CHAPTER 2

BASIC NOTIONS OF PARALLEL COMPUTING

2.1 Introduction

In this chapter, we shall discuss various architectural classification and memory organizations with the design of parallel algorithms (methodical design). Here, we will show how a program specification is translated into an algorithm that displays concurrency, scalability and locality. Our goal is to suggest a framework within which parallel algorithm design can be explored. We are able to explain the simple parallel algorithm in a methodical fashion. We also emphasis on network topology, interconnection networks encompassing the platform used in this thesis work. However, we present the various parallel programming models.

2.2 Architectural Classification

Between early 1960's and the mid-1990's, scientists and engineers explored a wide variety of parallel computer architectures. There are four criteria for standard parallel architecture classification (Blaise (1994)), this include: memory organization, control, interconnection topology, and granularity. Here, we will give a brief overview of the major classes of parallel and distributed architectures. A comprehensive and up-date resource on Parallel Computers architecture is Culler et al., (1999), and the current standard reference on clusters is Buyya, (1999).

2.2:1 Control (Flynn's Classification)

Flynn's taxonomy (Flynn, (1972)) is the best known classification scheme for parallel computers according to Quinn,(2001), Blaise (1994) and Barry (2003). A computer's category depends upon the parallelism it exhibits in its instruction stream and its data stream. The scheme is simple but famous and was introduced in 1966 by Michael Flynn and distinguishes between four classes of computers, as in Fig. 2.1: single instruction stream- single data stream (SISD), single instruction stream-multiple data stream (SIMD), multiple instruction stream-single data stream (MISD), and multiple instruction stream-multiple data stream (MIMD) computers. Parallel architectures can be classified into SIMD and MIMD, yet we see the single program multiple data (SPMD) and the multiple program multiple data (MPMD) which are sub-classes of MIMD. In SPMD paradigm, all processors carry out the same program on different data, while in MPMD paradigm different processors carry out different programs.

SISD	**SIMD**
MISD	**MIMD**

Fig. 2.1 (Flynn's classification)

The first two letters say whether multiple instructions can be carried out at the same time, and the last two letters say whether multiple data can be processed at the same time. The SISD computers are the Von-Neumann computers, named after a Hungarian mathematician John Von Neumann. The computer uses the stored-program concept, the central processing unit (CPU) executes a stored program that specifies a sequence of read and write operations on the memory. The SIMD machines are composed of a

powerful control processor and are array of related simple processing elements (PEs), shown in Fig. 2.2. Memory is distributed. SIMD computers maintain only a single copy of the program and a single program counter; both are stored in the control processor. Examples of SIMD architectures are the Connection Machines CM-2 of Thinking Machines and the Maspar MP-2 of Maspar Computer Corporation. The CM-2 contains up to about 65000 very simple processing elements (PEs). The current trend is in the direction of using commodity processors and of supporting SPMD instead of SIMD. Pure SIMD designs seem to be disappearing as general-purpose platform, are being replaced by MIMD machines with explicit support for SPMD mechanisms such as fast broadcast and synchronization.

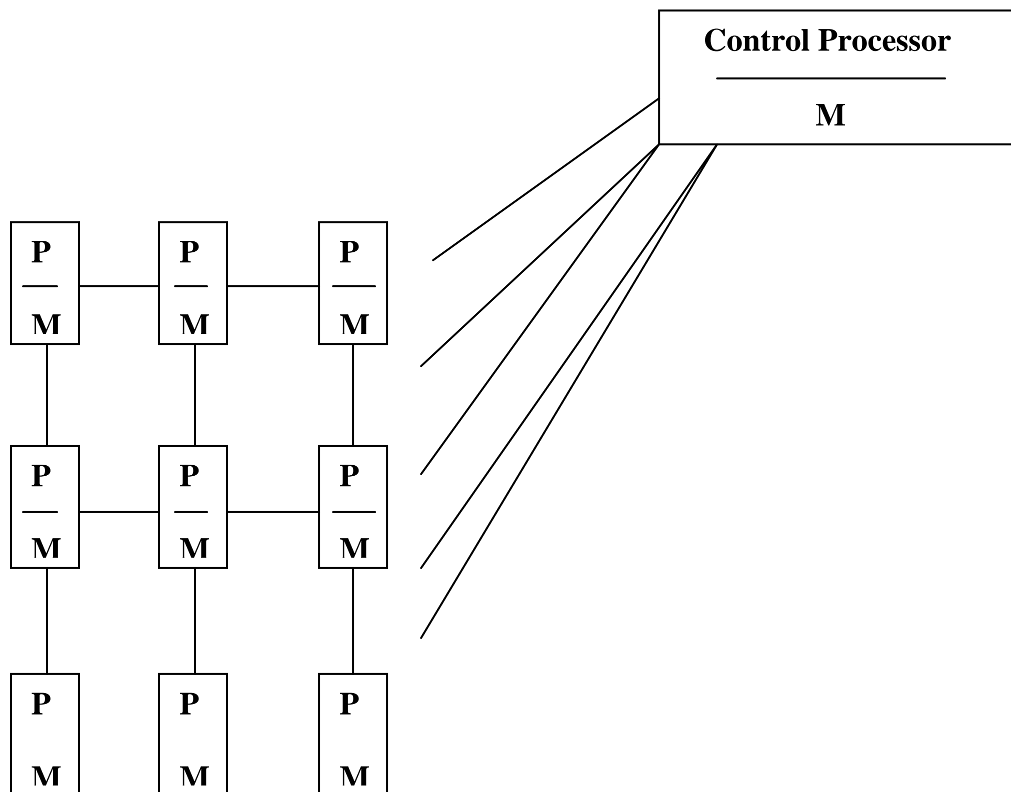

Fig. 2.2: A typical SIMD Architecture. P for processor, M for memory module.

The MISD computer does not exist unless one specifically classifies pipelined architectures in this group, or possibly some fault tolerant systems. Within the MIMD

classification, which are concerned with, each processor will have its own program to execute. This could be described as MPMD structure, as illustrated in Fig. 2.3.

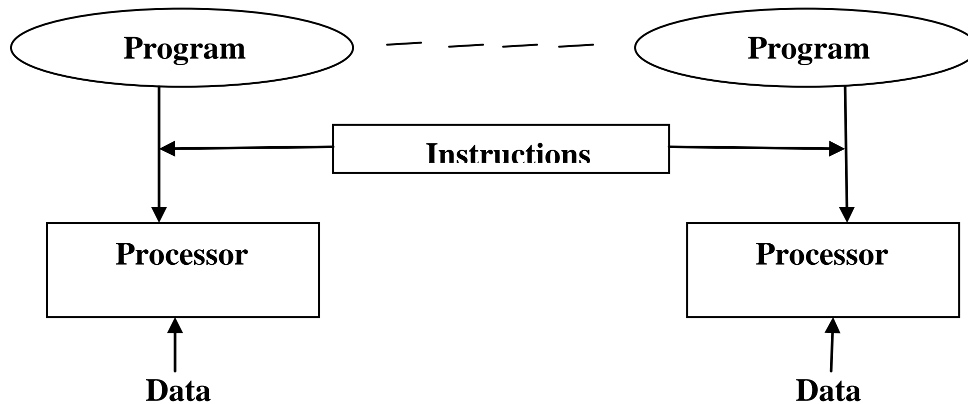

Fig. 2.3: MPMD Structure

2.2:2 Classifications by El-Rewini and Lewis

This scheme is proposed by Lewis and Rewini 1998; it refers to parallel and distributed computing, not merely to parallel computing like the previous scheme. The scheme uses two dimensions for classification: granularity and degree of coupling in the system architecture. The first dimension distinguishes between fine, medium and large grained problems, and the second dimension distinguishes between tightly coupled, coupled, loosely coupled, and shared-database system architectures.

