A Six Lecture Primer on Parallel Computing

Sauro Succi[1]
Bruce P. Ayati[2]
A. E. Hosoi[3]

[1]IBM European Center for Science and Engineering Computing, Rome, Italy. Present address: Institute for Computing Applications, National Research Council, Rome, Italy
[2]Computational and Applied Mathematics Program, The University of Chicago.
[3]Department of Physics, The University of Chicago.

Contents

Preface

This report came about as a result of a lecture series given at the University of Chicago by the first author in the Summer of 1995. The notes were compiled and critically revised by the last two authors.

One of the authors (SS) would like to kindly acknowledge financial support from the Computer Science Department, the Physics Department and the Material Research Science and Engineering Center of the University of Chicago. More specifically, he wishes to express his deep gratitude to Professors Todd Dupont, Leo Kadanoff and Robert Rosner for their warm and most enjoyable hospitality during his stay in Chicago.

This work made use of MRSEC Shared Facilities supported by the National Science Foundation under Award Number DMR-9400379 and was partially supported by ONR-AASERT N00014-94-1-0798.

Chapter 1

Basic Notions of Parallel Computing

1.1 Introduction

Computer simulations have opened up a third dimension in scientific investigation alongside theory and experimentation. This has sparked many reactions; some people believe that any problem can be solved with a computer, while others believe that computers are for those who are not good enough to do away with them. Fortunately, both of these groups are declining.

This book will take an intermediate point of view and address what one can reasonably expect from parallel computers today.

1.2 Motivation for Parallel Computing

The three primary goals in parallel computing are:

- Cut Turnaround Time – This is a common goal in industry: do more computations in less time without changing the size of the problem.
- Job Up-size – Common in academia where one would like to be able to look at the complex scenarios generated by the interactions of more degrees of freedom (e.g. fully developed turbulence).
- Both.

There are two factors that allow one to achieve these goals: S_1, the speed of a single processor and P, the number of processors or *granularity*. S_1 is typically measured in megaflops (millions of floating point operations per second). We would like to maximize the ratio between complexity and the time needed to

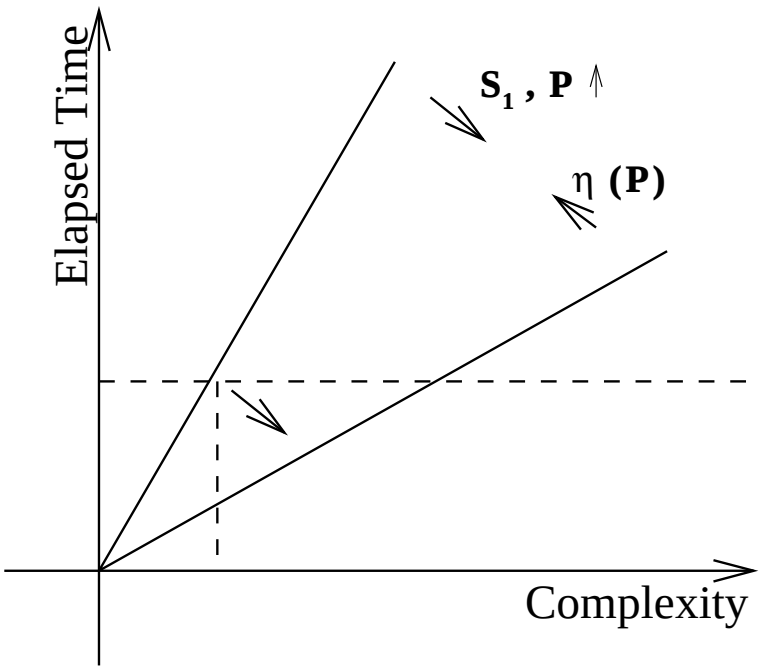

Figure 1.1: Processing speed $S(P)$ can be increased by either enhancing the single processor performance (higher S_1) and/or putting many processors at work together ($P > 1$). The result is shorter turnaround time at a given problem size (vertical dashed line) and/or larger problems solved at a given elapsed time (horizontal dashed line).

solve the problem. The complexity is the size (or rather an increasing function of the size) of the problem. The common aim of the three goals is to maximize the processing speed as defined by:

$$\mathrm{ProcessingSpeed} = \frac{\mathrm{Complexity}}{\mathrm{ElapsedTime}} = S_1 P \eta\,(P) \equiv S(P). \tag{1.1}$$

Traditionally, one would achieve this by cranking up the clock cycle, i.e. increase S_1. However, we are now reaching the technological limitations for this approach. It is generally believed that one cannot go beyond 10^9 flops/sec on a single chip. Alternatively one can bundle together a large number of processors and have them working concurrently on distinct sub-portions of the global problem, the "divide and conquer" approach.

Again, one cannot increase P indefinitely; there is always an interaction function which causes some loss of efficiency (factor η in eq. 1.1). One of the primary goals of theoretical parallel computing is to develop models which allow one to predict the type of efficiency one can obtain. This depends on many factors relating to both the details of the computational applications and

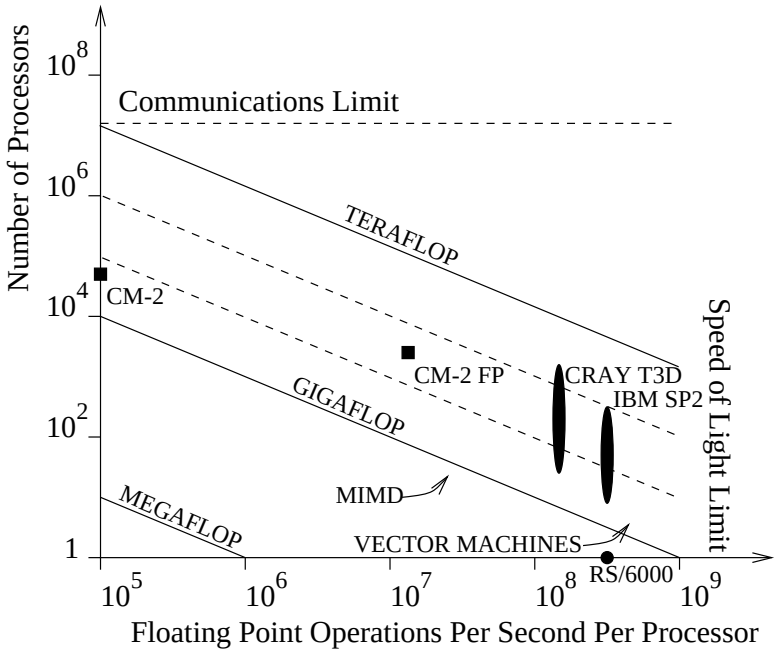

Figure 1.2: Evolution of Supercomputers. This figure is adapted from Boghosian.

the technological features of the computer architecture. However, there are scaling arguments which allow one to lump together many parameters and come out with analytic expressions.

1.3 Pushing the Frontiers of Supercomputing

Fig. 1.2 shows the state of the art today in terms of the two parameters S_1 and P. We have reached the technological boundary of 10^9 flops/sec/processor. We can bundle together many less powerful processors (y-axis). Again, if we go beyond a certain value ($\sim 10^5$), we are limited by $\eta(P)$, which would drop to zero for most real-life applications. Today, the intermediate approach of using a moderate number of moderately fast processors, called *Scalable Parallel Computing*, is the most popular. One can expect performance somewhere between a gigaflop and a teraflop.

1.3.1 Physical Limits

There are several physical factors which limit the performance of a single chip:

- Conduction, Dissipation and Transmission Losses.

- Off-chip communication.
- Propagation delays ($\tau_p \ll \tau_s$).

The lower value of τ_p, the propogation time, is limited by the speed of light. For various switching times, this gives

$\tau_s = 1\text{ns} \Longrightarrow l < 30\text{cm}$ 1 Gflop,
$\tau_s = 1\text{ps} \Longrightarrow l < 0.3\text{mm}$ 1 Tflop,

where τ_s switching time of a single transistor. Solid state physics has been striving to decrease τ_s. Today transistors can operate with a cut-off frequency in excess of 100G-Hz. This means one can switch the state of a single bit in 10ps. The problem occurs when crossing from one chip to another. There are parasitic currents that decrease the effective switching time by almost an order of magnitude (fig. 1.3). Also, one cannot make chips too small because of heat generation problems. For example, if one wants a switching time of 1ps, the chip cannot be bigger than 0.3mm (speed of light limit), too small for sufficient cooling. This is one of the most basic limitations, making the gigaflop a natural barrier due to propagation delays.

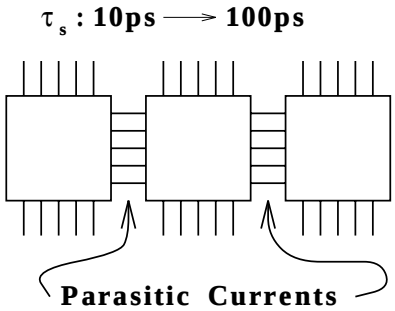

Figure 1.3: Physical Limits.

1.3.2 The "Killer" Micros

The situation today is that micro processors are catching up to traditional CRAY-like vector computers. From fig. 1.4 we see that in twenty years, the cycle time of the vector computers has decreased only by a factor of three. With microprocessor technology, there was a decrease in cycle time by a factor of four over five years. Some contend that the micros are killing off the vector machines. There is still a natural one nanosecond limit, so there is not much room for either processor to evolve on sheer hardware grounds.

1.3.3 Granularity

Typically, one classifies granularity as *low* (less than 10 processors), *medium* (10-100 processors), or *massive* (more than 100 processors). Fig. 1.5 should have

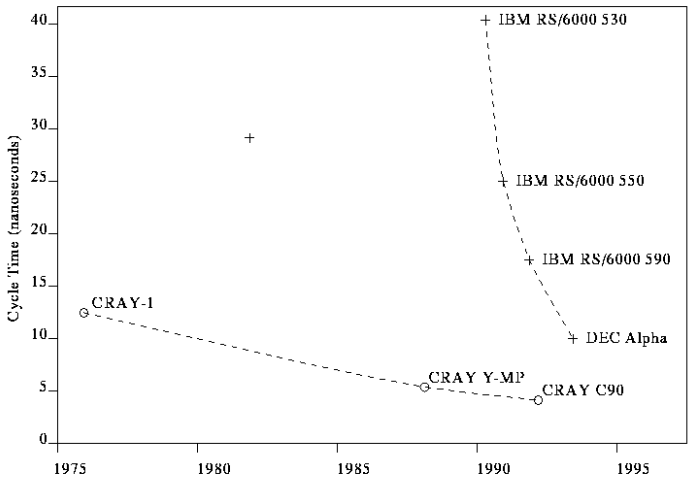

Figure 1.4: The "Killer" Micros. This graph shows the maximum configuration for each machine. Figure from Foster.

vertical error bars since each machine has several possible configurations. The IBM-SP2 can have between 8-512 processors.

1.3.4 Peak Performance

We see in figure 1.6 that S_1, the speed of a single processor, has grown exponentially. The graph is deceptive since it represents peak performance. The figure that matters is the sustained rate which depends on the program and the compilers. Compiler problems are a key issue!

1.3.5 Scalability

A problem is scalability if one receives "constant performance returns on processor investment." In other words, if one has twice as many processors, one would like to get twice as much power. Of course, scalability cannot last forever, since speed-up is given by:

$$\eta := \frac{T(1)}{PT(P)}; \qquad S = \eta P. \tag{1.2}$$

There is saturation at some point. This is illustrated by T, the amount of time

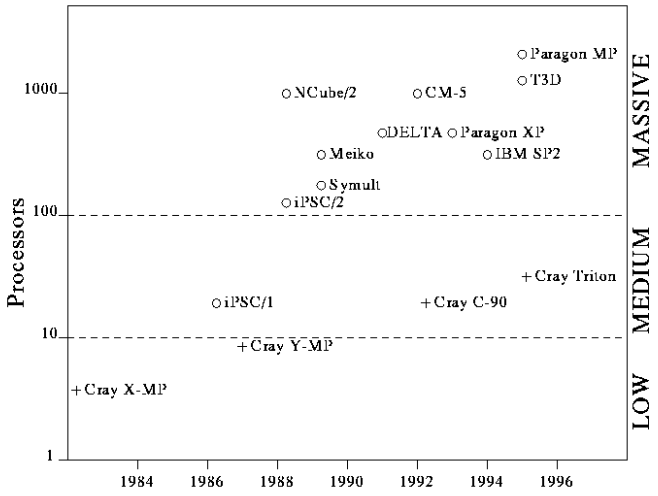

Figure 1.5: Granularity of Different Machines. Figure from Foster.

needed to solve a problem, approximately given by

$$T(P) = \frac{T(1)}{P} + T_{\rm com}(P) . \tag{1.3}$$

First, T depends on the number of processors, where each processor does one Pth of the work. However the different entities need to interact and exchange information, like a statistical mechanics n body problem. Calculation time goes down like P^{-1}. But communication time, at best, decays more slowly than P^{-1} since $T_{\rm com}$ is an increasing function of the number of processors. With an infinite number of processors, all time is spent exchanging information and nothing gets done. This is sometimes called "bad manager mode".

1.3.6 Formal Scaling Theory of Scalability

To illustrate scalability, we add another parameter to the efficiency; η now depends on the number of processors, P, and the size of the problem, N. Very often, to 0th order, we can recast η as

$$\eta(P, N) = \psi_{\rm U}(s_1, \cdots, s_M) . \tag{1.4}$$

ψ_U is a "universal" function which depends on a series, $M < N$, of non-dimensional parameters, the "Reynolds numbers of parallel computing". $P_{\rm ci}$ is the critical number of processors for the ith mechanism. $P_{\rm c}$ depends on the size of the problem. Very often, $P_{\rm c}$ can be expressed as a power law,

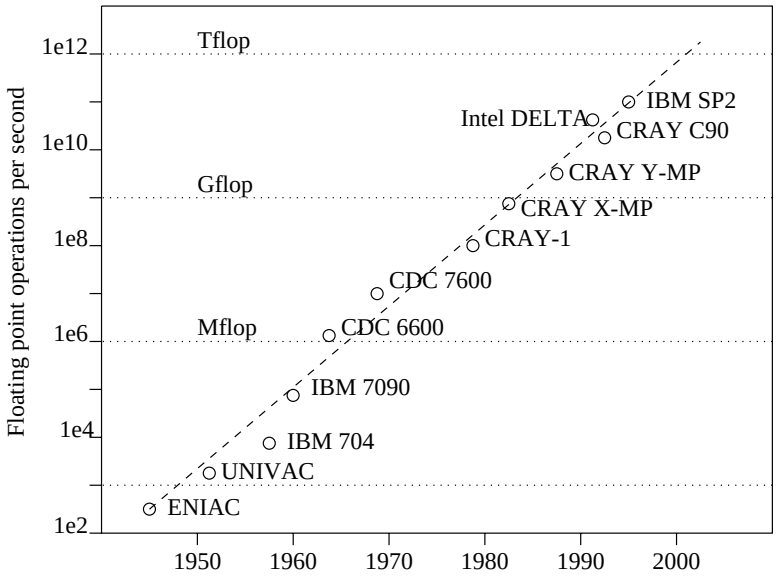

Figure 1.6: Peak Performance of a Single Processor. Figure from Foster.

$$s_i \equiv \frac{P}{P_{\rm c}\left(N\right)}, \qquad P_{\rm c}\left(N\right) \sim N^{\beta_i}. \tag{1.5}$$

The β_i are the "iso-scaling exponents". These are dimensionless quantities which measure the relative strengths between the different types of communication in the algorithm, and between communication and computation. $P_{\rm ci}$ is defined as the number of processors which gives a factor 2 loss in efficiency. For the kth kind of communication, communication is taking as much time as computation. Formally, ψ_U is required to fulfill the following relations:

$$\begin{aligned} l\psi_{\rm U}\left(s_1 \to 0, s_2 \to 0, \cdots s_M \to 0\right) &= 1 \\ \psi_{\rm U}\left(0, \cdots s_k = 1, \cdots 0\right) &= 1/2 \\ \psi_{\rm U}\left(1, \cdots 1, \cdots 1\right) &= 1/M \end{aligned} \tag{1.6}$$

The ideal situation is when $s_i = 0$ for all i, an extremely parallel application. Often,

$$\psi_{\rm U}\left(\mathbf{s}\right) \sim \frac{1}{1 + s_1^{\alpha_1} + \cdots s_M^{\alpha_M}}, \tag{1.7}$$

where α_i is the "scaling exponent", relating to the ith communication mechanism. Thus, small α's are desired.

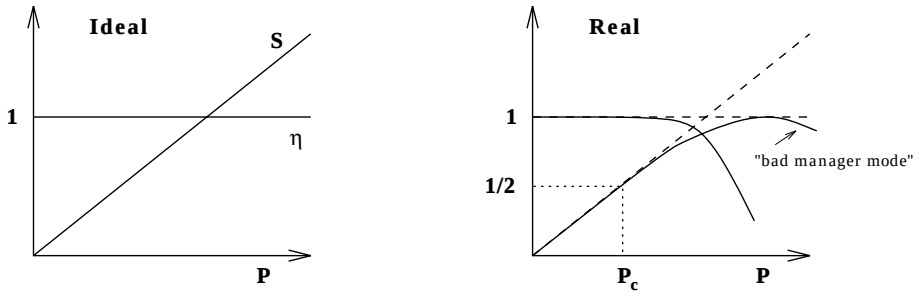

Figure 1.7: Scalability; speed-up and efficiency as a function of the number of processors.

1.3.7 Iso-scaling

We consider the physical interpretation [1]. There are different types of communication (local, global, etc.), and the fewer the better. When writing code for a parallel machine, always try get an estimate of P_c for your application. Three variables critical in the calculation of P_c are:

- M; number of overhead mechanisms (would like to decrease M).
- α_i; soft / hard loss of scalability (would like to decrease α).
- β_i; iso-scaling coefficients (would like to increase β)

The iso-scaling exponent can be interpreted as follows. $P = P_c(N) = N^\beta$ is the number of processors needed to keep $\eta =$ constant, as N increases. $N = N_c(P) = P^{1/\beta}$ is the problem size needed to keep $\eta =$ constant as P increases. For a given number of processors, stay far away from N^β! Also, $P^{1/\beta}$ is the *minimum* size of a problem to run given P processors, the *critical grain size*. To minimize power losses due to communication:

- Keep P *below* P_c for a given N.
- Keep N *above* N_c for a given P.

For regular large-scale lattice calculations, very often $P_c \sim 100$; for less regular geometries, involving irregular sparse matrix algebra, $P_c \sim 10 - 20$.

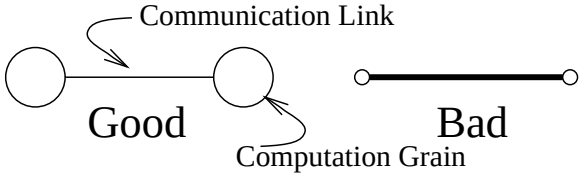

Figure 1.8: Communication (link) to Computation (circle) Ratio. ”Chunky” molecules make for optimal parallel efficiency.

1.3.8 Partitioning

Always consider the communication to computation ratio (fig 1.8). For example, in an N-body problem, we get N^2 communications but N^3 calculations. This is perfectly acceptable, considering only parallel efficiency.

1.4 Architecture Classification

There are four criteria for standard parallel architecture classification:

- Memory Organization
- Control
- Interconnection Topology
- Granularity

1.4.1 Control

There are four types of control options:

Single Instruction Single Data (SISD) . These are the simplest machines, also known as Von Neumann machines. Efficiency is increased by speeding up the processor. Memory stores both instruction and data.

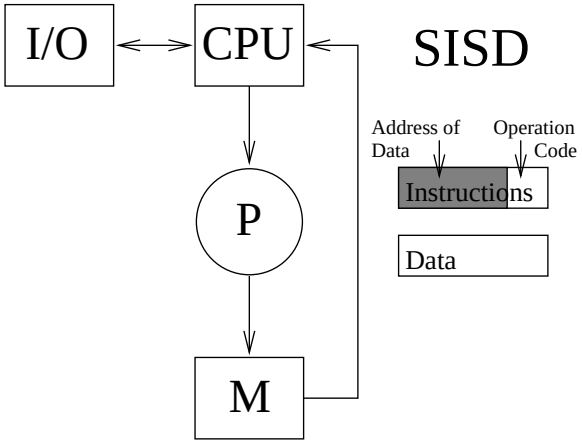

Figure 1.9: Single Instruction Single Data Architecture

Single Instruction Multiple Data (SIMD) . These machines have a single control unit with many processors, known as “Non Von”. Each processor acts on a different data set. This mimics the real world where there are

few physical laws, but a tremendous amount of data (10^{80} particles). Typically, each processor is not very powerful ($\sim$ 1 Mflop), but there are many of them (~ 64k). This is perfectly suited to regular (lattice) calcula-

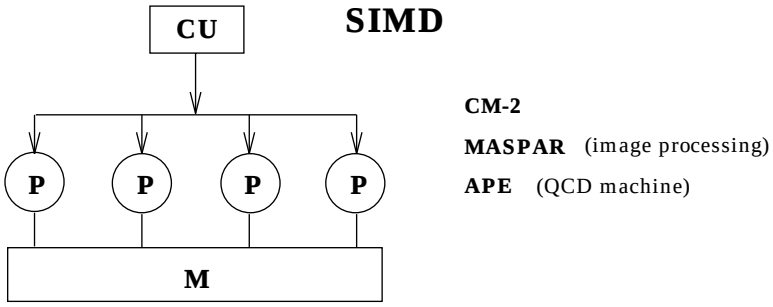

Figure 1.10: Single Instruction Multiple Data Architecture.

tions. Typical difficulties arise when processors must do different tasks. For example boundary conditions where something different is done on the interior). Keeping all processors busy at the same time may be difficult.

Multiple Instruction Single Data (MISD) . These machines are essentially nonexistent. As mentioned before, there are generally very few instructions and lots of data so it makes no sense to parallelize the instructions. However,

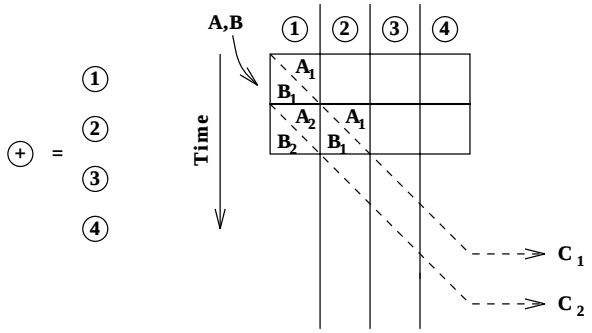

Figure 1.11: The Principle of Pipelining.

one can view vector computers as MISD; these computers work on the principle of a pipeline. For example an elementary operation like

$$C(i) = A(i) \oplus B(i) \tag{1.8}$$

is actually a four stage operation on a vector machine. The first operation is done on A_1 and B_1. Then the second operation is done on A_1 and B_1 while the first operation is done on A_2 and B_2. So after a delay of four

cycles, you get one result every cycle. The best increase in speed on a vector machine is *not* given by the number of values it can hold, but by the length of the pipeline.

Multiple Instruction Multiple Data (MIMD) These are the most flexible machines. Since every processor is controlled independently one can do very irregular, asynchronous calculations. This also makes them more difficult to program and debug.

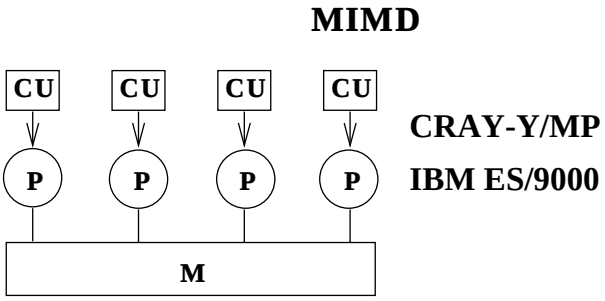

Figure 1.12: Multiple Instruction Multiple Data Shared Memory Architecture.

1.4.2 Memory Organization

There are two types of memory organization, shared memory and distributed memory. In shared memory, each processor can access any memory location at a uniform time. In distributed memory, each processor has access to its own memory.

