the tradeoff between supporting global- and local-view approaches is often one of high-level clarity versus low-level control. The goal of the ZPL programming language is to take advantage of the clarity offered by a global-view language without sacrificing the programmer’s ability to understand the low-level implementation and tune their code accordingly. Further chapters will develop this point and also provide a more comprehensive survey of parallel programming languages and libraries.

Evaluating Parallel Programs

For any of the parallel programming approaches described in the previous section, there are a number of metrics that can be used to evaluate its effectiveness. This section describes five of the most important metrics that will be used to evaluate parallel programming in this dissertation: performance, clarity, portability, generality, and a programmer’s ability to reason about the implementation.

Performance

Performance is typically viewed as the bottom line in parallel computing. Since improved performance is often the primary motivation for using parallel computers, failing to achieve good performance reflects poorly on a language, library, or compiler.
This dissertation will typically measure performance in terms of speedup, defined to be the fastest single-processor execution time (using any approach) divided by the execution time on $p$ processors:

$$speedup_p = \frac{T_{1_{\text{min}}}}{T_p}$$

If the original motivating goal of running a program $p$ times faster using $p$ processors is met, then $speedup_p = p$. This is known as linear speedup. In practice, this is challenging to achieve since the parallel implementation of most interesting programs requires work beyond that which was required for the sequential algorithm: in particular, communication and synchronization between processors. Thus, the amount of work per processor in a parallel implementation will typically be more than $1/p$ of the work of the sequential algorithm.

On the other hand, note that the parallelization of many algorithms requires allocating
approximately $1/p$ of the sequential program’s memory on each processor. This causes the working set of each processor to decrease as $p$ increases, allowing it to make better use of the memory hierarchy. This effect can often offset the overhead of communication, making linear, or even superlinear speedups possible.

Parallel performance is typically reported using a graph showing speedup versus the number of processors. Figure 1.3 shows a sample graph that displays fictional results for a pair of programs. The speedup of program A resembles a parallel algorithm like matrix addition that requires no communication between processors and therefore achieves nearly linear speedup. In contrast, program B’s speedup falls away from the ideal as the number of processors increases, as might occur in a matrix multiplication algorithm that requires communication.

Clarity

For the purposes of this dissertation, the clarity of a parallel program will refer to how clearly it represents the overall algorithm being expressed. For example, given that listings 1.2 and 1.3 both implement the SUMMA algorithm for matrix multiplication, how clear is each representation? Conversely, how much do the details of the parallel implementation interfere with a reader’s ability to understand an algorithm?

The importance of clarity is often brushed aside in favor of the all-consuming pursuit of performance. However, this is a mistake that should not be made. Clarity is perhaps the single most important factor that prevents more scientists and programmers from utilizing parallel computers today. Local-view libraries continue to be the predominant approach to parallel programming, yet their syntactic overheads are such that clarity is greatly compromised. This requires programmers to focus most of their attention on making the program work correctly rather than spending time implementing and improving their original algorithm. Ideally, parallel programming approaches should result in clear programs that can be readily understood.
Portability

A program’s portability is practically assured in the sequential computing world, primarily due to the universality of C and Fortran compilers. In the parallel world, portability is not as prevalent due to the extreme differences that exist between platforms. Parallel architectures vary widely not only between distinct machines, but also from one generation of a machine to the next. Memory may be organized as a single shared address space, a single distributed address space, or multiple distributed address spaces. Networks may be composed of buses, tori, hypercubes, sparse networks, or hierarchical combinations of these options. Communication paradigms may involve message passing, single-sided data transfers, or synchronization primitives over shared memory.

This multitude of architectural possibilities may be exposed by local-view approaches, making it difficult to implement a program that will run efficiently, if at all, from one machine to the next. Architectural differences also complicate the implementation of global-view compilers and libraries since they must run correctly and efficiently on all current parallel architectures, as well as those that may exist in the future.

Ideally, portability implies that a given program will behave consistently on all machines, regardless of their architectural features.

Generality

Generality simply refers to the ability of a parallel programming approach to express algorithms for varying types of problems. For example, a library which only supports matrix multiplication operations is not very general, and would not be very helpful for writing a parallel quicksort algorithm. Conversely, a global-view functional language might make it simple to write a parallel quicksort algorithm, but difficult to express the SUMMA matrix multiplication algorithm efficiently. Ideally, a parallel programming approach should be as general as possible.
Listing 1.4: Two matrix additions in C. Which one is better?

double A[m][n];
double B[m][n];
double C[m][n];

for (i=0; i<m; i++) {
    for (j=0; j<n; j++) {
        C[i][j] = A[i][j] + B[i][j];
    }
}

for (j=0; j<n; j++) {
    for (i=0; i<m; i++) {
        C[i][j] = A[i][j] + B[i][j];
    }
}

Performance Model

This dissertation defines a performance model as the means by which programmers understand the implementations of their programs. In this context, the performance model need not be a precise tool, but simply a means of weighing different implementation alternatives against one another.

As an example, C’s performance model indicates that the two loop nests in Listing 1.4 may perform differently in spite of the fact that they are semantically equivalent. C specifies that two-dimensional arrays are laid out in row-major order, and the memory models of modern machines indicate that accessing memory sequentially tends to be faster than accessing it in a strided manner. Using this information, a savvy C programmer will always choose to implement matrix addition using the first loop nest.

Note that C does not say how much slower the second loop nest will be. In fact, it does not even guarantee that the second loop nest will be slower. An optimizing compiler may reorder the loops to make them equivalent to the first loop nest. Or, hardware prefetching may detect the memory access pattern and successfully hide the memory latency normally
associated with strided array accesses. In the presence of these uncertainties, experienced C programmers will recognize that the first loop nest should be no worse than the second. Given the choice between the two approaches, they will choose the first implementation every time.

C’s performance model gives the programmer some idea of how C code will be compiled down to a machine’s hardware, even if the programmer is unfamiliar with specific details like the machine’s assembly language, its cache size, or its number of registers. In the same way, a parallel programmer should have some sense of how their code is being implemented on a parallel machine—for example, how the data and work are distributed between the processors, when communication takes place, what kind of communication it is, etc. Note that users of local-view languages and libraries have access to this information, because they specify it manually. Ideally, global-view languages and libraries should also give their users a parallel performance model with which different implementation alternatives can be compared and evaluated.

**This Dissertation**

This dissertation was designed to serve many different purposes. Naturally, its most important role is to describe the contributions that make up my doctoral research. With this goal in mind, I have worked to create a document that examines the complete range of effects that regions have had on the ZPL language, from their syntactic benefits to their implementation, and from their parallel implications to their ability to support advanced parallel computations. I also designed this dissertation to serve as documentation for many of my contributions to the ZPL compiler for use by future collaborators in the project. As such, some sections contain low-level implementation details that may not be of interest to those outside the ZPL community. Throughout the process of writing, my unifying concept has been to tell the story of regions as completely and accurately as I could in the time and space available.
In telling such a broad story, some of this dissertation’s contributions have been made as a joint effort between myself and other members of the ZPL project—most notably Sung-Eun Choi, Steven Deitz, E Christopher Lewis, Calvin Lin, Ton Ngo, and my advisor, Lawrence Snyder. In describing aspects of the project that were developed as a team, my intent is not to take credit for work that others have been involved in, but rather to make this treatment of regions as complete and seamless as possible.

The novel contributions of this dissertation include:

- A formal description and analysis of the region concept for expressing array computation, including support for replicated and privatized dimensions.

- A parallel interpretation of regions that admits syntax-based evaluation of a program’s communication requirements and concurrency.

- The design and implementation of a runtime representation of regions which enables parallel performance that compares favorably with hand-coded parallel programs.

- The design of the Ironman philosophy for supporting efficient paradigm-neutral communications, and an instantiation of the philosophy in the form of a point-to-point data transfer library.

- A means of parameterizing regions that supports the concise and efficient expression of hierarchical index sets and algorithms.

- Region-based support for sparse computation that permits the expression of sparse algorithms using dense syntax, and an implementation that supports general array operations, yet can be optimized to a compact form.

The chapters of this dissertation have a consistent organization. The bulk of each chapter describes its contributions. Most chapters contain an experimental evaluation of their
ideas along with a summary of previous work that is related to their contents. Each chapter concludes with a discussion section that addresses the strengths and weaknesses of its contributions, mentions side issues not covered in the chapter proper, and outlines possibilities for future work.

This dissertation is organized as follows. The next three chapters define and analyze the fundamental region concept. First, Chapter 2 describes the role of the region as a syntactic mechanism for sequential array-based programming, using ZPL as its context. Then, Chapter 3 explains the parallel implications of regions, detailing their use in defining ZPL’s performance model. The implementation of regions and of ZPL’s runtime libraries is covered in Chapter 4. The two chapters that follow each describe an extension to the basic region concept designed to support more advanced parallel algorithms. The notion of a parameterized region is defined in Chapter 5 and its use in implementing multigrid-style computations is detailed. Chapter 6 extends the region to support sparse sets of indices, and demonstrates its effectiveness in a number of sparse benchmarks.
SHARED MEMORY PROGRAMMING WITH OPENMP
OpenMP

- An API for shared-memory parallel programming.
- MP = multiprocessing
- Designed for systems in which each thread or process can potentially have access to all available memory.
- System is viewed as a collection of cores or CPU’s, all of which have access to main memory.
A shared memory system
Pragmas

- Special preprocessor instructions.
- Typically added to a system to allow behaviors that aren’t part of the basic C specification.
- Compilers that don’t support the pragmas ignore them.

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

void Hello(void); /* Thread function */

int main(int argc, char* argv[]) {
    /* Get number of threads from command line */
    int thread_count = strtol(argv[1], NULL, 10);

    #pragma omp parallel num_threads(thread_count)
    Hello();

    return 0;
} /* main */

void Hello(void) {
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    printf("Hello from thread %d of %d\n", my_rank, thread_count);
}
/* Hello */
gcc -g -Wall -fopenmp -o omp_hello omp_hello.c

./omp_hello 4

compiling
running with 4 threads

Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4

possible outcomes

Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4

Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
OpenMp pragmas

- # pragma omp parallel

  - Most basic parallel directive.
  - The number of threads that run the following structured block of code is determined by the run-time system.
A process forking and joining two threads
clause

- Text that modifies a directive.
- The num_threads clause can be added to a parallel directive.
- It allows the programmer to specify the number of threads that should execute the following block.

```
#pragma omp parallel num_threads (thread_count)
```
Of note...

- There may be system-defined limitations on the number of threads that a program can start.
- The OpenMP standard doesn’t guarantee that this will actually start thread_count threads.
- Most current systems can start hundreds or even thousands of threads.
- Unless we’re trying to start a lot of threads, we will almost always get the desired number of threads.
Some terminology

- In OpenMP parlance the collection of threads executing the parallel block — the original thread and the new threads — is called a team, the original thread is called the master, and the additional threads are called slaves.
In case the compiler doesn’t support OpenMP

```c
#include <omp.h>

#ifdef _OPENMP
#include <omp.h>
#endif
```
In case the compiler doesn’t support OpenMP

```c
#ifdef _OPENMP
    int my_rank = omp_get_thread_num ( );
    int thread_count = omp_get_num_threads ( );
#else
    int my_rank = 0;
    int thread_count = 1;
#endif
```
1) We identified two types of tasks:
   a) computation of the areas of individual trapezoids, and
   b) adding the areas of trapezoids.
2) There is no communication among the tasks in the first collection, but each task in the first collection communicates with task 1b.
3) We assumed that there would be many more trapezoids than cores.

- So we aggregated tasks by assigning a contiguous block of trapezoids to each thread (and a single thread to each core).
Assignment of trapezoids to threads
Unpredictable results when two (or more) threads attempt to simultaneously execute:

```
global_result += my_result;
```
Mutual exclusion

```c
#pragma omp critical
global_result += my_result;
```

only one thread can execute the following structured block at a time

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

void Trap(double a, double b, int n, double* global_result_p);

int main(int argc, char* argv[]) {
    double global_result = 0.0;    /* Store result in global_result */
    double a, b;                   /* Left and right endpoints */
    int n;                         /* Total number of trapezoids */
    int thread_count;

    thread_count = strtol(argv[1], NULL, 10);
    printf("Enter a, b, and n\n");
    scanf("%lf %lf %d", &a, &b, &n);
    #pragma omp parallel num_threads(thread_count)
    Trap(a, b, n, &global_result);

    printf("With n = %d trapezoids, our estimate\n", n);
    printf("of the integral from %f to %f = %.14e\n", a, b, global_result);
    return 0;
} /* main */
void Trap(double a, double b, int n, double* global_result_p) {
    double h, x, my_result;
    double local_a, local_b;
    int i, local_n;
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    h = (b—a)/n;
    local_n = n/thread_count;
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    my_result = (f(local_a) + f(local_b))/2.0;
    for (i = 1; i <= local_n—1; i++) {
        x = local_a + i*h;
        my_result += f(x);
    }
    my_result = my_result*h;

    #pragma omp critical
    *global_result_p += my_result;
}
*/ Trap */
SCOPE OF VARIABLES
Scope in OpenMP

- A variable that can be accessed by all the threads in the team has **shared** scope.

- A variable that can only be accessed by a single thread has **private** scope.