2.3 Memory Organizations

Barry & Michael (2003), Braunl, (1993), Succi et al., (1998), Ted & Rewini, (1992), emphasize three types of memory organizations i.e. Shared Memory, Distributed Memory and Distributed Shared Memory. The shared memory varies widely, but generally has in common the ability for all processors to access all memory as global

address space. 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. 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. 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.

2.3:1 Shared Memory Machines

Snir et al., (1998) and Li, (1986), wrote on hierarchical memory and shared virtual memory on loosely coupled multiprocessor. Consider Fig. 2.4 as the shared memory architecture where multiple processors can operate independently but share the same memory resources and changes in a memory location effected by one processor are visible to all other processors. Shared Memory machines can be divided into two main classes based upon memory access times: the Uniform Memory Access (UMA) and the Non-Uniform Memory Access (NUMA). The UMA mostly represented by the Symmetric Multiprocessor (SMP) machines, has identical processors and equal accesses and access time to memory and are sometimes called Cache Coherent UMA (CC-UMA). Cache Coherent means if one processor updates a location in Shared Memory, all the other processors know about the update, and it is accomplished at the hardware level. The NUMA often made physically linking two or more SMPs. Not all processors have equal access time to all memories and memory access link is slower. If Cache

Coherency is maintained, then it may also be called CC-NUMA- Cache Coherent NUMA.

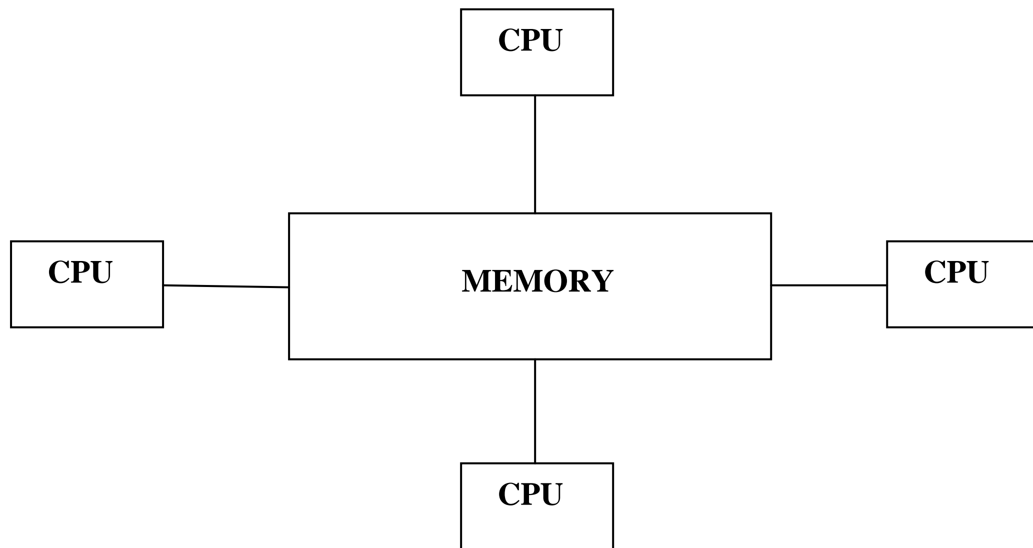

Fig. 2.4: Shared Memory Architecture.

2.3:2 Distributed Memory (DM) Machines

The Distributed Memory systems vary widely and share common characteristics. According to Blaise, (1994), they require a communication network to connect inter-processor memory. Hence, processors have their own local memory addresses in each processor. One processor does not map to another processor, so there is no concept of global address space across all processors. Each processor operates independently and the concept of Cache Coherency does not apply. When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data are communicated. Synchronization between tasks is likewise the programmer's responsibility. The network used for data transfer varies widely, though it can be as single as Ethernet. The structure is given in Figure 2.5; this is a collection of

independent computers that are each composed of processors, memory modules, input/output (IO) facilities, etc. This case corresponds to clusters.

There are two basic parameters which determine the performance of a DM machine: Bandwidth, the asymptotic flow rate of information and Latency, which is the overhead paid to establish a connection between two processors, i.e. the cost to send a zero length of message.

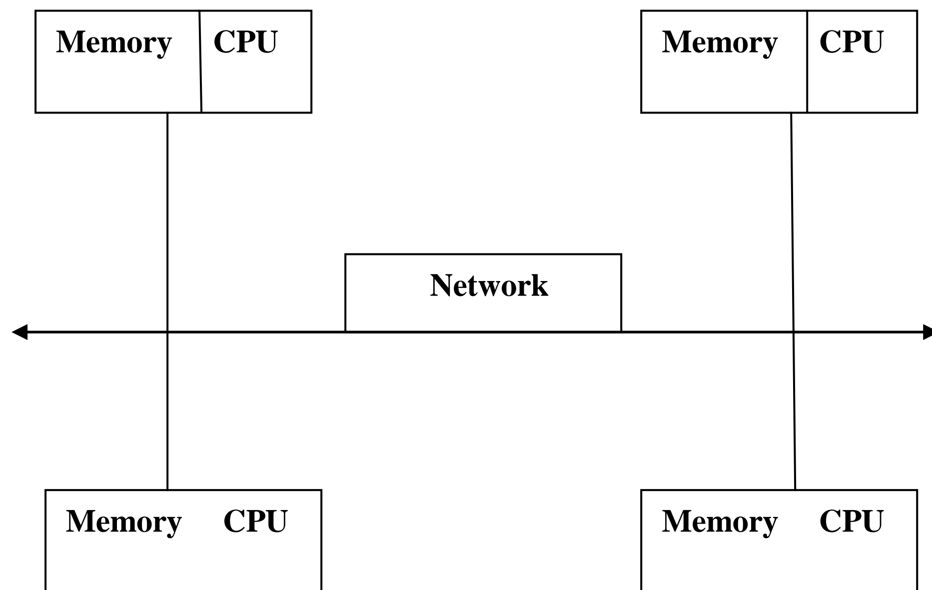

Fig. 2.5: Distributed Memory Parallel Computers

2.3:3 Distributed Shared Memory

The concept of a Distributed Shared Memory system in which each processor has access to the whole memory using a single memory address space was given by Barry & Michael, (2003). For a process to access a location not in its local memory, message-passing must occur to pass data from the processor to the location or from the location to the processor in some automatic way that hides the fact that the memory is distributed. This ideal is called Shared Virtual Memory, which gives the illusion of

shared memory as in Fig. 2.6. Distributed architectures, are distinguished as clusters, loosely coupled distributed systems, and grids. If course, remote accesses will occur a greater delay, and usually a significant greater delay, than for local accesses. One of the first to develop Shared Virtual Memory was Li, (1986).

