AN OVERVIEW OF MESSAGE PASSING ENVIRONMENTS

Oliver A. McBryan

CU-CS-717-94

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

AN OVERVIEW OF MESSAGE PASSING ENVIRONMENTS

Oliver A. McBryan

CU-CS-717-94   1994

Department of Computer Science
University of Colorado at Boulder
Campus Box 430
Boulder, Colorado 80309-0430 USA

An Overview of Message Passing Environments[*][†]

Oliver A. McBryan
Dept. of Computer Science
University of Colorado
Boulder, CO 80309.

Email: mcbryan@cs.colorado.edu

Abstract

A majority of the MPP systems designed to date have been MIMD distributed memory systems. For almost all of these systems, message passing environments have provided the primary mechanism for programming multiprocessor applications.

In this paper we provide an introduction to MPP systems in general. We then introduce current MPP message passing interfaces, by tracing their historical development over the last 10 years. In addition to their use within a single MPP architecture, we discuss the use of message passing systems to interconnect more loosely coupled processors in heterogeneous environments. Finally we review the development of "portability platforms" - message passing systems that have been devised solely to allow portability of message passing programs between different systems.

[*]Research supported in part by NSF Grand Challenges Applications Group grant ASC-9217394 and by NASA HPCC Group Grant NAG5-2218.
[†] To appear in Parallel Computing, April 1994.

TABLE OF CONTENTS

1. Introduction 1

2. Overview of MPP Architectures 4

2.1. Classification of Parallel Computers and Software 4

2.1.1. SIMD and MIMD 4

2.1.2. SPMD Paradigm 4

2.1.3. Massive Parallelism 5

2.1.4. Shared and Distributed Memory 5

2.1.5. Message Passing 5

2.1.6. Virtual Shared Memory 5

2.1.7. Scalable Systems 6

2.1.8. Scalable Algorithms 6

2.2. Approaches to Multicomputing 6

2.3. Node design 7

2.4. Communication Features 9

2.5. Parallel Software 11

3. Message Passing in MPP Environments: The Hypercube Example 13

3.1. Hypercube Architectures 13

3.2. The Caltech Hypercube 13

3.2.1. Caltech Hypercube Programming: CROS 14

3.3. The Intel iPSC1 Hypercube 15

3.3.1. Intel iPSC1 Programming: NX1 15

3.4. The Intel iPSC2 Computer 18

3.4.1. Intel iPSC2 Programming: NX2 18

3.5. nCUBE Hypercubes 20

3.5.1. nCUBE PSE 20

3.6. Cost/Benefit Aspects of Different Message Passing Environments 20

4. Other Message Passing Systems 22

4.1. IBM EUI 22

4.2. Meiko CS System 22

4.3. SUPRENUM 23

4.4. Thinking Machines CMMD 24

4.5. Virtual Shared Memory 25

5. Portability Platforms and Heterogeneous Environments 26

5.1. RPROC 26

5.2. P4 Macros 27

TABLE OF CONTENTS

5.3. PARMACS 28
5.4. EXPRESS 28
5.5. PVM 29
5.6. Zipcode 29
5.7. LINDA 30
5.8. MPI 30

References...32

An Overview of Message Passing Environments[*†]

Oliver A. McBryan
Dept. of Computer Science
University of Colorado
Boulder, CO 80309.

Email: mcbryan@cs.colorado.edu

1. Introduction

This paper provides an overview of massively parallel processing (MPP) hardware, an introduction to message passing systems used on MPP systems, and a discussion of more general message passing systems used to interconnect processes running on networks of computers of diverse types.

Sequential processors are controlled by a single program, which defines a set of control instructions that are scheduled for execution in the specified order and which access and use the available memory to effect the desired computation. MPP computers are far more complex, having some number P of sequential processors, and in many cases each processor has local memory which is not visible to other processors. An MPP will again be controlled by a program, which defines and schedules the operations to be executed in each of the P processors and which accesses and uses the available memory (of all of the processors). Logically this is equivalent to P programs, one running in each processor, plus a set of extra instructions that specify the relative scheduling of operations on the P processors and which provide access for each processor to the memory of other processors.