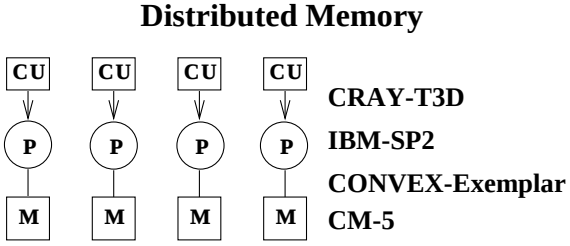

Figure 1.13: MIMD Distributed Memory Architecture.

From a programming point of view shared memory is much easier to handle. With distributed memory, the programmer must keep track of who owns what. A physical analogy is that shared memory is like a long distance interaction, e.g. a field whose action is felt globally, while distributed memory is like specifically treating the propagator between two particles, e.g. Feynman diagrams. Shared memory is good for a small number of processors. If one has too many processors, there is a large probability that there will be conflicts over

memory locations. Also, the cost of an interconnecting network grows with the square of the number of processors. With shared memory, one gets scalability only for $P \sim 10$. With distributed memory, scalability can be preserved up to $P \sim 10^2 - 10^3$.

1.5 Network Topology

Topology was a primary issue in first generation parallel computers, such as the Hypercube. The programmer was forced to be aware of the topology to be efficient. In modern machines, such as the IBM SP2, this is no longer the case, which is a great advantage for the programmer. For example, on the hyper-

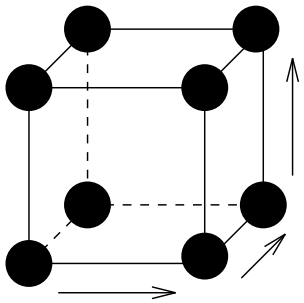

Figure 1.14: The hypercube topology.

cube, the generalization of a cube in n dimensions (fig. 1.14), communication between nearest neighbors is optimal. However, point-to-point remote communication is difficult since one needs a routing mechanism that selects the best path connecting two remote processors. This routing problem is no longer an issue on modern parallel machines.

Interconnecting networks fall within two broad categories:

- Static
- Dynamic

1.5.1 Static Networks

Static networks are those in which the interconnection topology is set once and does not change over time. Typical examples are the linear array, ring, 2-D mesh, and binary tree, as seen in fig. 1.15.

These interconnections are particularly efficient for those computations whose information flow pattern can be put in a homeomorphic correspondence with

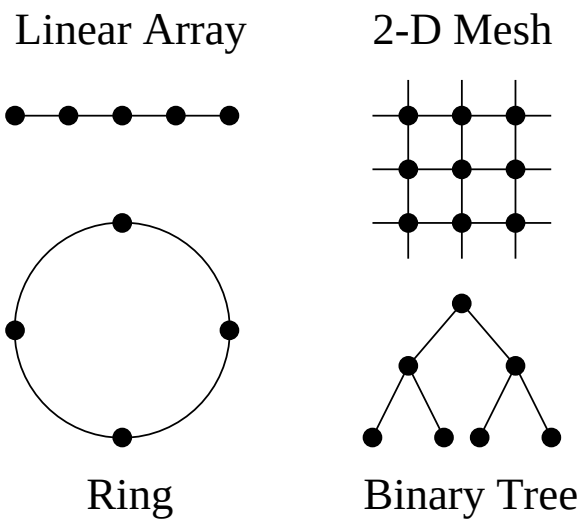

Figure 1.15: Static Network Topologies.

the interconnection topology. A typical example is the two-dimensional Poisson equation in a regular domain, solved with a regular finite-difference method. The ideal interconnection is the 2-D mesh.

1.5.2 Dynamic Networks

In dynamic networks, the connections between the processors and the memory can change over time. Both shared and distributed memory are dynamic networks. But distributed memory architectures scale, while shared memory architecture do *not* scale. The two most popular architectures for shared memory machines are the bus and the crossbar (fig. 1.16,1.17), both of which do not scale. For a bus there is a constant (high $\sim 1\mathrm{GB/s}$) flow rate of information.

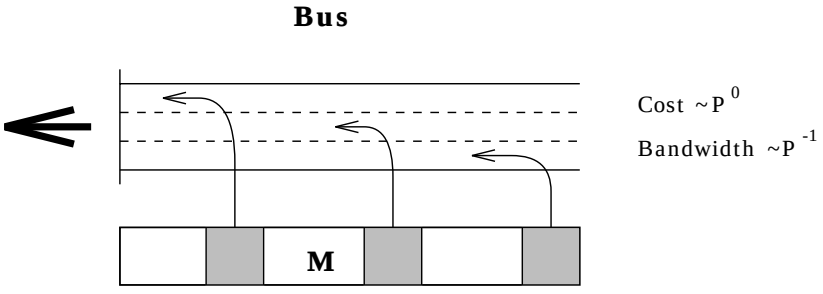

Figure 1.16: Bus Architecture.

However, if there are many processors accessing the memory, each of them

sees only its own share of the total bandwidth. Therefore, even though the cost is only weakly dependent on the number of processors, the bandwidth goes down like P^{-1}. In the scaling regime (highly parallel applications), the bandwidth for each processor vanishes as $P \to \infty$. This interconnection is very useful for low granularity systems, $P \sim 10$, such as the IBM ES/9000, but does not extend to massively parallel applications.

The second interconnection, as used by CRAY, is much more powerful and much more expensive. It is scalable in terms of connectivity. There is a certain number of processors and a certain number of memory banks and they are connected by an array of switches. Thus, the bandwidth is linear in the number of processors since we can activate many switches at the same time. However, since we need a full matrix, the cost goes like P^2. Again, the ratio of

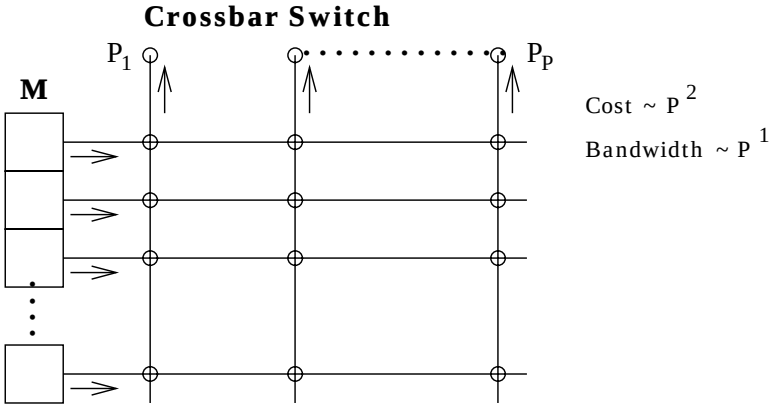

Figure 1.17: Crossbar Architecture.

bandwidth and cost goes like P^{-1}. So for shared memory machines, the number of processors that can be used efficiently is on the order of 10.

Typical values are a bandwidth of $\mathcal{O}\left(1\mathrm{GB/s}\right)$, roughly $1\frac{\mathrm{word}}{\mathrm{cycle}}$ This is very fast!. We can assume that the message transfer from memory to processor is almost infinitely fast. Compare this to Ethernet at roughly 1 MB/s. This means that distributed memory machines have a problem since the interconnection is relatively slow.

1.5.3 Multistage Switch

This is the interconnection used in the SP2 and is an intermediate step between the bus and the crossbar switch (fig. 1.18).
Given a pool of n processors and n memory banks, at any time, any processor can talk to any memory location. The processor can access any memory bank with the same efficiency. However, each memory bank can be accessed

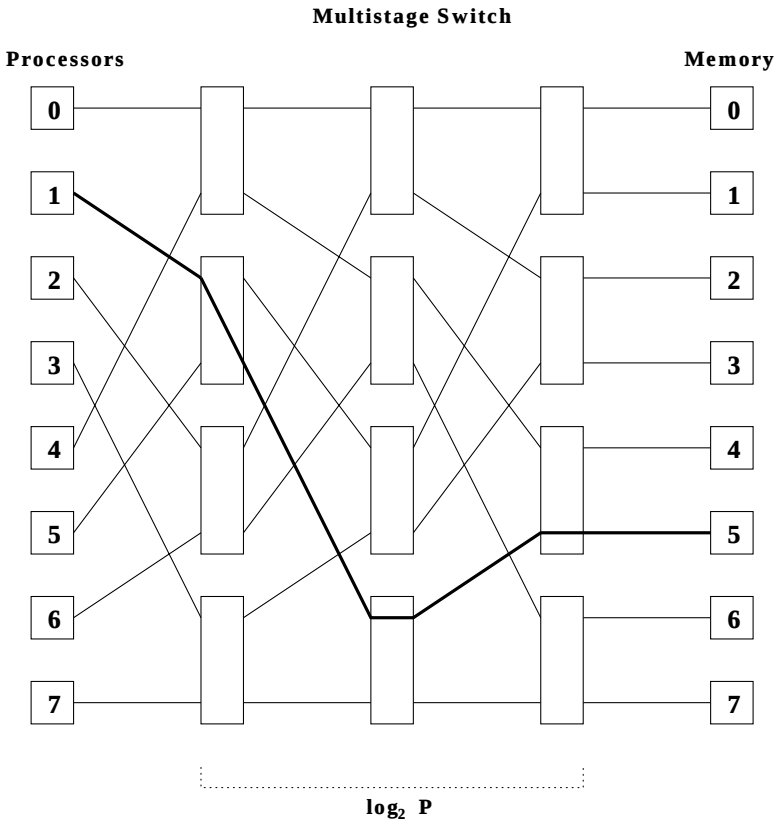

Figure 1.18: Multi-Stage Switch.

by only one processor in any given configuration (fig. 1.19). This architecture is designed to pass through $\log_2 P$ stages. At each pass, the number of paths is multiplied by two since some paths are redundant. So the cost is no longer P^2, but only $P \log_2 P$. Suppose one has a 4×4 array of processors and one would like to use the switch to maximize the number of one-to-one bonds at any time (fig. 1.20). The price to connect any processor to any memory location is $\log_2 P \times \{$the clock time of the switch$\}$. For the SP2, the time to go through the switch is 128ns, so the time it takes to make a connection is 4 (number of switches) times $128\text{ns} = 512\text{ns}$, regardless of the processor's identity. This number is the latency from hardware considerations, only. The actual latency will be much higher because of software decoding burdens.

Any connection can be represented in a "propagator" form:

$$\langle i|j\rangle = \langle i|k_1\rangle\langle k_1|k_2\rangle \cdots \langle k_N|j\rangle . \tag{1.9}$$

This structure is hidden from the programmer, so one doesn't need to worry

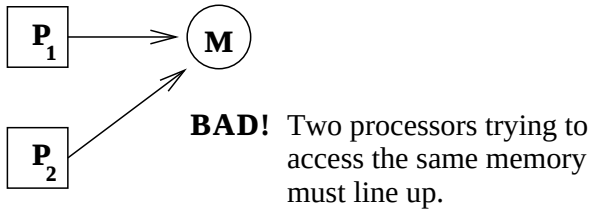

Figure 1.19: Two processors attempting to access the same memory at once.

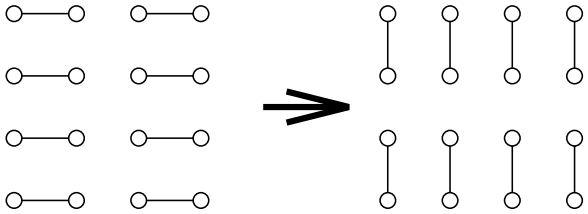

Figure 1.20: One-to-one bonding.

about it provided that two distinct processors i and i' are not simultaneously attempting to communicate with $j \neq i, i'$.

1.5.4 Tree Structure

A switch with tree structure, shown in figure 1.21, can be made even more elegant by increasing the bandwidth towards the top. This makes remote communication not much more expensive than local communication.

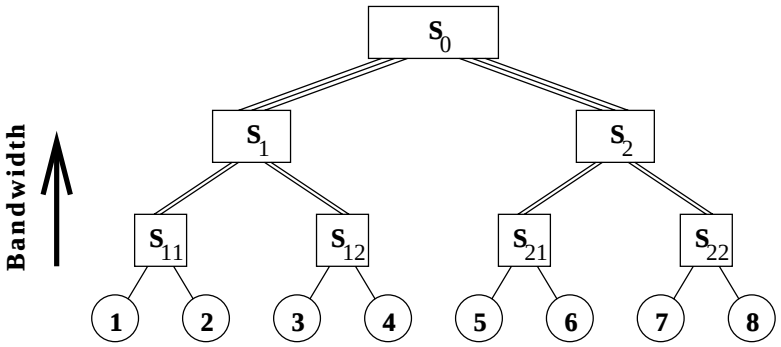

Figure 1.21: Switch with Tree Structure. Nodes 1 and 2 communicate through switch S_{11}. Nodes 1 and 8 communicate through switches S_{11}, S_1, S_0, S_2, S_{22}.

1.5.5 Three Different Architectures

There are two basic parameters which determine the performance of a distributed memory machine.

- *Bandwidth*, the asymptotic flow rate of information.
- *Latency*, the overhead payed to establish a connection between two processors, i.e. the cost to send a zero length message. This is very important for machines based on RISC processors.

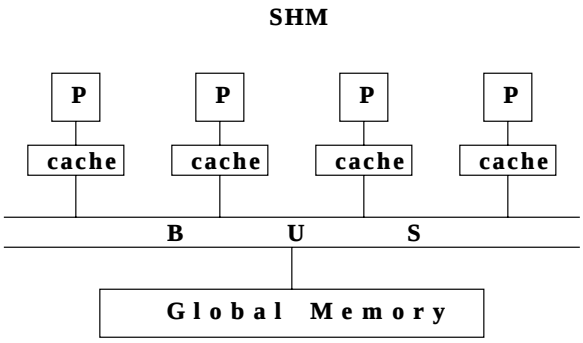

Figure 1.22: Shared Memory Architecture.

Typically for a shared memory architecture (fig. 1.22), one gets a bandwidth of about $1\mathrm{GB/s}$ and latency is negligible.

At the other extreme, Ethernet (fig. 1.23) gives bandwidths of about $\mathrm{MB/s}$ and very high latencies on the order of $1\mathrm{ms}$. This is the most primitive form of distributed parallel computing and the most widely used in an industrial environment. Ethernet is not at all efficient since thousands of floating point op-

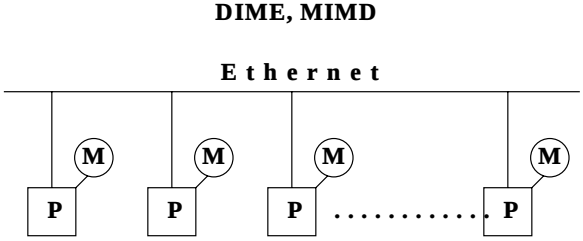

Figure 1.23: Ethernet-DIME,MIMD.

erations are wasted in latency. So, Ethernet is fine for problems characterized by low communication to computation ratios.

The DIME-SIMD machines (fig. 1.24) were very popular a few years ago. These have slow nodes of speeds at most 10 Mflop / node, bandwidths on the order of 10 MB / s and very low latencies on the order of 1μs. Since the latencies are so small, one can fragment the communications without paying a high price. This allows one to operate in the highly granular regime of thousands of processors.

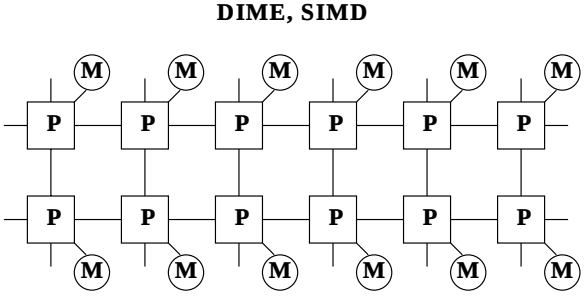

Figure 1.24: DIME-SIMD Architecture.

Today the granularity of the most powerful machines is a compromise between the shared memory machines and the DIME-SIMD, where P is about 100.

Three different examples of parallel architectures are shown in figures 1.22, 1.23, and 1.24. They represent a shared memory MIMD with bus interconnection (fig. 1.22), a distributed memory MIMD connected via a local area network (fig. 1.23), better known as a "cluster", and a distributed memory SIMD using a static mesh interconnection.

Chapter 2

Parallel Programming Models

In Chapter 1, we discussed the distinctive features of the main parallel architectures. In order to run parallel applications on these architectures, we need to implement them in a given programming language. The choice of languages is by no means unique, and a number of programming paradigms are available today. Like parallel architectures, parallel programming paradigms also lend themselves to a general classification according to the following criteria:

- Explicit vs. Implicit Message Passing.
- Local vs. Global Address-Space.
- Data vs. Control Parallelism.

2.1 Explicit vs. Implicit Message-Passing

One way to encode a parallel algorithm is to explicitly specify how the processors cooperate in order to solve a given problem in a coherent fashion. This makes life easy for the compiler, since it does not need to figure out automatically how interprocessor communication should be organized. Instead, the whole burden is placed on the programmer.

Implicit parallel programming does just the opposite. The programmer codes his or her application with a serial language, and the compiler inserts the appropriate parallel constructs. There is no question that implicit parallel programming is preferred over explicit parallel programming on account of the great savings in programming efforts and minimal exposure to programming errors. Unfortunately, the automatic parallelization of serial code is a very difficult task. The compiler needs to analyze and understand the dependencies of virtually every portion of the sequential code in order to generate a correct and efficient mapping onto a multiprocessor architecture.

So far, the goal of automatic parallelism has only been achieved for shared-memory machines. For these machines, all leading vendors are offering reliable parallel FORTRAN compilers and library extensions. The situation is much less developed for distributed memory machines. Only simple applications with pretty regular data structures can be dealt with by parallel compilers, such as FORTRAN D and High Performance FORTRAN.

It is worth mentioning that shared-address programming can also be implemented on distributed memory machines. The memory is physically distributed, but is logically shared (CRAY T3D, CONVEX Exemplar). This emulation is done by tagging part of the local memory of each processor as a communication buffer accessible to all other processors via remote read / write operations (fig. 2.1). This approach is more encompassing, but is also more expensive. The CPU of each processor needs to be able to resolve the addresses of another CPU in hardware, which adds to the complexity, and hence the cost, of the CPU.

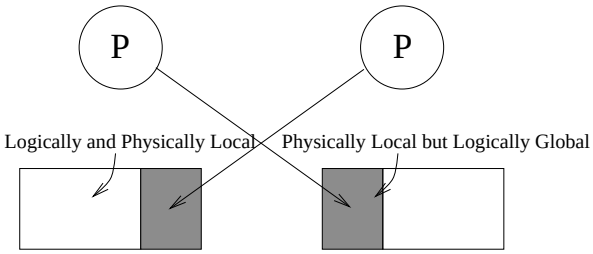

Figure 2.1: Shared-address programming on distributed memory machines: remote read / write.

On the other hand, message-passing only requires the CPU to be interfaced to the network via some standard input / output port (adapter, fig. 2.2).

2.2 Global vs. Local Address-Space

For the global address-space model, the programmer views his or her program as a collection of processes, all of which are allowed to access a central pool of shared variables. Two distinct processes communicate by reading / writing these shared variables from / to a globally accessible memory. This programming model is naturally suited to shared-memory parallel computers. Automatic parallelization is feasible, but the scalability limits described in Chapter 1 become manifest as soon as more than 10-20 processors are involved.

In the local address-space paradigm, each process is only entitled to access its private variables residing in its own local memory. Interprocessor communication can only take place via an explicit exchange of messages across an interconnecting network, whence the domination of the “message-passing”

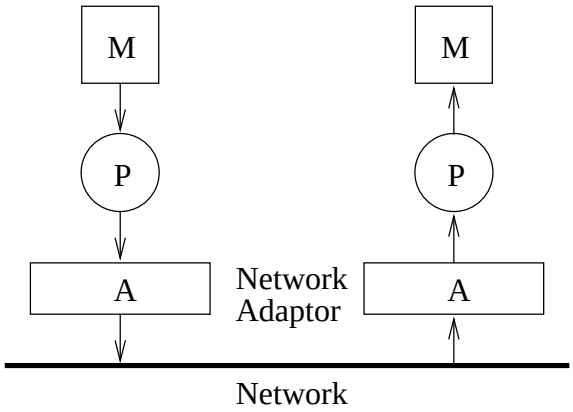

Figure 2.2: Message-passing across a network. The CPU's send and receive data via a relatively inexpensive network adaptor.

paradigm.

Message-passing is the natural choice for distributed memory machines. Being a form of explicit parallel programming, message-passing requires some programming labor to be put in place. The payoff, however, can be very rewarding in that scalability can be pushed way beyond the capabilities of shared-address programming. This is why several efforts are underway around the world to lay down message-passing programming standards, such as MPI, the Message Passing Interface.

2.3 Data vs. Control Parallelism

Many problems in the physical sciences come in the form of a few operations repeatedly applied to a large amount of data items. Such problems can be parallelized by assigning data elements to various processors, each of which performs the same operation on its own share of data. This is called data parallelism.

A typical example is matrix multiplication, $C_{ij} = \sum_k A_{ik} B_{ki}$. The element C_{ij} results from the dot product of the ith row of the matrix A and the jth column of the matrix B. Thus we have N^2 instances (A, B, and C have dimension $N \times N$) of the same operation, the dot product, executed on different data.

Several programming languages exist which make it easy to exploit data parallelism. Data parallel programs consist of a single sequence of instructions executed on each of the data sets in lockstep. Clearly, data parallel programs are naturally suited to SIMD computers.

Data parallel algorithims can also be executed on MIMD computers. However, the strictly synchronous execution typical of SIMD machines would set a needlessly heavy toll on MIMD computers, where global synchronization is a costly operation. A popular solution is to relax the constraint of strict synchronization and move to "loosely synchronous" execution. This means that each processor executes the same program, but not necessarily the same instruction, thus the definition of SPMD, Single Program Multiple Data (fig. 2.3). Synchronization only takes place when processors need to exchange data.

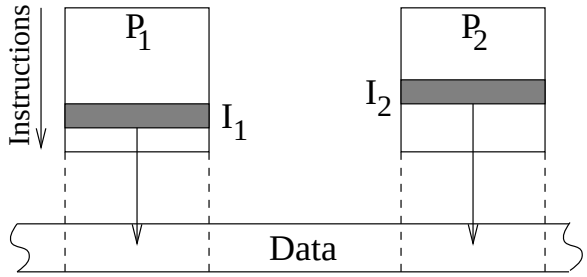

Figure 2.3: SPMD Execution: Processors P_1 and P_2 execute two different instructions of the same program on different data.

Another popular programming model which is unique to MIMD machines is the so-called "host-node" paradigm. Here the host plays the role of a master who coordinates the actions of the slaves without doing any work itself. This setting is particularly convenient in heterogeneous computing environments when one computer is significantly slower than the others. Of course, nothing prevents the master from doing any work, the "cooperative host-node" paradigm. These are illustrated in figure 2.4.

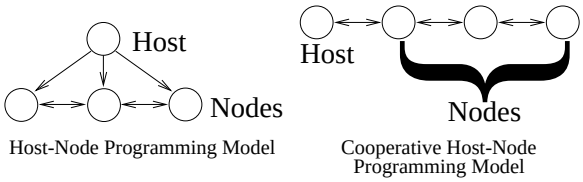

Figure 2.4: Illustrations of the host-node paradigms.

Control parallelism refers to the simultaneous execution of different instructions on the same or different data streams. A typical example is the pipelining process used in vector machines. Control parallelism is only possible on MIMD machines because multiple instructions are needed to execute the different operations. Control parallelism does not lend itself to fine grain parallelism since the number of different instructions is usually very limited, especially when

compared with the amount of different data to be acted upon.

2.4 Navigating Across the Pain-Gain Plane

There is more to parallel programming than just parallel languages. Parallel profiling and debugging tools are also of great importance if one is to navigate the "pain-gain" plane depicted in figure 2.5.