- The default scope for variables declared before a parallel block is **shared**.
THE REDUCTION CLAUSE
We need this more complex version to add each thread’s local calculation to get \texttt{global\_result}.

\begin{verbatim}
void Trap(double a, double b, int n, double* global_result_p);
\end{verbatim}

Although we’d prefer this.

\begin{verbatim}
double Trap(double a, double b, int n);

global_result = Trap(a, b, n);
\end{verbatim}
If we use this, there’s no critical section!

```c
double Local_trap(double a, double b, int n);
```

If we fix it like this…

```c
global_result = 0.0;
#pragma omp parallel num_threads(thread_count)
{
    #pragma omp critical
    global_result += Local_trap(double a, double b, int n);
}
```

… we force the threads to execute sequentially.
We can avoid this problem by declaring a private variable inside the parallel block and moving the critical section after the function call.

```c
global_result = 0.0;
#pragma omp parallel num_threads(thread_count)
{
    double my_result = 0.0; /* private */
    my_result += Local_trap(double a, double b, int n);
#pragma omp critical
    global_result += my_result;
}
```
Reduction operators

- A **reduction operator** is a binary operation (such as addition or multiplication).
- A **reduction** is a computation that repeatedly applies the same reduction operator to a sequence of operands in order to get a single result.
- All of the intermediate results of the operation should be stored in the same variable: the reduction variable.
A reduction clause can be added to a parallel directive.

```
reduction(<operator>: <variable list>)),
+,*,-, &, |, ^, &&, ||
```

```c
global_result = 0.0;
#pragma omp parallel num_threads(thread_count) \
    reduction(+: global_result)
global_result += Local_trap(double a, double b, int n);
```
THE "PARALLEL FOR" DIRECTIVE
Parallel for

- Forks a team of threads to execute the following structured block.
- However, the structured block following the parallel for directive must be a for loop.
- Furthermore, with the parallel for directive the system parallelizes the for loop by dividing the iterations of the loop among the threads.
\[ h = \frac{(b-a)}{n}; \]
\[ \text{approx} = \frac{(f(a) + f(b))}{2.0}; \]
\[ \text{for } (i = 1; i \leq n-1; i++) \]
\[ \quad \text{approx} += f(a + i*h); \]
\[ \text{approx} = h*\text{approx}; \]

```
# pragma omp parallel for num_threads(thread_count) \
   reduction(+: approx)
for (i = 1; i <= n-1; i++)
    approx += f(a + i*h);
approx = h*approx;
```
Legal forms for parallelizable for statements

\[
\begin{align*}
\text{for} & \quad \left\{ \begin{array}{l}
\text{index} = \text{start} ; \\
\text{index} \geq \text{end} ; \\
\text{index} > \text{end}
\end{array} \right. \\
& \quad \begin{array}{l}
\text{index} += \text{incr} \\
\text{index} -= \text{incr} \\
\text{index} = \text{index} + \text{incr} \\
\text{index} = \text{incr} + \text{index} \\
\text{index} = \text{index} - \text{incr}
\end{array}
\end{align*}
\]
Caveats

- The variable `index` must have integer or pointer type (e.g., it can’t be a float).

- The expressions `start`, `end`, and `incr` must have a compatible type. For example, if `index` is a pointer, then `incr` must have integer type.
Caveats

- The expressions `start`, `end`, and `incr` must not change during execution of the loop.

- During execution of the loop, the variable `index` can only be modified by the “increment expression” in the `for` statement.
Data dependencies

\[
\begin{align*}
  \text{fibo}[0] &= \text{fibo}[1] = 1; \\
  \text{for } (i = 2; i < n; i++) \\
  \quad \text{fibo}[i] &= \text{fibo}[i - 1] + \text{fibo}[i - 2]; \\
\end{align*}
\]

1 1 2 3 5 8 13 21 34 55

but sometimes we get this

1 1 2 3 5 8 0 0 0 0

this is correct

note 2 threads
Estimating $\pi$

\[
\pi = 4 \left[ 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots \right] = 4 \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}
\]

double factor = 1.0;
double sum = 0.0;
for (k = 0; k < n; k++) {
    sum += factor/(2*k+1);
    factor = -factor;
}
pi_approx = 4.0*sum;
OpenMP solution #1

double factor = 1.0;
double sum = 0.0;

#pragma omp parallel for num_threads(thread_count) \
  reduction(+:sum)
for (k = 0; k < n; k++) {
  sum += factor/(2*k+1);
  factor = -factor;
}
p_i_approx = 4.0*sum;

loop dependency
**OpenMP solution #2**

```c
double sum = 0.0;

#pragma omp parallel for num_threads(thread_count) \ reduction(+:sum) private(factor)
for (k = 0; k < n; k++) {
    if (k % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;
    sum += factor/(2*k+1);
}
```

Insures factor has private scope.
The default clause

- Lets the programmer specify the scope of each variable in a block.

  `default (none)`

- With this clause the compiler will require that we specify the scope of each variable we use in the block and that has been declared outside the block.
The default clause

double sum = 0.0;
#pragma omp parallel for num_threads(thread_count) \
  default(none) reduction(+:sum) private(k, factor) \
  shared(n)
for (k = 0; k < n; k++) {
  if (k % 2 == 0)
    factor = 1.0;
  else
    factor = -1.0;
  sum += factor/(2*k+1);
}
MORE ABOUT LOOPS IN OPENMP: SORTING
Bubble Sort

for (list_length = n; list_length >= 2; list_length--)
for (i = 0; i < list_length - 1; i++)
    if (a[i] > a[i+1]) {
        tmp = a[i];
        a[i] = a[i+1];
        a[i+1] = tmp;
    }
Serial Odd-Even Transposition Sort

```c
for (phase = 0; phase < n; phase++)
    if (phase % 2 == 0)
        for (i = 1; i < n; i += 2)
            if (a[i-1] > a[i]) Swap(&a[i-1], &a[i]);
    else
        for (i = 1; i < n-1; i += 2)
            if (a[i] > a[i+1]) Swap(&a[i], &a[i+1]);
```
### Serial Odd-Even Transposition Sort

<table>
<thead>
<tr>
<th>Phase</th>
<th>Subscript in Array</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>
```c
for (phase = 0; phase < n; phase++) {
    if (phase % 2 == 0)
        # pragma omp parallel for num_threads(thread_count) \
        default(none) shared(a, n) private(i, tmp)
        for (i = 1; i < n; i += 2) {
            if (a[i-1] > a[i]) {
                tmp = a[i-1];
                a[i-1] = a[i];
                a[i] = tmp;
            }
        }
    else
        # pragma omp parallel for num_threads(thread_count) \
        default(none) shared(a, n) private(i, tmp)
        for (i = 1; i < n-1; i += 2) {
            if (a[i] > a[i+1]) {
                tmp = a[i+1];
                a[i+1] = a[i];
                a[i] = tmp;
            }
        }
}
```
Second OpenMP Odd-Even Sort

```c
#pragma omp parallel num_threads(thread_count) \
 default(none) shared(a, n) private(i, tmp, phase)
for (phase = 0; phase < n; phase++) {
    if (phase % 2 == 0)
        #pragma omp for
        for (i = 1; i < n; i += 2) {
            if (a[i-1] > a[i]) {
                tmp = a[i-1];
                a[i-1] = a[i];
                a[i] = tmp;
            }
        }
    else
        #pragma omp for
        for (i = 1; i < n-1; i += 2) {
            if (a[i] > a[i+1]) {
                tmp = a[i+1];
                a[i+1] = a[i];
                a[i] = tmp;
            }
        }
}
```
Odd-even sort with two parallel for directives and two for directives.  
(Times are in seconds.)

<table>
<thead>
<tr>
<th>thread_count</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two parallel for directives</td>
<td>0.770</td>
<td>0.453</td>
<td>0.358</td>
<td>0.305</td>
</tr>
<tr>
<td>Two for directives</td>
<td>0.732</td>
<td>0.376</td>
<td>0.294</td>
<td>0.239</td>
</tr>
</tbody>
</table>
SCHEDULING LOOPS
We want to parallelize this loop.

```plaintext
sum = 0.0;
for (i = 0; i <= n; i++)
    sum += f(i);
```

<table>
<thead>
<tr>
<th>Thread</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0, ( n/t ), ( 2n/t ), ...</td>
</tr>
<tr>
<td>1</td>
<td>1, ( n/t + 1 ), ( 2n/t + 1 ), ...</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>( t - 1 )</td>
<td>( t - 1 ), ( n/t + t - 1 ), ( 2n/t + t - 1 ), ...</td>
</tr>
</tbody>
</table>

Assignment of work using cyclic partitioning.
double f(int i) {
    int j, start = i*(i+1)/2, finish = start + i;
    double return_val = 0.0;

    for (j = start; j <= finish; j++) {
        return_val += sin(j);
    }
    return return_val;
}

/* f */

Our definition of function $f$. 
Results

- f(i) calls the sin function \( i \) times.
- Assume the time to execute \( f(2i) \) requires approximately twice as much time as the time to execute \( f(i) \).

- \( n = 10,000 \)
  - one thread
  - run-time = 3.67 seconds.
Results

- $n = 10,000$
  - two threads
  - default assignment
  - run-time = 2.76 seconds
  - speedup = 1.33

- $n = 10,000$
  - two threads
  - cyclic assignment
  - run-time = 1.84 seconds
  - speedup = 1.99
The Schedule Clause

- Default schedule:

```c
sum = 0.0;
# pragma omp parallel for num_threads(thread_count) \
 reduction(+:sum)
for (i = 0; i <= n; i++)
    sum += f(i);
```

- Cyclic schedule:

```c
sum = 0.0;
# pragma omp parallel for num_threads(thread_count) \
 reduction(+:sum) schedule(static,1)
for (i = 0; i <= n; i++)
    sum += f(i);
```
PRODUCERS AND CONSUMERS
Queues

- Can be viewed as an abstraction of a line of customers waiting to pay for their groceries in a supermarket.

- A natural data structure to use in many multithreaded applications.

- For example, suppose we have several “producer” threads and several “consumer” threads.
  - Producer threads might “produce” requests for data.
  - Consumer threads might “consume” the request by finding or generating the requested data.
Message-Passing

- Each thread could have a shared message queue, and when one thread wants to "send a message" to another thread, it could enqueue the message in the destination thread's queue.

- A thread could receive a message by dequeuing the message at the head of its message queue.
for (sent_msgs = 0; sent_msgs < send_max; sent_msgs++) {
    Send_msg();
    Try_receive();
}

while (!Done())
    Try_receive();
Sending Messages

```c
mesg = random();
dest = random() % thread_count;
#pragma omp critical
Enqueue(queue, dest, my_rank, mesg);
```
Receiving Messages

if (queue_size == 0) return;
else if (queue_size == 1)
    # pragma omp critical
    Dequeue(queue, &src, &mesg);
else
    Dequeue(queue, &src, &mesg);
Print_message(src, msg);

queue_size = enqueued - dequeued;
if (queue_size == 0 && done_sending == thread_count)
    return TRUE;
else
    return FALSE;

each thread increments this after completing its for loop
When the program begins execution, a single thread, the master thread, will get command line arguments and allocate an array of message queues: one for each thread.

This array needs to be shared among the threads, since any thread can send to any other thread, and hence any thread can enqueue a message in any of the queues.
The Atomic Directive (1)

- Unlike the critical directive, it can only protect critical sections that consist of a single C assignment statement.

  ```c
  # pragma omp atomic
  ```

- Further, the statement must have one of the following forms:

  ```c
  x <op>= <expression>;
  x++;
  ++x;
  x--;
  --x;
  ```
Critical Sections

- OpenMP provides the option of adding a name to a critical directive:

  ```
  # pragma omp critical(name)
  ```

- When we do this, two blocks protected with critical directives with different names can be executed simultaneously.

- However, the names are set during compilation, and we want a different critical section for each thread’s queue.
UNIT 4 - DISTRIBUTED MEMORY PROGRAMMING WITH MPI
Identifying MPI processes

- Common practice to identify processes by nonnegative integer ranks.

- $p$ processes are numbered $0, 1, 2, \ldots p-1$
Our first MPI program

```c
#include <stdio.h>
#include <string.h>  /* For strlen */
#include <mpi.h>     /* For MPI functions, etc */

const int MAX_STRING = 100;

int main(void) {
  char greeting[MAX_STRING];
  int comm_sz;  /* Number of processes */
  int my_rank;  /* My process rank */

  MPI_Init(NULL, NULL);
  MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  if (my_rank != 0) {
    sprintf(greeting, "Greetings from process %d of %d!",
            my_rank, comm_sz);
    MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
             MPI_COMM_WORLD);
  } else {
    printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
    for (int q = 1; q < comm_sz; q++) {
      MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
                0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      printf("%s\n", greeting);
    }
  }

  MPI_Finalize();
  return 0;
} /* main */
```
mpicc -g -Wall -o mpi_hello mpi_hello.c

wrapper script to compile

source file

produce debugging information

turns on all warnings

create this executable file name (as opposed to default a.out)
Execution

mpiexec -n <number of processes> <executable>

---

mpiexec -n 1 ./mpi_hello

run with 1 process

mpiexec -n 4 ./mpi_hello

run with 4 processes
Execution

mpiexec -n 1 ./mpi_hello
Greetings from process 0 of 1 !

mpiexec -n 4 ./mpi_hello
Greetings from process 0 of 4 !
Greetings from process 1 of 4 !
Greetings from process 2 of 4 !
Greetings from process 3 of 4 !
MPI Programs

- Written in C.
  - Has main.
  - Uses stdio.h, string.h, etc.
- Need to add mpi.h header file.
- Identifiers defined by MPI start with “MPI_”.
- First letter following underscore is uppercase.
  - For function names and MPI-defined types.
  - Helps to avoid confusion.
MPI Components

- **MPI_Init**
  - Tells MPI to do all the necessary setup.

  ```
  int MPI_Init(
    int* argc_p /* in/out */,
    char*** argv_p /* in/out */);
  ```

- **MPI_Finalize**
  - Tells MPI we’re done, so clean up anything allocated for this program.

  ```
  int MPI_Finalize(void);
  ```
Basic Outline

```c
#include <mpi.h>

int main(int argc, char* argv[]) {
    ...  
    /* No MPI calls before this */
    MPI_Init(&argc, &argv);
    ...  
    MPI_Finalize();
    /* No MPI calls after this */
    ...  
    return 0;
}
```
Communicators

- A collection of processes that can send messages to each other.
- MPI_Init defines a communicator that consists of all the processes created when the program is started.
- Called MPI_COMM_WORLD.
Communicators

```c
int MPI_Comm_size(
    MPI_Comm comm    /* in */,
    int*           comm_sz_p   /* out */);
```

*number of processes in the communicator*

```c
int MPI_Comm_rank(
    MPI_Comm comm    /* in */,
    int*           my_rank_p   /* out */);
```

*my rank*

*(the process making this call)*
SPMD

- Single-Program Multiple-Data
- We compile **one** program.
- Process 0 does something different.
  - Receives messages and prints them while the other processes do the work.