Message passing (MP) is one of several paradigms used to write an MPP program. We briefly discuss a number of other well-known paradigms in section 2. Message passing provides the two key aspects of MPP programming: a) synchronization of processes and b) read/write access for each processor to the memory of all other processes. Message passing is defined by the limited way in which access to other memories is provided: a process can send a copy of any item in its memory (called a *message*) to another process, but no other access is provided. While apparently providing only a limited form of read access, this mechanism actually allows *cooperating* programs to effectively have read/write access to each

[*]Research supported in part by NSF Grand Challenges Applications Group grant ASC-9217394 and by NASA HPCC Group Grant NAG5-2218.
[†] To appear in Parallel Computing, April 1994.

others memory. A process receiving a message containing a data item may write the received item to a pre-agreed location in its own memory. Logically, the sender of that message has now modified the receivers memory. Furthermore cooperating processes can use the mutual receipt of messages as a pre-agreed synchronization mechanism. While clumsy, this indicates that message passing is indeed capable of providing the full functionality specified above for MP programming.

An MPP message passing program consists of *P* sequential programs, one run in each process. Each sequential program uses message passing instructions to synchronize with and to access memory of other programs. Typically these MP instructions come from a limited set of instructions that define the Message Passing Environment. Usually the instructions are made available to programmers in the form of a Message Passing Library The MPP programming environment then consists of a standard uniprocessor programming language such as C or Fortran, along with the Message Passing Library. In the extreme case, each processor would have a different program and each program would be synchronized with all others by inserting a synchronization call after each instruction. In practice programs are usually far more loosely coupled, with occasional synchronizations but also long periods where programs run independently of each other using only their local memories.

While all message passing systems are logically similar, there are differences in the ways in which the two key services (synchronization, data exchange) are provided. This has resulted in a large number of different incompatible systems. Well known systems include Caltech's CROS, IBM's EUI, Intel's NX, Meiko's CS, nCUBE's PSE and Thinking Machines CMMD. We will review each of these systems and others, indicating their historical development and providing some comparisons of features. Because the required services are fundamentally the same it is possible to implement any of these in terms of another - for example given a CMMD library one can create an NX library which makes one or more CMMD calls to provide each desired NX function. Later papers in this volume address each of these systems in greater detail.

Several groups have attempted to overcome the vendor/machine specificity of the MP environments by developing so called Portability Platform MP environments. The idea here is to choose one environment specification and implement it on all hardware platforms - either as a native MP library, or by making appropriate calls to the resident MP environment. Programs written using a Portability Platform MP environment become portable to all systems, which is obviously a great advantage. Examples of well-known portability platforms include EXPRESS, Linda, P4, PARMACS, PVM and Zipcode. We will introduce these systems later in this paper, and again papers following in the volume explore each of these systems in greater detail.

The large number of portability platforms presents some of the same problems as the number of vendor interfaces. Furthermore most of the portability platforms support only a subset of the full features of the vendor systems. It was for these reasons that the MPI Committee was formed in 1992 to define a new standard message passing interface called MPI (for Message Passing Interface). MPI was to support all of the best features of the various vendor and portability platforms. The MPI definition has now been published and it is expected that all major vendors will support it with native implementations. In the meantime implementations of MPI in terms of the portability platforms are underway, and in some cases already complete, providing portable MPI implementations that are available immediately for evaluation. One of the papers in this volume is a complete introduction to MPI.

Section 2 of this paper gives an introduction to design issues for MPP systems, including both software and hardware. In section 3 we take a more detailed look at the subset of hypercube architectures and introduce the major developments in message passing environments for these architectures that has resulted in today's programming systems. This section provides both an historical perspective and a tutorial introduction to message passing features - we highlight the key concept that led to introduction of each new feature. Section 4 looks briefly at several other message passing systems that have been developed for non hypercube architectures. Section 5 introduces the portability platforms that have been developed for message passing systems as well as messaging environments designed specifically for heterogeneous systems.

2. Overview of MPP Architectures

In this section we will provide a general discussion of design issues for massively parallel systems, with some discussion of the historical developments and systems that have resulted in the current massively parallel supercomputer architectures. We will present a detailed look at message passing systems developed for some of these architectures in sections 3 and 4.