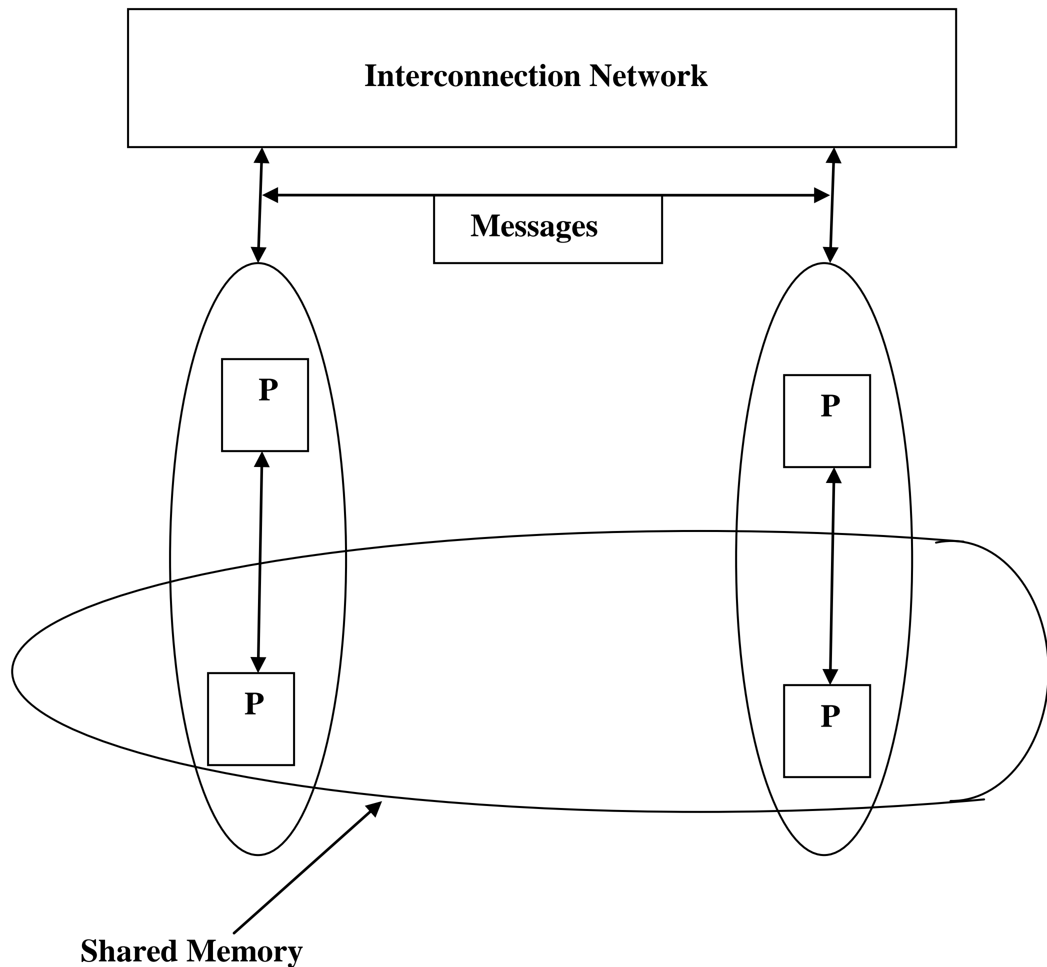

Fig. 2.6: Shared Memory Multiprocessor Implementation.

2.4 Network Topology

Information on inter-connected networks can be found in Duato et al., (1997). Barry & Michael (2003) and Beverly et al., (2005) gave a comprehensive view of network topology. The description of how the processors are connected to one another is often called the topology of the computers or precisely the inter-connect network. The pattern

of communication is called an application topology or virtual topology, by Groop et al., (1999).

2.4.1 Static Network

As seen in Fig. 2.6, a message-passing multicomputer requires some form of interconnection network to carry the messages. A common form of interconnection network is the static network. Static interconnection networks are those that have direct fixed physical links between computers (nodes), as shown in Fig. 2.7. Each node contains a processor, memory and a communication interface with links to other nodes. Static networks can also be used in a shared memory system. A link between two nodes could be bidirectional or there may be two separate uni-bidirectional links, one for each direction. We refer to connection between two nodes as a link. The key issues in network designs are the network bandwidth, network latency, and the cost as indicated by the number of links in the network. The bandwidth is the number of bits that can be transmitted in unit time, given as bits/sec. Network latency is the time to make a message transfer through the network. The communication latency is the total time to send the message including the software overhead and interface delay.

In exhaustive or completely connected networks, each node has a link to every other node. Hence, a node could be exhaustively interconnected with $n-1$ links from each node to other $n-1$ nodes. There are $n(n-1)/2$ links in all. Static networks with restricted interconnection includes: the line/ring, the mesh, the hypercube and the tree network. The ideal situation in passing message from a source node to a destination node occurs when there is a direct link between the source node and the destination node. There are two basic ways that messages can be transferred from source to a destination; Circuit Switching (CS) and Packet Switching (PS). CS involves

establishing the path and maintaining all the links in the path for the message to pass, uninterrupted, from the source to the destination. All the links are reserved for the transfer until the message transfer is complete, e.g. a simple telephone connection. It has been used in some multi-computers (e.g., the Intel IPSC-2 hypercube system), but it suffers from forcing all the links in the path to be reserved for the complete transfer. None of the links can be used for other messages until the transfer is completed. Interconnected networks have routine algorithms to find a path between nodes. In general, routine algorithms, unless properly designed can be prone to livelock and deadlock. Livelock describes the situation in which a packet keeps going around the network without ever finding its destination. Deadlock occurs when packets cannot be forwarded to the next node because they are blocked by other packets waiting to be forwarded and these packets are blocked in similar way such that none of the packets can move; see (Dally & Seitz (1987)).