- The **if-else** construct makes our program SPMD.
int MPI_Send(
    void* msg_buf_p,  /* in */,
    int msg_size,     /* in */,
    MPI_Datatype msg_type,  /* in */,
    int dest,         /* in */,
    int tag,          /* in */,
    MPI_Comm communicator /* in */);

Communication
### Data types

<table>
<thead>
<tr>
<th>MPI datatype</th>
<th>C datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_LONG_LONG</td>
<td>signed long long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>unsigned long long int</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>float</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
</tr>
</tbody>
</table>
```c
int MPI_Recv(  
    void* msg_buf_p,  /* out */,
    int buf_size,    /* in */,
    MPI_Datatype buf_type, /* in */,
    int source,      /* in */,
    int tag,         /* in */,
    MPI_Comm communicator, /* in */,
    MPI_Status* status_p  /* out */);
```
Message matching

MPI_Send (send_buf_p, send_buf_sz, send_type, dest, send_tag, send_comm);

MPI_Send
src = q

MPI_Recv
dest = r

MPI_Recv (recv_buf_p, recv_buf_sz, recv_type, src, recv_tag, recv_comm, &status);
Receiving messages

- A receiver can get a message without knowing:
  - the amount of data in the message,
  - the sender of the message,
  - or the tag of the message.
status_p argument

MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag, recv_comm, &status);

MPI_Status* status;

status.MPI_SOURCE
status.MPI_TAG
status.MPI_ERROR
How much data am I receiving?

```c
int MPI_Get_count(
    MPI_Status* status_p  /* in */,
    MPI_Datatype  type    /* in */,
    int* count_p         /* out */);
```
Issues with send and receive

- Exact behavior is determined by the MPI implementation.
- MPI_Send may behave differently with regard to buffer size, cutoffs and blocking.
- MPI_Recv always blocks until a matching message is received.
- Know your implementation; don’t make assumptions!
TRAPEZOIDAL RULE IN MPI
The Trapezoidal Rule
The Trapezoidal Rule

Area of one trapezoid \( = \frac{h}{2}[f(x_i) + f(x_{i+1})] \)

\[
h = \frac{b - a}{n}
\]

\( x_0 = a, \ x_1 = a + h, \ x_2 = a + 2h, \ldots, \ x_{n-1} = a + (n - 1)h, \ x_n = b \)

Sum of trapezoid areas \( = h[f(x_0)/2 + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + f(x_n)/2] \)
One trapezoid
Pseudo-code for a serial program

/* Input: a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 0; i <= n-1; i++) {
    x_i = a + i*h;
    approx += f(x_i);
}
approx = h*approx;
Parallelizing the Trapezoidal Rule

1. Partition problem solution into tasks.
2. Identify communication channels between tasks.
3. Aggregate tasks into composite tasks.
4. Map composite tasks to cores.
1 Get a, b, n;
2 h = (b-a)/n;
3 local_n = n/comm_sz;
4 local_a = a + my_rank*local_n*h;
5 local_b = local_a + local_n*h;
6 local_integral = Trap(local_a, local_b, local_n, h);
7 if (my_rank != 0)
     Send local_integral to process 0;
8 else /* my_rank == 0 */
9     total_integral = local_integral;
10 for (proc = 1; proc < comm_sz; proc++) {
11     Receive local_integral from proc;
12     total_integral += local_integral;
13 }
14 }
15 if (my_rank == 0)
16 print result;
Tasks and communications for Trapezoidal Rule

- Compute area of trap 0
- Compute area of trap 1
- ... Compute area of trap \( n - 1 \)

Add areas
```c
int main(void) {
    int my_rank, comm_sz, n = 1024, local_n;
    double a = 0.0, b = 3.0, h, local_a, local_b;
    double local_int, total_int;
    int source;

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

    h = (b-a)/n;                 /* h is the same for all processes */
    local_n = n/comm_sz;       /* So is the number of trapezoids */

    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    local_int = Trap(local_a, local_b, local_n, h);

    if (my_rank != 0) {
        MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0,
                 MPI_COMM_WORLD);
    }
```
else {
    total_int = local_int;
    for (source = 1; source < comm_sz; source++) {
        MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
                  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        total_int += local_int;
    }
}

if (my_rank == 0) {
    printf("With n = %d trapezoids, our estimate\n", n);
    printf("of the integral from %f to %f = %.15e\n", a, b, total_int);
}
MPI_Finalize();
return 0;
} /* main */
double Trap(
    double left_endpt /* in */,
    double right_endpt /* in */,
    int trap_count /* in */,
    double base_len /* in */) {

double estimate, x;
int i;

estimate = (f(left_endpt) + f(right_endpt))/2.0;
for (i = 1; i <= trap_count - 1; i++) {
    x = left_endpt + i*base_len;
    estimate += f(x);
}
estimate = estimate*base_len;

return estimate;
} /* Trap */
Dealing with I/O

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

int main(void) {
    int my_rank, comm_sz;

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    printf("Proc %d of %d > Does anyone have a toothpick?\n", my_rank, comm_sz);

    MPI_Finalize();
    return 0;
} /* main */
Running with 6 processes

Proc 0 of 6 > Does anyone have a toothpick?
Proc 1 of 6 > Does anyone have a toothpick?
Proc 2 of 6 > Does anyone have a toothpick?
Proc 3 of 6 > Does anyone have a toothpick?
Proc 4 of 6 > Does anyone have a toothpick?
Proc 5 of 6 > Does anyone have a toothpick?

unpredictable output
Most MPI implementations only allow process 0 in MPI_COMM_WORLD access to stdin.

Process 0 must read the data (scanf) and send to the other processes.

```c
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

Get_data(my_rank, comm_sz, &a, &b, &n);

h = (b-a)/n;
...
```
Function for reading user input

```c
void Get_input(
    int my_rank /* in */,
    int comm_sz /* in */,
    double* a_p /* out */,
    double* b_p /* out */,
    int* n_p /* out */) {

    int dest;

    if (my_rank == 0) {
        printf("Enter a, b, and n\n");
        scanf("%lf %lf %d", a_p, b_p, n_p);
        for (dest = 1; dest < comm_sz; dest++) {
            MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
            MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
            MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
        }
    } else { /* my_rank != 0 */
        MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

} /* Get_input */
```
COLLECTIVE COMMUNICATION
1. In the first phase:
   (a) Process 1 sends to 0, 3 sends to 2, 5 sends to 4, and 7 sends to 6.
   (b) Processes 0, 2, 4, and 6 add in the received values.
   (c) Processes 2 and 6 send their new values to processes 0 and 4, respectively.
   (d) Processes 0 and 4 add the received values into their new values.

2. (a) Process 4 sends its newest value to process 0.
   (b) Process 0 adds the received value to its newest value.
A tree-structured global sum
An alternative tree-structured global sum
MPI_Reduce

```c
int MPI_Reduce(
    void* input_data_p  /* in */,
    void* output_data_p /* out */,
    int count /* in */,
    MPI_Datatype datatype /* in */,
    MPI_Op operator /* in */,
    int dest_process /* in */,
    MPI_Comm comm /* in */);

MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0,
            MPI_COMM_WORLD);

double local_x[N], sum[N];
...
MPI_Reduce(local_x, sum, N, MPI_DOUBLE, MPI_SUM, 0,
           MPI_COMM_WORLD);
```
Predefined reduction operators in MPI

<table>
<thead>
<tr>
<th>Operation Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>Maximum</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Minimum</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>Sum</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Product</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>Logical and</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>Bitwise and</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Logical or</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>Bitwise or</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>Logical exclusive or</td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>Bitwise exclusive or</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>Maximum and location of maximum</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>Minimum and location of minimum</td>
</tr>
</tbody>
</table>
Collective vs. Point-to-Point Communications

- **All** the processes in the communicator must call the same collective function.

- For example, a program that attempts to match a call to `MPI_Reduce` on one process with a call to `MPI_Recv` on another process is erroneous, and, in all likelihood, the program will hang or crash.
Collective vs. Point-to-Point Communications

- The arguments passed by each process to an MPI collective communication must be “compatible.”

- For example, if one process passes in 0 as the `dest_process` and another passes in 1, then the outcome of a call to `MPI_Reduce` is erroneous, and, once again, the program is likely to hang or crash.
Collective vs. Point-to-Point Communications

- The output_data_p argument is only used on dest_process.

- However, all of the processes still need to pass in an actual argument corresponding to output_data_p, even if it’s just NULL.
Collective vs. Point-to-Point Communications

- Point-to-point communications are matched on the basis of tags and communicators.

- Collective communications don’t use tags.

- They’re matched solely on the basis of the communicator and the order in which they’re called.
Example (1)

<table>
<thead>
<tr>
<th>Time</th>
<th>Process 0</th>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>a = 1; c = 2</code></td>
<td><code>a = 1; c = 2</code></td>
<td><code>a = 1; c = 2</code></td>
</tr>
<tr>
<td>1</td>
<td><code>MPI_Reduce(&amp;a, &amp;b, ...)</code></td>
<td><code>MPI_Reduce(&amp;c, &amp;d, ...)</code></td>
<td><code>MPI_Reduce(&amp;a, &amp;b, ...)</code></td>
</tr>
<tr>
<td>2</td>
<td><code>MPI_Reduce(&amp;c, &amp;d, ...)</code></td>
<td><code>MPI_Reduce(&amp;a, &amp;b, ...)</code></td>
<td><code>MPI_Reduce(&amp;c, &amp;d, ...)</code></td>
</tr>
</tbody>
</table>

Multiple calls to MPI_Reduce
Example (2)

- Suppose that each process calls `MPI_Reduce` with operator `MPI_SUM`, and destination process 0.

- At first glance, it might seem that after the two calls to `MPI_Reduce`, the value of b will be 3, and the value of d will be 6.
Example (3)

- However, the names of the memory locations are irrelevant to the matching of the calls to `MPI_Reduce`.

- The order of the calls will determine the matching so the value stored in b will be $1+2+1 = 4$, and the value stored in d will be $2+1+2 = 5$. 
MPI_Allreduce

- Useful in a situation in which all of the processes need the result of a global sum in order to complete some larger computation.

```c
int MPI_Allreduce(
    void* input_data_p /* in */,
    void* output_data_p /* out */,
    int count /* in */,
    MPI_Datatype datatype /* in */,
    MPI_Op operator /* in */,
    MPI_Comm comm /* in */);
```
A global sum followed by distribution of the result.
A butterfly-structured global sum.
Broadcast

- Data belonging to a single process is sent to all of the processes in the communicator.

```c
int MPI_Bcast(
    void* data_p,   /* in/out */
    int count,      /* in */
    MPI_Datatype datatype, /* in */
    int source_proc, /* in */
    MPI_Comm comm,  /* in */
);```

A tree-structured broadcast.
A version of `Get_input` that uses `MPI_Bcast`

```c
void Get_input(
    int     my_rank  /* in */,
    int     comm_sz  /* in */,
    double* a_p      /* out */,
    double* b_p      /* out */,
    int*    n_p      /* out */) {

    if (my_rank == 0) {
        printf("Enter a, b, and n\n");
        scanf("%lf %lf %d", a_p, b_p, n_p);
    }
    MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
} /* Get_input */
Data distributions

\[
\begin{align*}
x + y &= (x_0, x_1, \ldots, x_{n-1}) + (y_0, y_1, \ldots, y_{n-1}) \\
&= (x_0 + y_0, x_1 + y_1, \ldots, x_{n-1} + y_{n-1}) \\
&= (z_0, z_1, \ldots, z_{n-1}) \\
&= z
\end{align*}
\]

Compute a vector sum.
Serial implementation of vector addition

```c
void Vector_sum(double x[], double y[], double z[], int n) {
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];
} /* Vector_sum */
```
Different partitions of a 12-component vector among 3 processes

<table>
<thead>
<tr>
<th>Process</th>
<th>Block</th>
<th>Cyclic</th>
<th>Block-cyclic Blocksize = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 1 2 3</td>
<td>0 3 6 9</td>
<td>0 1 6 7</td>
</tr>
<tr>
<td>1</td>
<td>4 5 6 7</td>
<td>1 4 7 10</td>
<td>2 3 8 9</td>
</tr>
<tr>
<td>2</td>
<td>8 9 10 11</td>
<td>2 5 8 11</td>
<td>4 5 10 11</td>
</tr>
</tbody>
</table>
Partitioning options

- Block partitioning
  - Assign blocks of consecutive components to each process.