2.1. Classification of Parallel Computers and Software

2.1.1. SIMD and MIMD

Parallel computers may be broadly categorized in two types - SIMD or MIMD [1]. SIMD and MIMD are acronyms for Single Instruction stream - Multiple Data stream, and Multiple Instruction stream - Multiple Data stream, respectively. In SIMD computers, every processor executes the same instruction at every cycle, whereas in a MIMD machine, each processor executes instructions independently of the others. The vector unit of a CRAY computer is an example of SIMD parallelism - the same operation must be performed on all components of a vector. The CM-200 and Masspar MP-2 are also examples of SIMD computers. Most of the interesting parallel computers are of MIMD type which greatly increases the range of computations in which parallelism may be effectively exploited using these machines. However, this occurs at the expense of programming ease - MIMD computers are much more difficult to program than SIMD machines. Many current designs incorporate both MIMD and SIMD aspects - often each node of a MIMD system is itself a vector processor, or as in the case of the CM-5, each node consists of a scalar Sparc processor and four vector processors.

2.1.2. SPMD Paradigm

A SIMD computer is controlled by a single program, which greatly eases the complexities of program development. A MIMD computer in principle will have a different program running on every node, or possibly several different programs on every node if multiprocessing nodes are involved. This makes for an extremely complex programming environment. A frequent programming paradigm for MIMD machines is the SPMD model, an acronym for Single Program - Multiple Data. In this model the same program text is run on all processors of a system, but the execution may follow different paths through the program on different processors. Clearly any set of P MIMD programs can be replaced by a single merged program, at the possible cost of some increased program memory. Thus there is essentially no loss in restricting to SPMD programming.

2.1.3. Massive Parallelism

While many interesting parallel machines involve only a few processors, we will restrict consideration to those machines which have moderate to large numbers of processors, since they represent the path to the highest performance. Furthermore, we will emphasize those machines which have a distributed memory architecture, including virtual shared memory systems, because at this point such systems are the only ones that appear to be truly scalable. Important classes of machines such as the CRAY C-90, CRAY-3, and several similar Japanese supercomputers are not the focus of our remarks because such systems do not appear to offer true scalability (see below) using any current approach.

2.1.4. Shared and Distributed Memory

Another easy categorization is between machines with global or local memories. In local memory machines, usually referred to as distributed memory systems, communication between processors is entirely handled by a communication network, whereas in global memory machines a single shared memory is accessible to all processors. Shared memory systems offer the advantage of much easier programming. Building massively parallel shared memory systems that also scale (see below) is extremely difficult however.

2.1.5. Message Passing

Processors in distributed memory systems have no direct access to the memory of other processors. In order to utilize multiple processors on one task it is necessary to exchange information between processors by sending packets of data - messages - between them using an available communication network. Software libraries to facilitate such exchange of data are called Message Passing Environments. While communication networks vary enormously in detail, they tend to provide broadly similar capabilities for exchanging data. As a result, message passing environments are often remarkably similar across architectures and this has led to the development of portable message passing environments as discussed in the Introduction. This paper deals almost entirely with message passing environments, both machine specific and portable.

2.1.6. Virtual Shared Memory

Even in distributed memory systems it is possible to simulate a global shared memory in software, or with a combination of software and hardware. Such systems, referred to generically as virtual shared memory (VSM) systems, offer the ease of use of a shared memory system, while preserving the scalability of distributed systems. VSM is a powerful paradigm, but is currently available on only a few architectures. Note that we distinguish here the case of disk-based virtual memory supported on individual processors which is unrelated to VSM.

2.1.7. Scalable Systems

The driving force behind supercomputer development has been the insatiable computational needs of the so-called Grand Challenge (GC) problems. GC problems are those major problems of interest to science and engineering whose solution is possible to contemplate using computers, either directly or through numerical simulation. In order that parallel supercomputers can make a serious attack on the Grand Challenge applications, it is essential that these systems have the property of scalability. Scalability essentially means that performance increases linearly with the number of processors P:

$$Perf(P) = O(P).$$

In practice if performance drops below linearity by not more than a logarithmic factor, it is also considered to be scalable:

$$Perf(P) = O(P / (\ln P)^c).$$

Scalability in parallel computer architecture is a critical issue for massively parallel computer designers. This is a major departure from conventional supercomputer design where at most eight or sixteen processors are involved. In such systems the design emphasis is primarily on node performance and I/O to disk.