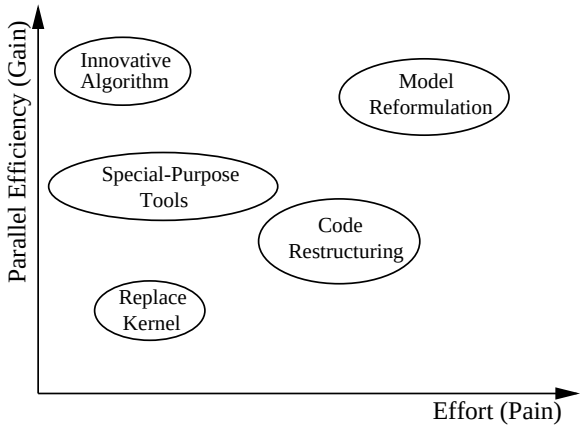

Figure 2.5: Approaches to Parallel Enablement: Pain vs. Gain.

In fact, the ultimate goal of the parallel programmer is by no means unique. Possibly the most fortunate of parallel programmers is the scientist who is given the task of designing a new, innovative algorithm from scratch, having parallel computers well in mind from the start. Less fortunate is the "parallel enabler", typically a computer company specialist, who is given the task of porting a pre-existing, and often rather complex, huge piece of code to a parallel machine. Fortunately, even the task of the latter unlucky parallel programmer is nowadays made easier by a series of serial-to-parallel automatic conversion tools. These are starting to consolidate the framework of software tools for parallel computing. The maturation of such software tools is pivotal to the broad expansion of parallel computing beyond the relatively narrow world of computer specialists. Software is the pacing issue in parallel computing.

Chapter 3

Programming Message-Passing Computers

3.1 Basic Communication Primitives

Message-passing computers are programmed via explicit exchange of messages between the communicating processes. We consider the following communication primitives:

- System query (“Who am I?”)
- Point-to-point (send / receive)
- Collective (one-to-all, all-to-one, all-to-all)

3.1.1 System Query

In order to undertake the proper actions, each process needs to be aware of its own identity (pid = process identity), as well as the number of processes contributing to the communication pool. This information can be obtained by issuing a query to the system. Generally, this query takes the form:

query(mid,numprocs)

where mid is the processor id and numprocs is the number of active processes.

3.1.2 Point-to-Point

These are the building blocks of any message-passing code, usually called send / receive. This code illustrates a small subset of the message-passing library. In addition to the usual send and receives, there are primitives that do both (fig. 3.1). These are recommended. Another important issue is collective

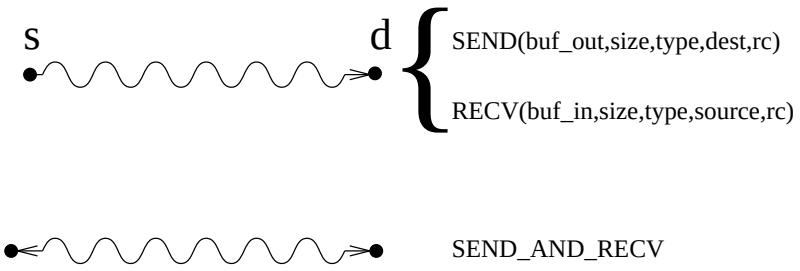

Figure 3.1: Point-to-point communication.

communication. For a numerical solution to a partial differential equation, we need a local stencil with local communication. When we need to synchronize or, for example, do a fast Fourier transform, we need collective communication. The different types are illustrated in fig. 3.2.

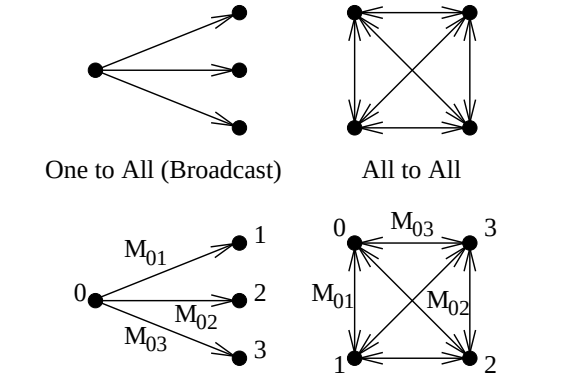

Figure 3.2: Collective communication.

3.2 Latency and Bandwidth

We will consider more general issues, like technological parameters. Once the code is running, we would like to have some idea of the efficiency. The time T it takes to exchange a message of length L is given by

$$T(L) = \Lambda + \frac{L}{B} = \frac{L_0 + L}{B} \equiv \frac{L}{B_{\text{eff}}} \equiv \frac{L_{\text{eff}}}{B}. \tag{3.1}$$

Λ is the start-up overhead, called *latency*, B is the bandwidth, and $L_0 = B\Lambda$ is the critical message length. Λ can constitute a significant penalty if one is

Machine	$\Lambda(\mu s)$	$B(MB/s)$	$S_1(M\text{flops})$	$L_0(kB)$	$F_0(k\text{flops})$
IBM-SP2	40	40	250	1.6	10
CM-5	80	10	128	0.8	10
Paragon	120	80	75	2.6	9
Delta	80	10		0.8	
Cray T-3D Virtual Shared Memory	1	300	150	0.3	0.15
SGI-P Challenge Shared Memory	0.01	1000	70	0.01	0.7
WS-Enet	1500	1	100	1.5	150
FDDI	1000	5	100	5.0	100

Table 3.1: Statistics for Present Day Machines. $F_0 = S_1\Lambda$ is the number of flops wasted in latency.

sending small messages. It is better to aggregate messages as much as possible. From here we can calculate an effective message length, L_{eff}, or an effective bandwidth, B_{eff}.

$$B_{\text{eff}} = \frac{B}{1 + L_0/L}, \qquad L_{\text{eff}} = L + L_0. \tag{3.2}$$

What we would like to know is:

- What is the shortest message that will not penalize our communication *or*
- How many floating point operations do we waste during latency.

These depend on the machine, as shown in Table 3.1. Latency is basically a *software* constraint. Recall that the hardware latency for a switch is roughly 100ns, while here the latency is on the order of microseconds.

3.3 Message-Passing Overheads

One reason for these high latency costs is historical. Originally, the networks were not designed for parallel computing. Only later, it was realized that the networks could be used to build parallel architectures. Typically, data has to flow through a number of software layers before it can be processed (fig. 3.3). So we would like to strip off as many layers as possible and use only the data transport layer. The main sources of overhead come from three places, the operating system, Application Programming Interface (A.P.I.), and protocol. In the operating system, system calls to protect or manage communication resources and context switches to field interrupts are sources of overhead. In A.P.I., buffer management and error checking are sources of overhead. In protocol, checking message tags and handshaking before incoming messages are accepted are sources of overhead.

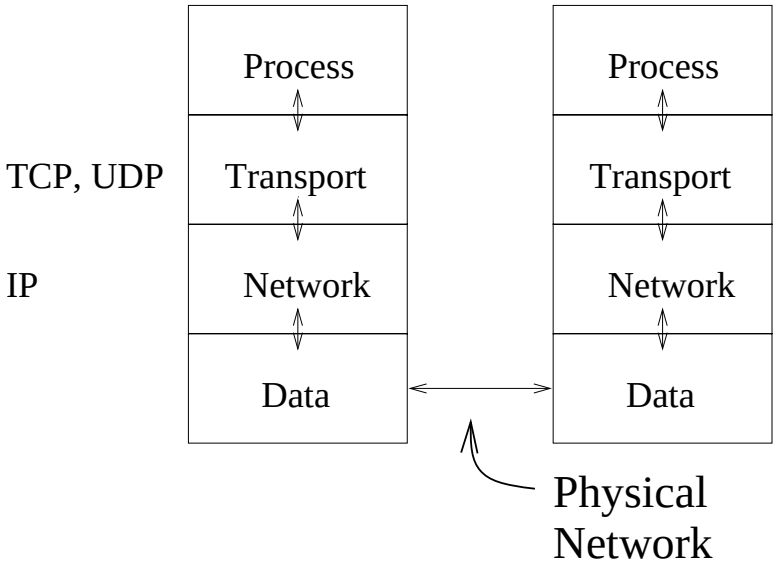

Figure 3.3: Message Passing Overheads.

3.4 SP2 System Structure

The early version IBM's used sockets. Today, on the SP2, one can bypass many of the communication policies (fig. 3.4). These are referred to as *light-weight*, or *lean*, protocols. Today there are two libraries available for this, MPL and PVMe.

3.5 PVM

The cheap way to do parallel computing is to use a type of public domain software known as Parallel Virtual Machine (PVM) (fig. 3.5). This software allows one to connect many different types of machines together. All of these machines become part of one heterogeneous computer. PVMe (PVM enhanced), fig. 3.6, allows one to run exceptionally fast if all of the components are of the same type, in this case IBM. PVMe is also completely compatible with PVM. PVMe bypasses most of the levels in fig. 3.3. Most significantly, PVMe bypasses data conversion between different types of machines. The main advantages of PVMe are error detection and lack of data conversion.

3.6 Data Communication

Once we understand how bandwidth and latency relate to efficiency, we can address the different types of send/receive statements. Message passing may be sufficient for now. However, many believe that in the long run it must be

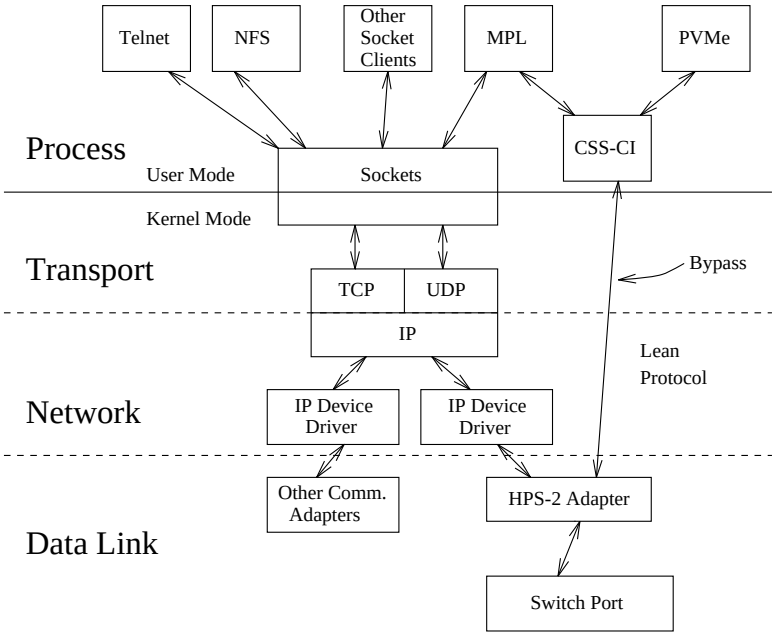

Figure 3.4: Bypassing communication protocol on the IBM SP2.

replaced by something that relieves the user of the burden of explicit message-passing programming. The main reason for this is that "send/receive" code is error-prone. A mismatch between the destination and the source is generally unforgiving. On the plus side, the system is very flexible so the rewards can be very high in terms of performance. In addition, since the program is so close to the processor distribution, it is able to take full advantage of hardware scalability (fig. 3.7).

3.7 Blocking vs. Non-Blocking Communication

There is a useful graphical convention for send/receive communications, as illustrated in figure 3.8. In a blocking receive, the task issues a request to receive and stops until the requested message is received. In a non-blocking receive, the task issues a request to receive, but continues to do work while waiting for the requested message. In a blocking send, the task issues a request to send and blocks until an acknowledgment of receipt is issued by the receiving process. In a non-blocking send, the task issues a request to send but goes on with its work without waiting for the message to be received. Due to a lack of idle times, non-blocking primitives are faster but less safe than blocking primitives. Non-blocking primitives should only be used when the program has already been well debugged.

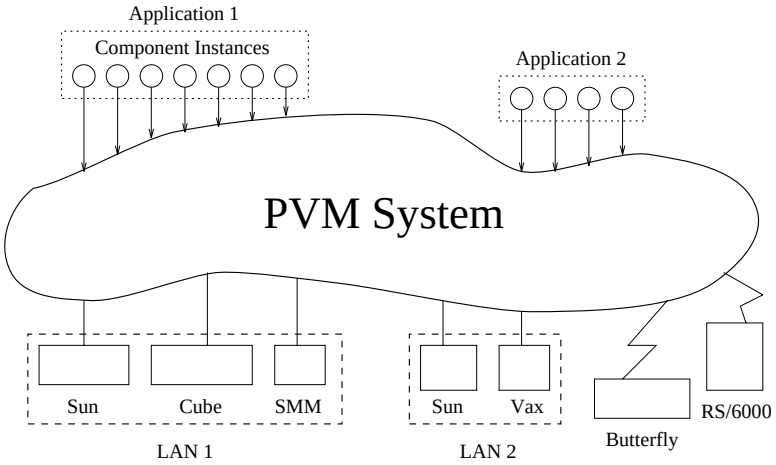

Figure 3.5: PVM: Parallel Virtual Machine made up of different computers linked by a local area network (LAN).

3.8 Switching Models

The interconnect networks of modern parallel architectures are designed in such a way that the programmer need not consider the network topology. This is achieved by letting interprocessor communication take place over switching networks that are able to support dynamic connectivity patterns. Two switching policies are very popular today:

- Circuit switching (telephone model).
- Packet switching (postal model).

3.8.1 Circuit Switching – Telephone Model

In a circuit switch (fig. 3.9), the link, once established, is kept busy until completion of the message exchange. The message travels as a rigid body. This is a non-optimal allocation of resources.

3.8.2 Packet Switching – Postal Model

In a packet switch (fig. 3.10), the packet proceeds on its own and can split into several sub-packets, each following its own path, “worm–hole” routing. The processor can proceed concurrently, “send and forget”, since the network is now intelligent and can steer the packet to its correct destination.

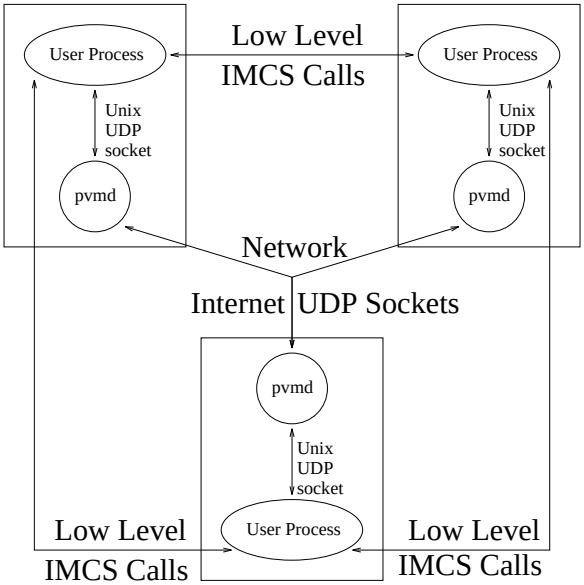

Figure 3.6: PVMe: a faster PVM.

3.9 Finite-Size Effects

Finite-size effects (fig. 3.11) can produce unpredictable results (“Heisenbugs”) such as buffer overflow, indefinite wait status (“bubbles”), and incorrect delivery. A solution is to receive a message as soon as possible after it was sent, “produce and consume asap”.

3.10 Examples in Point-to-Point Message Passing

These examples of the circular shift program are drawn ’verbatim’ from the IBM MPL library.

3.10.1 Example A-1 (Incorrect)

```
CALL MP_ENVIRON(NUMPE,ME)
IF (ME.LT.NUMPE-1)
   THEN RIGHT = ME+1
   ELSE RIGHT = 0
END IF
CALL MP_BRECV(INMSG1,MSGLEN,SOURCE,TYPE,NBYTES)
CALL MP_BSEND(OUTMSG1,MSGLEN,RIGHT,TYPE)
```

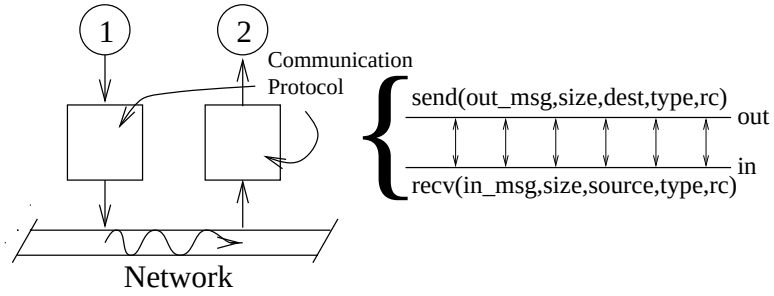

Figure 3.7: Basic send and receive primitives. Mismatch of call parameters is usually unforgiving.

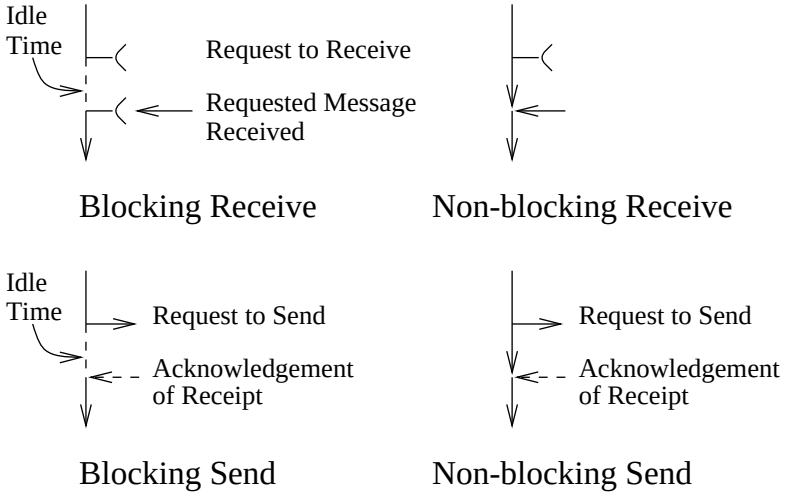

Figure 3.8: Graphical conventions for send and receive. Here time is moving along the vertical axis.

This program is incorrect because it will never complete. Reason is that *all* processors issue a request to receive prior to receiveing so that deadlock results.

3.10.2 Example A-2 (Correct but Unsafe)

```
CALL MP_ENVIRON(NUMPE,ME)
IF (ME.LT.NUMPE-1)
   THEN RIGHT = ME+1
   ELSE RIGHT = 0
END IF
CALL MP_RECV(INMSG1,MSGLEN,SOURCE,TYPE)
CALL MP_BSEND(OUTMSG1,MSGLEN,RIGHT,TYPE)
```

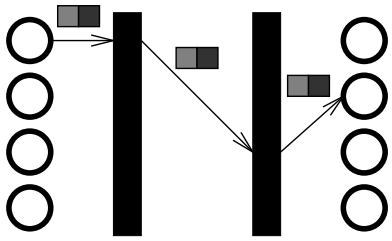

Figure 3.9: Circuit Switch – Telephone Model

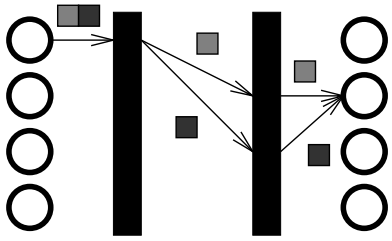

Figure 3.10: Packet Switch – Postal Model

This code solves the deadlock problem by replacing the blocking receive with a non-blocking receive. It is unsafe since we only have a finite-size buffer, and buffer space is required for both incoming and outgoing messages.

3.10.3 Example A-3 (Correct but Unsafe)

```
CALL MP_ENVIRON(NUMPE,ME)
IF (ME.LT.NUMPE-1)
   THEN RIGHT = ME+1
   ELSE RIGHT = 0
END IF
CALL MP_BSEND(OUTMSG1,MSGLEN,RIGHT,TYPE)
CALL MP_BRECV(INMSG1,MSGLEN,SOURCE,TYPE,NBYTES)
```

This code solves the deadlock problem by reversing the order of the blocking routines. Again, it is unsafe since we only have a finite-size buffer.

3.10.4 Example A-4 (Correct and Safe)

```
CALL MP_ENVIRON(NUMPE,ME)
IF (ME.LT.NUMPE-1)
   THEN RIGHT = ME+1
   ELSE RIGHT = 0
```

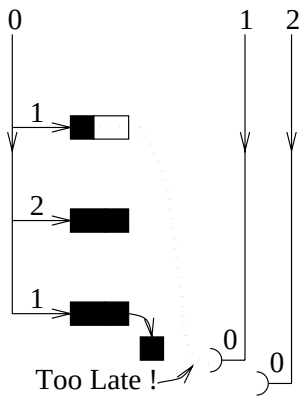

Figure 3.11: Finite-Size Effects – "Heisenbugs"

```
END IF
IF ((ME \ 2).EQ.0)
   CALL MP_BSEND(OUTMSG1,MSGLEN,RIGHT,TYPE)
   CALL MP_BRECV(INMSG1,MSGLEN,SOURCE,TYPE,NBYTES)
ELSE
   CALL MP_BRECV(INMSG1,MSGLEN,SOURCE,TYPE,NBYTES)
   CALL MP_BSEND(OUTMSG1,MSGLEN,RIGHT,TYPE)
END IF
```

Odd and even tasks alternate send and receive operations, so the only cause of a buffer overflow is a single message that is larger than the buffer size. Given the actual size of present-day buffers (a few MBytes), such a condition is very unlikely to occur under ordinary circumstances.

3.10.5 Example A-5 (Correct and Safe)

```
CALL MP_ENVIRON(NUMPE,ME)
IF (ME.LT.NUMPE-1)
   THEN RIGHT = ME+1
   ELSE RIGHT = 0
END IF
CALL MP_BSENDRECV(OUTMSG1,MSGLEN,RIGHT,TYPE,INMSG1,MSGLEN,SOURCE,NBYTES)
```

The best solution to the problem uses a combined blocking send and receive call. Here the routine has internal buffer management schemes that minimize the demands on system-buffer space.

3.10.6 Example B-1 (Correct but Unsafe)

```
CALL MP_ENVIRON(NUMPE,ME)
IF (ME.EQ.1)
   THEN
     DO I = 1, LARGE
       DEST=2
       TYPE=I
       CALL MP_SEND(OUTMSG(I),MSGLEN,DEST,TYPE)
     END DO
     CALL MP_WAIT(ALLMSG)
   ELSE IF (ME.EQ.2)
     DO I = LARGE, 1, -1
       SOURCE=1
       TYPE=I
       CALL MP_BRECV(INMSG(I),MSGLEN,SOURCE,TYPE,NBYTES)
     END DO
END IF
```

This will probably cause a buffer overflow due to the way the messages are received by type. All messages must be sent before any of them can be received.

3.10.7 Example B-2 (Correct and Safe)

```
CALL MP_ENVIRON(NUMPE,ME)
IF (ME.EQ.1)
   THEN
     DO I = 1, LARGE
       DEST=2
       TYPE=I
       CALL MP_BSEND(OUTMSG(I),MSGLEN,DEST,TYPE)
     END DO
   ELSE IF (ME.EQ.2)
     DO I = LARGE, 1, -1
       SOURCE=1
       TYPE=I
       CALL MP_BRECV(INMSG(I),MSGLEN,SOURCE,TYPE,NBYTES)
     END DO
END IF
```

This program is safe since it reverses the order in which messages are received. In general, to avoid deadlock and buffer overflow:

- Do not call a blocking routine that can never be satisfied.
- Guarantee that you have a receive for every message sent.

- Do not allow a number of messages to accumulate without being received.

3.10.8 Three Body Interaction

All of the above recommendations hold even more true in the presence of "multi-point" communication patterns such as the one depicted in fig. 3.12.

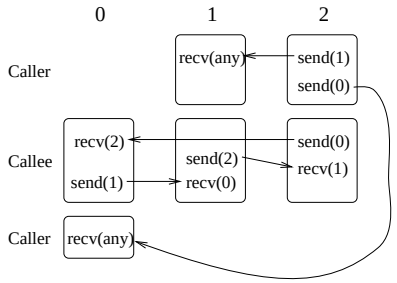

Figure 3.12: Three-Body Interaction

3.11 Collective Communication

Collective communication helps clarify the message-passing programs by replacing several elementary primitive calls with a single "macro" call. Besides clarity, this is also beneficial in terms of code efficiency.

3.11.1 Broadcasting

Figures 3.13- 3.16 illustrate various broadcasting models:

- Simple broadcast.
- Recursive broadcast.
- Binary broadcast.
- Fibonnaci broadcast.