- Cyclic partitioning
  - Assign components in a round robin fashion.

- Block-cyclic partitioning
  - Use a cyclic distribution of blocks of components.
Parallel implementation of vector addition

```c
void Parallel_vector_sum(
    double local_x[] /* in */,
    double local_y[] /* in */,
    double local_z[] /* out */,
    int local_n /* in */) {
    int local_i;

    for (local_i = 0; local_i < local_n; local_i++)
        local_z[local_i] = local_x[local_i] + local_y[local_i];
} /* Parallel_vector_sum */
```
Scatter

- MPI_Scatter can be used in a function that reads in an entire vector on process 0 but only sends the needed components to each of the other processes.

```c
int MPI_Scatter(
    void* send_buf_p, /* in */,
    int send_count /* in */,
    MPI_Datatype send_type /* in */,
    void* recv_buf_p, /* out */,
    int recv_count /* in */,
    MPI_Datatype recv_type /* in */,
    int src_proc /* in */,
    MPI_Comm comm /* in */);
```
Reading and distributing a vector

```c
void Read_vector(
    double local_a[], /* out */,
    int local_n, /* in */,
    int n, /* in */,
    char vec_name[], /* in */,
    int my_rank, /* in */,
    MPI_Comm comm, /* in */
) {

    double* a = NULL;
    int i;

    if (my_rank == 0) {
        a = malloc(n*sizeof(double));
        printf("Enter the vector \n", vec_name);
        for (i = 0; i < n; i++)
            scanf("%lf", &a[i]);
        MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE, 0, comm);
        free(a);
    } else {
        MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE, 0, comm);
    }
} /* Read_vector */
```
Gather

- Collect all of the components of the vector onto process 0, and then process 0 can process all of the components.

```c
int MPI_Gather(
    void* send_buf_p /* in */,
    int send_count /* in */,
    MPI_Datatype send_type /* in */,
    void* recv_buf_p /* out */,
    int recv_count /* in */,
    MPI_Datatype recv_type /* in */,
    int dest_proc /* in */,
    MPI_Comm comm /* in */);
```
void Print_vector(
    double local_b[] /* in */,
    int local_n /* in */,
    int n /* in */,
    char title[] /* in */,
    int my_rank /* in */,
    MPI_Comm comm /* in */) {

    double* b = NULL;
    int i;

if (my_rank == 0) {
    b = malloc(n*sizeof(double));
    MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE,
               0, comm);
    printf("%s\n", title);
    for (i = 0; i < n; i++)
        printf("%f ", b[i]);
    printf("\n");
    free(b);
} else {
    MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE,
               0, comm);
}
} /* Print_vector */
Allgather

- Concatenates the contents of each process’ `send_buf_p` and stores this in each process’ `recv_buf_p`.
- As usual, `recv_count` is the amount of data being received from each process.

```c
int MPI_Allgather(
    void* send_buf_p  /* in */,
    int send_count    /* in */,
    MPI_Datatype send_type /* in */,
    void* recv_buf_p  /* out */,
    int recv_count    /* in */,
    MPI_Datatype recv_type /* in */,
    MPI_Comm comm     /* in */);
```
Matrix-vector multiplication

\[ A = (a_{ij}) \text{ is an } m \times n \text{ matrix} \]

\[ x \text{ is a vector with } n \text{ components} \]

\[ y = Ax \text{ is a vector with } m \text{ components} \]

\[ y_i = a_{i0}x_0 + a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{i,n-1}x_{n-1} \]

- \text{i-th component of } y
- \text{Dot product of the ith row of } A \text{ with } x.$
Matrix-vector multiplication

\[
\begin{array}{cccc}
  a_{00} & a_{01} & \cdots & a_{0,n-1} \\
  a_{10} & a_{11} & \cdots & a_{1,n-1} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\
\end{array}
\]

\[
\begin{array}{c}
  x_0 \\
  y_0 \\
  x_1 \\
  y_1 \\
  \vdots \\
  y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1} \\
  \vdots \\
  y_m = a_{m-1,0}x_0 + a_{m-1,1}x_1 + \cdots + a_{m-1,n-1}x_{n-1} \\
\end{array}
\]
Multiply a matrix by a vector

/* For each row of A */
for (i = 0; i < m; i++) {
   /* Form dot product of ith row with x */
   y[i] = 0.0;

   for (j = 0; j < n; j++)
      y[i] += A[i][j]*x[j];
}

Serial pseudo-code
C style arrays

\[
\begin{pmatrix}
0 & 1 & 2 & 3 \\
4 & 5 & 6 & 7 \\
8 & 9 & 10 & 11
\end{pmatrix}
\]

stored as

0 1 2 3 4 5 6 7 8 9 10 11
void Mat_vect_mult(
    double A[] /* in */,
    double x[] /* in */,
    double y[] /* out */,
    int m /* in */,
    int n /* in */) {

    int i, j;

    for (i = 0; i < m; i++) {
        y[i] = 0.0;
        for (j = 0; j < n; j++)
            y[i] += A[i*n+j]*x[j];
    }
} /* Mat_vect_mult */
void Mat_vect_mult(
    double    local_A[]   /* in */,
    double    local_x[]   /* in */,
    double    local_y[]   /* out */,
    int       local_m    /* in */,
    int       n           /* in */,
    int       local_n    /* in */,
    MPI_Comm  comm       /* in */) {

double* x;
int local_i, j;
int local_ok = 1;
An MPI matrix-vector multiplication function (2)

```c
x = malloc(n*sizeof(double));
MPI_Allgather(local_x, local_n, MPI_DOUBLE,
               x, local_n, MPI_DOUBLE, comm);

for (local_i = 0; local_i < local_m; local_i++) {
    local_y[local_i] = 0.0;
    for (j = 0; j < n; j++)
        local_y[local_i] += local_A[local_i*n+j]*x[j];
}
free(x);
/* Mat_vec_mult */
```
MPI DERIVED DATATYPES
Derived datatypes

- Used to represent any collection of data items in memory by storing both the types of the items and their relative locations in memory.

- The idea is that if a function that sends data knows this information about a collection of data items, it can collect the items from memory before they are sent.

- Similarly, a function that receives data can distribute the items into their correct destinations in memory when they're received.
Derived datatypes

- Formally, consists of a sequence of basic MPI data types together with a displacement for each of the data types.

- Trapezoidal Rule example:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>24</td>
</tr>
<tr>
<td>b</td>
<td>40</td>
</tr>
<tr>
<td>n</td>
<td>48</td>
</tr>
</tbody>
</table>

\{(\text{MPI\_DOUBLE}, 0), (\text{MPI\_DOUBLE}, 16), (\text{MPI\_INT}, 24)\}
MPI_Type create_struct

- Builds a derived datatype that consists of individual elements that have different basic types.

```c
int MPI_Type_create_struct(
    int count,           /* in */
    int array_of_blocklengths[], /* in */
    MPI_Aint array_of_displacements[], /* in */
    MPI_Datatype array_of_types[],    /* in */
    MPI_Datatype* new_type_p          /* out */) {
```
MPI_Get_address

- Returns the address of the memory location referenced by `location_p`.
- The special type `MPI_Aint` is an integer type that is big enough to store an address on the system.

```c
int MPI_Get_address(
    void* location_p /* in */,
    MPI_Aint* address_p /* out */);
```
MPI_Type_commit

- Allows the MPI implementation to optimize its internal representation of the datatype for use in communication functions.

```c
int MPI_Type_commit(MPI_Datatype* new_mpi_t_p /* in/out */);
```
**MPI_Type_free**

- When we’re finished with our new type, this frees any additional storage used.

```c
int MPI_Type_free(MPI_Datatype* old_mpi_t_p /* in/out */);
```
Get input function with a derived datatype (1)

```c
void Build_mpi_type(
    double* a_p, /* in */,
    double* b_p, /* in */,
    int* n_p, /* in */,
    MPI_Datatype* input_mpi_t_p /* out */) {

    int array_of_blocklengths[3] = {1, 1, 1};
    MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};
    MPI_Aint a_addr, b_addr, n_addr;
    MPI_Aint array_of_displacements[3] = {0};
```
Get input function with a derived datatype (2)

```c
MPI_Get_address(a_p, &a_addr);
MPI_Get_address(b_p, &b_addr);
MPI_Get_address(n_p, &n_addr);
array_of_displacements[1] = b_addr - a_addr;
MPI_Type_create_struct(3, array_of_blocklengths, 
                        array_of_displacements, array_of_types, 
                        input_mpi_t_p);
MPI_Type_commit(input_mpi_t_p);
/*/ Build_mpi_type */
```
void Get_input(int my_rank, int comm_sz, double* a_p, double* b_p, int* n_p) {
    MPI_Datatype input_mpi_t;

    Build_mpi_type(a_p, b_p, n_p, &input_mpi_t);

    if (my_rank == 0) {
        printf("Enter a, b, and n\n\n");
        scanf("%lf %lf %d", a_p, b_p, n_p);
    }
    MPI_Bcast(a_p, 1, input_mpi_t, 0, MPI_COMM_WORLD);

    MPI_Type_free(&input_mpi_t);
} /* Get_input */
PERFORMANCE EVALUATION
**Elapsed parallel time**

- Returns the number of seconds that have elapsed since some time in the past.

```c
double MPI_Wtime(void);

double start, finish;
...
start = MPI_Wtime();
/* Code to be timed */
...
finish = MPI_Wtime();
printf("Proc %d > Elapsed time = %e seconds\n"
    my_rank, finish-start);
```
In this case, you don’t need to link in the MPI libraries.

Returns time in microseconds elapsed from some point in the past.

```c
#include "timer.h"
...
double now;
...
GET_TIME(now);
```
#include "timer.h"

... 

**double** start, finish;

... GET_TIME(start);

/* Code to be timed */

... GET_TIME(finish);

printf("Elapsed time = %e seconds\n", finish−start);
MPI_Barrier

- Ensures that no process will return from calling it until every process in the communicator has started calling it.

```c
int MPI_Barrier(MPI_Comm comm /* in */);
```
double local_start, local_finish, local_elapsed, elapsed;
...
MPI_Barrier(comm);
local_start = MPI_Wtime();
/* Code to be timed */
...
local_finish = MPI_Wtime();
local_elapsed = local_finish - local_start;
MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE, 
            MPI_MAX, 0, comm);

if (my_rank == 0)
    printf("Elapsed time = %e seconds\n", elapsed);
Run-times of serial and parallel matrix-vector multiplication

<table>
<thead>
<tr>
<th>comm_sz</th>
<th>Order of Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1024</td>
</tr>
<tr>
<td>1</td>
<td>4.1</td>
</tr>
<tr>
<td>2</td>
<td>2.3</td>
</tr>
<tr>
<td>4</td>
<td>2.0</td>
</tr>
<tr>
<td>8</td>
<td>1.7</td>
</tr>
<tr>
<td>16</td>
<td>1.7</td>
</tr>
</tbody>
</table>

(Seconds)
Speedup

\[ S(n, p) = \frac{T_{\text{serial}}(n)}{T_{\text{parallel}}(n, p)} \]
Efficiency

\[ E(n, p) = \frac{S(n, p)}{p} = \frac{T_{\text{serial}}(n)}{p \times T_{\text{parallel}}(n, p)} \]
## Speedups of Parallel Matrix-Vector Multiplication

<table>
<thead>
<tr>
<th>comm_sz</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
<th>8192</th>
<th>16,384</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>1.8</td>
<td>1.9</td>
<td>1.9</td>
<td>1.9</td>
<td>2.0</td>
</tr>
<tr>
<td>4</td>
<td>2.1</td>
<td>3.1</td>
<td>3.6</td>
<td>3.9</td>
<td>3.9</td>
</tr>
<tr>
<td>8</td>
<td>2.4</td>
<td>4.8</td>
<td>6.5</td>
<td>7.5</td>
<td>7.9</td>
</tr>
<tr>
<td>16</td>
<td>2.4</td>
<td>6.2</td>
<td>10.8</td>
<td>14.2</td>
<td>15.5</td>
</tr>
</tbody>
</table>
### Efficiencies of Parallel Matrix-Vector Multiplication

<table>
<thead>
<tr>
<th>comm_sz</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
<th>8192</th>
<th>16,384</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>0.89</td>
<td>0.94</td>
<td>0.97</td>
<td>0.96</td>
<td>0.98</td>
</tr>
<tr>
<td>4</td>
<td>0.51</td>
<td>0.78</td>
<td>0.89</td>
<td>0.96</td>
<td>0.98</td>
</tr>
<tr>
<td>8</td>
<td>0.30</td>
<td>0.61</td>
<td>0.82</td>
<td>0.94</td>
<td>0.98</td>
</tr>
<tr>
<td>16</td>
<td>0.15</td>
<td>0.39</td>
<td>0.68</td>
<td>0.89</td>
<td>0.97</td>
</tr>
</tbody>
</table>
Scalability

- A program is **scalable** if the problem size can be increased at a rate so that the efficiency doesn’t decrease as the number of processes increase.
Scalability

- Programs that can maintain a constant efficiency without increasing the problem size are sometimes said to be **strongly scalable**.