2.1.8. Scalable Algorithms

A second critical aspect of Grand Challenge applications is the need to design scalable algorithms. An algorithm is scalable on a scalable architecture if performance scales linearly with the number of processors, provided that the problem size is scaled so as to need that many processors. The need for a particular minimum number of processors is usually driven by the memory needs of the application. Again a reduction from linearity by a logarithmic factor is still considered scalable.

The design of scalable machines and the search for scalable algorithms for Grand Challenge applications on such machines are the two dominant forces controlling HPC activities today.

2.2. Approaches to Multicomputing

There are several hundred recent or current parallel computer projects worldwide. Table 1 lists a selection of such projects, a mix of university and industrial enterprises. This is just a sample of the diverse projects but covers a wide range of different architectures chosen more or less at random. While some of these projects are unlikely to lead to practical machines, a substantial number have already, or will, lead to useful prototypes. Many commercial parallel computers are included in the table and these have been or are already in production (e.g. Alliant FX/8, FX/2800 and CAMPUS/800, Connection Machine CM-2, CM-5, CRAY Research T3D, Denelcor HEP-1, Evans and Sutherland ES-1, FPS T-Series, IBM SP-1, ICL DAP, Intel iPSC2 and Paragon, Kendall Square KSR1, Meiko CS-2, Myrias SPS-3, nCUBE-2, Parsytec GC(T805), SUPRENUM-1, Symult 2010) and more are under development. Some of these products have been commercial failures (e.g. Denelcor HEP, ETA-10, ES-1, FPS T-Series, Symult 2010), yet they have provided important insights into parallelism. One should also remember that the latest CRAY Research vector computers, (e.g. CRAY-2, CRAY Y-MP and CRAY Y-MP/C90) involve multiple processors, and other vector computer manufacturers such as NEC, Fujitsu and Hitachi have similar strategies.

Beyond the simple classification into SIMD or MIMD computers we recognize a vast array of different approaches to the task of building a parallel architecture. We will now look at the reasons for this broad range by discussing some of the possibilities encountered for both node and communication facilities.

Table 1. Some Parallel Computer Projects

Alliant Campus/800	Alliant FX/2800
Alliant FX/8	BBN Butterfly
CalTech Hypercube	Cedar Project
Connection Machine CM-2	Connection Machine CM-5
CRAY Y-MP and CRAY-2	Data-flow Machines
Denelcor HEP-1	Encore Multimax
ETA-10	Evans and Sutherland ES1
Flex 1	FPS T-Series
Fujitsu AP-1000	Goodyear MPP
IBM GF-11 and TF1	IBM SP-1
ICL DAP	Illiac IV
Intel DELTA	Intel iPSC/860 Hypercube
Intel iPSC2 Hypercube	Intel iWarp
Intel Paragon	Intel Paragon
Intel Touchstone	Kendall Square KSR1
Masspar MP1000	Meiko MK860 and CS-2
Multiflow	Myrias SPS-3
Navier-Stokes Machine	nCUBE2 Hypercube
NYU/IBM RP3	Parsytec
Sequent Balance	SUPRENUM-1
Symult 2010	TERA

2.3. Node design

By *node* we mean the individual computational processor, along with its associated communications hardware, and local memory if available. Node design tends to be far less variable than other aspects of parallel computers. The main reason for this is that most architects have relied on off-the-shelf products for the node - standard microprocessors, floating point accelerators and memory chips. The advantage is that startup time for a project may be substantially reduced. Additionally, there is a substantial body of low-level software available for such processors - such as compilers, assemblers and debuggers. Thus we find that an enormous number of the recent or current parallel computer products are based on one or more of the DEC Alpha processor, IBM R6000 processor, Inmos T800 and T9000 transputers, Intel 80X86 and i860, Motorola 680X0, Sparc processors and Weitek floating point accelerators. Typically one of these microprocessors will be combined on a board with a floating point coprocessor (e.g. 80387 or 68881), possibly a Weitek processor and several megabytes of memory. The more recent processors (e.g. i860) tend to have built-in floating point processing, and sometimes a graphics processing capability. Despite these general comments, it should be mentioned that some manufacturers have developed custom processors specifically for parallel computers. In the list above we would point to the CM-2, DAP, ES-1, HEP-1, iWarp, KSR1, Navier-Stokes and nCUBE machines as examples.