The Fibonnaci broadcast takes 6 stages instead of the 7 required by the binary-tree pattern. The Fibonacci broadcast supersedes binary-tree communication patterns whenever the latency takes more than one unit of computation. The time of a Fibonnaci broadcast (fig. 3.16) is given by

$$f_\lambda(n) = \min\{t, F_\lambda \geq n\}, \tag{3.3}$$

$$F_\lambda = \begin{cases} 1 & \text{if } 0 \leq t \leq \lambda, \\ F_\lambda(t-1) + F_\lambda(t-\lambda) & \text{if } t > \lambda. \end{cases} \tag{3.4}$$

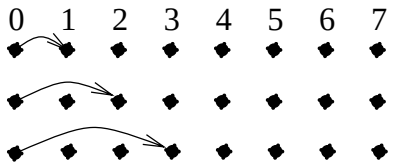

Figure 3.13: Simple Broadcast, $\Lambda = 1$.

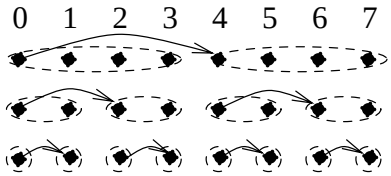

Figure 3.14: Recursive Broadcast (Binary Tree), $\Lambda = 1$.

3.11.2 Routines from the IBM Message Passing Library (MPL)

Figures 3.18- 3.25 illustrate routines from the IBM Message Passing Library (MPL). These implement various forms of collective communication, fig. 3.17

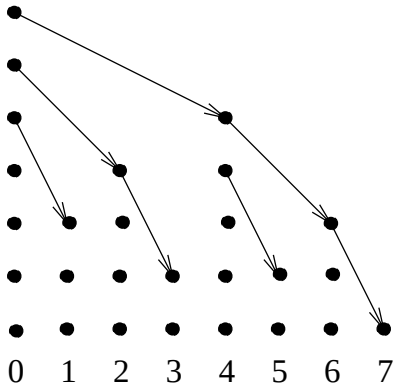

Figure 3.15: Binary Broadcast, $\Lambda = 2$. For binary tree recursive doubling, $t_{\text{com}} = \Lambda \ln_2 P$.

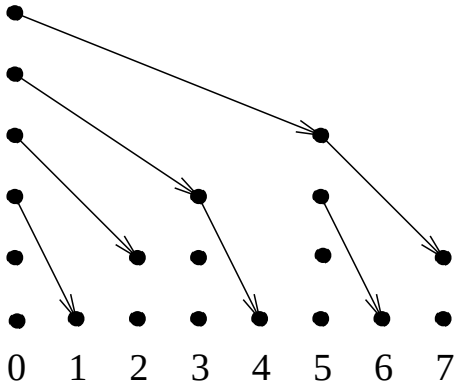

Figure 3.16: Fibonacci Broadcast, $\Lambda = 2$

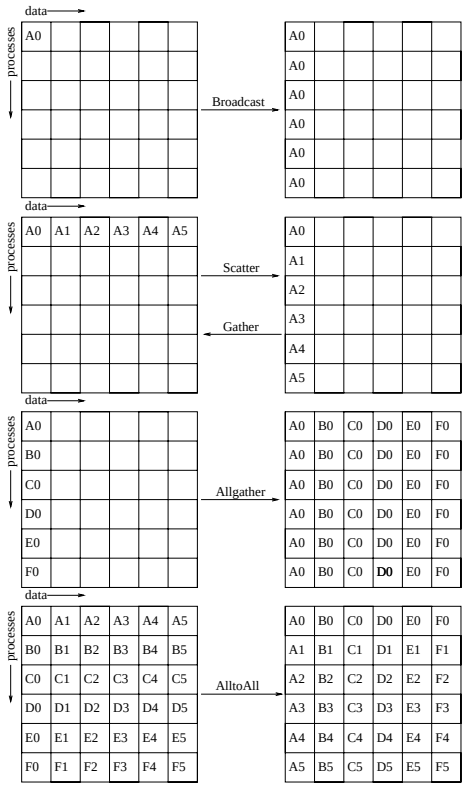

Figure 3.17: Various Forms of Collective Communication.

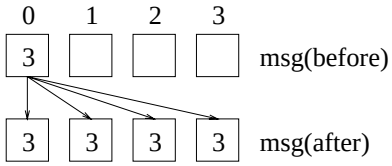

Figure 3.18: MP_BCAST broadcasts a message from one task to all tasks in the group. Here, task “0”, is the source task. This is useful in host-to-node global data distribution.

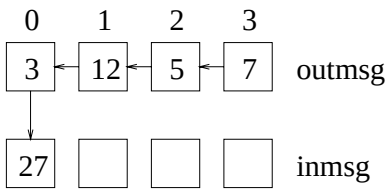

Figure 3.19: MP_REDUCE applies a reduction operation on all the tasks in the group and places the result in one task. Here, the reduction operator is integer addition and task "0" is the destination task. This is useful in convergence checks during iterative computations.

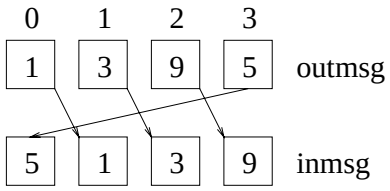

Figure 3.20: MP_SHIFT shifts data up or down some number of steps in the group. Here, data is shifted one task to the "right" with wraparound. This is as easily implemented on a MIMD machine as on a SIMD machine. This is useful in regular lattice calculations.

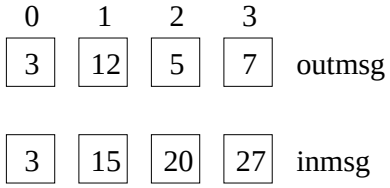

Figure 3.21: MP_PREFIX applies a parallel prefix with respect to a reduction operation across a task group and places the corresponding result in each task in the group. This is useful in multilevel, hierarchical computations.

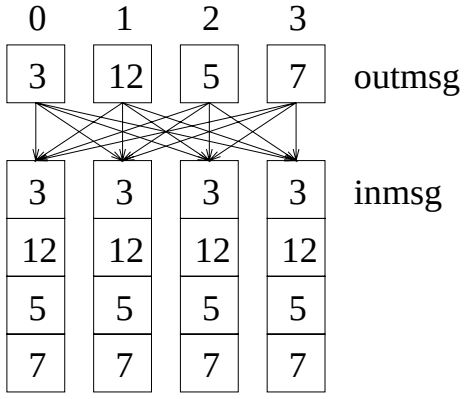

Figure 3.22: In MP_CONCAT, each task performs a one-to-all broadcast.

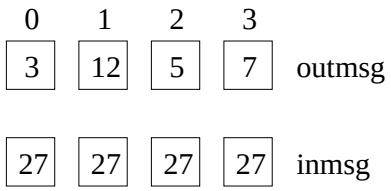

Figure 3.23: In MP_INDEX, each task in a group performs a scatter operation, sending a distinct message to all the tasks in the group in order by index. This is useful for matrix transposition, such as multidimensional Fast Fourier Transform computations.

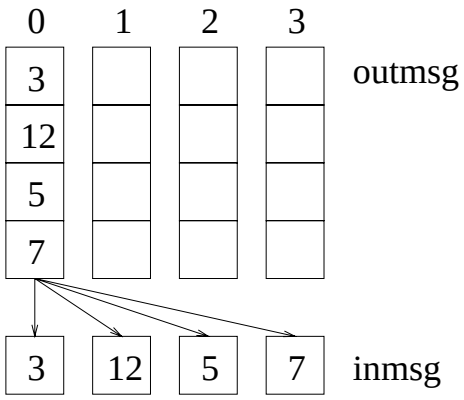

Figure 3.24: MP_SCATTER scatters distinct messages from a single source task to each task in the group. Here, task "0" is the source task. This is useful in irregular, finite element or molecular dynamics computations

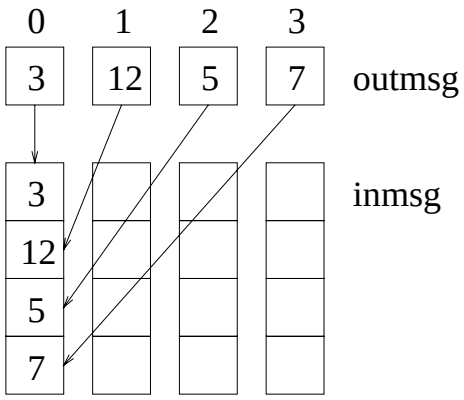

Figure 3.25: MP_GATHER gathers distinct messages from each task in the group to a single destination task. It is the reverse operation of scatter. This is useful in irregular computations.

Chapter 4

Factors Controlling Parallel Efficiency

The main factors controlling parallel efficiency are:

- Fraction of parallel content
- Communication and synchronization overheads
- Load balance across the various processors
- Redundancy

We shall express the parallel efficiency in the following general form:

$$\eta(P) = \prod_{i}^{4} \eta_i \equiv \prod_{i}^{4} \frac{1}{1 + OVH_i}, \tag{4.1}$$

where OVH_i refers to the parallel overhead introduced by the ith mechanism. Typically OVH_i is expressed as the ratio between CPU time wasted setting up parallel operations versus CPU time usefully spent in computation.

A problem is said to be *scalable* as long as the parallel overhead is much smaller than 1 so that efficiency is basically unaffected. Mostly, parallel overhead is an increasing function of time and therefore scalability cannot last forever. More specifically, one can always introduce the notion of a *critical* processor number P_c beyond which parallel efficiency drops rapidly down to zero.

$$\lim_{P>>P_c} \eta(P) \simeq 0. \tag{4.2}$$

Scalability requires high P_c's and, to a minor extent, smooth fallout for $P > P_c$.

Equation 4.1 implies that in order to achieve a satisfactory parallel efficiency, *all* parallel overheads must be kept small. Thus, parallel computing, albeit transparent from the conceptual viewpoint, is by no means an easy job in practice.

f	S_∞	P_c	$S(P_c)$	P	f_c	$S(f_c)$
1/4	4/3			4	2/3	2
1/2	2	1	1	8	6/7	4
3/4	4	3	2	16	14/15	8
0.9	10	9	5	64	0.94	32
0.99	100	99	50	128	0.982	64
0.999	1000	999	500	1024	0.999	512

Table 4.1: Amdahl's Law

4.1 Parallel Content: Amdahl's law

The parallel potential, f, is the fraction of time spent by the CPU in the parallel section of the code. If $T(1)$ is the time taken by a single processor, the corresponding time with P processors is given by:

$$T(P) = T(1) \cdot \left(\frac{f}{P} + (1 - f) \right). \tag{4.3}$$

To determine the optimal number of processors, we consider the parallel speedup:

$$S(P) = \frac{T(1)}{T(P)} = \frac{P}{f + (1 - f)P} \longrightarrow \frac{1}{1 - f} \equiv S_\infty, \text{ as } P \to \infty. \tag{4.4}$$

Thus, the parallel speed-up for a given problem is bounded above by $(1-f)^{-1}$. Since there are not many massively parallel problems, P can be taken fairly low. The maximum number of processors for a given f is $P_c = f/(1 - f)$ since

$$\frac{S}{S_\infty} = \frac{1}{1 + P_c/P}. \tag{4.5}$$

The minimum parallel fraction for a given number of processors is given by $g_c = (P - 1)^{-1}$ since

$$\frac{S}{S_1} = \frac{1}{1 + g/g_c}, \qquad S \longrightarrow S_1 = P, \text{ as } f \to 1, \tag{4.6}$$

where $g = 1 - f$.

Finally, we can write down Amdahl's law as follows

$$\eta(f) = \frac{1}{1 + g/g_c}. \tag{4.7}$$

Table 4.1 gives values for Amdahl's Law with $f_c = 1-g_c$. From this table we get a clear perception that massive parallelism, thousands of processors or more, is only viable for applications spending no more than one part per thousands of their CPU time in the non-parallelizable section of the code.

Kernel	Complexity	$1-f$
I/O	$N \cdot \mathcal{O}(1)$	1
Direct	$N \cdot \mathcal{O}(N^{2/3}) \cdot NT$	$N^{-5/3}$
Iterative	$N \cdot \mathcal{O}(N^{1/3}) \cdot NT$	$N^{-4/3}$
FFT	$N \cdot \mathcal{O}(\log_2 N) \cdot NT$	$N^{-1}/\log_2 N$
Explicit	$N \cdot \mathcal{O}(1) \cdot NT$	N^{-1}
N-Body	$N \cdot \mathcal{O}(N) \cdot NT$	N^{-2}

Table 4.2: Gustafson's Rescue from Amdahl's Law. Here NT is the number of time steps to complete a job. It is assumed $NT \sim N$.

4.2 Scaled Speed-Up: Gustafson's Law

The two uses of parallel computing are to increase the size of the problem (thermodynamic limit), and cut turnaround (continuum limit). The continuum limit is the Amdahl's Law case, where as $P \to \infty$ and $N \to C$, a constant, $N/P \to 0$. This leads to rather gloomy conclusions as far as massively parallel computing, $P > 1000$, is concerned.

The best use of parallel computing is to increase the size of the problem. As $P \to \infty$ and $N \to \infty$, $N/P \to C$, a constant. The parallel potential, f, increases as the problem size N, increases. Thus we have Gustafson's Rescue from Amdahl's Law, as seen in table 4.2.

In this table, we report the computational complexity of some typical kernels of scientific applications as a function of the problem size. Assuming that the serial fraction of the code, g, scales linearly with the problem size (like an I/O operation), we obtain the data reported in the 3rd column; $f \to 1$ as $N \to \infty$.

4.3 Synchronization Overheads

In this section we will work through some examples to estimate the different types of overheads which arise when calculating parallel efficiencies. Recall that the efficiency, η, is given by

$$\eta = \frac{T(1)}{P\left(T_{\rm cal}(P) + T_{\rm com}(P)\right)}, \tag{4.8}$$

where P is the number of processors, $T(1)$ the time it takes to run on a single processor, $T_{\rm cal}(P)$ and $T_{\rm com}(P)$ is the time for calculation and the time for communication running on P processors. If we assume that $PT_{\rm cal}(P) = T(1)$ then

$$\eta = \frac{1}{1 + \frac{T_{\rm com}(P)}{T_{\rm cal}(P)}} = \frac{1}{1 + {\rm CCR}} \tag{4.9}$$

where CCR is the communication to computation ratio.

We have already seen that there are various forms of penalties for communication. We will specifically address synchronization overheads. Synchronization means that we have spawned some processes and at some point we need to re-synchronize (fig. 4.1).

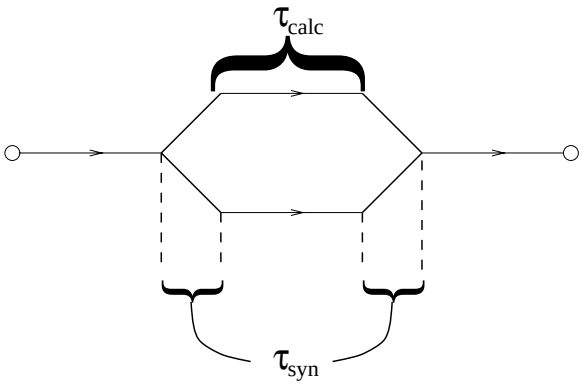

Figure 4.1: Synchronization overheads.

This concept is illustrated in figure 4.1. However, in practice, there are several algorithms which will implement this function (the alpha trees, the binary trees, etc.). In figure 4.1, we show just two processes, but we are dealing with global communication since we are spawning work for all the processors. Practical instances of synchronization overhead include checking iteration convergence, advancing the time step, and I/O.

We would like to know what the penalties associated with these kinds of operation are. The time spent in synchronization is given by

$$\tau_{\text{syn}} = \Lambda\varphi(P) \longrightarrow \Lambda \begin{cases} P(P-1)/2 \\ (P-1) \\ 2\log_2 P \end{cases} + \delta\Lambda(P). \tag{4.10}$$

Again, Λ is the latency, φ depends on the kind of algorithm we use to make the broadcast, and $\delta\Lambda(P)$ is the "traffic congestion", when two processors want to speak to a single processor. Suppose we want to try three different types of global communication (fig. 4.2 and table 4.3).

The three types of communication illustrated are:

- All-to-all, which gives very poor efficiency.
- Linear, where one master processor takes information from the others.

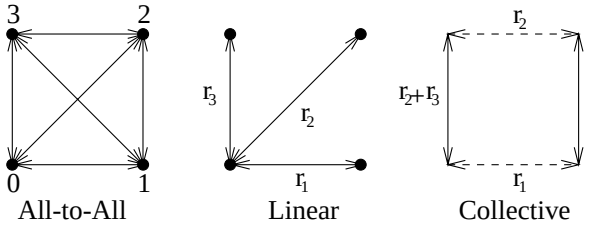

Figure 4.2: Three different types of communication

$\Lambda \sim 100\mu s, \quad S_1 \sim$ Mflops, $\quad N_c = \Lambda S_1/\chi$			
	All-to-All	Linear	Collective
	P		
$\chi = 10^2, \quad N = 10^6$	$10^{4/3}$	10^2	$10^4/4$
$\chi = 10, \quad N = 10^5$	$10^{2/3}$	10	10^2

Table 4.3: Orders of Magnitude of P for three different communication architectures

- Collective, which is the best case.

P for the all-to-all communication is better than that observed in practice. The best solution is to use collective communication. Since the limit as P tends to infinity of CCR is infinity, we have broken scalability.

Note that in all these examples, we have assumed that there is only one synchronization point per time step. This is not necessarily the case, such as for conjugate gradient methods, so there is another factor, the number of synchronization points, that needs to be multiplied in. It should be made clear that no matter which scheme is used, $\varphi(P)$ grows with P, while τ_{calc} decreases. So global communication is doomed to kill efficiency beyond a certain P.

For now, we will assume that we are doing a blocking send or receive. The case where $\tau_{\text{syn}} \sim P^2$ is a simple linear do loop, which loops over all processors, is highly inefficient and yields unacceptable performance results, even for small values of P. The synchronization communication to computation ratio is

$$\text{CCR} = \frac{\tau_{\text{syn}}}{\tau_{\text{calc}}} = \left(\frac{\Lambda S_1}{\chi}\right) \frac{P\varphi(P)}{N} \equiv N_c \frac{P\varphi(P)}{N}. \tag{4.11}$$

Here $\tau_{\text{calc}} = \chi N/(S_1 P)$ and $N_c = \Lambda S_1/\chi$ is the critical grain size. χ is the number of flops per grid point and S_1 is the speed of a single processor. N is the grain size (number of grid points or number of degrees of freedom in the problem). Recall that $S_1\Lambda$ is essentially the number of floating point operations wasted in establishing a connection between two processors. This is the most important parameter. The synchronization-free condition is CCR $\ll 1$,

$$P\varphi(P) \ll \frac{N}{N_c} \equiv \frac{\text{flops to Update}}{\text{flops wasted in latency}}. \tag{4.12}$$

We clearly see that we need to minimize the number of flops that are wasted in latency. The dimensionless number N/N_c can be regarded as a sort of Reynolds number of parallel computing. N_c is limited by technology and N is numerical, i.e. the density of the algorithm. Both of these need to be adjusted so that the ratio is in the scaling regime.

4.4 Orders of Magnitude

With reference to fluid dynamics algorithms, we typically assume that ξ is on the order of 100 flops/grid point and the target architecture has a latency of about 100 μs The problem size is N, which is typically 10^6 for industrial fluid dynamics, and S_1 about 100 Mflops. This gives us a critical size of

$$N_c = \frac{\Lambda S_1}{\xi} \sim 10^2. \tag{4.13}$$

Fully developed turbulence has N roughly 10^9.

In 2D, the synchronization time is equivalent to the time spent updating a 10×10 square over one time step. Similarly, in 3D, we are losing the time it would take to update a $5 \times 5 \times 5$ cube. This means that if we have a partitioning scheme such that each processor gets less than a 10×10 square (in 2D), then the efficiency is completely lost. Again, this depends on the scheme that we use.

4.5 Communication Overheads

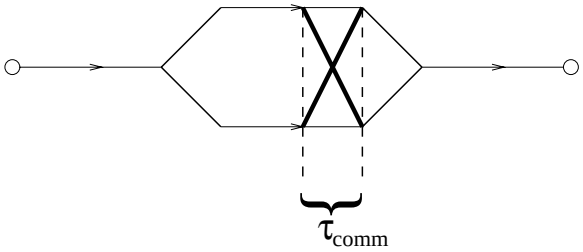

Figure 4.3: Communication overhead

The same exercise we did with synchronization overheads can be done with local communication requirements. We do the same calculation for communication cost due to data passing across a local stencil (fig. 4.3). The time spent in communication is given by

$$\tau_{\text{comm}} = \frac{\sigma\Sigma}{B} + N_M \Lambda, \tag{4.14}$$

where σ is the "surface density" measured in bytes per grid point on the surface (if there is just one variable being passed, then σ is 8 or 4), Σ is the exchange surface measured in grid points, B is the bandwidth, and N_M is the number (granularity) of messages. Then the communication to computation ratio, assuming no extra cost for long range communication, is given by (neglecting $N_M \Lambda$)

$$\text{CCR} = \frac{\tau_{\text{comm}}}{\tau_{\text{calc}}} = \left(\frac{\sigma\Sigma}{\chi N}\right)\left(\frac{B}{S_1}\right). \tag{4.15}$$

The first factor on the right hand side is the ratio of bytes to be sent per flop required to advance the calculation one time step. It is a measure of the communicativity of the algorithm; there are no hardware constraints here. For B approximately 10 MB/s and S_1 roughly 100 Mflops/s, we get $\sigma\Sigma/\chi N < 10^{-1}$. At least 10 flops per each byte is sent off. These are all order of magnitude estimates to give some idea of whether the code is running efficiently.

Hint: Use at least 10 *flops per bytes sent off (*~ 100 *flop/word).*

4.6 Domain Decomposition

Communication overheads are very sensitive to the way that data are partitioned in the "divide-and-conquer" strategy. The method of choice for many problems in physics and engineering is domain decomposition. Domain decomposition is very straight-forward in the case of structured problems, where calculations can be mapped onto a cube. First we need to decide what type of partitioning to use (fig. 4.4).

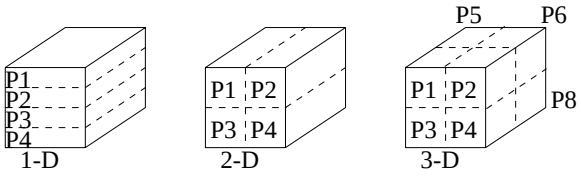

Figure 4.4: Domain decomposition: The three types of partitions are pictured above. Set l to be the length of one edge of a cube in a three-dimensional partition. Set $N = L^3 = Pl^3$.

Generally, the most efficient partition is 3-dimensional. We want to minimize the surface to volume ratio since data is flowing across the surfaces. Using N as the global size of the problem, and l as the linear size of the problem on

a single processor ($N = Pl^3 = L^3$), we can estimate the efficiencies for the different types of partitioning.

- 3-Dimensional Decomposition.

$$\Sigma = 6Pl^2, \tag{4.16}$$

$$P < P_c = \left(\frac{B\chi L}{6S_1\sigma}\right)^3 \propto N, \tag{4.17}$$

$$\frac{\Sigma}{N} \sim P^{1/3}L^{-1}. \tag{4.18}$$

  The ratio of calculation to communication goes up like $P^{1/3}$. A three-dimensional partition is best if there is a large number of processors.

- 2-Dimensional Decomposition.

$$\Sigma = 4PlL, \tag{4.19}$$

$$P < P_c = \left(\frac{B\chi L}{4S_1\sigma}\right)^{3/2} \propto N^{1/2}, \tag{4.20}$$

$$\frac{\Sigma}{N} \sim 4P^{2/3}L^{-1}. \tag{4.21}$$

  A two-dimensional partition is a fair compromise between one and three dimensions. This works well on the SP2 which has roughly 100 processors.

- 1-Dimensional Decomposition.

$$\Sigma = 2PL^2, \tag{4.22}$$

$$P < P_c = \left(\frac{B\chi L}{2S_1\sigma}\right) \propto N^{1/3}, \tag{4.23}$$

$$\frac{\Sigma}{N} \sim 2PL^{-1}. \tag{4.24}$$

  A one-dimensional partition is the easiest to program, since there is only one exchange surface. It is less scalable, but also less latency exposed. There are less messages and they are longer. A one-dimensional partition is better suited for large latency machines or low granularity.