- Programs that can maintain a constant efficiency if the problem size increases at the same rate as the number of processes are sometimes said to be **weakly scalable**.
CASE STUDIES: n-BODY SOLVERS

Introduction

Many physical phenomena directly or indirectly (when solving a discrete version of a continuous problem) involve, or can be simulated with particle systems, where each particle interacts with all other particles according to the laws of physics. Examples include the gravitational interaction among the stars in a galaxy or the Coulomb forces exerted by the atoms in a molecule. The challenge of efficiently carrying out the related calculations is generally known as the N-body problem.

Mathematically, the N-body problem can be formulated as

\[ U(x_0) = \sum_i F(x_0, x_i) \]  

(1)

where \( U(x_0) \) is a physical quantity at \( x_0 \) which can be obtained by summing the pairwise interactions \( F(x_0, x_i) \) over the particles of the system. For instance, assume a system of \( N \) particles, located at \( x_i \) and having a mass of \( m_i \). The gravitational force exerted on a particle \( x \) having a mass \( m \) is then expressed as

\[ F(x) = \sum_{i=1}^{N} \frac{Gm_i m}{|x - x_i|^3} \]  

(2)

where \( G \) is the gravitational constant.

The task of evaluating the function \( U(x_0) \) for all \( N \) particles using (1) requires \( O(n) \) operations for each particle, resulting in a total complexity of \( O(n^2) \). In this paper we will see how this complexity can be reduced to \( O(n \log n) \) or \( O(n) \) by using efficient methods to approximate the sum in the right hand term of (1), while still preserving such important physical properties as energy and momentum.

Example N-body problem: 10 million-star galaxy

The summation techniques presented in this paper may be applied to a wide range of N-body problems. However, in order to clarify it is beneficial to have an actual problem to apply the ideas on. In this case, I have chosen the problem of simulating the movements of the stars of a galaxy. Let’s assume that there are about \( N := 10 \) million starts in the galaxy, although this is clearly much less than “in the real world”. Furthermore, for the sake of clarity the simulation will be done in two dimensions.

In the model, each star has the following quantities associated with it:

- mass, \( m_i \)
- position, \( x_i \) (depends on time)

Newtonian (let’s not allow Einstein to mess things up) physics tells us that for each star

\[ \frac{\partial^2}{\partial x^2} x_j(t) = \sum_{i=1, i \neq i}^{N} \frac{Gm_i}{|x_j - x_i|^3} \]  

(3)

Then, for each timestep from time \( t_k \) to \( t_{k+1} := \Delta t + t_k \) we need to integrate the right hand term of equation (3) in order to obtain the change in position:
\[ \Delta x_j = \int_{[t_i,t_{i+1}]} \mathbf{F}(x_j(t)) dt dt \]  

(4)

where \[ F(x_j) = \sum_{i=1}^{N} Gm_i \frac{x_j - x_i}{|x_j - x_i|^3} \]  

(5)

(4) is a somewhat difficult integral equation, since \( x_j \) is present on both sides. Also, \( x_i \) is dependent on \( t \), which means we have a system of \( N \) coupled integral equations for each time step.

A discrete version of (4) (which can be obtained by making certain assumptions) has the general form

\[ \Delta x_j = \sum_{i=1}^{k} c_i \mathbf{F}(x_j(t + h_i)), \quad k < \infty \]  

(6)

and is thus a linear combination of the function \( \mathbf{F} \) evaluated at different time points; different discrete integration schemes yield different coefficients \( c_i \) and \( h_i \). A commonly used integrator is the so-called Leapfrog integration scheme.

We can now formulate an algorithm for the simulation:

1. Set initial positions
2. for each timestep \( \Delta t \) do
3. for each particle \( j \) do
4. evaluate \( \mathbf{F}(x_j(t)) \) at timepoints required by the integrator
5. use the integrator to calculate \( \Delta x_j \)
6. \( x_j(t + \Delta t) = x_j(t) + \Delta x \)
7. endfor
8. endfor

The function \( \mathbf{F} \) is of the form (1), and thus the N-body force calculation algorithms presented in this paper can be used to speed up step 4 of the algorithm.

**The Particle-Particle (PP) method**

The method of evaluating the right hand side of (1) directly is generally referred to as the Particle-Particle (PP) method\(^2\). This brute-force approach is clearly not feasible for large amount of particles due to the \( O(n^2) \) time requirement, but can effectively be used for small amounts of particles\(^3\). As this method does not approximate the sum, the accuracy equals machine precision.

**Tree codes**

Let \( a \) be the radius of the smallest disc, call it \( D \), so that the set of particles \( P := (x_{i1}, \ldots, x_{iN}) \) are inside the disc. Many physical systems have the property that the field \( U(x) \) generated by the the particle set \( P \) may be very complex inside \( D \), but smooth (“low on information content”) at some distance \( c \cdot a \) from \( D \). The gravitational force, for instance, has this property.\(^4\)

This observation is used in the so-called tree code-approach to the N-body problem: Clusters of particles at enough distance from the “target particle” \( x_0 \) of equation 1 are
given a computationally simpler representation in order to speed up summation. This approach can be illustrated by the following example: when calculating the gravitational force exerted by earth on an orbiting satellite, we do not try to sum the gravitational forces of all atoms that constitute planet earth. Instead we approximate the force with that of an infinitesimal particle, which have the same mass as earth and is located at earth’s center of mass.

![Figure 1 Approximating the set P](image)

The tree-code approach is effective because it can be applied recursively. For instance, in Figure 5, we might also be able to use a subset of $P' \subseteq P$ to approximate the force on the particle down to the left, as the radius $a'$ of the disc $D'$ enclosing $P'$ is less than $a$, and thus the minimal distance for approximation, $ca'$, is also smaller.

We need a data structure that supports the recursive nature of the idea, in 2D this is a *quadtree* and in 3D an *octree*. Each node in the quadtree is a square, and has four child nodes (unless it is a leaf node), representing a break-up into four smaller squares, $\frac{1}{4}$ the size of the original square (see Figure 2). An octree is built in a similar manner (each node has 8 children).

![Figure 2 Quadtree](image)

We now construct a quadtree, so that each leaf node contains only 1 particle. This is done by recursively subdividing the computational box; each node is further subdivided if it contains more than one particle.
Assuming the particles are not at arbitrarily small distances from each other (at least the machine precision sets a limit) a quadtree can be built in $O(n \min(b, \log n))$ time, where $b$ is the machine precision.

Now assume that the distance between a cluster and a particle must be at least the length of the side of the cluster box in order to obtain an accurate approximation. When calculating the force on a particle $x_0$, the tree is recursively traversed from the root. At each level, there may be no more than 9 boxes (the ones surrounding the box containing the particle) which need further subdivision, limiting the number of force calculations on the next level to 27 ($=2^6 + 3^3 - 9$, see Figure 4). Thus as each level a maximum of 27 $O(1)$ operations are performed. The depth of the tree is $\min(b, \log(n))$ yielding a total complexity (for all $N$ particles) of $O(n \min(b, \log n))$.

Figure 4 Quadtree nodes calculated at each level for particle $x$

Tree codes thus reduces the computational complexity from $O(n^2)$ to $O(n \log n)$ or $O(n)$ depending on your point of view - certainly a vast improvement! But as the
saying goes, there’s no such thing as a free lunch: tree codes are less accurate than simple PP, and require more auxiliary storage.

The Barnes-Hut algorithm: applying tree codes.

The Barnes-Hut algorithm is a good example of how to use tree codes in an algorithm. The algorithm was presented in [Barnes&Hut] in 1986, and is widely used in astrophysics. The algorithm has also been successfully parallized. This discussion of the algorithm is mainly based on [Demmel1], which describes Barnes-Hut for a 2D N-body system similar to our example.

The main idea is to approximate long-range forces by aggregating particles into one particle, and using the force exerted by this particle. A quadtree structure (or octree in 3D), as described in the previous section, is used to store the particles. The tree steps of the algorithm are

1. Build quadtree as described in section 0
2. Traverse the quadtree from the leaves to the root, computing center of mass and total mass for each parent node.
3. For each particle, traverse the tree from the root, calculating the force during the traversal

Step 2

Step 2 calculates the approximations for the long-range force. The approximation is made by considering several particles as one, with a position equal to the center of mass of the approximated particles, and a mass the sum of the approximated particles’ masses. More formally, to find the mass and position associated with a node N:

\[
\text{calculate_approximations}( N )
\]

if N is a leaf node
  return; // Node has a (real) particle => has mass & position
for all children n of N do
  calculate_approximations( n )
M := 0
cm := (0,0)
for all children n of N do
  M := M + mass of n
  cm := cm + mass of n * position of n
endfor
cm := 1/M * cm
mass of N := M
position of N := cm
end

Step 3

Consider the ratio,

\[
\theta = \frac{D}{r}
\]

where \( D \) is the size of the current node (call it A) “box” and \( r \) is the distance to the center of mass of another node (called B). If this ratio is sufficiently small, we can use the center of mass and mass of B to compute the force in A. If this is not the case, we need to go to the children of B and do the same test. Figure 4 shows this for \( \theta = 1.0 \); the numbers indicate the relative depth of the nodes. It can clearly be seen that further away from the particle, x, large nodes are used and closer to it smaller. The method is accurate to approximately 1% with \( \theta = 1.0 \). Expressed in pseudocode
treeForce(x, N)
    if N is a leaf or size(N)/|x-N|<θ
        return force(x,N)
    else
        F := 0
        for all children n of N do
            F := F + treeForce(x, n)
        endfor
        return F
    endif
end

Whereas we have discretized time, position is still continuous. The PM method goes one step further: it effectively discretizes position too\textsuperscript{12}. However, before we explore this idea further, we need the concept of potential.

Let us assume that there exists a quantity $\Phi$, which is related to the physical quantity $U$ we are studying according to:

$$U = \nabla \Phi$$

and, furthermore, that

$$\nabla \cdot U = c\rho$$

where $\rho$ is the density function (e.g. mass, charge) obtained from the particle distribution and $c$ is a constant. This leads to the Poisson equation\textsuperscript{13}

$$\nabla^2 \Phi = c\rho$$

In the sample N-body problem $U$ corresponds to the force, and $\Phi$ to the potential energy in the gravitational field generated by the particles. $\rho$ is the mass density (mass/area unit). In the continued discussion of the PM method the quantities used will be these.

The idea of the PM method is that we set up a mesh (grid) over the computational box, and then solve the potential (i.e. Poisson’s equation) at the meshpoints. Forces at the meshpoints can then be obtained by calculating the gradient of the potential. To find the force on a particle not located at a meshpoint we can either use the force at the nearest meshpoint, or interpolate the force from the closest meshpoints.\textsuperscript{14}

![Figure 5 Mesh density function assignment](image)

**Figure 5** Mesh density function assignment

**Creating the density function** $\rho$

There are several ways of assigning the particle masses to the density function\textsuperscript{15} $\rho$:
Nearest gridpoint (NGP): The mass of each particle \( m_i \) is assigned to the gridpoint closest to the mass. In Figure 5, this would mean assigning the entire mass of the particle to cell 2. NGP is also referred to as zero-order interpolation.

Cloud-in-Cell (CIC): The mass of each particle is weighted over the four (in 2D) closest cells; the weighting is proportional to the intersection of the “cloud” surrounding the particle and the cell. In Figure 5 almost all mass would be assigned to cell 2, then approximately the same amount to cells 1 and 4, and finally about 1/16 to cell 3. CIC implements first order (linear) interpolation.

Higher order interpolations: The “cloud” (weighting function) around the particle can be made to cover even more cells, resulting in higher order interpolations, e.g. TSC (triangular shaped cloud).

Calculating the potentials

Now that we have the density function \( \rho \), we can solve (10) in order to obtain the potential at the meshpoints. This is done by rewriting (10) as a system of discrete difference equations, and solving the system. The system can be solved in \( O(G \log G) \) time, where \( G \) is the number of gridpoints, by using the Fast Fourier Transform.\(^{16}\)

Finding the forces

Thus the potential is solved. Calculating the force is easy, we only need to calculate the gradient of the potential by taking the difference between two potential values. Some special considerations regarding the degree of force interpolation must however be taken in order to conserve such physical properties as momentum and energy.\(^{17}\)

PM pros and cons

The PM method has a complexity of \( O(n) \) with respect to the particles, the slowest steps being the FFT requiring \( O(G \log G) \) operations.\(^{18}\) The cost of the speed is the limited spatial resolution of PM - phenomena at a smaller scale than the mesh spacing are not modelled accurately, i.e. it is only suited for modelling of systems where collisions occur. On the other hand, large-scale phenomena can be shown quite accurately.\(^{19}\)

The Particle-Particle/Particle-Mesh algorithm (P³M)

The Particle-Particle/Particle-Mesh algorithm is a hybrid algorithm that strives to correct the shortcomings of PM when it comes to modelling of short-range phenomena. The following discussion is based on [Hockney]\(^{20}\) and applied to our sample N-body problem.

The general idea is to split up the gravitational force in short-range and long-range forces, just as we did in section 0. The PM method is used to determine the long range force, and PP to correct the PM force at short distances.

Let \( r_e \) be the distance (typically 2-3 times the size of the mesh cell), below which we want to use the PP method for force calculation. Denote by \( x_0 \) the position where the force is to be evaluated. In order to be able to efficiently find the particles \( x_i \) whose distance from \( x_0 \) is less than \( r_e \), we employ as so-called chaining mesh for the particles,
see Figure 6. As can be see from the figure, those \( x_i \) closer to \( x_0 \) than \( r_c \) must be in the neighbouring 8 chaining mesh cells of \( x_0 \)'s chaining mesh cell.