Memory consumes substantial space, and current systems have in the range of 1 to 32 Mbytes per node, although up to 128 Mbytes has been announced for some systems. Most systems support several levels of memory on a node: frequently main memory, cache memory and registers. Effective management of cache memory is often critical for good node performance, and as a result both the design of the cache and the quality of the single-node compilers are essential aspects of all MPP systems.

Recently there has been a trend towards complexifying the physical node concept in order to increase packaging density and reduce cost. Early examples were the CM-1 and CM-2 machines which have 16 processors on a single chip. In the CM-2 this is further confused by the fact that a single Weitek vector processor is shared by the 32 processors on each pair of adjacent chips. The CM-2 and CM-200 slicewise operating systems in fact recognize this in that they present a view of the system as consisting of 2048 vector processing nodes, essentially ignoring the view of the system as composed of 65,536 bit-serial processors. The CM-5 node consists of a Sparc processor and 4 vector processors, which may only be treated as a unit. Intel has announced that sometime in 1994 they will switch from the current Paragon node design of two i860 processors (one computational and one communicational), to a highly integrated chip containing 5 i860 processors - and in fact these processors share memory.

The increasing complexity of the node concept is a major challenge for any simulation system for massively parallel computers. It is also a major challenge for users and software designers who would prefer not know about these details. One resolution of this dichotomy is to separate the concepts of logical and physical node. A logical node is a unit that supports a single thread of computation and memory access at any time. A physical node is a level in the physical hierarchy constituting a massively parallel system, and is the smallest hierarchical level that contains at least one CPU.

2.4. Communication Features

The range of interprocessor communication facilities is what really characterizes the differences in architecture among various parallel machines. While we have previously distinguished the shared memory and distributed memory classes, one should observe that this distinction should not be taken too seriously. A distributed memory computer can certainly simulate a global shared memory and vice versa.

Communication pathways are typically built either from direct point-to-point connections, or from busses. Busses have the advantage that many processors may be serviced by one communication path, but have the disadvantage of slower bandwidth performance as the number of processors increases. With point-to-point connections, processors that are directly connected will have very efficient communication, but indirectly connected processors will likely incur substantial extra overheads including increased latency as well as lower bandwidth. Over the last seven years, interconnection performance has improved almost as fast as processor performance. For example, the Intel systems have moved from performance in the range of hundreds of Kbytes/sec on the iPSC1 to 200 Mbytes/sec (peak) on the Paragon. Communication latency has also improved substantially although to a lesser extent over this period, from about 5000 microsecs on the iPSC1 to 60 microseconds on the Intel Paragon for example.

The most popular interconnection strategies involve simple symmetric arrangements including rings, meshes, hypercubes, trees and complete connections or crossbars. The prevalence of hypercube designs is explained by the fact that the architecture supplies substantial parallel bandwidth for many standard algorithms, for example the Fast Fourier Transform, while at the same time incurring only relatively modest fan-in and fan-out of connections which grow in number only logarithmically with the processors. However, total hypercube wiring complexity grows super linearly with the number of processors and the likelihood increases that most wires are unused most of the time. Table 2 compares several simple topologies as a function of processor number P from the point of view of amount of wiring (difficulty of building), connectivity (ease of programming) and maximal path (efficiency of long-range communication).