4.7 Data Layout

A large amount of research has gone into more complex models of data layout. Again we would like to minimize the surface to volume ratio. If the geometry is non-trivial, identifying the best partition is also non-trivial. One has to ensure that each processor gets approximately the same amount of work (*load balancing*) and that the boundary surface for every processor is approximately the same, while minimizing the global surface. For the majority of industrial applications, such as aircraft design, automobile design or chemical engineering, the geometry is the major problem. There is a lot of research in computer science to identify automatic domain decomposition tools for non-trivial geometries.

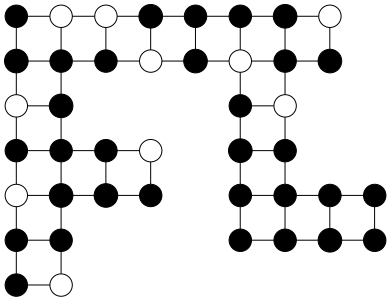

Figure 4.5: Here is a bad data layout. The data assigned to each processor is scattered, thus triggering a lot of communication.

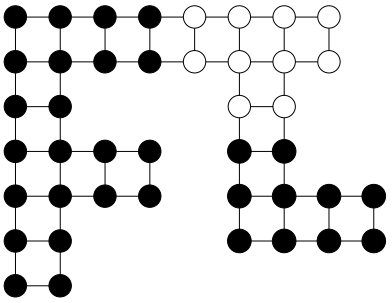

Figure 4.6: Here is a good data layout. Related data is mapped to the same processor. Communication is minimal.

4.8 Load Balance

A difficult but important goal is to keep all processors equally busy. The problem is that the pace is dictated by the slowest processor, thus we are working in

the infinity norm since we have to give infinite weight to the slowest processor. We are dealing with a scenario where, once a processor has finished its job, it cannot go on to something else. There is no overlapping communication. This is especially difficult to achieve if one has irregular lattice structures or even regular lattice structures if the physics is inhomogeneous, such as shocks or a lot of activity in a narrow domain, or in systems with a time varying geometry. Some causes of load imbalance are:

- $N \bmod P \neq 0$,
- Boundary conditions,
- Irregular data structures,
- Inhomogeneous computation or communication loads,
- Time-varying geometry or topology.

4.9 Load Balance Methods

4.9.1 Static Load Balancing

In static load balancing, the problem is divided up and tasks are assigned to processors once and only once. The number or size of tasks may be varied to account for different computational powers of machines. This is the method that is used in regular problems with static geometries. It is well adapted to both SIMD and MIMD machines. For complex geometries, the task is much more difficult, it is NP-complete. In fact, the simultaneous matching of the following three requirements:

1. Even computational work (volume),
2. Even communication,
3. Minimal Surface-to-Volume,

cannot be solved exactly for arbitrary geometries. As a result, quasi-optimal optimization techniques are used. Among them, perhaps the most popular is Recursive Spectral Bisection.

Recursive Spectral Bisection

Suppose we have a finite element geometry with no regularity in the mesh (see fig. 4.7). The number of neighbors varies from point to point. There is a recipe we can use to obtain the minimum surface to volume ratio partition. First, the Laplace matrix is defined as $\mathcal{L}_{ij} = D_{ij} - A_{ij}$, where

$$D_{ij} = \nu \delta_{ij}\,, \tag{4.25}$$

$$A_{ij} = \begin{cases} 1, & \text{if } |i-j| = 1 \\ 0, & \text{otherwise.} \end{cases} \tag{4.26}$$

A_{ij} is a connectivity matrix which is 1 if two nodes are connected and zero if they are not (for $j \neq i$). For a regular grid, this reduces to the usual discretization of the Laplace operator. The Fiedler vector, $\overrightarrow{x}^{(1)}$, is the eigenvector that corresponds with the first non-zero eigenvalue of the Laplace matrix. It turns out that the Fiedler vector contains the "metric of the system." Once we get the coordinates of the Fiedler vector, we can sort the nodes, $\overrightarrow{x}_n$, along $\overrightarrow{x}$. In other words, we project our domain onto the x-axis and all of the nodes that fall between two adjacent coordinates in the Fiedler vector are given to the same processor. This optimizes the partitioning of the computational domain. The scheme can then be implemented recursively. This is by far the most common scheme used in engineering calculations.

Figure 4.7: This is a cartoon of the domain for internal engine combustion. This illustrates a well divided complex geometry. The cuts are always made orthogonal to the leading dimension (different colors go with different processors). This is done with a method called Recursive Coordinate Bisection.

4.9.2 Dynamic Load Balancing by Pool of Tasks

This method is typically used with the master / slave scheme. The master keeps a queue of tasks and continues to give them to idle slaves until the queue is empty. Faster machines get more tasks naturally.

4.9.3 Dynamic Load Balancing by Coordination

This method is typically used in the SPMD programming model. All the tasks stop and redistribute their work at fixed times or if some condition occurs, such

as if the load imbalance between tasks exceeds some limit. This method may be ill-suited to message-passing machines if the processor workload undergoes abrupt changes in time.

4.10 Redundancy

Optimal parallel algorithms are not necessarily the plain extension of their optimal serial counterpart. In order to achieve optimal parallel efficiency, one is sometimes forced to introduce extra computations which are simply not needed by the serial algorithm. Perhaps the most straightforward example is the so-called "overlapped domain decomposition" technique, where, in order to speed-up convergence of the parallel algorithm, the subdomains are forced to overlap along the interprocessor boundaries. Since the overlapping regions are computed by all processors which own them, a certain amount of extra computation cannot be avoided. The bargain is to offset these extra computations by the gain in convergence speed. A typical layout of a matrix resulting from overlapped domain decomposition is shown in Fig. 4.8.

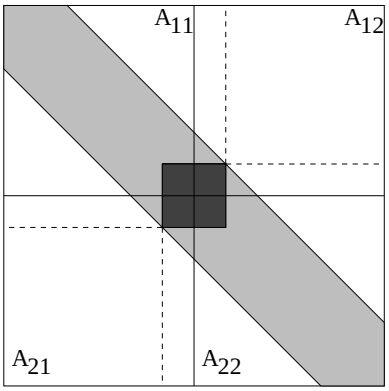

Figure 4.8: Overlap partitioning for sparse matrices.

Chapter 5

Example Programs

In this chapter, we shall deal with the problem of solving a simple partial differential equation, the heat equation (5.1) on the line, illustrating four distinct programming styles:

1. Serial Program.
2. Parallel Program on Shared-Memory MIMD Machine.
3. Parallel Program on Distributed-Memory MIMD Machine.
4. Parallel Program on Distributed-Memory SIMD Machine.

5.1 The Heat Equation

We shall consider the one-dimensional diffusion equation

$$\frac{\partial \psi}{\partial t} = D \frac{\partial^2 \psi}{\partial x^2}, \tag{5.1}$$

with the following initial and boundary conditions

$$\psi(t=0,x) = \psi_0(x), \tag{5.2}$$
$$\psi(x=0,t) = \psi(x=L,t), \qquad \text{(Periodic)}. \tag{5.3}$$

Using centered finite differences in space and Euler forward time stepping, equation (5.1) yields

$$\hat{\psi}_l = b\psi_{l-1} + a\psi_l + b\psi_{l+1}, \tag{5.4}$$

where a caret denotes $t + \Delta t$, and l labels the spatial lattice. The numerical coefficients are given by

$$a = 1 - 2D\frac{\Delta t}{(\Delta x)^2}, \tag{5.5}$$

$$b = D\frac{\Delta t}{(\Delta x)^2}. \tag{5.6}$$

Boundary conditions are handled by "ghost" nodes, as seen in fig. 5.1.

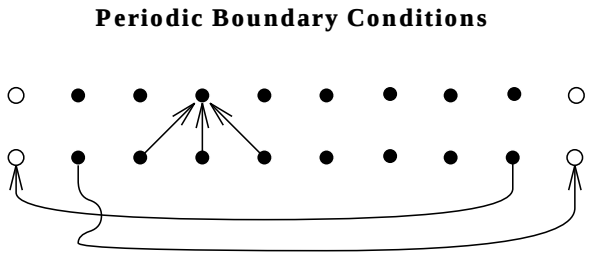

Figure 5.1: Boundary Conditions as handled by "ghost" nodes.

5.2 SISD Code for the Heat Equation.

The serial code consists of four logical steps (fig. 5.2):

1. Initialize the system (INPUT).
2. Update the state of the system at time $t + \Delta t$ (UPDATE).
3. Advance the system by preparing for the next time step (ADVANCE).
4. OUTPUT.

Note that boundary conditions are implemented via "ghost" nodes (locations 0 and $n + 1$), serving as temporary buffers for the boundary values. Also note that two levels of storage are explicitly retained (**psi** and **psi_new**) for the sake of simplicity.

The Code

```
c-----------------------------------------------------------------------
c   Heat Equation: serial Program
c   IBM XL Fortran 3.1
c-----------------------------------------------------------------------
        parameter (n=100,nstep=100)
        dimension psi(0:n+1),psi_new(0:n+1)
c -----------------------------------------------------------------------
```

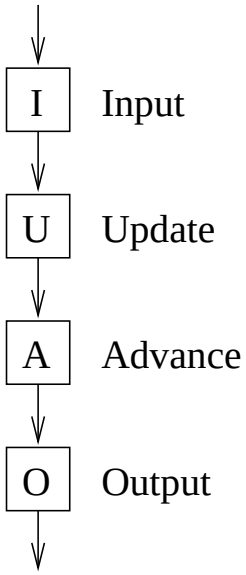

Figure 5.2: Time line of the serial code.

```
        open(unit=21,file='sisd.out')
        dt = 1.
        dx = 1.
        D  = 0.1
        b = D*dt/(dx*dx)
        a = 1.- 2.*b

        pi = 4.*atan(1.)

c  time-step control
        if(b.ge.1.) stop "STOP HERE, DT TOO HIGH!!!"
c  initialize <-------------------------------------------
         do i=1,n
           psi(i) = sin(pi*float(i))
           write(21,*) i,psi(i)
         end do
c  time loop
        do istep=1,nstep

c periodic boundary conditions
         psi(0) = psi(n)
         psi(n+1)= psi(1)
c update  <--------------------------------------------
        do i=1,n
```

```
          psi_new(i) = b*psi(i-1) + a*psi(i) + b*psi(i+1)
        end do
c advance <-----------------------------------------------
         do i=1,n
           psi(i) = psi_new(i)
         end do
c output  <-----------------------------------------------
         if(mod(istep,50).eq.0) then
          write(21,*) 'start step n.   ',istep
          do i=1,n
           write(21,*) i,psi(i)
          end do
         endif

        end do      ! end time looping

        stop
        end
------- end -------
```

5.3 MIMD Code for the Heat Equation – Shared Memory

As discussed in chapter 1, on a shared-memory machine, each parallel task is entitled to reference the whole addressing space as a single shared resource. With respect to the serial code, four additional actions are required:

1. Task generation (ORIGINATE).
2. Task operation (DISPATCH).
3. Task synchronization (WAIT).
4. Task termination (TERMINATE).

Task Generation

This is accomplished by the ORIGINATE ANY TASK parallel FORTRAN call (IBM VS FORTRAN). Each task defines the left-most boundary of its own computational domain (**ibegin**) as well as the corresponding size (**isize**). This information is needed to allow each task to proceed concurrently just on its own share of data.

Task Operation

Here, each task invokes a separate instance of the same routine (UPDATE), which performs the update of the discrete variables owned by the given task.

Global address space is made available via the **psicom** common, and specific data are properly addressed via **ibegin** and **isize**. Note that having used two distinct arrays for time t and $t + \Delta t$, no errors due to data corruption can occur. What may eventually occur is competition for the boundary cells, which may be needed for two tasks at a time (fig. 5.3). Such a problem is no concern for the routine ADVANCE, which just copies **psi** onto **psi_new** with no memory shifts.

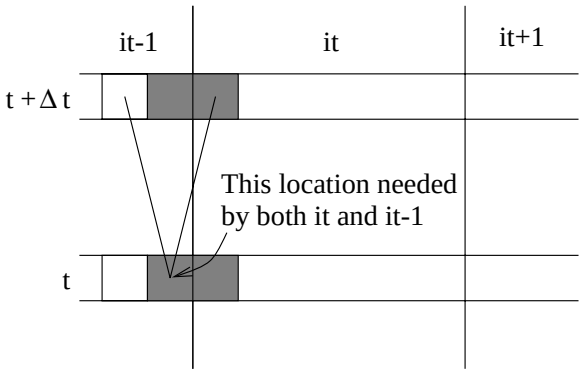

Figure 5.3: Competition for the same memory location by two different tasks.

Task Synchronization

In order to guard against time-misalignment between the various tasks, a global barrier (WAIT FOR ALL TASKS) is implemented at completion of both the UPDATE and ADVANCE steps.

Task Termination

Once all the time-cycles are completed, the tasks have exhausted their function and can consequently be annihilated (fig. 5.4). Note that we have been cavalier about parallel I/O operations.

The Code

```
c ------------------------------------------------------------
c   Heat Equation: MIMD Program for shared_memory multiprocessor
c                  Coarse-grain implementation
c                  IBM VS PARALLEL FORTRAN
c ------------------------------------------------------------
        parameter (n=100,n1=n+1)
        parameter (ntask=4)
        common /psicom/ psi,psi_new
        dimension psi(0:n1),psi_new(0:n1)
```

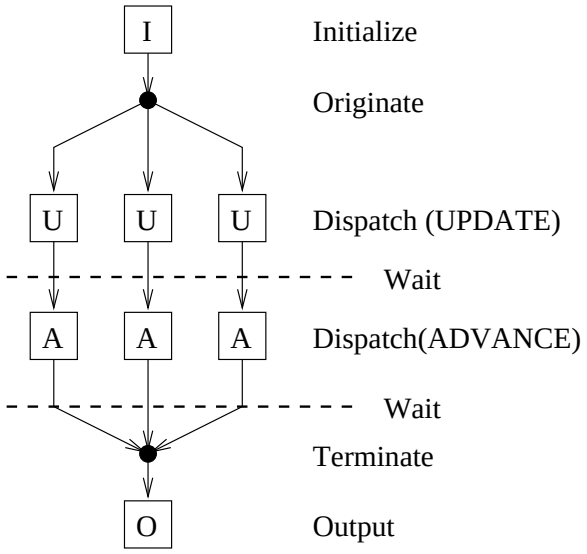

Figure 5.4: Timeline of the shared-memory code.

```
        dimension ibegin(ntask),isize(ntask)
c -------------------------------------------------------------
        open(unit=21,file='shm.out')
        nstep = 100
        dt = 1.
        dx = 1.
        D  = 0.1
        a = D*dt/(dx*dx)
        b = 1. -2.*a
        pi= 4.*atan(1.)
c  time-step control
        if(a.ge.1.) stop "STOP HERE, DT TOO HIGH!!!"
c ----------------------------------------> initialize (one task for al
        do i=1,n
         psi(i) = sin(pi*float(i))
        end do

        do 10 it=1,ntask                        ! spawn ntask tasks
         isize  (it) = n/ntask
         ibegin(it) = (it-1)*isize(it) + 1
         ORIGINATE ANY TASK ITASK(it)
10      continue
c ----------------------------------------> time evolution
        do 1000 istep=1,nstep
```

```
         do 20 it=1,ntask                      ! tasks in action
           dispatch task itask(it),
     +     sharing(psicom),
     +     calling(UPDATE(ibegin(it),isize(it),nstep,a,b)

           WAIT FOR ALL TASKS

           dispatch task itask(it),
     +     sharing(psicom),
     +     calling(ADVANC(ibegin(it),isize(it),nstep,a,b)

           WAIT FOR ALL TASKS
20       continue
c ------------------------------------------------> output (one task for all)
        if(mod(istep,50).eq.0) then
         do i=1,n
          write(21,*) psi(i)
         end do
        endif

1000    continue                                   ! end time-loop
c -------------------------------------------------------------
        do 1100 it=1,ntask
1100    TERMINATE TASK ITASK(it)                   ! fork-in

        stop
        end
c =====================================================
        subroutine UPDATE (imin,ichunk,nstep,a,b)
c =====================================================
        parameter (n=100,n1=n+1)
        common /psicom/ psi,psi_new
        dimension psi(0:n1),psi_new(0:n1)
c -------------------------------------------------------
c   boundary conditions        (no risk, psi is read-only at this stage)
        psi(0) = psi(n)
        psi(n1)= psi(1)
*  loop within processor data
        do i=imin,imin+ichunk
          psi_new(i) = b*psi(i-1) + a*psi(i) + b*psi(i+1)
        end do

        return
        end
c =====================================================
        subroutine ADVANC(imin,ichunk,nstep,a,b)
```

```
c ================================================
c   advance
        parameter (n=100,n1=n+1)
        common /psicom/ psi,psi_new
        dimension psi(0:n1),psi_new(0:n1)
c -------------------------------------------------------
        do i=imin,imin+ichunk
         psi(i) = psi_new(i)
        end do

        return
        end
------- end -------
```

5.4 MIMD Code for the Heat Equation – Distributed Memory

As opposed to a shared-memory environment, in a distributed-memory (DM) context, data intercommunication among the various processors takes place via explicit messages. This requires the programmer to specify in full detail all the information needed to complete the message-passing operation. The sample DM code presented here consists of the following logical sections:

1. System query (MP_ENVIRON).
2. Setting up the interprocessor communication topology.
3. Local initialization.
4. Task synchronization (MP_WAIT).
5. Message passing (MP_BSEND / MP_BRECV).
6. Update.
7. Advance.
8. Output.

System Query

Each task queries the number of tasks involved in the parallel job (**ntask**) as well as its own task identity (**taskid**, an integer between 0 and **ntask**-1).

Communication Topology

Each processor defines a left and right neighbor, according to a one-dimensional cyclic geometry, as seen in figure 5.5.

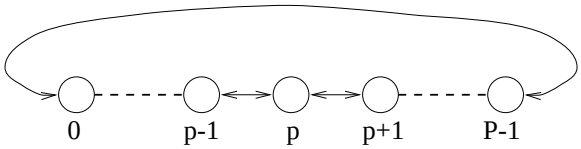

Figure 5.5: One-dimensional cyclic geometry.

Initialization

Each task initializes its own sub-piece of the array **psi**. Note that a linear map between the local (i) and global (ig) addresses is needed.

Task Synchronization

As usual, all processes must be aligned and synchronized before starting and after completing each time cycle.

Message-Passing

This is the tricky part of the code. Message-passing is organized in 4 steps (fig. 5.6):

1. Even tasks send to left $\equiv$ Odd tasks receive from right.
2. Even tasks receive from left $\equiv$ Odd tasks send to right.
3. Even tasks send to right $\equiv$ Odd tasks receive from left.
4. Even tasks receive from right $\equiv$ Odd tasks send to left.

The reason for organizing communication according to this sort of "parlor game" is to make sure that each request for sending / receiving is matched quickly by a corresponding request for receive / send. This is to "produce and consume asap" the messages, thus minimizing the network traffic. In addition, this pattern is contention-free in the sense that at each stage, disjoint pairs of processors are in communication.

Update and Advance

Once the parlor game is over, the code can proceed as if it were serial, since the communication buffers, **psi**(0) and **psi**(np-1), have been properly set up during the message-passing stage. Note that no explicit barrier is required since synchronization has been performed implicitly by the blocking send and receive primitives.

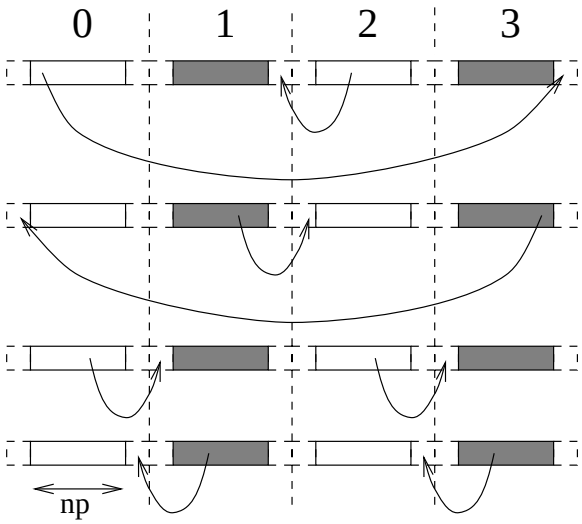

Figure 5.6: Message-passing in four steps.

Output

Each processor writes its share of data on a different file, much the same way a serial code would. Given the fact that output is produced for a series of time snapshots, say at $t = 0, 50, 100$, a recomposition tool is needed to reproduce a globally consistent output file (fig. 5.7). Such a recomposition step will be automatically taken care of by parallel I/O facilities whose description is beyond the scope of this book.

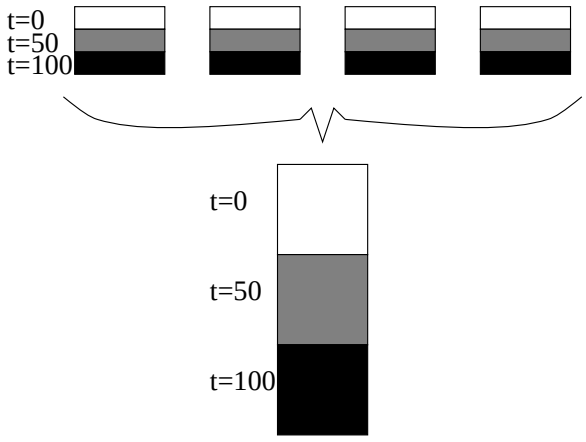

Figure 5.7: Recomposition of data from different processors.

The Code

Note that this program could have been written in a much more compact form using collective communication primitives (MP_SHIFT).

```
c-------------------------------------------------------------------------
c   Heat Equation: MIMD Program for Distributed memory multiprocessor
c                  SPMD: all execute the same code
c                  IBM MPL Library
c-------------------------------------------------------------------------
        parameter (n=100)
        parameter (ntask=4,np=n/ntask,np1=np+1)
        dimension psi(0:np1),psi_new(0:np1)
        integer taskid
        integer left(0:ntask-1),right(0:ntask-1)
c ------------------------------------------------------------------------
        dt = 1.0
        dx = 1.0
        D  = 0.1
        b = D*dt/(dx*dx)
        a = 1. -2.*b
        nstep = 100
        pi = 4.*atan(1.)
c  time-step control
        if(b.ge.1.) stop "STOP HERE, DT TOO HIGH!!!"
c ------------------------------------------------------------------
c  message passing prologue: how many procs? whoami?

        call MP_ENVIRON(ntask,taskid)

c interprocessor communication topology (redundant)

        do iproc = 0,ntask-1
         left(iproc)  = iproc-1
         right(iproc) = iproc+1
        end do
        left     (0) = ntask-1
        right(ntask-1) = 0

c   code message type: moving left  +0 ; right  +1
c ---------------------------------------------------------
c   initialize: each processor its piece-only (SPMD-mode)
        do i=1,np
          ig = i + np*taskid         ! note global address ig
          psi(i) = sin(pi*float(ig))
        end do
```

```
c -------------------------------------------------------
        call MP_WAIT(allmsg,nbytes)     !  global barrier: all aligned befor
        write(6,*) 'barrier passed'

c   time loop
        do istep=1,nstep

c   message passing machinery in action !

        msglen = 4                      !    # of bytes transmitted

       if (mod(taskid,2).eq.0) then

c even send to left odd
c       0   1<-----2     3<----0
        mtype  = 0
        mdest  = left(taskid)
        call MP_BSEND(psi(1),msglen, mdest, mtype)
c even receive from left odd
c       0   1----->2     3---->0
        mtype  = +1
        msrce  = left(taskid)
        call MP_BRECV(psi(0),msglen, msrce, mtype,nbytes)
c even send to right odd
c       0----->1     2---->3
        mtype  = +1
        mdest  = right(taskid)
        call MP_BSEND(psi(np1),msglen, mdest, mtype)
c even receive from right odd
c       0<-----1     2<----3
        mtype  = 0
        msrce  = right(taskid)
        call MP_BRECV(psi(np1),msglen, msrce, mtype,nbytes)

       else         ! ======================================================

c  odd receive from right even
c       0   1<-----2     3<----0
        mtype  = 0
        msrce  = right(taskid)

        call MP_BRECV(psi(np1),msglen, msrce, mtype,nbytes)

c  odd send to right even
c       0   1----->2     3---->4     5  ......
        mtype  = +1
```

```
        mdest  = right(taskid)

        call MP_BSEND(psi(np),msglen, mdest, mtype)

c  odd  receive from left even
c       0----->1     2---->3
        mtype  = +1
        msrce  = left(taskid)

        call MP_BRECV(psi(0),msglen, msrce, mtype,nbytes)

c  odd send to left even
c       0<-----1     2<----3
        mtype  = 0
        mdest  = left(taskid)

        call MP_BSEND(psi(0),msglen, mdest, mtype)


       endif

c   boundary conditions: not needed; embedded in the update step
c -------------------------------------------------------
c    serial lookalike section
c    from now on the code is serial lookalike

        do i=1,np
          psi_new(i) = b*psi(i-1) + a*psi(i) + b*psi(i+1)
        end do

c   advance

         do i=1,np
           psi(i) = psi_new(i)
         end do

        call MP_WAIT(allmsg,nbytes)               ! all aligned before next step
c ---------------------------------------------------------------------
c output (distributed, each on its file: 21, 22, 23 ....)
c these files need to be sensibly recomposed in the absence of parallel I/O‘
        if(mod(istep,50).eq.0) then
          do i=1,np
           write(21+taskid,*) psi(i)
          end do
        endif
```

```
        if(taskid.eq.0) write(6,*) 'completed step n. ',istep

        end do      ! end time looping

        stop
        end
------- end -------
```

5.5 SIMD Code for the Heat Equation

SIMD machines, and the corresponding data-parallel programming model exhibit a natural match to regular grid problems, such as the one under inspection. No wonder that this programming paradigm gives rise to the most compact and readable code, if not the most efficient. The logic of our SIMD code barely needs any further explanation, except for a few technical remarks.