![Figure 6 Mesh cells and chaining mesh](image)

Mathematically, we define the correction, or short-range, force \( f^{sr} \) as

\[
f^{tot} = R + f^{sr} \Rightarrow f^{sr} = f^{tot} - R
\]

(11)

where \( f^{tot} \) is the total force and \( R \) is the force obtained using the PM method.

Suppose we have a particle \( x_i \) closer to than \( r_c \). We’d like to calculate the total force on \( x_0 \), using direct PP to correct the force \( R \) obtained by PM calculations. With some lengthy mathematical manipulations it can be shown that PM treats the particle \( x_i \) as a “mass cloud” (the density and shape of which depend on the details of the PM method). Thus, we can correct the PM force by subtracting the force induced by such a “mass cloud” (this force is included in the PM force) and then adding the direct PP interaction force. Mathematically, in the case of pointlike masses

\[
f^{sr} = Gmm \frac{x_0 - x_i}{|x_0 - x_i|^3} - e(x_0, x_i)
\]

(12)

where \( e(x_0, x_i) \) is the function for the gravitational force of the “mass cloud” the PM method treats \( x_i \) as. We can now formulate the P"M algorithm:

1. for each particle \( x \) do
2. find force on \( x =: F \) using the PM method
3. locate particles closer than \( r_c \) for PP calculation using the chaining mesh. Calculate short range force using these particles
4. force on \( x := F + \) short range forces
5. endfor

The P"M method has been used widely in cosmological simulations, and is easy to use when forces can be easily split into short and long-range (i.e. gravity). A problem is that the algorithm easily becomes dominated by the PP phase.\(^{21}\)

**Fast multipole method (FMM)**

Multipole expansion can be viewed as a tree code, with the following main differences
• Potential instead of force is used
• Multipole expansions of the potential of a box are more accurate than a simple center of mass substitution, but also computationally “heavier”. This is compensated by having more than 1 particle per tree leaf.\textsuperscript{22}
• FMM only require evaluation of $F(x_0, x_i)$ at leaf nodes, and possesses a “richer analytical structure”, meaning it can be adapted to a wider variety of problems\textsuperscript{23}

**Potentials and multipole expansion of the gravitational field in 2D**

The mathematics associated with FMM is somewhat lengthy, but not excessively advanced. This section will deal with multipole expansions of the gravitational field of our example N-body problem, similar methods are used when expanding other quantities and/or in other dimensionalities\textsuperscript{24}. The following presentation of the mathematical ideas is based on [Demmel2].

Recall that the potential of a particle satisfies Poisson’s equation (equation 10). A solution to (10) for a point mass located at $x_0$ in 2D is

$$\Phi(x) = \log(|x - x_0|)$$

(13)

Using the complex number $z=a+bi$ to represent the point $x=(a,b)$ the potential can be rewritten as the real part of the complex logarithm, which is analytic. Remembering that potentials are additive, the total potential from $n$ particles can be expressed as

$$\Phi(z) = \sum_{i=1}^{n} m_i \log(z - z_i) = \sum_{i=1}^{n} m_i (\log(z) + \log(1 - z_i / z))$$

(14)

Since $\Phi(z)$ is analytic, it is possible to do a Taylor expansion around origo of $\log(1-z_i/z)$, yielding

$$\Phi(z) = M \log(z) + \sum_{j=1}^{\infty} \alpha_j z^{-j}$$

(12)

where $M$ is the combined mass of all particles and

$$\alpha_j = \sum_{i=1}^{n} m_i z_i^{-j}$$

(13)

We approximate the potential $\Phi(z)$ by summing a finite number, $p$, of terms in (12). The error is then proportional to

$$\varepsilon = \left( \frac{\max(|z_i|)}{|z|} \right)^{p+1}$$

(14)

Now, suppose that all $z_i$ lie inside a DD square centered at origin and $z$ is evaluated outside a 3D-3D square centered at origin, then $\varepsilon \propto 2.12^p$ (see Figure 7). We say that $z$ and $z_i$ are well-separated when this condition holds. Also note that the potential is expressed as a power series, and the gradient can thus easily be computed analytically, avoiding further discretization errors (compare to PM!).
Expansion around a point $z_c$ instead of origo is similar. Denote by $\text{outer}(M, \alpha, \ldots, \alpha, z_c)$ this expansion.

Similarly, we can do an inner expansion which approximates the potential from particles outside a 3D box and is valid in a D box in the center of the 3D box. This is denoted by $\text{inner}(M, \beta, \ldots, \beta, z_c)$

Furthermore, we’ll need functions to translate the center of expansion for the inner and outer expansions. These functions are defined as

$$\text{outer}(M, \alpha, \ldots, \alpha, z_c') = \text{outer-shift}(\text{outer}(M, \alpha, \ldots, \alpha, z_c), z_c')$$

and

$$\text{inner}(M, \beta, \ldots, \beta, z_c') = \text{inner-shift}(\text{inner}(M, \beta, \ldots, \beta, z_c), z_c')$$

These functions can be implemented in $O(p^3)$

Finally a conversion function between outer and inner expansions is needed (i.e. when an outer expansion of a box A is valid in the region of another box B, the outer expansion of A can be converted to an inner expansion around B’s center). This can also be performed in $O(p^3)$ and is denoted with

$$\text{inner}(M, \beta, \ldots, \beta, z_c') = \text{convert}(\text{outer}(M, \alpha, \ldots, \alpha, z_c), z_c')$$

To further compress the notation, I’ll use only the cell as identifier for an expansion, e.g. the inner expansion of cell A centered at $z_c$ is written as:

$\text{inner}(A)$

**The FMM algorithm: Treecodes with a twist**

Now that we are done with the math, let’s see how all these expansions help us to build a fast and accurate force calculation algorithm. The discussion is based on [Demmel2]

We begin by constructing a quadtree, as in section 0, with the exception that subdivision is not continued until there is 1 particle per cell, but less particles than some limit $s$. Then the quadtree is traversed from the leaves toward the root, computing outer expansions for each quadtree node. Next, the quadtree is traversed from the root to the leaves, computing inner expansions for the nodes (using the outer expansions provided in the previous step). Now, the force on the particles can be calculated by
adding the inner expansion for the node (accounts for the potential for all particles well separated from the node) and the direct sum of potentials from particles in nearby (non well-separated) nodes. A more detailed description follows:

**Step 1: Building the quadtree**

For simplicity, we will assume that the tree is fully populated, that is each leave is at the same distance from the root (this can be achieved by augmenting the tree with empty leaves). A version of FMM using adaptive trees can be found in [Carrier].

**Step 2: Computing \textit{outer}() for each node**

Recall that we were able to move the center of expansion with the \textit{outer\_shift} function. Now we can move the center of expansion for the child cells \(B_1..B_4\) to the center of parent cell \(A (z_a)\), and then simply add the coefficients \(\alpha_i\) of the shifted expansions to obtain an outer expansion for cell \(A\). The expansions of \(B_1..B_4\) converge at distances larger than \(D\) from \(z_a\), and thus the criterion of convergence for the outer expansion of \(A\) is satisfied.

We start from the leaves, by calculating their outer expansions directly, and then proceed towards the root, “merging” outer expansions as described above. Note the similarity to the center of mass calculation step in the Barnes&Hut algorithm.

**Step 3: Computing \textit{inner}() for each node**

The idea is that we want to be able to calculate an expansion, which is valid inside a cell \(A\), by using the outer expansions of other cells. Since the outer expansion of a cell \(B_i\) only converges (with the desired accuracy) at a distance \(\geq 1.5D\) from the center of the cell, the immediate neighbour cells of \(A\) cannot be used to calculate the inner expansion. More formally, we define the interaction set \(I\) of \(A\), in which our particle resides:
I(A) := \{ \text{nodes } B_i \text{ such that } B_i \text{ is a child of a neighbour of parent}(A), \\
but B_i \text{ is not itself a neighbour of } A \}

The interaction set of A is pictured in Figure 9.

In this stage, we start traversing the tree from the root. At each level, we have an inner expansion which is generated from the interaction set of the previous level (the root does not have any interaction set, and thus it’s expansion from “previous levels” is empty). Now, by converting the outer expansions of the interaction set at the current level to inner expansions for A, summing them up, and finally adding the shifted inner expansion of A’s parent, we have obtained an inner expansion for A that includes the potential of all cells except the neighbours of A. If A is itself a parent node, we then recurse once more. More formally:

```
Build_inner(A)
  P := parent(A)
  inner(A) := EMPTY
  for all B_i \in I(A) do
    inner(A) := inner(A) + convert(outer(B_i), A)
  endfor
  inner(A) := inner(A) + inner_shift(inner(P), A)
  for all C := children of A do
    build_inner(C)
  endfor
end
```

Note the difference from Barnes-Hut: This step does not calculate the force on a single particle, rather the potential of an entire leaf cell.
A "binary search tree" (BST) or "ordered binary tree" is a type of binary tree where the nodes are arranged in order: for each node, all elements in its left subtree are less-or-equal to the node (<=), and all the elements in its right subtree are greater than the node (>). The tree shown above is a binary search tree -- the "root" node is a 5, and its left subtree nodes (1, 3, 4) are <= 5, and its right subtree nodes (6, 9) are > 5. Recursively, each of the subtrees must also obey the binary search tree constraint: in the (1, 3, 4) subtree, the 3 is the root, the 1 <= 3 and 4 > 3. Watch out for the exact wording in the problems -- a "binary search tree" is different from a "binary tree".

The nodes at the bottom edge of the tree have empty subtrees and are called "leaf" nodes (1, 4, 6) while the others are "internal" nodes (3, 5, 9).

**Binary Search Tree Niche**

Basically, binary search trees are fast at insert and lookup. The next section presents the code for these two algorithms. On average, a binary search tree algorithm can locate a node in an N node tree in order $\lg(N)$ time (log base 2). Therefore, binary search trees are good for "dictionary" problems where the code inserts and looks up information indexed by some key. The $\lg(N)$ behavior is the average case -- it's possible for a particular tree to be much slower depending on its shape.

**Strategy**

Some of the problems in this article use plain binary trees, and some use binary search trees. In any case, the problems concentrate on the combination of pointers and recursion. (See the articles linked above for pointer articles that do not emphasize recursion.)

For each problem, there are two things to understand...

- The node/pointer structure that makes up the tree and the code that manipulates it
- The algorithm, typically recursive, that iterates over the tree

When thinking about a binary tree problem, it's often a good idea to draw a few little trees to think about the various cases.
Typical Binary Tree Code in C/C++

As an introduction, we'll look at the code for the two most basic binary search tree operations -- lookup() and insert(). The code here works for C or C++.

In C or C++, the binary tree is built with a node type like this...

```c
struct node {
    int data;
    struct node* left;
    struct node* right;
}
```

Lookup()

Given a binary search tree and a "target" value, search the tree to see if it contains the target. The basic pattern of the lookup() code occurs in many recursive tree algorithms: deal with the base case where the tree is empty, deal with the current node, and then use recursion to deal with the subtrees. If the tree is a binary search tree, there is often some sort of less-than test on the node to decide if the recursion should go left or right.

```c
/*
Given a binary tree, return true if a node
with the target data is found in the tree. Recurs
don the tree, chooses the left or right
branch by comparing the target to each node.
*/
static int lookup(struct node* node, int target) {
    // 1. Base case == empty tree
    // in that case, the target is not found so return false
    if (node == NULL) {
        return(false);
    }
    else {
        // 2. see if found here
        if (target == node->data) return(true);
        else {
            // 3. otherwise recur down the correct subtree
            if (target < node->data) return(lookup(node->left, target));
            else return(lookup(node->right, target));
        }
    }
}
```

The lookup() algorithm could be written as a while-loop that iterates down the tree. Our version uses recursion to help prepare you for the problems below that require recursion.

Pointer Changing Code

There is a common problem with pointer intensive code: what if a function needs to change one of the pointer parameters passed to it? For example, the insert() function below may want to change the root pointer. In C and C++, one solution uses pointers-to-pointers (aka "reference parameters"). That's a fine technique, but here we will use the simpler technique that a function that wishes to change a pointer passed to it will return the new value of the pointer to the caller. The caller is responsible for using the new value. Suppose we have a change() function
that may change the the root, then a call to change() will look like this...

```c
// suppose the variable "root" points to the tree
root = change(root);
```

We take the value returned by change(), and use it as the new value for root. This construct is a little awkward, but it avoids using reference parameters which confuse some C and C++ programmers, and Java does not have reference parameters at all. This allows us to focus on the recursion instead of the pointer mechanics.

**Insert()**

Insert() -- given a binary search tree and a number, insert a new node with the given number into the tree in the correct place. The insert() code is similar to lookup(), but with the complication that it modifies the tree structure. As described above, insert() returns the new tree pointer to use to its caller. Calling insert() with the number 5 on this tree...

```
2
/ \
1 10
```

returns the tree...

```
2
/ \
1 10
/ \
5
```

The solution shown here introduces a newNode() helper function that builds a single node. The base-case/recursion structure is similar to the structure in lookup() -- each call checks for the NULL case, looks at the node at hand, and then recurs down the left or right subtree if needed.

```c
/*
Helper function that allocates a new node
with the given data and NULL left and right
pointers.
*/
struct node* NewNode(int data) {
  struct node* node = new(struct node);    // "new" is like "malloc"
  node->data = data;
  node->left = NULL;
  node->right = NULL;
  return(node);
}
```

```c
/*
Give a binary search tree and a number, inserts a new node
with the given number in the correct place in the tree.
Returns the new root pointer which the caller should
then use (the standard trick to avoid using reference
parameters).
*/
struct node* insert(struct node* node, int data) {
```
```c

// 1. If the tree is empty, return a new, single node
if (node == NULL) {
    return(newNode(data));
}
else {
    // 2. Otherwise, recur down the tree
    if (data <= node->data) node->left = insert(node->left, data);
    else node->right = insert(node->right, data);

    return(node); // return the (unchanged) node pointer
}
}
```

The shape of a binary tree depends very much on the order that the nodes are inserted. In particular, if the nodes are inserted in increasing order (1, 2, 3, 4), the tree nodes just grow to the right leading to a linked list shape where all the left pointers are NULL. A similar thing happens if the nodes are inserted in decreasing order (4, 3, 2, 1). The linked list shape defeats the \( \lg(N) \) performance. We will not address that issue here, instead focusing on pointers and recursion.