Table 2. Properties of Interconnection Networks

Network	Wires	Connectivity	Max Path
Cross Bar	P(P - 1)	P	1
1D Grid	P-1	2	P-1
2D Grid	2P	4	$2\sqrt{P}$
Binary Tree	P	3	$2\log_2 P$
Hypercube	$\frac{1}{2}P\log_2 P$	$\log_2 P$	$\log_2 P$

While cross bar switches are extremely difficult to build for large numbers of processors, they have tremendous flexibility in terms of efficiency and ease of use. It is conceivable that a technological breakthrough such as optical switching might allow cross bars to be built that would connect thousands of processors. For the time being, crossbars are restricted to small systems of at most 64 processors, or to providing interconnects among the processors of sub-clusters within larger machines (e.g. the ES-1 and Alliant FX/2800).

Bus based connection networks are attractive for moderate numbers of processors, for example 16 to 32. Beyond this point bandwidth begins to suffer intolerably. Architectures based on busses therefore tend to be hierarchical beyond that number of processors. As an example, the SUPRENUM-1 computer used a fast local bus to connect within a cluster of 16 processors. Clusters are arranged in a rectangular grid and connected by row and column busses, which has the added attraction of providing redundancy and double bandwidth. The Myrias SPS-3 is similar, utilizing three levels of bus: 4 processors connected on a single board by a bus, cages of 16 boards connected by a pair of backplane busses, and finally cages connected by third-level busses. The KSR1 connects 32 processors in a ring as a basic cluster, with rings of clusters used to scale the system up to 1088 nodes.

New configurations of processors continue to be proposed. One common technical approach is the use of "worm-hole" routing in distributed systems. The basic idea here is to allow circuits to be established between remote processors, without the necessity of interrupting any intermediate nodes. While there may be a small overhead for circuit creation, subsequently all data traverses the circuit without overheads such as multiple startup costs at intermediate nodes. Once a circuit is established, communication proceeds essentially in pipelined fashion. Frequently it suffices to create logical rather than physical connections. These allow messages to proceed on virtual worm-hole channels, but with the possibility that physically the channels are multiplexed. This is particularly convenient as a means for preventing dead-locks and blocking of small messages by large ones. The resulting communication performance tends to be essentially independent of distance.

Worm-hole routing is utilized in the CM-2, CM-5, the iPSC2, iPSC/860, the Intel Paragon and the Intel iWarp among others. The CM-2 and the Symult 2020 were the first systems to emphasize wormhole routing. In the case of Symult, the designers were so confident of the advantages of wormhole routing that they abandoned a hypercube architecture from their first generation in favor of a simple two-dimensional rectangular grid. The Intel iPSC2 hypercube has similarly given way recently to a rectangular-grid based Intel Paragon, similar in spirit to the Symult. This could not have been attempted without worm-hole routing.

2.5. Parallel Software

Software for currently available parallel computers is extremely limited. In all cases manufacturers provide Fortran and C compilers, which are frequently just a single processor compiler. These compilers usually have no concept of parallelism or of communication capability. Typical examples are the systems supplied by Intel, Meiko and nCUBE. In these systems, all communication and process control is initiated explicitly by the user, resulting in substantial code modification as well as a loss of portability of software. Typically libraries of low-level communication primitives are supplied with these systems to allow the user to initiate communication operations.

A few manufacturers have gone beyond this by providing language extensions that capture aspects of the parallel hardware. Thinking Machines provides a parallel Fortran, CMF, for their Connection Machine CM-2 and CM-5 computers. The compiler supports the Fortran 90 array extensions to Fortran 77, and the convention is that objects used as arrays are understood to be distributed across the parallel processors. Communication among processors is supported by the F90 shift operations, as well as the various reduction operators such as vector sum. While the CMF environment is remarkably elegant and user-friendly, one should point out that the task is much simplified by the SIMD nature of the CM-2 hardware onto which array operations map extremely well. CMF also runs on the MIMD CM-5, but is less natural there. New language standards such as HPF (High Performance Fortran) extend on languages such as CMF, but offer significant advantages only for essentially SIMD computations.

Myrias Corporation (SPS-2, SPS-3) and Evans and Sutherland (ES-1, no longer in production) both supported a virtual address space across processors. If a processor attempts to access a memory location not in its physical memory, then a page fault occurs and the appropriate memory page is fetched from the processor which has it. Myrias in particular have implemented a sophisticated mechanism for load balancing and rapid access to memory. The system attempts to localize page table information and to provide access to it in a distributed fashion. The Myrias system was the first to provide virtual shared memory on a distributed memory architecture. The ES-1 was actually a cross-bar based shared memory computer, and here again virtual memory was provided to make the memory system more transparent.