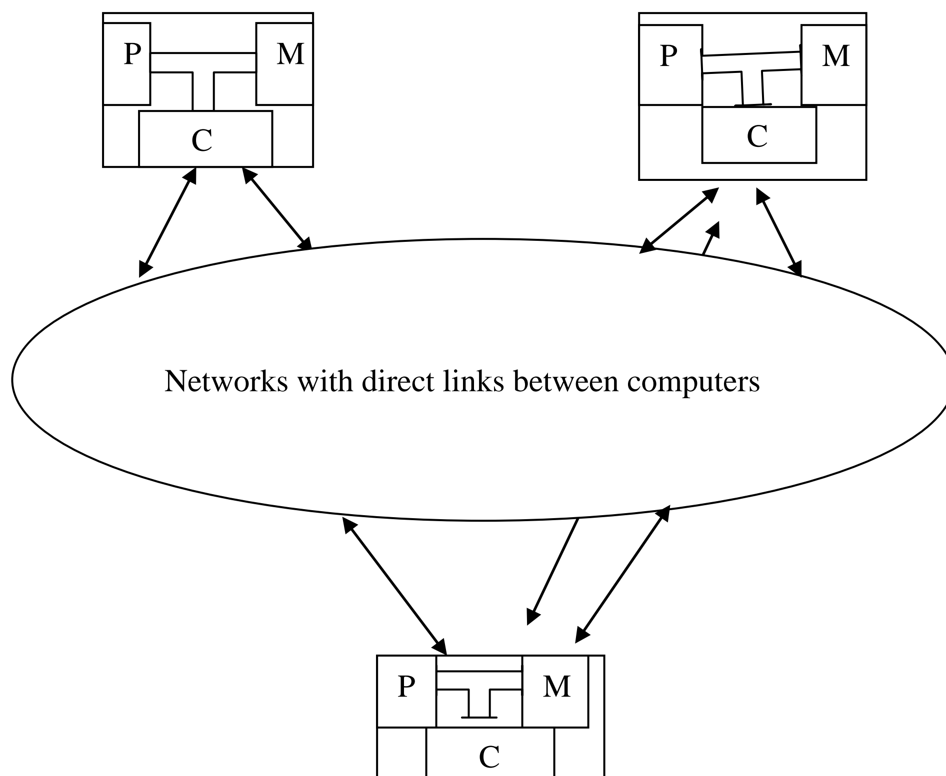

Figure 2.7: Static link multi-processor

The problem of deadlock appears in communication networks using store-and-forward routing and has been studied extensively by (Sahni (1996)). The mathematics conditions and solutions for deadlock free routing in any network can be found in (Dally & Seitz (1987)). A general solution to deadlock is to provide virtual channels, each with separate buffers, for classes of messages.

2.4:2 TCP/IP and UDP/IP

Skillicorn et al., (1998), emphasize predictable communication on unpredictable networks implementing on TCP/IP and UDP/IP. The major current protocol family for Wide Area Networks (WAN) is TCP/IP. It is very popular since it is used in the internet. TCP/IP is actually two main protocols. The Internet Protocol (IP) runs on relatively low layer of the protocol hierarchy. It is a connectionless protocol and provides a so-called best effort service. Best-effort means that IP tries to transfer all packets, but it does not give any guarantees. Packets loss is frequently caused by congestion, since IP allows an intermediate node to throw away packets when it is overloaded. The Transmission Control Protocol (TCP) runs on the layer above IP and expands the IP functionality into a reliable and connection-oriented service. TCP is concerned with the transfer of whole messages, which can be of variable sizes. The messages are first split up into packets at the source, then handed over to IP for transfer, and finally reassembled at the destination. TCP checks if all packets have arrived and if their contents are correct according to some checksum. In case of errors, retransmission is required. TCP also accomplishes flow control; it adapts the speed of injecting packets into the network to the receiver speed. The reliable service of TCP is desirable; it comes at the price of high overhead that makes TCP slow. In several applications speed is more important than reliability.

2.4:3 Interconnection Networks

At the level of the physical hardware interconnect, multicores have initially employed buses or crossbar switches between the cores and cache banks, but such solutions are not scalable to thousands of cores. We need on-chip topologies that scale close to linearly with system size to prevent the complexity of the interconnect from dominating cost of manycore systems. Scalable on-chip communication networks use ideas from larger scale packet-switched networks (Dally & Towles, (2001)). Already chip implementations such as the IBM cell employ multiple ring networks to interconnect the nine processors on the chip and use software-managed memory to communicate between the cores rather than conventional cache-coherency protocols.

There have been researches into statistical traffic models to help refine the design of networks-on-chip (Soterion et al., (2006)), based on studies of the communication requirements of existing massively concurrent scientific applications.

2.5 Models and Paradigms

Models and Paradigms cover almost the whole area of general-purpose Parallel and Distributed Computing. Fig. 2.8 gives a simplified view of the area. This shows how Parallel and Distributed Computing mediates between technology (at the bottom), and application (at the top). Moreover, both technologies and the Parallel and Distributed Computing are very complex. As described by Claudia (2001), we have the model abstraction as shown below.