**Binary Tree Problems**

Here are 14 binary tree problems in increasing order of difficulty. Some of the problems operate on binary search trees (aka "ordered binary trees") while others work on plain binary trees with no special ordering. The next section, shows the solution code in C/C++. gives the background and solution code in Java. The basic structure and recursion of the solution code is the same in both languages -- the differences are superficial.

Reading about a data structure is a fine introduction, but at some point the only way to learn is to actually try to solve some problems starting with a blank sheet of paper. To get the most out of these problems, you should at least attempt to solve them before looking at the solution. Even if your solution is not quite right, you will be building up the right skills. With any pointer-based code, it's a good idea to make memory drawings of a a few simple cases to see how the algorithm should work.

**build123()**

This is a very basic problem with a little pointer manipulation. (You can skip this problem if you are already comfortable with pointers.) Write code that builds the following little 1-2-3 binary search tree...

```
2
/  \
1   3
```

Write the code in three different ways...

- a: by calling newNode() three times, and using three pointer variables
- b: by calling newNode() three times, and using only one pointer variable
- c: by calling insert() three times passing it the root pointer to build up the tree

(In Java, write a build123() method that operates on the receiver to change it to be the 1-2-3 tree with the given coding constraints.

```java
struct node* build123() {

size()
```
This problem demonstrates simple binary tree traversal. Given a binary tree, count the number of nodes in the tree.

```c
int size(struct node* node) {
}
```

**maxDepth()**

Given a binary tree, compute its "maxDepth" -- the number of nodes along the longest path from the root node down to the farthest leaf node. The maxDepth of the empty tree is 0, the maxDepth of the tree on the first page is 3.

```c
int maxDepth(struct node* node) {
}
```

**minValue()**

Given a non-empty binary search tree (an ordered binary tree), return the minimum data value found in that tree. Note that it is not necessary to search the entire tree. A maxValue() function is structurally very similar to this function. This can be solved with recursion or with a simple while loop.

```c
int minValue(struct node* node) {
}
```

**printTree()**

Given a binary search tree (aka an "ordered binary tree"), iterate over the nodes to print them out in increasing order. So the tree...

```
    4
   / \
  2   5
 / \
1   3
```

Produces the output "1 2 3 4 5". This is known as an "inorder" traversal of the tree.

**Hint:** For each node, the strategy is: recur left, print the node data, recur right.

```c
void printTree(struct node* node) {
}
```

**printPostorder()**

Given a binary tree, print out the nodes of the tree according to a bottom-up "postorder" traversal -- both subtrees of a node are printed out completely before the node itself is printed, and each left subtree is printed before the right subtree. So the tree...

```
    4
   / \
  2   5
 / \
1   3
```

Produces the output "1 3 2 5 4". The description is complex, but the code is simple. This is the sort of bottom-up traversal that would be used, for example, to evaluate an expression tree where a node is an operation like '+' and its subtrees are, recursively, the two subexpressions for the '+'.
void printPostorder(struct node* node) {

hasPathSum()

We'll define a "root-to-leaf path" to be a sequence of nodes in a tree starting with the root node and proceeding downward to a leaf (a node with no children). We'll say that an empty tree contains no root-to-leaf paths. So for example, the following tree has exactly four root-to-leaf paths:

```
      5
     / \
    4   8
   /   / \
  11  13  4
 / \    
7   2   1
```

Root-to-leaf paths:
- path 1: 5 4 11 7
- path 2: 5 4 11 2
- path 3: 5 8 13
- path 4: 5 8 4 1

For this problem, we will be concerned with the sum of the values of such a path -- for example, the sum of the values on the 5-4-11-7 path is 5 + 4 + 11 + 7 = 27.

Given a binary tree and a sum, return true if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum. Return false if no such path can be found. (Thanks to Owen Astrachan for suggesting this problem.)

int hasPathSum(struct node* node, int sum) {

printPaths()

Given a binary tree, print out all of its root-to-leaf paths as defined above. This problem is a little harder than it looks, since the "path so far" needs to be communicated between the recursive calls. **Hint:** In C, C++, and Java, probably the best solution is to create a recursive helper function printPathsRecur(node, int path[], int pathLen), where the path array communicates the sequence of nodes that led up to the current call. Alternately, the problem may be solved bottom-up, with each node returning its list of paths. This strategy works quite nicely in Lisp, since it can exploit the built in list and mapping primitives. (Thanks to Matthias Felleisen for suggesting this problem.)

Given a binary tree, print out all of its root-to-leaf paths, one per line.

void printPaths(struct node* node) {

mirror()

Change a tree so that the roles of the left and right pointers are swapped at every node.

So the tree...
```
  4
 / \
 2   5
 / \
```

The solution is short, but very recursive. As it happens, this can be accomplished without changing the root node pointer, so the return-the-new-root construct is not necessary. Alternately, if you do not want to change the tree nodes, you may construct and return a new mirror tree based on the original tree.

```c
void mirror(struct node* node) {
}
```

**doubleTree()**

For each node in a binary search tree, create a new duplicate node, and insert the duplicate as the left child of the original node. The resulting tree should still be a binary search tree.

So the tree...
```c
type
  / 
  1 3
```
is changed to...
```c
type
  / 
  2 3
  / 
  1 3
  / 
  1
```
As with the previous problem, this can be accomplished without changing the root node pointer.

```c
void doubleTree(struct node* node) {
}
```

**sameTree()**

Given two binary trees, return true if they are structurally identical -- they are made of nodes with the same values arranged in the same way. (Thanks to Julie Zelenski for suggesting this problem.)

```c
int sameTree(struct node* a, struct node* b) {
}
```

**countTrees()**

This is not a binary tree programming problem in the ordinary sense -- it's more of a math/combinatorics recursion problem that happens to use binary trees. (Thanks to Jerry Cain for suggesting this problem.)

Suppose you are building an N node binary search tree with the values 1..N. How many structurally different binary search trees are there that store those values? Write a recursive function that, given the number of distinct values, computes the number of structurally unique binary search trees that store those values. For example,
countTrees(4) should return 14, since there are 14 structurally unique binary search trees that store 1, 2, 3, and 4. The base case is easy, and the recursion is short but dense. Your code should not construct any actual trees; it's just a counting problem.

```c
int countTrees(int numKeys) {
```

**Binary Search Tree Checking (for problems 13 and 14)**

This background is used by the next two problems: Given a plain binary tree, examine the tree to determine if it meets the requirement to be a binary search tree. To be a binary search tree, for every node, all of the nodes in its left tree must be \( \leq \) the node, and all of the nodes in its right subtree must be \( > \) the node. Consider the following four examples...

a. \( 5 \) -> TRUE
   / \  
  2 7

b. \( 5 \) -> FALSE, because the 6 is not ok to the left of the 5
   / \  
  6 7

c. \( 5 \) -> TRUE
   / \  
  2 7
   /  
  1

d. \( 5 \) -> FALSE, the 6 is ok with the 2, but the 6 is not ok with the 5
   / \  
  2 7
   / \  
  1 6

For the first two cases, the right answer can be seen just by comparing each node to the two nodes immediately below it. However, the fourth case shows how checking the BST quality may depend on nodes which are several layers apart -- the 5 and the 6 in that case.

**isBST() -- version 1**

Suppose you have helper functions minValue() and maxValue() that return the min or max int value from a non-empty tree (see problem 3 above). Write an isBST() function that returns true if a tree is a binary search tree and false otherwise. Use the helper functions, and don’t forget to check every node in the tree. It’s ok if your solution is not very efficient. (Thanks to Owen Astrachan for the idea of having this problem, and comparing it to problem 14)

Returns true if a binary tree is a binary search tree.

```c
int isBST(struct node* node) {
```

**isBST() -- version 2**

```c
```
Version 1 above runs slowly since it traverses over some parts of the tree many times. A better solution looks at each node only once. The trick is to write a utility helper function isBSTRecurs(struct node* node, int min, int max) that traverses down the tree keeping track of the narrowing min and max allowed values as it goes, looking at each node only once. The initial values for min and max should be INT_MIN and INT_MAX -- they narrow from there.

```c
/*
Returns true if the given tree is a binary search tree
(efficient version).
*/
int isBST2(struct node* node) {
  return(isBSTRecurs(node, INT_MIN, INT_MAX));
}

/*
Returns true if the given tree is a BST and its
values are >= min and <= max.
*/
int isBSTRecurs(struct node* node, int min, int max) {
```

Tree-List

The Tree-List problem is one of the greatest recursive pointer problems ever devised, and it happens to use binary trees as well. CLibrary works through the Tree-List problem in detail and includes solution code in C and Java. The problem requires an understanding of binary trees, linked lists, recursion, and pointers. It's a great problem, but it's complex.

C/C++ Solutions

Make an attempt to solve each problem before looking at the solution -- it's the best way to learn.

Build123() Solution (C/C++)

```c
// call newNode() three times
struct node* build123a() {
  struct node* root = newNode(2);
  struct node* lChild = newNode(1);
  struct node* rChild = newNode(3);

  root->left = lChild;
  root->right = rChild;

  return(root);
}

// call newNode() three times, and use only one local variable
struct node* build123b() {
  struct node* root = newNode(2);
  root->left = newNode(1);
  root->right = newNode(3);

  return(root);
}
```
/*
 * Build 123 by calling insert() three times.
 * Note that the '2' must be inserted first.
 */
struct node* build123c() {
    struct node* root = NULL;
    root = insert(root, 2);
    root = insert(root, 1);
    root = insert(root, 3);
    return(root);
}

size() Solution (C/C++)

/*
 * Compute the number of nodes in a tree.
 */
int size(struct node* node) {
    if (node==NULL) {
        return(0);
    } else {
        return(size(node->left) + 1 + size(node->right));
    }
}

maxDepth() Solution (C/C++)

/*
 * Compute the "maxDepth" of a tree -- the number of nodes along
 * the longest path from the root node down to the farthest leaf node.
 */
int maxDepth(struct node* node) {
    if (node==NULL) {
        return(0);
    } else {
        // compute the depth of each subtree
        int lDepth = maxDepth(node->left);
        int rDepth = maxDepth(node->right);

        // use the larger one
        if (lDepth > rDepth) return(lDepth+1);
        else return(rDepth+1);
    }
}

minValue() Solution (C/C++)

/*
 * Given a non-empty binary search tree,
 * return the minimum data value found in that tree.
 *
Note that the entire tree does not need to be searched. */
int minValue(struct node* node) {
    struct node* current = node;

    // loop down to find the leftmost leaf
    while (current->left != NULL) {
        current = current->left;
    }

    return(current->data);
}

printTree() Solution (C/C++)

/*
Given a binary search tree, print out its data elements in increasing sorted order.
*/
void printTree(struct node* node) {
    if (node == NULL) return;

    printTree(node->left);
    printf("%d ", node->data);
    printTree(node->right);
}

printPostorder() Solution (C/C++)

/*
Given a binary tree, print its nodes according to the "bottom-up" postorder traversal.
*/
void printPostorder(struct node* node) {
    if (node == NULL) return;

    // first recur on both subtrees
    printTree(node->left);
    printTree(node->right);

    // then deal with the node
    printf("%d ", node->data);
}

hasPathSum() Solution (C/C++)

/*
Given a tree and a sum, return true if there is a path from the root down to a leaf, such that adding up all the values along the path equals the given sum.
*/
Strategy: subtract the node value from the sum when recurring down, and check to see if the sum is 0 when you run out of tree.

```c
int hasPathSum(struct node* node, int sum) {
    // return true if we run out of tree and sum==0
    if (node == NULL) {
        return(sum == 0);
    }
    else {
        // otherwise check both subtrees
        int subSum = sum - node->data;
        return(hasPathSum(node->left, subSum) ||
                hasPathSum(node->right, subSum));
    }
}
```

printPaths() Solution (C/C++)

/*
   Given a binary tree, print out all of its root-to-leaf paths, one per line. Uses a recursive helper to do the work.
*/
void printPaths(struct node* node) {
    int path[1000];
    printPathsRecur(node, path, 0);
}

/*
   Recursive helper function -- given a node, and an array containing the path from the root node up to but not including this node, print out all the root-leaf paths.
*/
void printPathsRecur(struct node* node, int path[], int pathLen) {
    if (node == NULL) return;

    // append this node to the path array
    path[pathLen] = node->data;
    pathLen++;

    // it's a leaf, so print the path that led to here
    if (node->left == NULL && node->right == NULL) {
        printArray(path, pathLen);
    }
    else {
        // otherwise try both subtrees
        printPathsRecur(node->left, path, pathLen);
        printPathsRecur(node->right, path, pathLen);
    }
}

// Utility that prints out an array on a line.
void printArray(int ints[], int len) {
    int i;
    for (i=0; i<len; i++) {

printf("%d ", ints[i]);
}
printf("\n");
}

mirror() Solution (C/C++)

/*
  Change a tree so that the roles of the
  left and right pointers are swapped at every node.