The most recent system of this type is the KSR1 (and now KSR2) which combines a two-level hierarchical memory based on rings, with real supercomputer performance (1088 nodes, 40 Mflops/node peak). The system implements a global 40-bit address space using hardware assistance for virtual shared memory operations. The KSR-1 system supports interprocessor communication at rates of 34 Mbytes/sec with a latency of only 6 microsecs.

One should also note the tendency to support virtual processes. This is an important aid to software development as it allows an application to simulate a larger number of processors than are physically present. Virtual processing actually can increase system throughput by allowing nodes to remain computationally active while a process is suspended waiting for memory or messages. Virtual processing is supported by the majority of systems in one form or another. Examples include CM-2, iPSC, Myrias and SUPRENUM.

Another recent trend is towards providing a complete UNIX capability on every node. For example the Alliant CAMPUS/800, the KSR1 and the Intel Paragon provide this feature. In some cases (e.g. Paragon) this is accomplished by supporting a microkernel on computational nodes which relies on services provided by special server and I/O nodes. In addition to UNIX, other standard software products are being offered by a wide range of vendors: for example NFS file systems, TCP/IP networking to nodes, and support for graphics systems such as AVS and Explorer.

3. Message Passing in MPP Environments: The Hypercube Example

3.1. Hypercube Architectures

Hypercube architectures have been very frequently developed because of the advantages of high connectivity and low wiring costs already referred to above in Table 2. Mathematically, a hypercube may be easily defined recursively, as a set of vertices and edges, as follows. A 0 dimensional hypercube consists of one vertex. A d dimensional hypercube consists of a pair of $d-1$ dimensional hypercubes in which corresponding vertices of each hypercube are joined by a new edge. It follows that the number of vertices P in a hypercube doubles and the number of edges C leaving a vertex (i.e. the connectivity) increases by one, with each increase in dimension. Therefore we have $P = 2^d$ and $C = \log_2(P)$, which establishes several of the entries in Table 2.

From the computational point of view, a major significance of hypercubes is that topologies for many important algorithms map naturally and efficiently to hypercubes. For example, rings, grids, binary and other trees and butterfly networks all map to hypercubes. Furthermore algorithms that utilize these structures will typically be able to fully overlap communications on different wires - i.e. there is sufficient parallelism in the hypercube wiring to allow communication, as well as computation, to be fully parallelized. This is a key aspect of developing scalable algorithms.

As an illustration of developments in message passing interfaces we will now trace the progress in such interfaces for hypercubes over the last decade. This discussion is intended as both an historical review and a tutorial introduction.

3.2. The Caltech Hypercube

The original distributed memory MPP computer from which many later designs were derived, was the Caltech Hypercube. This system, built around 1984 was based on the simple d-dimensional hypercube model. As noted above, a d dimensional hypercube has 2^d vertices with d edges radiating from each. In the original Caltech design, which was a 6 dimensional hypercube, each cube vertex (termed an Element, or ELT for short) contained an Intel 286/287 processor and 64KB of memory. The cube edges were data communication channels allowing data transfer between adjacent processors. In addition to the 64 nodes, an additional processor called the Intermediate Host (IH for short) was provided as the user interface and was connected to node 0. All user communication with the hypercube occurred via the IH. The software environment of the system consisted of standard

Fortran 77 and C compilers, plus a library of communication calls, known as the Crystalline Operating System or CROS [2].

3.2.1. Caltech Hypercube Programming: CROS

Assignment of a numbering scheme for nodes of an MPP is a necessary prelude to any communication operations. The topology of an architecture usually suggests natural numbering schemes. For example, in a two-dimensional grid architecture, coordinate pairs provide a natural numbering system for nodes. In a hypercube it is possible to represent each vertex (processor) by a binary number such that physically adjacent processors always differ only in 1 bit - in fact there are many different numberings of vertices that satisfy this condition.