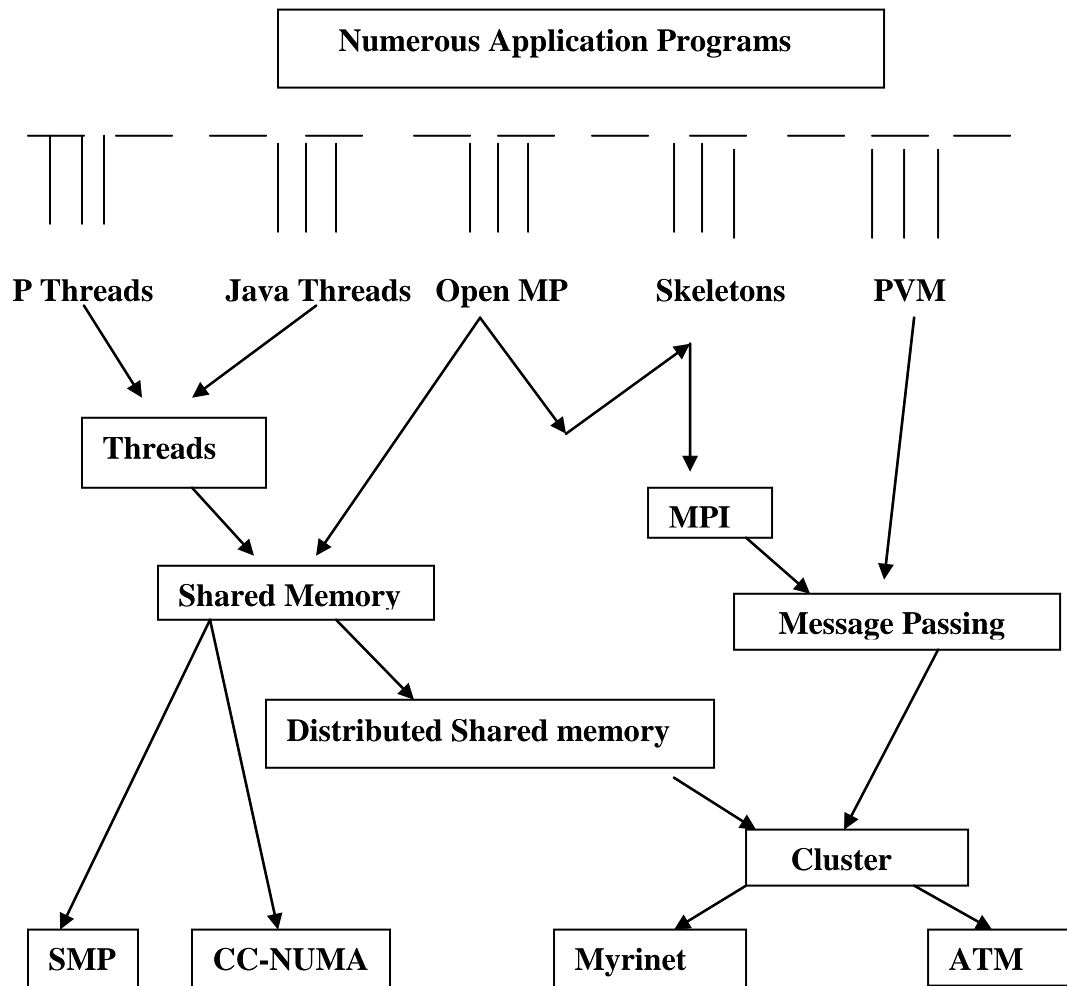

Fig. 2.8: A typical simplified view of the Parallel & Distributed computing field.

To manage the complexity, a number of conceptual levels have been introduced; we called such an abstract view of the figure above a model. A Model is a simplified or idealized description or conception of a particular system. The models more towards the bottom or top of the conceptual hierarchy are called low-level or high-level, respectively. We concentrate on models at about the middle levels. A Paradigm is a kind of model that describes typical structures or patterns of application programs. Abstractions from below and abstractions from above are combined in the design of a model. Thus the distinction between models and paradigms is rather vague. Programming Models are programming languages together with libraries, and run-time system. The term Application Programming Interface (API) is used instead of

programming model. Models give the programmer or algorithm designer a simplified but hopefully sufficiently correct view of a system whose complexity he or she would otherwise not be able to manage. A model should make the complexity of program design manageable for the programmer, expressing algorithmic intentions. Other considerations include support for maintenance, reusability, and correction.

Lewis & Rewini, (1992), gave two general methods of coordinating parallel computers: asynchronous and synchronous. The first approach is to build synchronous or lockstep coordination into the hardware by forcing all operations to be performed at the same time and in a manner that removes the dependency of one task on another. The second form called asynchronous because there is no lockstep coordination, relies on coordination mechanisms called locks to coordinate processors. Here processors operate freely on tasks, without regard for global synchronization.

2.5:1 Clusters

A Cluster is a type of Parallel or Distributed system that consists of a collection of interconnected whole computers, and is used as a single, unified computing resource. The term "whole computer" stands for a note that comprises one or several processors, a cache, a main memory, a disk, an input/output (IO) interface, and an own operating system. It can be PC, a workstation, or even an SMP. The interconnection network uses either standard technologies such as Ethernet, or high-performance technologies such as Myrinet. The term "single, unified computing resource" stands for what we have called a Single System Image (SSI). Clusters are classified into high-availability and high-performance clusters. We restrict ourselves to HPC which can be run in execute parallel programs. In this thesis work, the Cluster design for the project is called Armadillo.

Clusters vary widely. First, dedicated clusters are solely designed for use as parallel machines, the nodes are located in a small physical area such as a single room and do not contains peripherals such as keyboards and monitors.

2.5:2 Descriptions of Armadillo and Geranium Cadcam Clusters.

The implementation was done on a distributed computing environment of AGDS consisting of Armadillo Generation 1 (AG-1) and Armadillo Generation 2 (AG-2). AG-1 has 8 PCs Intel Pentium IV with 1.66GHZ speed and 0.74GB RAM distributed memory and AG-2 consists of 8 Intel Pentium Dual-Core at 1.73GHZ and 0.99GB RAM. Communication is through a fast Ethernet of 100Mbps connected through fast Ethernet running Linux, located at the University of Malaya, Institute of Mathematical Sciences laboratory. The AG-1 and AG-2 cluster performance have high memory bandwidth with a message passing supported by PVM which is public-domain software from Oak Ridge National Laboratory (Geist et al., (1994)). The AGDS consists of 16 processing units. The Geranium Cadcam Cluster operating from the frontend node in the Department of Design and Manufacturing Faculty of Engineering, University of Malaya, consists of 16 DELL Power Edge 2650 Xeon 2.8GHZ, 32GB DDR ECC SDRAM. Our platform is non localized and different from the platform used in (Foster et al., 1998) for MPI implementation and in (Dou & Phan-Thien (1997) & Sahimi et al., (2001)) for PVM implementation. Programs written in Fortran, C, or C++ are provided access to PVM through calling PVM library routines for functions such as process initiation, message transmission and reception. The program written in C provides access to MPI through calling MPI library routines.

2.6 Parallel Designs

Parallel algorithm designs by Foster, (1996), Barry & Michael, (2003) describe models that facilitate the development of efficient parallel programs, particularly the running on distributed memory parallel computers.

2.6:1 Foster's Design Methodology

Foster, (1996), has proposed a four-step process for designing parallel algorithms. It encourages the development of scalable parallel algorithms by delaying machine-dependent considerations to the later steps. The four design steps are called partitioning, communication, agglomeration and mapping (Fig.2.9).

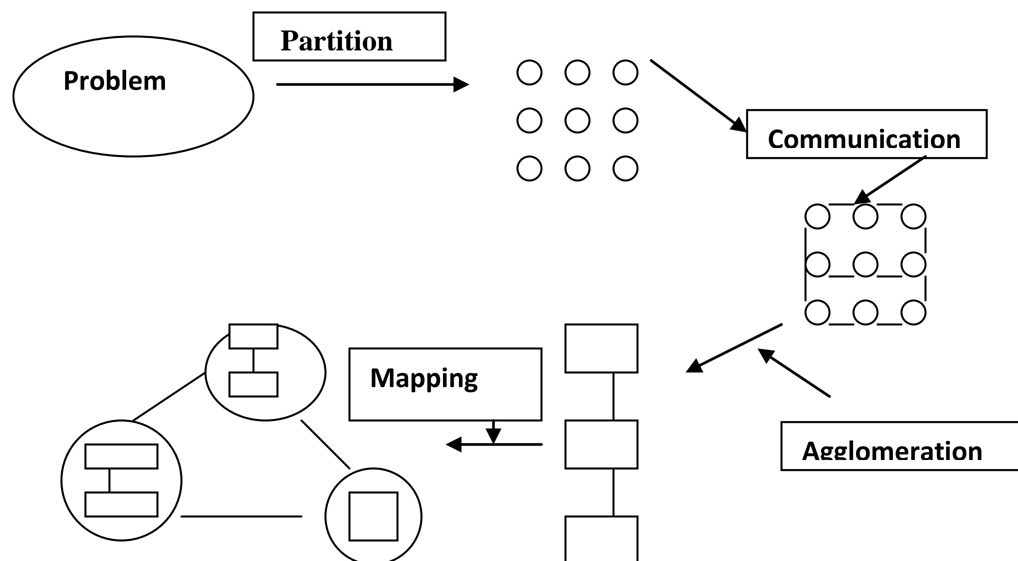

Fig. 2.9: Foster's Parallel Algorithm Design Methodology.