First, we note that the parallel structure of the code is entirely hidden from the programmer, who just sees a single global address space as in a serial program. The code only manipulates global arrays with no need to reference indexed memory locations, thus avoiding any loops. The interprocessor communication involved by the update stage is triggered by "stencil software" routines, like **cshift**, with no need for the programmer to spell out any details of the communication topology. These will be passed to the compiler by a data distribution directive, such as

```
       DISTRIBUTE psi(block),psi_new(block)
```

informing the compiler to drop data into **nprocs** blocks, each of size **n/nprocs**.

The Code

```
c-----------------------------------------------------------------------
c   Heat Equation: Data parallel SIMD program
c                  IBM XL FORTRAN 3.1 (FORTRAN 90 features inside)
c                  emulates data parallel execution
c-----------------------------------------------------------------------
        parameter (n=100,nstep=100)
        real :: x(1:n)
        real :: psi(1:n),psi_new(1:n)
c ----------------------------------------------------------------------
        open(unit=21,file='simd.out')

        dt = 1.
        dx = 1.
        D  = 0.1
        b = D*dt/(dx*dx)
```

```
        a = 1.- 2.*b

        pi = 4.*atan(1.)

c  time-step control
        if(b.ge.1.) stop "STOP HERE, DT TOO HIGH!!!"
c -------------------------------------------------------
c   initialize
           do i=1,n
            x(i) = pi*float(i)
           end do

           psi = sin(x)         ! parallel intrinsic function
c -------------------------------------------------------
c  time loop
        do istep=1,nstep

c update: data communication is automatically taken care of by the compiler

         psi_new = b*cshift(psi,-1,1) + a*psi + b*cshift(psi,+1,1)

c advance: perfect alignement: no communication

         psi = psi_new

c output
         if(mod(istep,50).eq.0) then
          write(21,*) 'start step n.   ',istep
          do i=1,n
           write(21,*) i,psi(i)
          end do
         endif

        end do      ! end time looping

        stop
        end
------- end -------
```

Chapter 6

Sample Application: Lattice Boltzmann Fluid Dynamics on the IBM SP2

In this chapter we shall present in a great detail an extended excerpt of a message-passing code for three-dimensional fluid dynamics based upon the Lattice Boltzmann method. The text reported here refers to an early version of the code written for the IBM SP1 scalable parallel computer by F. Massaioli, G. Punzo and the first author.

6.1 LBE Dynamics

The Lattice Boltzmann Equation method can be regarded as a finite difference map, even though this theory was derived from kinetic theory rather than from finite difference methods. At each lattice site, we evolve a population, f_i, which is bounded between 0 and 1. These f_i propagate in the directions shown in fig. 6.1, with velocities, c_i. This mimics the free-streaming in the Boltzmann Equation or in the Navier-Stokes equation. The second step is particle interaction, i.e. scattering from direction i to direction j. This scattering is active as long as the distribution function has not yet relaxed to a local equilibrium. Note that the right hand side of equation (6.1) is completely local, while the left hand side involves nearest neighbor propagation only.

This problem is well suited for parallel computing because the calculation to communication ratio is high. Here communication is minimal since we are only looking at nearest neighbor interactions but the complexity of the calculation is roughly b^2 where b, the number of discrete speeds, is approximately 20 in three dimensions.

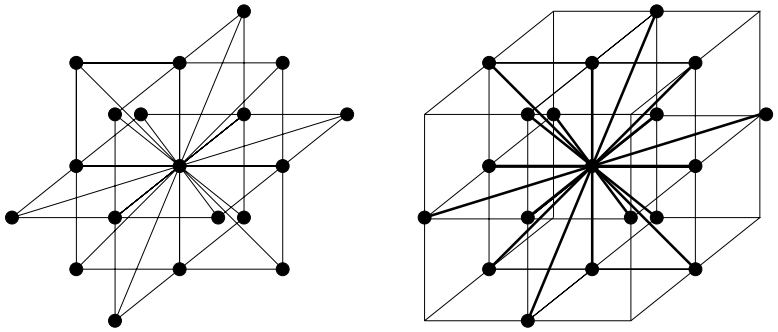

Figure 6.1: Stencil of LBE computation.

The Lattice Boltzmann Equations for gas dynamics are:

$$f_i(\vec{x}+\vec{c}_i,t+1)-f_i(\vec{x},t) = \sum_{j=1}^{b} A_{ij}(f_j-f_j^e)(x,t), \tag{6.1}$$

$$f_j^e(x,t) = d(1+\frac{\vec{c}_i\vec{u}}{c_s^2}+\frac{1}{2c_s^4}\vec{\vec{Q}}_i:\vec{u}\vec{u}), \tag{6.2}$$

$$\vec{\vec{Q}}_i=\vec{c}_i\vec{c}_i-c_s^2\vec{\vec{I}}, \qquad c_s^2=1/2. \tag{6.3}$$

For the specific example we will look at, $b = 24$. The scheme is elementary. However, since the stencil is somewhat "chunky," there is a considerable amount of bookkeeping that needs to be done in the code.

6.2 LBE Parallelization

The code consists of three stages:

1. Pre-processing, which includes initial conditions, boundary conditions, grid set-up, etc.

2. Solving the equations. This is done in parallel.

3. Collecting the data to get back to a serial environment.

The initial step can be trivially parallelized. The final step is less trivial, especially if the geometry is complicated. One would like to use existing tools, rather than writing this step on one's own. In the pre-processing stage, we set the initial conditions and the topology information. The topology includes the boundaries of the computational domain, the list of neighbors, etc.

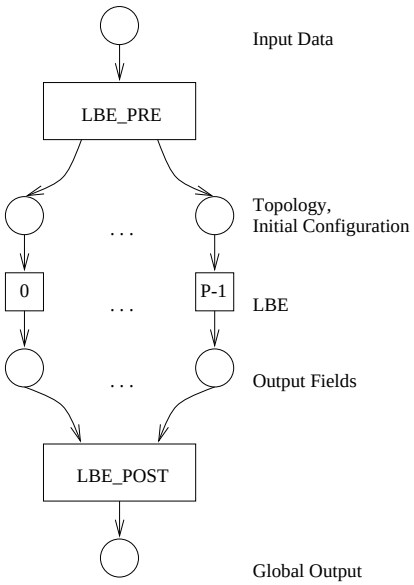

Figure 6.2: LBE Parallelization

6.2.1 LBE_PRE

The preprocessing stage, LBE_PRE, does two things. First, it initializes the population, f_i (A01 to A24 in the code), and writes them onto a set of separate files, one per processor. Second, it prepares the interprocessor communication topology and writes the corresponding information on the files mentioned above (20 + processor identity).

The interconnection topology is a two-dimensional mesh along y and z with processors numbered in lexicographic y–z order, as in figure 6.3.

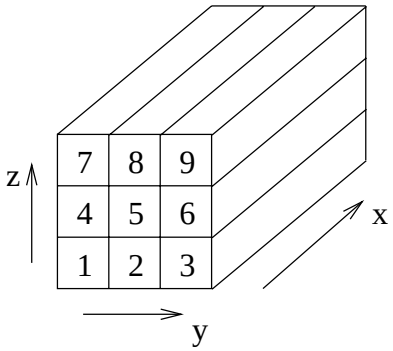

Figure 6.3: 2-D partition.

Each processor gets four neighbors, known as **lefty** (left along y), **righty** (right along y), **leftz** (left along z), and **rightz** (right along z). For example, processor 5 in figure 6.3 has **lefty**=4, **righty**=6, **leftz**=2, and **rightz**=8. Processors lying on the border get assigned -1 for non-existing neighbors. For example, **lefty**(1) = -1 and **rightz**(8) = -1. Each subdomain of size $l \times m \times n$ is identified by a pair of offsets, **jstart** and **kstart**, whose meaning is made clear in figure 6.4.

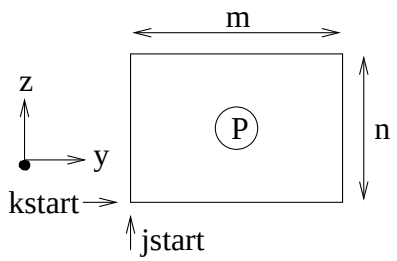

Figure 6.4: Offsets **jstart** and **kstart**.

The Pre-processing Code

lbehw.copy

```
C
C INCLUDE FILE WITH LATTICE DIMENSIONS FOR LBE 3D CONVECTION
C

cmicro                 PARAMETER (LGL = 32 ,  MGL = 32  , NGL = 32  )
          PARAMETER (LGL = 32 ,  MGL = 48  , NGL = 64  )  ! 18M
cmed            PARAMETER (LGL = 64 ,  MGL = 96  , NGL = 128 )  ! 144M
cbig            PARAMETER (LGL = 128,  MGL = 192 , NGL = 256 )  ! 1152
cgiga           PARAMETER (LGL = 256,  MGL = 192 , NGL = 256 )  ! 2304
cgiga2          PARAMETER (LGL = 256,  MGL = 256 , NGL = 256 )  ! 3272
cgiga3          PARAMETER (LGL = 256,  MGL = 512 , NGL = 256 )  ! 6544
cgiga4          PARAMETER (LGL = 512,  MGL = 512 , NGL = 512 )  ! 6544
```

lbepar.copy

```
C
C INCLUDE FILE WITH LATTICE DIMENSIONS FOR LBE 3D CONVECTION
C
           include 'lbehw.copy'

c1           PARAMETER ( NPY = 1, NPZ = 1 )
c2           PARAMETER ( NPY = 1, NPZ = 2 )
c4             PARAMETER ( NPY = 2, NPZ = 2 )
c6           PARAMETER ( NPY = 2, NPZ = 3 )
           PARAMETER ( NPY = 2, NPZ = 4 )
```

```
c12           PARAMETER ( NPY = 3, NPZ = 4 )
c16           PARAMETER ( NPY = 4, NPZ = 4 )
c24           PARAMETER ( NPY = 4, NPZ = 6 )
c32           PARAMETER ( NPY = 4, NPZ = 8 )
c64           PARAMETER ( NPY = 8, NPZ = 16)

            PARAMETER (L = LGL,  M = MGL/NPY , N = NGL/NPZ )
```

lbeproc.copy

```
C ******************************************************************
C * INCLUDE FILE FOR LBECNV3D
C ******************************************************************

        INCLUDE 'lbepar.copy'

        PARAMETER (LGL1=LGL+1, MGL1=MGL+1, NGL1=NGL+1)

        PARAMETER (L1=L+1, M1=M+1, N1=N+1 )
CG LEN=number of bytes involved in each transmission
CG if using real*4 => *4, if real*8 => *8 in the following multiply
        PARAMETER ( LEN = (M+2)*(L+2)*4 )
CG needed by EUI simulator
      parameter (mode=3)

        REAL*4
     *  A01(0:LGL1,0:MGL1,0:NGL1), A02(0:LGL1,0:MGL1,0:NGL1),
     *  A03(0:LGL1,0:MGL1,0:NGL1), A04(0:LGL1,0:MGL1,0:NGL1),
     *  A05(0:LGL1,0:MGL1,0:NGL1), A06(0:LGL1,0:MGL1,0:NGL1),
     *  A07(0:LGL1,0:MGL1,0:NGL1), A08(0:LGL1,0:MGL1,0:NGL1),
     *  A09(0:LGL1,0:MGL1,0:NGL1), A10(0:LGL1,0:MGL1,0:NGL1),
     *  A11(0:LGL1,0:MGL1,0:NGL1), A12(0:LGL1,0:MGL1,0:NGL1),
     *  A13(0:LGL1,0:MGL1,0:NGL1), A14(0:LGL1,0:MGL1,0:NGL1),
     *  A15(0:LGL1,0:MGL1,0:NGL1), A16(0:LGL1,0:MGL1,0:NGL1),
     *  A17(0:LGL1,0:MGL1,0:NGL1), A18(0:LGL1,0:MGL1,0:NGL1),
     *  A19(0:LGL1,0:MGL1,0:NGL1), A20(0:LGL1,0:MGL1,0:NGL1),
     *  A21(0:LGL1,0:MGL1,0:NGL1), A22(0:LGL1,0:MGL1,0:NGL1),
     *  A23(0:LGL1,0:MGL1,0:NGL1), A24(0:LGL1,0:MGL1,0:NGL1)

      INTEGER NTASK, TASKID, ALLMSG, ALLGRP, QBUF(4)
      INTEGER LEFTY, LEFTZ, RGHTY, RGHTZ

        COMMON /CPOI3/
     *  A01,A02,A03,A04,A05,A06,A07,A08,A09,
     *  A10,A11,A12,A13,A14,A15,A16,A17,A18,
     *  A19,A20,A21,A22,A23,A24
```

```
        COMMON /PAR/ SVISC,SCHI,ALFAG,DELTAT

        COMMON /COLL/ A,B,C,D,E,F,AMF,CPF,EMF,FRCEZ,FRCET

        COMMON /EIGEN/ ELAMBDA,EMU,TAU,SIGMA

        COMMON /NDOSTAMO/ NTASK, TASKID, ALLMSG, ALLGRP, QBUF

        COMMON /TOPOLOGY/ LEFTY, LEFTZ, RGHTY, RGHTZ
```

preproc.f

```
          PROGRAM PREPROC

C
C STARTING CONDITION FOR 3D LBE CONVECTION SIMULATION
C
C VELOCITY FIELD WITH ZERO DIVERGENCE
C
C     20 IS OUTPUT FILE
C

          INCLUDE 'preproc.copy'

          PARAMETER(QF = 1./12., RF = 1./288.)

          PARAMETER(PI=3.1415926)

          PARAMETER(AM1=0.16, WX1=1., WY1=2., WZ1=3.)

          PARAMETER(AM2=0.21, WX2=4., WY2=5., WZ2=6.)

          REAL*4 CXJ(LGL,MGL,NGL),
     *           CYJ(LGL,MGL,NGL),
     *           CZJ(LGL,MGL,NGL),
     *           CTJ(LGL,MGL,NGL)
          logical serial
          dimension ipp(0:npy-1,0:npz-1)
c -------------------------------------------------------------
CG          WRITE(6,*) 'DELTA T?'
CG          READ(5,*) DELTAT
           DELTAT = .2
c          WRITE(6,*) 'SERIAL? (T/F)'
c          READ(5,*) SERIAL
C          SERIAL = .FALSE.

          WRITE(6,*) 'BUILDING STARTING CONFIGURATION'
```

```
C
C VELOCITY FIELD
C

          AL1 = WX1*PI/FLOAT(LGL)
          BE1 = WY1*PI/FLOAT(MGL)
          GA1 = WZ1*PI/FLOAT(NGL)

          AL2 = WX2*PI/FLOAT(LGL)
          BE2 = WY2*PI/FLOAT(MGL)
          GA2 = WZ2*PI/FLOAT(NGL)

          DO K = 1, NGL
          DO J = 1, MGL
          DO I = 1, LGL

            X = FLOAT(I)-0.5
            Y = FLOAT(J)-0.5
            Z = FLOAT(K)-0.5

            CXJ(I,J,K) =
     *          AM1*SIN(AL1*X)*(GA1*COS(GA1*Z)-BE1*COS(BE1*Y))
     *         +AM2*SIN(AL2*X)*(GA2*COS(GA2*Z)-BE2*COS(BE2*Y))

            CYJ(I,J,K) =
     *          AM1*SIN(BE1*Y)*(AL1*COS(AL1*X)-GA1*COS(GA1*Z))
     *         +AM2*SIN(BE2*Y)*(AL2*COS(AL2*X)-GA2*COS(GA2*Z))

            CZJ(I,J,K) =
     *          AM1*SIN(GA1*Z)*(BE1*COS(BE1*Y)-AL1*COS(AL1*X))
     *         +AM2*SIN(GA2*Z)*(BE2*COS(BE2*Y)-AL2*COS(AL2*X))

          END DO
          END DO
          END DO

C
C BUILDS THE POPULATIONS
C

          DO K = 1, NGL

             TJ = -DELTAT * 24. *
     C        (FLOAT(NGL1)/FLOAT(2*NGL)-FLOAT(K)/FLOAT(NGL))
```

```
      DO J = 1, MGL
      DO I = 1, LGL

         XJ = 24.*CXJ(I,J,K)
         YJ = 24.*CYJ(I,J,K)
         ZJ = 24.*CZJ(I,J,K)

         VSQ=XJ*XJ+YJ*YJ+ZJ*ZJ+TJ*TJ

      A01(I,J,K) = QF*( XJ-YJ      ) + RF*((XJ-YJ)*(XJ-YJ)- .5*VSQ)
      A02(I,J,K) = QF*( XJ   -ZJ   ) + RF*((XJ-ZJ)*(XJ-ZJ)- .5*VSQ)
      A03(I,J,K) = QF*( XJ+YJ      ) + RF*((XJ+YJ)*(XJ+YJ)- .5*VSQ)
      A04(I,J,K) = QF*( XJ   +ZJ   ) + RF*((XJ+ZJ)*(XJ+ZJ)- .5*VSQ)
      A05(I,J,K) = QF*( XJ      +TJ) + RF*((XJ+TJ)*(XJ+TJ)- .5*VSQ)
      A06(I,J,K) = QF*(       ZJ+TJ) + RF*((ZJ+TJ)*(ZJ+TJ)- .5*VSQ)
      A07(I,J,K) = QF*(    YJ+ZJ   ) + RF*((YJ+ZJ)*(YJ+ZJ)- .5*VSQ)
      A08(I,J,K) = QF*(    YJ   +TJ) + RF*((YJ+TJ)*(YJ+TJ)- .5*VSQ)
      A09(I,J,K) = QF*(    YJ-ZJ   ) + RF*((YJ-ZJ)*(YJ-ZJ)- .5*VSQ)
      A10(I,J,K) = QF*(-XJ-YJ      ) + RF*((XJ+YJ)*(XJ+YJ)- .5*VSQ)
      A11(I,J,K) = QF*(-XJ   -ZJ   ) + RF*((XJ+ZJ)*(XJ+ZJ)- .5*VSQ)
      A12(I,J,K) = QF*(-XJ+YJ      ) + RF*((XJ-YJ)*(XJ-YJ)- .5*VSQ)
      A13(I,J,K) = QF*(-XJ   +ZJ   ) + RF*((XJ-ZJ)*(XJ-ZJ)- .5*VSQ)
      A14(I,J,K) = QF*(-XJ      +TJ) + RF*((XJ-TJ)*(XJ-TJ)- .5*VSQ)
      A15(I,J,K) = QF*(      -ZJ+TJ) + RF*((ZJ-TJ)*(ZJ-TJ)- .5*VSQ)
      A16(I,J,K) = QF*(   -YJ-ZJ   ) + RF*((YJ+ZJ)*(YJ+ZJ)- .5*VSQ)
      A17(I,J,K) = QF*(   -YJ   +TJ) + RF*((YJ-TJ)*(YJ-TJ)- .5*VSQ)
      A18(I,J,K) = QF*(   -YJ+ZJ   ) + RF*((YJ-ZJ)*(YJ-ZJ)- .5*VSQ)
      A19(I,J,K) = QF*( XJ      -TJ) + RF*((XJ-TJ)*(XJ-TJ)- .5*VSQ)
      A20(I,J,K) = QF*(       ZJ-TJ) + RF*((ZJ-TJ)*(ZJ-TJ)- .5*VSQ)
      A21(I,J,K) = QF*(    YJ   -TJ) + RF*((YJ-TJ)*(YJ-TJ)- .5*VSQ)
      A22(I,J,K) = QF*(-XJ      -TJ) + RF*((XJ+TJ)*(XJ+TJ)- .5*VSQ)
      A23(I,J,K) = QF*(   -YJ   -TJ) + RF*((YJ+TJ)*(YJ+TJ)- .5*VSQ)
      A24(I,J,K) = QF*(      -ZJ-TJ) + RF*((ZJ+TJ)*(ZJ+TJ)- .5*VSQ)

      END DO
      END DO
      END DO

      WRITE(6,*) 'START SAVING '

c     goto 1234
c      if (serial) then

      REWIND(20)

      WRITE(20) 0
```

```
           DO K=1,NGL
           DO J=1,MGL

              WRITE(20)
     *       (A01(I,J,K),A02(I,J,K),A03(I,J,K),A04(I,J,K),
     *        A05(I,J,K),A06(I,J,K),A07(I,J,K),A08(I,J,K),
     *        A09(I,J,K),A10(I,J,K),A11(I,J,K),A12(I,J,K),
     *        A13(I,J,K),A14(I,J,K),A15(I,J,K),A16(I,J,K),
     *        A17(I,J,K),A18(I,J,K),A19(I,J,K),A20(I,J,K),
     *        A21(I,J,K),A22(I,J,K),A23(I,J,K),A24(I,J,K),
     *        I=1,LGL)

           END DO
           END DO

c           else
c1234        continue

      np = npz*npy
      write(6,*) 'processor topology'
      write(6,*) '------------------'
      do ip = 1, np
c topology: process allocation: as fortran allocates data
c neighbours 0 if it is a wall
         lefty = (ip-1) * min( mod(ip-1,npy) , 1 ) -1
         leftz = max(ip - npy,0) -1
         rghty = (ip+1) * min( mod(ip,npy) , 1 ) -1
         rghtz = mod( min(ip+npy,np+1), (np+1) ) -1
         write(6,*) ip-1,lefty,rghty,leftz,rghtz

         rewind(20+ip)
         write(20+ip) 0, lefty, rghty, leftz, rghtz

         jstart = mod(ip-1,npy) * m
         kstart = ((ip-1)/npy) * n

         do k = kstart+1, kstart+n
            do j = jstart+1, jstart+m
               WRITE(20+ip)
     *    (A01(I,j,k),A02(I,j,k),A03(I,j,k),A04(I,j,k),
     *     A05(I,j,k),A06(I,j,k),A07(I,j,k),A08(I,j,k),
     *     A09(I,j,k),A10(I,j,k),A11(I,j,k),A12(I,j,k),
     *     A13(I,j,k),A14(I,j,k),A15(I,j,k),A16(I,j,k),
     *     A17(I,j,k),A18(I,j,k),A19(I,j,k),A20(I,j,k),
     *     A21(I,j,k),A22(I,j,k),A23(I,j,k),A24(I,j,k),
```

```
     *     I=1,L)
           end do
         end do
       end do

       WRITE(6,*) 'SAVE COMPLETED'

c plot the processor topology
       do 100 ipz=0,npz-1
       do 100 ipy=0,npy-1
        ipp(ipy,ipz) = ipz*npy + ipy
100    continue

       do 200 ipz=npz-1,0,-1
        write(6,*) (ipp(ipy,ipz),ipy=0,npy-1)
200    continue

       stop
       END
```

6.2.2 LBE

This is the fluid solver, the program to which parallelization applies. The core of the parallelization is lumped into a single routine, BCOND, which deals with the boundary conditions. There are actually two BCOND's, BCOND_EVEN and BCOND_ODD, which manage the communication from odd to even processors, and vice-versa. This split is motivated by the same needs discussed in the sample code for the heat equation, the "produce and consume asap" philosophy and a desire to be free of contention between processors.