  So the tree...
       4
      / \
     2   5
    / \
   1   3

  is changed to...
       4
      / \
     5   2
    / \
   3   1
*/
void mirror(struct node* node) {
    if (node==NULL) {
        return;
    }
    else {
        struct node* temp;

        // do the subtrees
        mirror(node->left);
        mirror(node->right);

        // swap the pointers in this node
        temp = node->left;
        node->left = node->right;
        node->right = temp;
    }
}

doubleTree() Solution (C/C++)

/*
  For each node in a binary search tree,
  create a new duplicate node, and insert
  the duplicate as the left child of the original node.
  The resulting tree should still be a binary search tree.

  So the tree...
       2
      / \
  */
```c
void doubleTree(struct node* node) {
    struct node* oldLeft;
    if (node==NULL) return;
    // do the subtrees
    doubleTree(node->left);
    doubleTree(node->right);
    // duplicate this node to its left
    oldLeft = node->left;
    node->left = newNode(node->data);
    node->left->left = oldLeft;
}
```

```c
int sameTree(struct node* a, struct node* b) {
    // 1. both empty -> true
    if (a==NULL && b==NULL) return(true);
    // 2. both non-empty -> compare them
    else if (a!=NULL && b!=NULL) {
        return(
            a->data == b->data &&
            sameTree(a->left, b->left) &&
            sameTree(a->right, b->right)
        );
    }
    // 3. one empty, one not -> false
    else return(false);
}
```

```c
int countTrees(int numKeys) {
    int count = 0;
    // Count the number of structurally unique trees
    // For the key values 1...numKeys, how many structurally unique
    // trees can be constructed?
    return count;
}
```
binary search trees are possible that store those keys.

Strategy: consider that each value could be the root.
Recursively find the size of the left and right subtrees.

```
*/
int countTrees(int numKeys) {

    if (numKeys <=1) {
        return(1);
    }

    else {
        // there will be one value at the root, with whatever remains
        // on the left and right each forming their own subtrees.
        // Iterate through all the values that could be the root...
        int sum = 0;
        int left, right, root;

        for (root=1; root<=numKeys; root++) {
            left = countTrees(root - 1);
            right = countTrees(numKeys - root);

            // number of possible trees with this root == left*right
            sum += left*right;
        }

        return(sum);
    }
}
```

**isBST1() Solution (C/C++)

/*
 * Returns true if a binary tree is a binary search tree.
 */
int isBST(struct node* node) {
    if (node==NULL) return(true);

    // false if the min of the left is > than us
    if (node->left!=NULL && minValue(node->left) > node->data)
        return(false);

    // false if the max of the right is <= than us
    if (node->right!=NULL && maxValue(node->right) <= node->data)
        return(false);

    // false if, recursively, the left or right is not a BST
    if (!isBST(node->left) || !isBST(node->right))
        return(false);

    // passing all that, it's a BST
    return(true);
}
Message Passing Interface

The Message Passing Interface, MPI[12], is a controlled API standard for programming a wide array of parallel architectures. Though MPI was originally intended for classic distributed memory architectures, it is used on various architectures from networks of PCs via large shared memory systems, such as the SGI Origin 2000, to massive parallel architectures, such as Cray T3D and Intel paragon. The complete MPI API offers 186 operations, which makes this a rather complex programming API. However, most MPI applications use only six to ten of the available operations.

MPI is intended for the Single Program Multiple Data (SPMD) programming paradigm – all nodes run the same application-code. The SPMD paradigm is efficient and easy to use for a large set of scientific applications with a regular execution pattern. Other, less regular, applications are far less suited to this paradigm and implementation in MPI is tedious.

MPI's point-to-point communication comes in four shapes: standard, ready, synchronous and buffered. A standard-send operation does not return until the send buffer has been copied, either to another buffer below the MPI layer or to the network interface, (NIC). The ready-send operations are not initiated until the addressed process has initiated a corresponding receive-operation. The synchronous call sends the message, but does not return until the receiver has initiated a read of the message. The fourth model, the buffered send, copies the message to a buffer in the MPI-layer and then allows the application to continue. Each of the four models also comes in asynchronous (in MPI called non-blocking) modes. The non-blocking calls return immediately, and it is the programmer's responsibility to check that the send has completed before overwriting the buffer. Likewise a non-blocking receive exist, which returns immediately and the programmer needs to ensure that the receive operation has finished before using the data.

MPI supports both group broadcasting and global reductions. Being SPMD, all nodes have to meet at a group operation, i.e. a broadcast operation blocks until all the processes in the context have issued the broadcast operation. This is important because it turns all group-operations into synchronization points in the application. The MPI API also supports scatter-gather for easy exchange of large data-structures and virtual architecture topologies, which allow source-code compatible MPI applications to execute efficiently across different platforms.

Experiment Environment

Cluster

The cluster comprises 51 Dell Precision Workstation 360s, each with a 3.2GHz Intel Prescott processor, 2GB RAM and a 120GB Serial ATA hard-disk. The nodes are connected using Gigabit Ethernet over two HP Procurve 2848 switches. 32 nodes are connected to the first switch, and 19 nodes to the second switch. The two switches are trunked with 4 copper cables, providing 4Gbit/s bandwidth between the switches, see Figure 1. The nodes are running RedHat Linux 9 with a patched Linux 2.4.26 kernel to support Serial ATA. Hyperthreading is switched on, and Linux is configured for Symmetric Multiprocessor support.

---

1 The computers have a motherboard with Intel’s 875P chipset. The chipset supports Gigabit Ethernet over Intel’s CSA (Communication Streaming Architecture) bus, but Dell’s implementation of the motherboards use an Intel 82540EM Gigabit Ethernet controller connected to the PCI bus instead.
2 Trunking is a method where traffic between two switches is loadbalanced across a set of links in order to provide a higher available bandwidth between the switches.

MPICH

MPICH is the official reference implementation of MPI and has a high focus on being portable. MPICH is available for all UNIX flavors and for Windows, a special GRID enabled version, MPICH-G2, is available for Globus[11]. Many of the MPI implementations for specialized hardware, i.e. cluster interconnects, are based on MPICH. MPICH version 1.2.52 is used for the below experiments.

LAM-MPI

Local Area Multicomputer-MPI, LAM-MPI, started out as an implementation for running MPI applications on LANs. An integrated part of this model was ‘on-the-fly’ endian-conversion to support different architectures to collaborate on an MPI execution. While endian-conversion still is supported, it is no longer performed per default as it is assumed that most executions will be on homogenous clusters. The experiments in this paper are performed with LAM-MPI 7.0.5.

MESH-MPI

MESH-MPI is only just released and the presented results are thus brand-new. MESH-MPI is ‘yet-another-commercial-MPI’, but with a strong focus on performance, rather than simply improved support over the open-source versions. In addition to improved performance, MESH-MPI also promotes true non-blocking operations, thread safety, and scalable collective operations. Future versions have announced support for a special Low Latency Communication library (LLC) and a Runtime Data Dependency Analysis (RDDA) functionality to schedule communication. These functions are not available in the current version which is 1.0a.

Benchmarks

This section describes the benchmark suites we have chosen for examining the performance of the three MPI implementations. One suite, Pallas, is a micro-benchmark suite, which gives a lot of information about the performance of the different MPI functions, while the other, NPB, is an application/kernel suite, which describes the application level performance. The NPB suite originates from NASA and is used as the basis for deciding on new systems at NASA. This benchmark tests both the processing power of the system and the communication performance.
**Pallas Benchmark Suite**

The Pallas benchmark suite[9] from Pallas GmbH is a suite, which measures the performance of different MPI functions. The performance is measured for individual operations rather than on the application level. The results can thus be used in two ways; either to choose an MPI implementation that performs well for the operations one uses, or to determine which operations performs poorly on the available MPI implementation so that one can avoid them when coding applications. The tests/operations that are run in Pallas are:

- **PingPong**
  *The time it takes to pass a message between two processes and back*
- **PingPing**
  *The time it takes to send a message from one process to another*
- **SendRecv**
  *The time it takes to send and receive a message in parallel*
- **Exchange**
  *The time it takes to exchange contents of two buffers*
- **Allreduce**
  *The time it takes to create a common result, i.e. a global sum*
- **Reduce**
  *The same as Allreduce but the result is delivered to only one process*
- **Reduce Scatter**
  *The same as Reduce but the result is distributed amongst the processes*
- **Allgather**
  *The time it takes to collect partial results from all processes and deliver the data to all processes*
- **Allgatherv**
  *Same as Allgather, except that the partial results need not have the same size*
- **Alltoall**
  *The time it takes for all processes to send data to all other processes and receive from all other processes – the data that is sent is unique to each receiver*
- **Bcast** - *the time it takes to deliver a message to all processes*

**NAS Parallel Benchmark Suite**

The NAS Parallel Benchmark, NPB, suite is described as:

*The NAS Parallel Benchmarks (NPB) are a set of 8 programs designed to help evaluate the performance of parallel supercomputers. The benchmarks, which are derived from computational fluid dynamics (CFD) applications, consist of five kernels and three pseudo-applications. The NPB come in several flavors. NAS solicits performance results for each from all sources.* [10]

NPB is available for threaded, OpenMP and MPI systems and we naturally run the MPI version. NPB is available with five different data-sets, A through D, and W which is for workstations only. We use dataset C since D won’t fit on the cluster, and also since C is the most widely reported dataset.
The application kernels in NPB are:

- MG – Multigrid
- CG – Conjugate Gradient
- FT – Fast Fourier Transform
- IS – Integer Sort
- EP – Embarrassingly Parallel
- BT – Block Tridiagonal
- SP – Scalar Pentadiagonal
- LU – Lower Upper Gauss-Seidel

**Results**

In this section we present and analyze the results of running the benchmarks from section 3 on the systems described in section 2. All the Pallas benchmarks are run on 32 CPUs (they run on $2^3$ sized systems) as are the NPB benchmarks except BT and SP which are run on 36 CPUs (they run on $2^4$ sized systems).

**Pallas Benchmark Suite**

First in the Pallas benchmark is the point to point experiments, the extreme case is the concurrent Send andRecv experiments where MPICH uses more than 12 times longer than MESH-MPI, but otherwise all three are fairly close. MPICH performs worse than the other two and the commercial MESH-MPI loses only on the ping-ping experiment.

The seemingly large differences on ping-pong and ping-pong are not as significant as they may seem since they are the result of the interrupt throttling rate on the Intel Ethernet chipsets which – when set at the recommended 8000, discretises latencies in chunks of 125us, thus the difference between 62.5 us and 125us is not as significant as it may seem and would probably be much smaller on other Ethernet chipsets.

![Figure 2: point-to-point latencies from Pallas 0B messages.](image-url)
Switching to large messages, 4MB, the picture is more uniform and MPICH consistently looses to the other two. LAM-MPI and MESH-MPI are quite close in all these experiments and are running within 2% of each other. The only significant exception is in the ping-ping experiment where LAM-MPI outperforms MESH-MPI with 5%.

In the collective operations, the small data is tested on 8B (eight bytes) rather than 0B because 0B on group-operations are often not performed at all and resulting times are reported in the 0.05us range, thus to test the performance on small packages we use the size of a double precision number. The results are shown in Figure 4.

In the collective operations, the extreme case is Allgatherv using LAM-MPI which reports a whopping 4747us or 11 times longer than when using MESH-MPI. Except for the Alltoall benchmark where LAM-MPI is fastest, MESH-MPI is consistently the faster, and for most experiments, the advantage is significant, measured in multiples rather than percentages. The Bcast operation, which is a frequently used operation in many applications, shows MESH-MPI to be 7 times faster than MPICH and 12 times faster than LAM-MPI.
For large messages, the results have been placed in two Figures, 5 and 6, in order to fit the time-scale better. With the large messages, MESH-MPI is consistently better than both open-source candidates, ranging from nothing, -1%, to a lot: 11 times. On average MESH-MPI outperforms LAM-MPI with 4.6 times and MPICH with 4.3 times. MESH-MPI is on average 3.5 times faster than the best of the two open-source implementations.

![Figure 5: collective Operations latencies from Pallas 4MB messages.](image)

![Figure 6: collective Operations latencies from Pallas 4MB messages.](image)

**NPB Benchmark Suite**

While micro-benchmarks are interesting from an MPI perspective, users are primarily interested in the performance at application level. Here, according to Amdahl’s law, improvements are limited by the fraction of time spent on MPI operations. Thus the runtime of the NPB suite is particularly interesting, since it allows us to predict the value of running a commercial MPI, and it will even allow us to determine if the differences at the operation level performance can be seen at the application level.
The results are in favour of the commercial MPI; MESH-MPI finished the suite 14.5% faster than LAM and 37.1% faster than MPICH. Considering that these are real-world applications doing real work and taking Amdahl’s law into consideration, this is significant.

![Figure 7: runtime of the NPB benchmark](image)

If we break down the results in the individual applications the picture is a little less obvious and LAM-MPI actually outperforms MESH-MPI on two of the experiments; the FT by 3% and the LU by 6%. Both of these makes extensive use of the Alltoall operation where MESH-MPI has the biggest problems keeping up with LAM-MPI in the Pallas tests.

![Figure 8: runtime of the individual NPB benchmarks](image)