Parallel algorithms have two kinds of communication patterns: local and global. A local communication is when a task needs values from a small number of other tasks in order to perform a computation, we created channel from the tasks supplying the data to the task consuming the data. A global communication exists when a significant number of the primitive tasks must contribute data in order to perform computation. We call communication among tasks part of the overhead of a parallel algorithm because it is something the sequential algorithm does not need to do. Agglomeration is the process of grouping tasks into larger tasks in order to improve performance or simplifying programming. One of the goals of agglomeration is to lower communication overhead. We call this increasing the locality of the parallel algorithm. Another way is to combine groups of sending and receiving tasks, reducing the number of messages being sent. Sending fewer, longer messages takes less time than sending more, shorter messages with the same total length because there is a message startup cost called the message latency which is incurred every time a message is sent, and this time is independent of the length of the message. Second goal is to maintain scalability; third goal is to reduce software engineering costs. However, mapping is the process of assigning tasks to processors.

2.6:2 Other Design Strategies (Barry, (2003) and Succi (1998))

Barry & Michael (2003) and Succi et al., (1998), also made mention of the following parallel algorithm design strategies:

1. Partitioning Strategies: partitioning simply divides the problem into parts. Most partitioning formulations, however, require the results of the parts to be combined to obtain the desired result. Partitioning can be applied to the program data i.e. dividing the data and operating upon the divided data concurrently. This is called data

partitioning or Domain Decomposition (DD). Partitioning can also be applied to the functions of a program, that is, dividing the program into independent functions and executing the functions concurrently. This is called function decomposition. Data partitioning is the main strategy for parallel programming. Consider Fig. 2.10. Suppose a sequence of numbers, $x_o, ... x_{n-1}$, is to be added. We might divide the sequence into m parts of n/m numbers each, $(x_o ... x_{n/m-1})$, . . ., $(x_{n/m} ... x_{(2n/m)-1})$, . . ., $(x_{(m-1)n/m} ... x_{n-1})$, at which point m processors can each add one sequence independently to create partial sums. The m partial sums need to be added together to form the final sum. In a message-passing system, the numbers would need to be passed to the processors individually. For a master-slave approach, the numbers are first sent from the master processor to slave processors. The slave processors add their numbers, operating independently and concurrently. Next, the partial sums are sent from the slaves to the master processor. Finally, the master processor adds the final sums to form the result, as in Fig. 2.10.

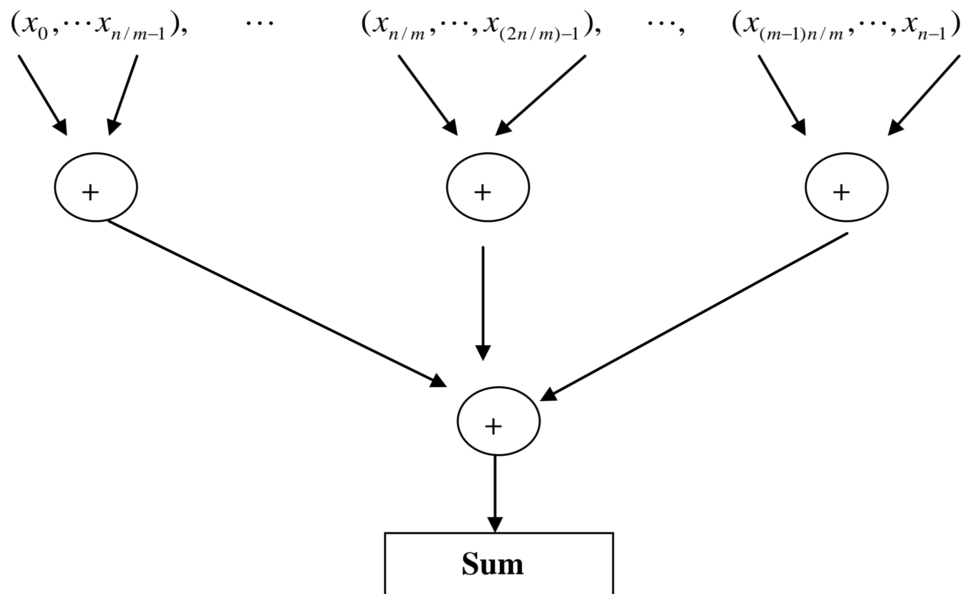

Fig. 2.10: Partitioning a sequence of numbers into parts and adding the parts.

2. Divide and Conquer**:** Jaja, (1992), differentiates between (i) dividing the problem, and (ii) combining the results. He categorizes the method as divide and conquer when the main work is combining the results, and categorizes the method as partitioning when the main work is dividing the problem.

3. Load Balancing**:** a difficult but important goal is to keep all processors equally busy. A problem is divided into a fixed number of processes that are to be executed in parallel. Each process performs a known amount of work. In this work, the load balancing is an important factor considered and is affected by the manner in which the domain is decomposed. With static load balancing the computation time of parallel subtasks should be relatively uniform across processors (Xue et al., (2010)); otherwise, some processors will be idle waiting for others to finish their subtasks. A better load balancing is achieved with the pool of tasks strategy, which is often used in master-slave programming (Coelho et al., (1993). With this strategy, the number of sub-domains should be relatively large compared to the number of processors. Otherwise, the slave solving the last sent block will force others to wait for the completion of this task; this is especially true if this processor happens to be the least powerful in the distributed system.

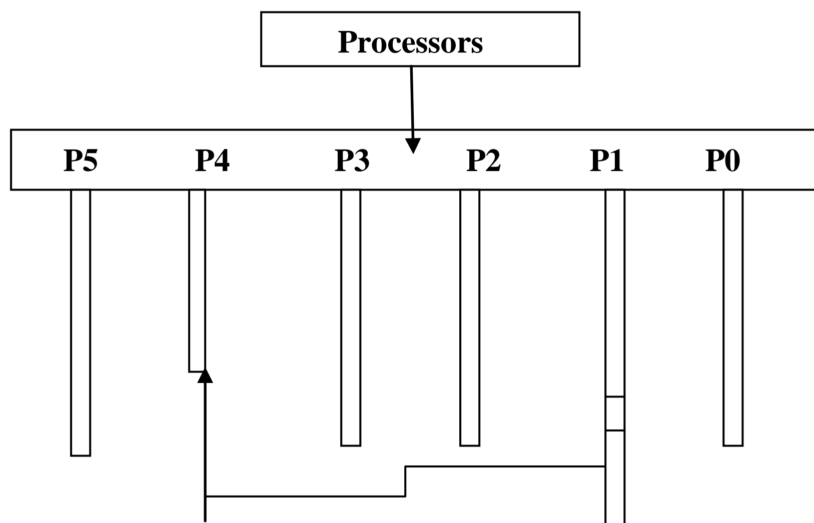

Fig. 2.11: Imperfect load balancing leading to increased execution time.