For BCOND_EVEN, we distinguish three steps:

1. Message-passing preparation.
2. Intersparse boundary conditions.
3. Actual reception of messages.

Message-Passing Preparation

We first stipulate a naming convention for the messages:

- Message Type = 100 + ip, 200 + ip, 300 + ip. This is for the propogation of population ip ($1 \leq$ ip ≤ 24) along x, y, z respectively.
- Message Identity = mi$\alpha\beta\gamma$, where
  - $\alpha = s, r$, depending on whether the message is sent or received.

- $\beta = x, y, z$, depending on the direction of propagation.
- γ, the population number.

The program starts with message-passing preparation across the surface "west" and "east" of populations 7, 9, 16, and 18, which propagate according to figure 6.5. Note that diagonal propagation (yz) is decomposed into two subsequent steps, first along $y(z)$ and then along $z(y)$. This is to avoid latency overheads which would be triggered by sending "corner" elements or a separate message.

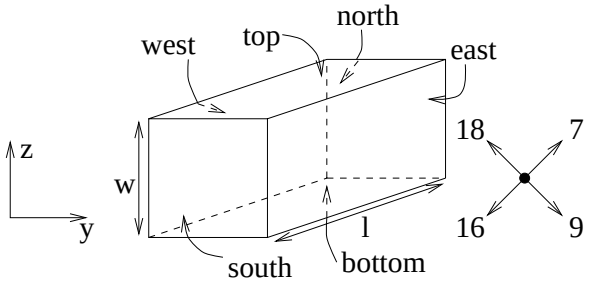

Figure 6.5: Propagation of populations 7, 9, 16, and 18.

We will focus on the "west" surface; 16 and 18 have to be sent and 7 and 9 received. Population 16 is sent with message type 200+16 = 216 and message id misy16 (send population 16 along y). For the sake of clarity, the array a16 is first packed into a buffer array, **buffone**, which stores the $l \cdot n$ memory locations corresponding to the surface "west" in a contiguous format. Today, this operation would not be needed since the IBM MPL takes care of direct send / receive of messages consisting of non-contiguous memory locations.

The sequence of operations is (fig. 6.6):

1. send 16.
2. receive 7.
3. send 18.
4. receive 9.

For the sake of compactness, we could have combined 16 and 18 into a single message, as well as 7 and 9. Since there are enough data to offset latency overheads, there is no compelling need for combining these messages. It is important to note that the non-blocking character of the send / receive primitives allows the CPU, while waiting for the message-passing operation to be completed, to move on to independent work. This reduces part of the communication overhead. This is the reason why the code, once the message-passing

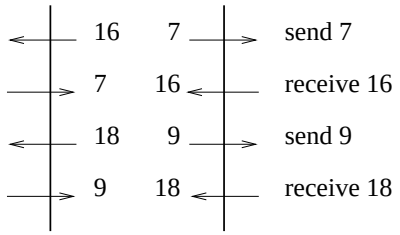

Figure 6.6: Sequence of operations for populations 7, 9, 16, and 18.

requests to send / receive on "west / east" surfaces are over, moves on to process independent boundary conditions, rather than going on with other message-passing requests. More specifically, on "north / south" walls, the code handles no-slip (zero velocity) boundary conditions, and on "west / east" walls, it handles free-slip (zero normal velocity) boundary conditions. Readers can convince themselves that these boundary conditions do not involve any data needed by the message-passing across "west / east", and can therefore proceed concurrently. Before moving to other message-passing operations, we need to secure correct completion of the message-passing across "west / east". This is performed in section 3 of the code, where selective barriers, **mp_wait**, are invoked upon completion of the messages issued in section 1. Note that selective barriers make use of the specific message-passing identities (misy16 .. miry09).

The Code

lbe_main.f
lbe.copy

```
        PROGRAM LBECNV3D
C PARALLEL PROGRAM: version for argonne: may 10 1994
c from S. Succi, G. Punzo, F. Massaioli, IBM Rome
C-------------------------------------------------------
C VQ = THETA = T-TEQ = T-(DELTAT/2-DELTAT/M*Y)
C-------------------------------------------------------
C        SVISC    : VISCOSITY
C        SCHI     : CONDUCTIVITY
C        TAU      : GHOST DENSITY EIGENVALUE
C        SIGMA    : GHOST MOMENTUM EIGENVALUE
C        ALFAG    : THERMAL EXPANSION COEFFICIENT * GRAVITY ACCELERATION
C        DELTAT   : LOWER-UPPER BOUNDARY TEMPERATURE DIFFERENCE
C        A01,A24  : ARRAY OF POPULATION, DEFINED AS : (NI-D)/D
C-------------------------------------------------------
C        ITFIN    : FINAL TIME STEP
C        ITSTART  : STARTING TIME STEP
C        IVTIM    : TIME LAP BETWEEN TWO MACRO-CONFIGURATION COMPUTATION
C        ITEMP    : TIME LAP BETWEEN TWO PROBES MEASURES
```

```
C        ITSAVE   : TIME LAP BETWEEN TWO SALVAGES
C        ICHECK   : TIME LAP BETWEEN TWO CONSERVATIONS CHECKS
C-------------------------------------------------------
C    INPUT AND OUTPUT FILES
C
C        20       : STARTING CONFIGURATION
C        16       :.COPY OF SCREEN OUTPUT
C        40       : SAVED CONFIGURATION (TO CONTINUE AT A LATER TIME)
C        60       : MACRO VARIABLES OUTPUT
C        95       : TEMPERATURE GRADIENTS
C        99       : TEMPERATURE MEASURES
C-------------------------------------------------------
C        DIRECTION  1    UNIT VECTOR = ( 1,-1, 0, 0)
C        DIRECTION  2    UNIT VECTOR = ( 1, 0,-1, 0)
C        DIRECTION  3    UNIT VECTOR = ( 1, 1, 0, 0)
C        DIRECTION  4    UNIT VECTOR = ( 1, 0, 1, 0)
C        DIRECTION  5    UNIT VECTOR = ( 1, 0, 0, 1)
C        DIRECTION  6    UNIT VECTOR = ( 0, 0, 1, 1)
C        DIRECTION  7    UNIT VECTOR = ( 0, 1, 1, 0)
C        DIRECTION  8    UNIT VECTOR = ( 0, 1, 0, 1)
C        DIRECTION  9    UNIT VECTOR = ( 0, 1,-1, 0)
C        DIRECTION 10    UNIT VECTOR = (-1,-1, 0, 0)
C        DIRECTION 11    UNIT VECTOR = (-1, 0,-1, 0)
C        DIRECTION 12    UNIT VECTOR = (-1, 1, 0, 0)
C        DIRECTION 13    UNIT VECTOR = (-1, 0, 1, 0)
C        DIRECTION 14    UNIT VECTOR = (-1, 0, 0, 1)
C        DIRECTION 15    UNIT VECTOR = ( 0, 0,-1, 1)
C        DIRECTION 16    UNIT VECTOR = ( 0,-1,-1, 0)
C        DIRECTION 17    UNIT VECTOR = ( 0,-1, 0, 1)
C        DIRECTION 18    UNIT VECTOR = ( 0,-1, 1, 0)
C        DIRECTION 19    UNIT VECTOR = ( 1, 0, 0,-1)
C        DIRECTION 20    UNIT VECTOR = ( 0, 0, 1,-1)
C        DIRECTION 21    UNIT VECTOR = ( 0, 1, 0,-1)
C        DIRECTION 22    UNIT VECTOR = (-1, 0, 0,-1)
C        DIRECTION 23    UNIT VECTOR = ( 0,-1, 0,-1)
C        DIRECTION 24    UNIT VECTOR = ( 0, 0,-1,-1)
C-------------------------------------------------------
      INCLUDE 'lbe.copy'

      CHARACTER*80 TITLE
      integer cpu1, cpu2, elap1, elap2, cputot, elaptot
      real*8 ttot1, ttot2, tmov1, tmov2, tcol1, tcol2, ttmov, ttcol
      real*8 trun1, trun2, tbcon1, tbcon2, ttbcon
      real*8 atime
C
C following routines are needed when using EUI
```

```
C
C this routine returns the number of tasks and the task identifier
C when the program is running in parallel mode
      call mp_environ(ntask,taskid)
c                                    ''WHO AM I''
c timing statements
      open(15,file='lbe.input')
      ttmov = 0.d0
      ttcol = 0.d0
      ttbcon = 0.d0
c -------------------------------------------------
      ttot1 = atime()

C
C READ INPUT PARAMETERS
C
                 CALL INPUT
     &    (TITLE,ITFIN,IVTIM,ITEMP,ITSAVE,ICHECK,IBENCH)

C WRITE RUN DATA ON OUTPUT FILE
C
                CALL OUTDAT
     &  (TITLE,ITFIN,ITSTART,IVTIM,ITEMP,ITSAVE,ICHECK)
C
C COLLISION PARAMETERS
C
                CALL HENCOL
C
C GET STARTING CONFIGURATION
C
C ibench = 0 => production
C ibench = 1 => benchmark

                if (ibench.eq.0) then
                  CALL PICK(ITSTART)
                else
                  call INIT
                endif

               CALL DIAGNO(ITSTART)
C
C>>>>>>>>>>   time loop    >>>>>>>>>>>>>>>>>>>>>>>>
C
         trun1=atime()

         DO 1000 ITIME=ITSTART+1,ITFIN
```

```
          if(taskid.eq.0) WRITE(6,*) 'itime',ITIME
C
C BOUNDARY CONDITIONS
C
       tmov1 = atime()
       tbcon1 = atime()

          if(mod(taskid,2).eq.0) then
C                                              Even talk to Odd
           CALL BCOND_EVEN
          else
C                                              Odd talk to Even
           CALL BCOND_ODD
          endif

       tbcon2 = atime()
       ttbcon = ttbcon + tbcon2 - tbcon1
C
C GET MACROSCOPIC VALUES
C

       IF (MOD(ITIME,IVTIM).EQ.0) THEN
                 CALL VARM(ITIME)
       END IF

       IF (MOD(ITIME,ITEMP).EQ.0) THEN
                 CALL VARTMP(ITIME)
       END IF

       IF (MOD(ITIME,ICHECK).EQ.0) THEN
                 CALL DIAGNO(ITIME)
       END IF
C
C PROPAGATION
C
       CALL MOVEF

      tmov2 = atime()
      ttmov = ttmov + tmov2 - tmov1
C
C COLLISION
C
      tcol1 = atime()

      CALL COL
```

```
      tcol2 = atime()
      ttcol = ttcol + tcol2 - tcol1
C
C SAVE ALL POPULATIONS
C
      IF (MOD(ITIME,ITSAVE).EQ.0) THEN
         CALL SAVE(ITIME)
      END IF

1000   CONTINUE

      trun2 = atime()
      ttot2 = atime()

      write(6,*) taskid,'tot time',ttot2-ttot1
      write(6,*) taskid,'tot run ',trun2-trun1
      write(6,*) taskid,'tot move',ttmov
      write(6,*) taskid,'tot bcond ',ttbcon
      write(6,*) taskid,'tot col ',ttcol

      stop
      END

\noindent {\bf bcond.f}
\begin{verbatim}
          SUBROUTINE BCOND_EVEN
css send receive for even processors
C
C BOUNDARY CONDITIONS
C LATERAL WALLS (EAST/WEST): FREE SLIP
C TOP & BOTTOM WALLS : NO SLIP & THETA = 0
C
c message types = i*100 + population number, where i = 1,2,3 for movemen
c    along x,y,z
c message id names = mi + {s|r (send/receive)} + {x|y|z (axis)} + popula
c                    number
c ---------------------------------------------------------------------
         INCLUDE 'lbe.copy'
         real*4 buffone((l+2)*n,8,2)
c    (L+2)*n; size of the surface
c    8      ; # of messages per processor
c    1,2    ; message sent/received across the surface
c
c---------------------------------------------------------------------
c                     SECTION 1
```

```
c--------------------------------------------------------------------
c 1st step: transmission of yz-populations along y: surface SY_MINUS (WEST)
      nllen = n * l * rlen

      if (lefty.ne.-1) then
         call my_pack(a16(1,1,1),buffone(1,1,1),l,yst,n)
         call mp_send(buffone(1,1,1),nllen,lefty,216,misy16)
         call mp_recv(buffone(1,3,2),nllen,lefty,207,miry07)

         call my_pack(a18(1,1,1),buffone(1,2,1),l,yst,n)
         call mp_send(buffone(1,2,1),nllen,lefty,218,misy18)
         call mp_recv(buffone(1,4,2),nllen,lefty,209,miry09)
      endif

      if (rghty.ne.-1) then
         call my_pack(a07(1,m,1),buffone(1,3,1),l,yst,n)
         call mp_send(buffone(1,3,1),nllen,rghty,207,misy07)
         call mp_recv(buffone(1,1,2), nllen,rghty,216,miry16)

         call my_pack(a09(1,m,1),buffone(1,4,1),l,yst,n)
         call mp_send(buffone(1,4,1),nllen,rghty,209,misy09)
         call mp_recv(buffone(1,2,2),nllen,rghty,218,miry18)
      endif
c
c--------------------------------------------------------------------
c                     SECTION 2
c--------------------------------------------------------------------
c Boundary Conditions on North/South.  Profit from non-blocking
c send/receive to keep on with independent work.
      jstart = 1
      jstop = m
      if (lefty.eq.-1) jstart = 2
      if (rghty.eq.-1) jstop = m-1

      do k = 1, n
         do j = jstart, jstop
            a10(l1,j,k) = a01(l,j,k)
            a12(l1,j,k) = a03(l,j,k)
            a01(0,j,k)  = a10(1,j,k)
            a03(0,j,k)  = a12(1,j,k)
         end do
      end do
c for xz- and yz-populations boundary conditions on x and y walls
      kstart = 1
      kstop = n
      if (leftz.eq.-1) kstart = 2
```

```
      if (rghtz.eq.-1) kstop = n-1

c x walls for xz-populations : before xz-populations z-transmissions
      do k = kstart, n
         do j = 1, m
            a11(l1,j,k) = a02(l,j,k)
            a02(0,j,k) = a11(1,j,k)
         end do
      end do
      do k = 1, kstop
         do j = 1, m
            a13(l1,j,k) = a04(l,j,k)
            a04(0,j,k) = a13(1,j,k)
         end do
      end do
c y walls for yz-populations : before yz-populations z-transmissions
      if (lefty.eq.-1) then
         do k = kstart, n
            do i = 1, l
               a09(i,0,k) = a16(i,1,k)
            end do
         end do
         do k = 1, kstop
            do i = 1, l
               a07(i,0,k) = a18(i,1,k)
            end do
         end do
      end if

      if (righty.eq.-1) then
         do k = kstart, n
            do i = 1, l
               a16(i,m1,k) = a09(i,m,k)
            end do
         end do
         do k = 1, kstop
            do i = 1, l
               a18(i,m1,k) = a07(i,m,k)
            end do
         end do
      end if
c
c-----------------------------------------------------------------------
c                     SECTION 3
c-----------------------------------------------------------------------
c SECURE RECEPTION BEFORE STORING
```

```
c      allmsg = qbuf(2)
c      call mp_wait(allmsg,nb)
       if (lefty.ne.-1) then
         call mp_wait(misy16, nb)
         call mp_wait(misy18, nb)
         call mp_wait(miry07, nb)
         call mp_wait(miry09, nb)
       endif
       if (rghty.ne.-1) then
         call mp_wait(misy07, nb)
         call mp_wait(misy09, nb)
         call mp_wait(miry16, nb)
         call mp_wait(miry18, nb)
       endif
c
c-----------------------------------------------------------------------
c                     SECTION 4
c-----------------------------------------------------------------------
c ACTUAL RECEIVE COMPLETED: STORE CAN PROCEED
c x walls for xy-populations : before xy-populations y-transmissions
       if (lefty.ne.-1) then
          call my_unpack(a07(1,0,1),buffone(1,3,2),l,yst,n)
          call my_unpack(a09(1,0,1),buffone(1,4,2),l,yst,n)
c Note:stored into buffer. Move will take care of shifting into the
c physical domain
       endif
       if (rghty.ne.-1) then
          call my_unpack(a16(1,m1,1),buffone(1,1,2),l,yst,n)
          call my_unpack(a18(1,m1,1),buffone(1,2,2),l,yst,n)
       endif
css ----------------------------------------------------
c 2nd step: transmission of yz-populations along z
css ----------------------------------------------------
       if (leftz.ne.-1) then
          call mp_send(a09(0,0,1),yst*rlen,leftz,309,misz09)
          call mp_recv(a07(0,0,0),yst*rlen,leftz,307,mirz07)
          call mp_send(a16(0,0,1),yst*rlen,leftz,316,misz16)
          call mp_recv(a18(0,0,0),yst*rlen,leftz,318,mirz18)
       endif
       if (rghtz.ne.-1) then
          call mp_send(a07(0,0,n),yst*rlen,rghtz,307,misz07)
          call mp_recv(a09(0,0,n1),yst*rlen,rghtz,309,mirz09)
          call mp_send(a18(0,0,n),yst*rlen,rghtz,318,misz18)
          call mp_recv(a16(0,0,n1),yst*rlen,rghtz,316,mirz16)
       endif
```

```
c transmission of xy-populations along y: 1,10
      if (lefty.ne.-1) then
       call my_pack(a01(0,1,1),buffone(1,1,1),(l+2),yst,n)
       call mp_send(buffone(1,1,1),n*(l+2)*rlen,lefty,201,misy01)
       call mp_recv(buffone(1,3,2),n*(l+2)*rlen,lefty,203,miry03)

       call my_pack(a10(0,1,1),buffone(1,2,1),(l+2),yst,n)
       call mp_send(buffone(1,2,1),n*(l+2)*rlen,lefty,210,misy10)
       call mp_recv(buffone(1,4,2),n*(l+2)*rlen,lefty,212,miry12)
      endif

      if (rghty.ne.-1) then
       call my_pack(a03(0,m,1),buffone(1,3,1),(l+2),yst,n)
       call mp_send(buffone(1,3,1),n*(l+2)*rlen,rghty,203,misy03)
       call mp_recv(buffone(1,1,2),n*(l+2)*rlen,rghty,201,miry01)

       call my_pack(a12(0,m,1),buffone(1,4,1),(l+2),yst,n)
       call mp_send(buffone(1,4,1),n*(l+2)*rlen,rghty,212,misy12)
       call mp_recv(buffone(1,2,2),n*(l+2)*rlen,rghty,210,miry10)
      endif
c transmission of y-populations along y
      if (lefty.ne.-1) then
         call my_pack(a17(1,1,1),buffone(1,5,1),l,yst,n)
         call mp_send(buffone(1,5,1),n*l*rlen,lefty,217,misy17)
         call mp_recv(buffone(1,7,2),n*l*rlen,lefty,208,miry08)

         call my_pack(a23(1,1,1),buffone(1,6,1),l,yst,n)
         call mp_send(buffone(1,6,1),n*l*rlen,lefty,223,misy23)
         call mp_recv(buffone(1,8,2),n*l*rlen,lefty,221,miry21)
      endif
      if (rghty.ne.-1) then
         call my_pack(a08(1,m,1),buffone(1,7,1),l,yst,n)
         call mp_send(buffone(1,7,1),n*l*rlen,rghty,208,misy08)
         call mp_recv(buffone(1,5,2),n*l*rlen,rghty,217,miry17)

         call my_pack(a21(1,m,1),buffone(1,8,1),l,yst,n)
         call mp_send(buffone(1,8,1),n*l*rlen,rghty,221,misy21)
         call mp_recv(buffone(1,6,2),n*l*rlen,rghty,223,miry23)
      endif

c transmission of xz-populations along z

      if (leftz.ne.-1) then
         call mp_send(a02(0,1,1),(l+2)*m*rlen,leftz,302,misz02)
         call mp_recv(a04(0,1,0),(l+2)*m*rlen,leftz,304,mirz04)
         call mp_send(a11(0,1,1),(l+2)*m*rlen,leftz,311,misz11)
```

```
         call mp_recv(a13(0,1,0),(l+2)*m*rlen,leftz,313,mirz13)
      endif
      if (rghtz.ne.-1) then
         call mp_send(a04(0,1,n),(l+2)*m*rlen,rghtz,304,misz04)
         call mp_recv(a02(0,1,n1),(l+2)*m*rlen,rghtz,302,mirz02)
         call mp_send(a13(0,1,n),(l+2)*m*rlen,rghtz,313,misz13)
         call mp_recv(a11(0,1,n1),(l+2)*m*rlen,rghtz,311,mirz11)
      endif
c transmission of z-populations along z
      if (leftz.ne.-1) then
         call mp_send(a15(0,1,1),(l+2)*m*rlen,leftz,315,misz15)
         call mp_recv(a06(0,1,0),(l+2)*m*rlen,leftz,306,mirz06)
         call mp_send(a24(0,1,1),(l+2)*m*rlen,leftz,324,misz24)
         call mp_recv(a20(0,1,0),(l+2)*m*rlen,leftz,320,mirz20)
      endif

      if (rghtz.ne.-1) then
         call mp_send(a06(0,1,n),(l+2)*m*rlen,rghtz,306,misz06)
         call mp_recv(a15(0,1,n1),(l+2)*m*rlen,rghtz,315,mirz15)
         call mp_send(a20(0,1,n),(l+2)*m*rlen,rghtz,320,misz20)
         call mp_recv(a24(0,1,n1),(l+2)*m*rlen,rghtz,324,mirz24)
      endif

c x walls for x-populations : commute with y- and z-transmissions
      do k = 1, n
         do j = 1, m
            a14(L1,j,k) = a05(L,j,k)
            a22(L1,j,k) = a19(L,j,k)
            a05(0,j,k) = a14(1,j,k)
            a19(0,j,k) = a22(1,j,k)
         end do
      end do

c y walls for y-populations : commute with z-transmissions
      if (lefty.eq.-1) then
         do k = 1, n
            do i = 1, l
               a08(i,0,k) = a17(i,1,k)
               a21(i,0,k) = a23(i,1,k)
            end do
         end do
      end if

      if (rghty.eq.-1) then
         do k = 1, n
            do i = 1, l
```

```
               a17(i,m1,k) = a08(i,m,k)
               a23(i,m1,k) = a21(i,m,k)
            end do
         end do
      end if

c y walls for xy-populations : commute with z-transmissions
      if (lefty.eq.-1) then
         do k = 1, n
            a03(0,0,k) = a10(1,1,k)
            do i = 2, l-1
               a03(i,0,k) = a01(i,1,k)
               a12(i,0,k) = a10(i,1,k)
            end do
            a12(l1,0,k) = a01(l,1,k)
         end do
      end if

      if (rghty.eq.-1) then
         do k = 1, n
            a01(0,m1,k) = a12(1,m,k)
            do i = 2, l-1
               a01(i,m1,k) = a03(i,m,k)
               a10(i,m1,k) = a12(i,m,k)
            end do
            a10(l1,m1,k) = a03(l,m,k)
         end do
      end if

c z walls : commute with y-transmissions
      if (leftz.eq.-1) then
         do j = 1, m
            do i = 1, l
               a06(i,j,0)   = a24(i,j,1)
               a20(i,j,0)   = a15(i,j,1)
               a04(i-1,j,0) = a11(i,j,1)
               a13(i+1,j,0) = a02(i,j,1)
               a07(i,j-1,0) = a16(i,j,1)
               a18(i,j+1,0) = a09(i,j,1)
            end do
         end do
      end if

      if (rghtz.eq.-1) then
         do j = 1, m
            do i = 1, l
```

```
                a24(i,j,n1) = a06(i,j,n)
                a15(i,j,n1) = a20(i,j,n)
                a02(i-1,j,n1) = a13(i,j,n)
                a11(i+1,j,n1) = a04(i,j,n)
                a09(i,j-1,n1) = a18(i,j,n)
                a16(i,j+1,n1) = a07(i,j,n)
              end do
            end do
          end if
css ====================== BARRIER # 2 ======================
c          allmsg = qbuf(2)
c          call mp_wait(allmsg,nb)
          if (leftz.ne.-1) then
            call mp_wait(misz09, nb)
            call mp_wait(misz16, nb)
            call mp_wait(mirz07, nb)
            call mp_wait(mirz18, nb)
            call mp_wait(misz02, nb)
            call mp_wait(mirz04, nb)
            call mp_wait(misz11, nb)
            call mp_wait(mirz13, nb)
            call mp_wait(misz15, nb)
            call mp_wait(mirz06, nb)
            call mp_wait(misz24, nb)
            call mp_wait(mirz20, nb)
          endif
          if (rghtz.ne.-1) then
            call mp_wait(misz07, nb)
            call mp_wait(misz18, nb)
            call mp_wait(mirz09, nb)
            call mp_wait(mirz16, nb)
            call mp_wait(misz04, nb)
            call mp_wait(mirz02, nb)
            call mp_wait(misz13, nb)
            call mp_wait(mirz11, nb)
            call mp_wait(misz06, nb)
            call mp_wait(mirz15, nb)
            call mp_wait(misz20, nb)
            call mp_wait(mirz24, nb)
          endif
          if (lefty.ne.-1) then
            call mp_wait(misy01, nb)
            call mp_wait(miry03, nb)
            call mp_wait(misy10, nb)
            call mp_wait(miry12, nb)
            call mp_wait(misy17, nb)
```

```
            call mp_wait(miry08, nb)
            call mp_wait(misy23, nb)
            call mp_wait(miry21, nb)
          endif
          if (rghty.ne.-1) then
            call mp_wait(misy03, nb)
            call mp_wait(misy12, nb)
            call mp_wait(miry01, nb)
            call mp_wait(miry10, nb)
            call mp_wait(misy08, nb)
            call mp_wait(miry17, nb)
            call mp_wait(misy21, nb)
            call mp_wait(miry23, nb)
          endif
css ====================== BARRIER # 2 ======================

c transmission of xy-populations along y
      if (lefty.ne.-1) then
         call my_unpack(a03(0,0,1),buffone(1,3,2),l+2,yst,n)
         call my_unpack(a12(0,0,1),buffone(1,4,2),l+2,yst,n)
      endif

      if (rghty.ne.-1) then
         call my_unpack(a01(0,m1,1),buffone(1,1,2),l+2,yst,n)
         call my_unpack(a10(0,m1,1),buffone(1,2,2),l+2,yst,n)
      endif

c transmission of y-populations along y
      if (lefty.ne.-1) then
         call my_unpack(a08(1,0,1),buffone(1,7,2),l,yst,n)
         call my_unpack(a21(1,0,1),buffone(1,8,2),l,yst,n)
      endif

      if (rghty.ne.-1) then
         call my_unpack(a17(1,m1,1),buffone(1,5,2),l,yst,n)
         call my_unpack(a23(1,m1,1),buffone(1,6,2),l,yst,n)
      endif

      return
      end
c -------------------------------------------------------
         SUBROUTINE BCOND_ODD
c -------------------------------------------------------
css  receive/send sequence for odd-numbered processors
C
C BOUNDARY CONDITIONS
```

```
C LATERAL WALLS : FREE SLIP
C LOWER & UPPER WALLS : NO SLIP & THETA = 0
C
c message types = i*100 + population number, where i = 1,2,3 for movemen
c    along x,y,z
c message id names = mi + {s|r (send/receive)} + {x|y|z (axis)} + popula
c                    number
c ---------------------------------------------------------------------
         INCLUDE 'lbe.copy'
         real*4 buffone((l+2)*n,8,2)
c    (L+2)*n; size of the surface
c    8      ; # of messages per processor
c    1,2    ; message sent/received across the surface
c ---------------------------------------------------------------------
c 1st step: transmission of yz-populations along y: surface SY_MINUS
      nllen = n * l * rlen

      if (lefty.ne.-1) then
         call mp_recv(buffone(1,3,2),nllen,lefty,207,miry07)
         call my_pack(a16(1,1,1),buffone(1,1,1),l,yst,n)
         call mp_send(buffone(1,1,1),nllen,lefty,216,misy16)

         call mp_recv(buffone(1,4,2),nllen,lefty,209,miry09)
         call my_pack(a18(1,1,1),buffone(1,2,1),l,yst,n)
         call mp_send(buffone(1,2,1),nllen,lefty,218,misy18)
      endif

      if (rghty.ne.-1) then
         call mp_recv(buffone(1,1,2), nllen,rghty,216,miry16)
         call my_pack(a07(1,m,1),buffone(1,3,1),l,yst,n)
         call mp_send(buffone(1,3,1),nllen,rghty,207,misy07)

         call mp_recv(buffone(1,2,2),nllen,rghty,218,miry18)
         call my_pack(a09(1,m,1),buffone(1,4,1),l,yst,n)
         call mp_send(buffone(1,4,1),nllen,rghty,209,misy09)
      endif
      jstart = 1
      jstop = m
      if (lefty.eq.-1) jstart = 2
      if (rghty.eq.-1) jstop = m-1

      do k = 1, n
         do j = jstart, jstop
            a10(l1,j,k) = a01(l,j,k)
            a12(l1,j,k) = a03(l,j,k)
            a01(0,j,k)  = a10(1,j,k)
```

```
            a03(0,j,k)  = a12(1,j,k)
         end do
      end do
c for xz- and yz-populations boundary conditions on x and y walls
      kstart = 1
      kstop = n
      if (leftz.eq.-1) kstart = 2
      if (rghtz.eq.-1) kstop = n-1

c x walls for xz-populations : before xz-populations z-transmissions
      do k = kstart, n
         do j = 1, m
            a11(l1,j,k) = a02(l,j,k)
            a02(0,j,k) = a11(1,j,k)
         end do
      end do
      do k = 1, kstop
         do j = 1, m
            a13(l1,j,k) = a04(l,j,k)
            a04(0,j,k) = a13(1,j,k)
         end do
      end do
c y walls for yz-populations : before yz-populations z-transmissions
      if (lefty.eq.-1) then
         do k = kstart, n
            do i = 1, l
               a09(i,0,k) = a16(i,1,k)
            end do
         end do
         do k = 1, kstop
            do i = 1, l
               a07(i,0,k) = a18(i,1,k)
            end do
         end do
      end if

      if (rghty.eq.-1) then
         do k = kstart, n
            do i = 1, l
               a16(i,m1,k) = a09(i,m,k)
            end do
         end do
         do k = 1, kstop
            do i = 1, l
               a18(i,m1,k) = a07(i,m,k)
            end do
```

```
         end do
      end if

css ======================== BARRIER # 1 ================
c      allmsg = qbuf(2)
c      call mp_wait(allmsg,nb)
      if (lefty.ne.-1) then
        call mp_wait(misy16, nb)
        call mp_wait(misy18, nb)
        call mp_wait(miry07, nb)
        call mp_wait(miry09, nb)
      endif
      if (rghty.ne.-1) then
        call mp_wait(misy07, nb)
        call mp_wait(misy09, nb)
        call mp_wait(miry16, nb)
        call mp_wait(miry18, nb)
      endif
css ======================== BARRIER # 1 ================
c x walls for xy-populations : before xy-populations y-transmissions
      if (lefty.ne.-1) then
         call my_unpack(a07(1,0,1),buffone(1,3,2),l,yst,n)
         call my_unpack(a09(1,0,1),buffone(1,4,2),l,yst,n)
      endif
      if (rghty.ne.-1) then
         call my_unpack(a16(1,m1,1),buffone(1,1,2),l,yst,n)
         call my_unpack(a18(1,m1,1),buffone(1,2,2),l,yst,n)
      endif
css ----------------------------------------------------
c 2nd step: transmission of yz-populations along z
css ----------------------------------------------------
      if (leftz.ne.-1) then
         call mp_recv(a07(0,0,0),yst*rlen,leftz,307,mirz07)
         call mp_send(a09(0,0,1),yst*rlen,leftz,309,misz09)
         call mp_recv(a18(0,0,0),yst*rlen,leftz,318,mirz18)
         call mp_send(a16(0,0,1),yst*rlen,leftz,316,misz16)
      endif
      if (rghtz.ne.-1) then
         call mp_recv(a09(0,0,n1),yst*rlen,rghtz,309,mirz09)
         call mp_send(a07(0,0,n),yst*rlen,rghtz,307,misz07)
         call mp_recv(a16(0,0,n1),yst*rlen,rghtz,316,mirz16)
         call mp_send(a18(0,0,n),yst*rlen,rghtz,318,misz18)
      endif

c transmission of xy-populations along y: 1,10
      if (lefty.ne.-1) then
```

```
       call mp_recv(buffone(1,3,2),n*(l+2)*rlen,lefty,203,miry03)
       call my_pack(a01(0,1,1),buffone(1,1,1),(l+2),yst,n)
       call mp_send(buffone(1,1,1),n*(l+2)*rlen,lefty,201,misy01)

       call mp_recv(buffone(1,4,2),n*(l+2)*rlen,lefty,212,miry12)
       call my_pack(a10(0,1,1),buffone(1,2,1),(l+2),yst,n)
       call mp_send(buffone(1,2,1),n*(l+2)*rlen,lefty,210,misy10)
      endif

      if (rghty.ne.-1) then
       call mp_recv(buffone(1,1,2),n*(l+2)*rlen,rghty,201,miry01)
       call my_pack(a03(0,m,1),buffone(1,3,1),(l+2),yst,n)
       call mp_send(buffone(1,3,1),n*(l+2)*rlen,rghty,203,misy03)

       call mp_recv(buffone(1,2,2),n*(l+2)*rlen,rghty,210,miry10)
       call my_pack(a12(0,m,1),buffone(1,4,1),(l+2),yst,n)
       call mp_send(buffone(1,4,1),n*(l+2)*rlen,rghty,212,misy12)
      endif
c transmission of y-populations along y
      if (lefty.ne.-1) then
         call mp_recv(buffone(1,7,2),n*l*rlen,lefty,208,miry08)
         call my_pack(a17(1,1,1),buffone(1,5,1),l,yst,n)
         call mp_send(buffone(1,5,1),n*l*rlen,lefty,217,misy17)

         call mp_recv(buffone(1,8,2),n*l*rlen,lefty,221,miry21)
         call my_pack(a23(1,1,1),buffone(1,6,1),l,yst,n)
         call mp_send(buffone(1,6,1),n*l*rlen,lefty,223,misy23)
      endif
      if (rghty.ne.-1) then
         call mp_recv(buffone(1,5,2),n*l*rlen,rghty,217,miry17)
         call my_pack(a08(1,m,1),buffone(1,7,1),l,yst,n)
         call mp_send(buffone(1,7,1),n*l*rlen,rghty,208,misy08)

         call mp_recv(buffone(1,6,2),n*l*rlen,rghty,223,miry23)
         call my_pack(a21(1,m,1),buffone(1,8,1),l,yst,n)
         call mp_send(buffone(1,8,1),n*l*rlen,rghty,221,misy21)
      endif

c transmission of xz-populations along z

      if (leftz.ne.-1) then
         call mp_recv(a04(0,1,0),(l+2)*m*rlen,leftz,304,mirz04)
         call mp_send(a02(0,1,1),(l+2)*m*rlen,leftz,302,misz02)
         call mp_recv(a13(0,1,0),(l+2)*m*rlen,leftz,313,mirz13)
         call mp_send(a11(0,1,1),(l+2)*m*rlen,leftz,311,misz11)
      endif
```

```
      if (rghtz.ne.-1) then
         call mp_recv(a02(0,1,n1),(l+2)*m*rlen,rghtz,302,mirz02)
         call mp_send(a04(0,1,n),(l+2)*m*rlen,rghtz,304,misz04)
         call mp_recv(a11(0,1,n1),(l+2)*m*rlen,rghtz,311,mirz11)
         call mp_send(a13(0,1,n),(l+2)*m*rlen,rghtz,313,misz13)
      endif
c transmission of z-populations along z
      if (leftz.ne.-1) then
         call mp_recv(a06(0,1,0),(l+2)*m*rlen,leftz,306,mirz06)
         call mp_send(a15(0,1,1),(l+2)*m*rlen,leftz,315,misz15)
         call mp_recv(a20(0,1,0),(l+2)*m*rlen,leftz,320,mirz20)
         call mp_send(a24(0,1,1),(l+2)*m*rlen,leftz,324,misz24)
      endif

      if (rghtz.ne.-1) then
         call mp_recv(a15(0,1,n1),(l+2)*m*rlen,rghtz,315,mirz15)
         call mp_send(a06(0,1,n),(l+2)*m*rlen,rghtz,306,misz06)
         call mp_recv(a24(0,1,n1),(l+2)*m*rlen,rghtz,324,mirz24)
         call mp_send(a20(0,1,n),(l+2)*m*rlen,rghtz,320,misz20)
      endif

c x walls for x-populations : commute with y- and z-transmissions
      do k = 1, n
         do j = 1, m
            a14(L1,j,k) = a05(L,j,k)
            a22(L1,j,k) = a19(L,j,k)
            a05(0,j,k) = a14(1,j,k)
            a19(0,j,k) = a22(1,j,k)
         end do
      end do

c y walls for y-populations : commute with z-transmissions
      if (lefty.eq.-1) then
         do k = 1, n
            do i = 1, l
               a08(i,0,k) = a17(i,1,k)
               a21(i,0,k) = a23(i,1,k)
            end do
         end do
      end if

      if (rghty.eq.-1) then
         do k = 1, n
            do i = 1, l
               a17(i,m1,k) = a08(i,m,k)
               a23(i,m1,k) = a21(i,m,k)
```

```
            end do
         end do
      end if

c y walls for xy-populations : commute with z-transmissions
      if (lefty.eq.-1) then
         do k = 1, n
            a03(0,0,k) = a10(1,1,k)
            do i = 2, l-1
               a03(i,0,k) = a01(i,1,k)
               a12(i,0,k) = a10(i,1,k)
            end do
            a12(l1,0,k) = a01(l,1,k)
         end do
      end if

      if (rghty.eq.-1) then
         do k = 1, n
            a01(0,m1,k) = a12(1,m,k)
            do i = 2, l-1
               a01(i,m1,k) = a03(i,m,k)
               a10(i,m1,k) = a12(i,m,k)
            end do
            a10(l1,m1,k) = a03(l,m,k)
         end do
      end if

c z walls : commute with y-transmissions
      if (leftz.eq.-1) then
         do j = 1, m
            do i = 1, l
               a06(i,j,0)   = a24(i,j,1)
               a20(i,j,0)   = a15(i,j,1)
               a04(i-1,j,0) = a11(i,j,1)
               a13(i+1,j,0) = a02(i,j,1)
               a07(i,j-1,0) = a16(i,j,1)
               a18(i,j+1,0) = a09(i,j,1)
            end do
         end do
      end if

      if (rghtz.eq.-1) then
         do j = 1, m
            do i = 1, l
               a24(i,j,n1) = a06(i,j,n)
               a15(i,j,n1) = a20(i,j,n)
```

```
               a02(i-1,j,n1) = a13(i,j,n)
               a11(i+1,j,n1) = a04(i,j,n)
               a09(i,j-1,n1) = a18(i,j,n)
               a16(i,j+1,n1) = a07(i,j,n)
            end do
         end do
      end if
css ====================== BARRIER # 2 ======================
c         allmsg = qbuf(2)
c         call mp_wait(allmsg,nb)
         if (leftz.ne.-1) then
           call mp_wait(misz09, nb)
           call mp_wait(misz16, nb)
           call mp_wait(mirz07, nb)
           call mp_wait(mirz18, nb)
           call mp_wait(misz02, nb)
           call mp_wait(mirz04, nb)
           call mp_wait(misz11, nb)
           call mp_wait(mirz13, nb)
           call mp_wait(misz15, nb)
           call mp_wait(mirz06, nb)
           call mp_wait(misz24, nb)
           call mp_wait(mirz20, nb)
         endif
         if (rghtz.ne.-1) then
           call mp_wait(misz07, nb)
           call mp_wait(misz18, nb)
           call mp_wait(mirz09, nb)
           call mp_wait(mirz16, nb)
           call mp_wait(misz04, nb)
           call mp_wait(mirz02, nb)
           call mp_wait(misz13, nb)
           call mp_wait(mirz11, nb)
           call mp_wait(misz06, nb)
           call mp_wait(mirz15, nb)
           call mp_wait(misz20, nb)
           call mp_wait(mirz24, nb)
         endif
         if (lefty.ne.-1) then
           call mp_wait(misy01, nb)
           call mp_wait(miry03, nb)
           call mp_wait(misy10, nb)
           call mp_wait(miry12, nb)
           call mp_wait(misy17, nb)
           call mp_wait(miry08, nb)
           call mp_wait(misy23, nb)
```

```
            call mp_wait(miry21, nb)
          endif
          if (rghty.ne.-1) then
            call mp_wait(misy03, nb)
            call mp_wait(misy12, nb)
            call mp_wait(miry01, nb)
            call mp_wait(miry10, nb)
            call mp_wait(misy08, nb)
            call mp_wait(miry17, nb)
            call mp_wait(misy21, nb)
            call mp_wait(miry23, nb)
          endif
css ====================== BARRIER # 2 ======================

c transmission of xy-populations along y
      if (lefty.ne.-1) then
         call my_unpack(a03(0,0,1),buffone(1,3,2),l+2,yst,n)
         call my_unpack(a12(0,0,1),buffone(1,4,2),l+2,yst,n)
      endif

      if (rghty.ne.-1) then
         call my_unpack(a01(0,m1,1),buffone(1,1,2),l+2,yst,n)
         call my_unpack(a10(0,m1,1),buffone(1,2,2),l+2,yst,n)
      endif

c transmission of y-populations along y
      if (lefty.ne.-1) then
         call my_unpack(a08(1,0,1),buffone(1,7,2),l,yst,n)
         call my_unpack(a21(1,0,1),buffone(1,8,2),l,yst,n)
      endif

      if (rghty.ne.-1) then
         call my_unpack(a17(1,m1,1),buffone(1,5,2),l,yst,n)
         call my_unpack(a23(1,m1,1),buffone(1,6,2),l,yst,n)
      endif

      return
      end
c ----------------------------------------
      subroutine my_pack(a, b, l, stride, n)
c ----------------------------------------
      real a(*), b(*)
      integer stride
      do i = 0, n-1
         do k = 1, l
            b(k+l*i) = a(k+stride*i)
```

```
         enddo
      enddo
      return
      end
c ----------------------------------------
      subroutine my_unpack(a, b, l, stride, n)
c ----------------------------------------
      real a(*), b(*)
      integer stride
      do i = 0, n-1
         do k = 1, l
            a(k+stride*i) = b(k+l*i)
         enddo
      enddo
      return
      end
```

6.2.3 LBE_POST

This is the post-processing program designed to recompose data into a single file for post-processing purposes. All that it does is read off the sub-portions of the population arrays, A01...A24, out of the corresponding files and recompose the partial results into a single series of global arrays, taking into account the proper offsets **jstart** and **kstart**. As a test, the arrays, A01...A24, are compared with B01...B24, where B is the solution from the serial code.

The Post-processing Code

postpop.copy

```
C
C INCLUDE FILE WITH LATTICE DIMENSIONS FOR LBE 3D CONVECTION
C
            include 'lbehw.copy'

            PARAMETER ( NPY = 1, NPZ = 1 )

            PARAMETER (L = LGL,  M = MGL , N = NGL )
```

postpop.f

```
          PROGRAM POSTPROC
C
C STARTING CONDITION FOR 3D LBE CONVECTION SIMULATION
C
C VELOCITY FIELD WITH ZERO DIVERGENCE
C
```

```
C     40 IS INPUT FILE
C

          INCLUDE 'postpop.copy'
          integer cx, cy, cz

          EQUIVALENCE (a01,avec)
          EQUIVALENCE (b01,bvec)

          write(6,*) 'npy, npz',npy, npz

c first read parallel files
      do ip = 1, npz*npy

         read(40+ip) itime, lefty, rghty, leftz, rghtz

         jstart = mod(ip-1,npy)  * m
         kstart = ((ip-1)/npy) * n

         do k = kstart+1, kstart+n
            do j = jstart+1, jstart+m
               read(40+ip)
     *    (A01(I,j,k),A02(I,j,k),A03(I,j,k),A04(I,j,k),
     *     A05(I,j,k),A06(I,j,k),A07(I,j,k),A08(I,j,k),
     *     A09(I,j,k),A10(I,j,k),A11(I,j,k),A12(I,j,k),
     *     A13(I,j,k),A14(I,j,k),A15(I,j,k),A16(I,j,k),
     *     A17(I,j,k),A18(I,j,k),A19(I,j,k),A20(I,j,k),
     *     A21(I,j,k),A22(I,j,k),A23(I,j,k),A24(I,j,k),
     *     I=1,L)
            end do
         end do
      end do

          REWIND(40)

          read(40) itime

          DO K=1,NGL
          DO J=1,MGL

             read(40)
     *       (B01(I,J,K),B02(I,J,K),B03(I,J,K),B04(I,J,K),
     *        B05(I,J,K),B06(I,J,K),B07(I,J,K),B08(I,J,K),
     *        B09(I,J,K),B10(I,J,K),B11(I,J,K),B12(I,J,K),
     *        B13(I,J,K),B14(I,J,K),B15(I,J,K),B16(I,J,K),
     *        B17(I,J,K),B18(I,J,K),B19(I,J,K),B20(I,J,K),
```

```
     *        B21(I,J,K),B22(I,J,K),B23(I,J,K),B24(I,J,K),
     *        I=1,L)

          END DO
          END DO

200      write(6,*) 'which matrix? (0 exit)'
       read(5,*) nmat
       if (nmat.eq.0) stop

       sum = 0.
       max = -9999.
       max1 = -9999.
       max2 = -9999.
           DO 3 K= (nmat-1)*lgl*mgl*ngl+1, nmat*lgl*mgl*ngl
           c = abs ( avec(k) - bvec(k) )
           c1 = abs ( avec(k) )
           c2 = abs ( bvec(k) )
           if ( c .gt. max ) then
              max = c
              kmax = k
           endif
c           if ( c .gt. 1.e-6 ) then
c             cz = ( k - (nmat-1)*lgl*mgl*ngl + 1 ) / ngl
c             write(6,*) k,avec(k), bvec(k)
c           endif
           if ( c1.gt. max1) max1= c1
           if ( c2.gt. max2) max2= c2
3           sum = sum + c
       sum = sum / (1.*ngl*mgl*lgl)

       write(6,*) 'sum',sum
       write(6,*) 'max',max
       write(6,*) 'max par',max1
       write(6,*) 'max ser',max2
c       write(6,*) 'kmax',kmax

       goto 200

       stop
       END
```

Chapter 7

Bibliography

General

1. V. Kumar, A. Grama, A. Gupta, G. Karypis, *Introduction to Parallel Computing*, Benjamin Cummings, 1994

2. I. Foster, *Designing and Building Parallel Programs*, Addison-Wesley, 1995

3. G. Almasi, A. Gottlieb, *Higly Parallel Computing*, 2nd edition, Benjamin Cummings, 1994

4. K. Hwang, *Advanced Computer Architecture; Parallelism, Scalability, Programmability*, Mc Graw Hill, NY, 1993

SIMD Programming

5. B. Boghosian, Computational Physics on the Connection Machine, *Computers in Physics*, Jan / Feb 1990

Message-Passing

6. A. Geist, A. Beguelin, J. Dongarra, R. Manchek, V. Sunderam, *PVM:Parallel Virtual Machine*, MIT Press, 1994

7. IBM AIX Parallel Environment, User's Guide and Reference, sh23-0019, 1994 available thru IBM branch offices.

LBE Application

8. G. Punzo, F. Massaioli, S. Succi, High resolution Lattice Boltzmann Computing on the IBM SP1 scalable parallel computer, *Computers in Physics*, 5, 1-7, 1994