4. Synchronization Overheads: Succi et al., (1998) look at the work through estimating the different types of overheads which arise when calculating parallel efficiencies. Synchronization can be referred to as a group of separate computation that must at times wait for each other before proceeding, thereby becoming synchronized. The same computation is applied to a set of data points, and the operation starts at the same time in a lock-step manner. Synchronization means that we have spawned some processes and at some point we need to re-synchronize, 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.

5. Domain Decomposition: communication overheads are very sensitive to the way that data are positioned in the "divide and conquer" strategy (see Rathish et al., (2001)). The method of choice for many problems in physics and engineering is Domain Decomposition. DD is very straight forward in the case of structured problems, where calculations can be mapped onto a cube.

6. 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 straight forward example is the "overlapped domain decomposition" technique, where, in order to speed-up convergence of the parallel algorithm, the sub-domains are forced to overlap along the inter-processor 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.

2.7 Programming Message Passing

Athas & Seit (1998), considered message-passing concurrent computers. Data formated on different computers are often incompatible, this incompatibility is an important point in distributed computing because data sent from one computer may be unreadable on the receiving computer. Message-Passing packages developed for heterogeneous environments must make sure all the computers understand the exchanged data. More facts can be seen in Geist et al., (1994), details of message-passing can be found in Athas & Seitz (1998). In this thesis work, we specify two systems of message passing; PVM and the MPI. We use the high-level language such as C, augmented with message-passing library calls that performs direct process-process message passing. It is necessary to say explicitly what process are to be executed, when to pass messages between concurrent processors and what to pass in the message. Two primary methods are needed in this form of message-passing system (Barry & Michael (2003)):

1. A method of creating separate processors for execution on different computers.
2. A method of sending and receiving messages.

2.7.1 Process Creation

There are two methods of creating processes according to Barry & Michael (2003): Static and Dynamic process creation. In the so-called SPMD model, the different processes are merged into one program. Within the program are control statements that will customize the code, that is, select different parts for each process. After the source program is constructed to separate the action of each process, the program is compiled into executable codes for each processor. Each processor loads a copy of this code into its local memory for execution. The most general model for dynamic process creation is

the MPMD model in which a completely separate different programs are written for different processors.

2.7.2 Sending and Receiving Messages

Send and receive message-passing library calls often have the following form: send (parameter_list) and recv (parameter_list). Here send ( ) is placed in the send process, originating the message, and recv ( ) is placed in the destination process to collect the message being sent. The following are different types of sending and receiving messages:

(1) Synchronous message-passing: the term synchronous is used for a routine that actually return when the message transfer has been completed. This routine does not need message buffer storage. A pair of processes, one with a synchronous send operation and one with a matching synchronous receive operation, will be synchronized, with neither the source operation processor nor the destination process being able to proceed until the message has been passed from the source process to the destination process. Hence, synchronization routines perform two actions: they transfer data and they synchronize processes. Blocking and Non-blocking Message-Passing: the term blocking is used to describe routines that do not return. The routines are “blocked” from continuity. The term non-blocking was used to describe routines that return whether or not the message has been received. Generally, a message buffer is needed between the source and destination to hold messages.

(2) Collective Message-Passing: a frequent requirement for the process originating the message is to send the same message to more than one destination process. The term ‘broadcast’ is used to describe sending the same message to all the

processes concerned with the problem. The term ‘multicast’ is used to describe sending the same message to a defined group of process. Process 0 is the root process and holds the data to be broadcast in buf. The term ‘scatter’ is used to describe sending each element of array of data in the root to a separate process. The content of the ith location of the array is sent to the ith process. The term ‘gather’ is used to describe having one process collect individual values from a set of processes. ‘Gather’ is used after some computations have been done by these processes. ‘Gather’ is the opposite of ‘scatter’. The data from the ith process is received by the root process and placed in the ith location of array set aside to receive the data.

2.7.3 Parallel Virtual Machine (PVM)

The tool called PVM allows a heterogeneous collection of workstations and supercomputers to function as a single high-performance parallel machine (Geist et al., 1994). PVM is the mainstay of the heterogeneous network computing research project, a collective venture between Oak Ridge National Laboratory, the University of Tennessee, Emory University, and Carnegie Mellon University. The project began in the summer of 1989 at the Oak Ridge National Laboratory. PVM 1.0, was constructed by Vaidy Sunderam and Al Geist; version 2 was written at the University of Tennessee and released in march 1991, and a number of changes (PVM 2.1 – 2.4), and version 3 was completed in 1993. PVM version 3.3 is used in this work for our computational application. A key concept in PVM is that it makes a collection of computers appear as one large virtual machine, hence its name. PVM transparently handles all message routine, data conversion, and task scheduling across a network of incompatible computer architectures. These routine allows the initialization and termination of tasks

across the network as well as communication and synchronization between tasks. Communication constructs include those for sending and receiving data structures as well as high-level primitives such as broadcast, barrier synchronization, and global sum.

2.7.3:1 PVM Systems

According to Geist et al., (1994), the PVM System is composed of two parts. The first part is a daemon, called pvmd3 and sometimes abbreviated pvmd that resides on all the computers making up the virtual machine. The second part of the system is a library of PVM interface routines. The library contains user callable routines for message-passing, spawning processes, coordinating tasks, and modifying the virtual machine. Barry & Michael (2003) shows how PVM allows any number of processes to be created without any relationship to the number of processors, and it will allocate processes to processors automatically. PVM programs are usually organized in a Master/Slave arrangement whereby the single master program is executed and all others are spawned from the master process. The $pvm_spawn()$ is used to start executing one or more identical processes. Here, we describe the C versions of the system calls. Spawning options include specifying which computer to use. Processes are identified by the task IDs, returned in an array specified in $pvm_spawn()$. One of the first steps a process must take when it executes is to be enrolled in PVM, and this can be done with $pvm_mytid()$, which returns the task ID. Processors are disenrolled with $pvm_exit()$, which all processes should call before they exit. Programs communicate by PVM library routines such as $pvm_send()$ and $pvm_recv()$, which are embedded into the programs prior to compilation. All PVM send routines are non-blocking (or asynchronous in PVM terminology) while PVM receive routines can be either blocking (synchronous) or non-blocking. Operations of sending and receiving data are done

through message buffers. PVM uses message tag (msgtag), attached to a message to differentiate between types of message being sent. If the data to be sent is composed of various types, the data has to be packed into a PVM send buffer prior to sending the data. There are specific packing and unpacking routines for each data type, for example, $pvm_pk\text{int}()$ and $pvm_upk\text{int}()$ for integers, $pvm_pkstr()$ and $pvm_upkstr()$ for strings, $pvm_pkfloat()$ and $pvm_upkfloat()$ for floats. The basic message-passing routines for packed messages are $pvm_send()$ (non-blocking), $pvm_recv()$ (blocking), and $pvm_nrecv()$ (non-blocking). The unpacking must be done in the same order as the packing. The default send buffer, which is normally used, must first be initialized by the source process with $pvm_initsend()$.

2.7.4 Message Passing Interface (MPI) System

MPI systems are discussed in (Ananth et al., (2003), Braunl (1993), Groop et al., (1999), Neil et al., (1994) of the Edinburgh parallel computing centre, Barry & Michael (2003), and Pacheco (1997). There are differences between PVM and MPI. A fundamental aspect of MPI is a “standard” that has implementations. MPI has a large number of routines over 120 and growing, and an important objective in developing MPI is the desire to make message-passing portable and easy to use. The first version of MPI was finalized in May 1994 and omitted some advanced or controversial features that may be added to subsequent versions. It is suggested by Groop, Lusk, & Skjellum (1999) that successful programs could be written with only six of the 120+ functions. The function calls are available for both C and FORTRAN, but we consider C versions. All MPI routines start with the prefix $MPI_$ and the next letter is capitalized. Generally, routines return information indicating the success or failure of the call which can be found in Snir et al., (1998) and Bruck et al., (1997). A significant difference from

PVM is that only static process creation is supported in MPI version 1, i.e., all processes must be defined prior to execution and started together. There is no equivalence to the PVM call $pvm_spawn()$. MPI has support for defining topologies (meshes), and hence it has the potential for automatic mapping. Before any MPI call, the code must be initialized with $MPI_Init()$, and after the MPI function calls, the code must be terminated with $MPI_Finalize()$.

```
Main(int argc, char *argv[] )
{
    MPI_Init ( &argc, &argv );              /* initialize MPI */
    ⋮
    MPI_Finalize ();                        /* terminate MPI */
}
```

All process is enrolled in a "universe" called MPI_COMM_WORLD, and each process is given a unique rank, a number from 0 to $n-1$, where there are n processes. MPI_COMM_WORLD, is a communicator.

2.8 Motivation for PVM

Static groups and message contexts are in PVM. The PVM sources and destinations are always absolute in terms of the "task ids". PVM was the effort of a single research group, allowing it great flexibility in design and also enabling it to respond incrementally to the experiences of a large user community. Moreover, the implementation team was the same as the design team, so design and implementation could interact quickly. PVM had, the exception of support for heterogeneous computing and a different approach to extensibility. In particular, PVM was aimed at providing a portable, heterogeneous environment for using clusters of machines using socket communications over TCP/IP as a parallel computer. Because of PVM focuses on socket-based communication between loosely-coupled systems, PVM places a greater

emphasis on providing a distributed computing environment and on handling communication failures.

Initially a PVM implementation simply aborted when an error was detected. Attempts were made to provide a useful error indication and to allow the run to continue. PVM is a virtual machine (implemented as the PVM daemons) that provides a simple yet useful distributed operating system. Special interfaces such as the pvm_reg_tasker, allow the PVM System to interface with other resource management systems, but MPI does not define a virtual machine. The pvm_config function provides the information on the virtual machine. This information can be used by the programmer to attempt to manage resources directly, for example, by specifying particular hosts in pvm_spawn.

In PVM, created processes have only one parent; this reflect PVM's use of the fork / execution or system spawn model of process creation as separate from connecting processes for communication. PVM ensures uniqueness because there is a single virtual machine. PVM provides asynchronous operation, It has specific functions to pack special data types into buffers, it is defined primarily by a single implementation for workstation networks, with freedom to add features appropriate for that environment. Hence, PVM provides more support for fault tolerance and recovery by exposing to the programmer some of the properties of sockets.

2.9 Motivation for MPI

MPI, the current dominant programming model for parallel scientific programming, forces coders to be aware of the exact mapping of communicational tasks to processors. This style has been recognized for years to increase the cognitive load on programmers, and has persisted primarily because it is expressive and delivers the best performance

(Snir et al. (1998), Gursoy & Kale, (2004)). Because we anticipate an increase in exploitable concurrency, we believe that this model will break down in the near future, as programmer have to explicitly deal with decomposing data, mapping tasks, and performing synchronization over thousands of processing elements. Recently efforts in programming languages have focused on this problem and their offerings have provided models where the number of processors is not exposed (Allen et al., (2006), Callahan et al., (2004)). While attractive, these models have the opposite problem-delivering performance. In addition, because the program is not over specified, the system has quite a bit of freedom in mapping and scheduling that in theory can be used to optimize performance.

Dynamic processes functionality is improved with MPI. In MPI, sources and destinations are relative to a group. MPI was designed by the MPI forum (a diverse collection of implementers, library writers, and end users) quite independently of any specific implementation, but with the expectation that all of participating vendors would implement it. Some of the goals of MPI Forum are:

- MPI would be a library for writing application programs, not a distributed operating system. This goal has implications for resource management issues.
- MPI would not mandate thread-safe implementations, but its specification would allow them. Thread safety implies that there can be no notion of a "current" buffer, message, error code, and so on. As the "nodes" in the network becomes increasingly important in a heterogeneous networked environment.
- MPI would be capable of delivering high performance on high-performance systems. Hence, no memory copies would be mandated by the design. Scalability, combined with correctness, for collective operations required that groups be "static".

- MPI would be modular, to accelerate the development of portable parallel libraries. Hence, process source / destination must be specified by rank in a group rather than by an absolute identifier, and context must not be a visible value.
- MPI would be extensible to meet future needs and developments. This requirement led to an object-oriented approach without a commitment to an object-oriented language.
- MPI would support heterogeneous computing (the MPI_Data_type object allows implementations to be heterogeneous), though not require that all implementations be heterogeneous.
- MPI would require well-defined behavior (no race conditions or avoidable implementation-specific behavior. See Groop (2001), for a discussion of the importance of these goals to the success of MPI.
- In MPI, standards specifications tend to specify the minimum level of complies, while any implementation offers more functionality. In the MPI Forum, many such "added-value" features are listed as expected of a "high-quality implementation. Error handling and recovery are a good example. Thus, most MPI implementations do not simply abort when an error is detected; just as the PVM implementation does, they attempt to provide a useful error indication and allow the user to continue.

Daemons are not required by the MPI specification. Its implementations are required to ignore unrecognized fields; this strategy encourages users to provide extra information when possible. Hence, the MPI_Comm_spawn call combines process creation with information on the needed resources. Combining operations is a classic approach for solving race conditions, and this solution is used in many places in MPI. Because of the presence of race condition MPI forms the MPI communicator. MPI uses

“nonblocking” library routines for communication activities. It provides an extension set of nonblocking operation. It provides such operation not only to allow for overlapping communication, but also to make it easier to write portable, correct programs.