In the Caltech Hypercube, this property was used to define the concept of the *channel* connecting neighboring processors. Each processor has $d = \log_2(P)$ wires connecting it to neighbors. The wire connecting to the neighbor whose processor number differs in bit c is called channel c, $0 \le c < d$. Thus each processor sees d channels numbered $0,1,\ldots,d-1$. It is easy to verify that if processor A sees processor B at the end of channel c, then processor B will also report that processor A is on the end of its channel c. An extra channel is also defined to connect from the Intermediate Host to Node 0. All data transfers are then specified by channel. Data packets were restricted in the original CROS to a size of 8 bytes.

The complete communication interface for the Caltech Hypercube is summarized in Table 3. While a few extra calls were supported, these typically combined two of the operations into a compound operation.

Table 3: Caltech Hypercube Communication Calls

Host-Node Routines:	
wtIH(data,CUBE)	Write a packet from IH to node 0
rdsig(data)	Data from IH is read by each node
wtres(data)	Data sent from node to IH
rdbufIH(datas,CUBE,P)	Read in all data sent by the nodes
Node Routines:	
wtELT(data,chan)	Send data on channel chan
rdELT(data,chan)	Read data on channel chan

The first four routines listed provide communication between the IH and the hypercube nodes. Typically a program begins with a call to *wtIH* which sends an item of data to the cube (CUBE denotes the unique extra channel provided to node 0). The 8 bytes of data might contain information in the form of two integers or floats, a double, or a string of up to 8 characters. The cube nodes respond to the *wtIH* call by themselves calling *rdsig* to receive the incoming data. The name *rdsig* denotes "read signal" - the incoming data from the host being typically used to signal the nodes to start their work. On completion of work, the nodes can return their results by calling *wtres* which allows each node to return an 8-byte result to the IH. Finally the IH can receive the incoming stream of P results by calling *rdbufIH* and providing it a buffer of length $8P$. The final two routines are used for inter-node communication within the cube, whose nodes were called Elements, hence the ELT. The routine *wtELT* sends an 8-byte packet to the neighboring processor on channel number *chan*, while *rdELT* is used to read an incoming message on channel *chan*.

This system provided a highly efficient representation of the hypercube physical communication structure. Note that only nearest neighbor communication is allowed, corresponding to the actual hardware wiring. One major restriction imposed by CROS is that only SIMD communication patterns are allowed. If one node issues a *wtELT* call, then all nodes must do so simultaneously. Similarly if one node returns a result to the IH, all must do so, even if they have nothing useful to return. Because only such regular communication patterns were accepted, the communication library was referred to as the Crystalline Operating System.

3.3. The Intel iPSC1 Hypercube

The iPSC1 was developed by Intel shortly after the Caltech Hypercube was built. It was very close in design, although differing in several features. The system was based on 80286 nodes, each supported by 512K of memory. Intel used some off the shelf Ethernet chips to implement a fairly fast communication structure. The iPSC1 design allowed up to a 7 dimensional hypercube to be constructed, i.e. 128 nodes. As with the Caltech machine, the system used a host machine, consisting in this case of a Xenix workstation.

3.3.1. Intel iPSC1 Programming: NX1

While the underlying hardware of the Intel iPSC1 was fundamentally similar to the Caltech Hypercube, the software environment developed - known as the NX1 operating system - was fundamentally different. The system was based on the Reactive Kernel [3], also developed at Caltech. We highlight key developments from the Reactive Kernel paradigm as these have all become important aspects of modern message passing systems. All of the key communication routines are shown in Table 4, and their purpose is described here.

Topology Independence: To begin with, the communication software attempts to hide as much as possible the details of the connection topology. Each processor is represented by an integer - the node number - in the range $[0, P-1]$ where P is the number of processors. The host is denoted by a special integer, HOSTID, which is a negative integer such as -1.

Multiprocessing Nodes: In a distinct departure from the Caltech design, each processor of the iPSC1 may have several processes running simultaneously, each distinguished by a process identity number, or *pid,* which is local to that node. Since processes rather than processors are the logical communicating entities in any system, it follows that rather than node to node communication, we are now dealing with process to process communication. Thus a communication operation must specify the destination node *and* pid.

Transparent Access: In another major departure from the Caltech approach, the NX1 system allows each process to communicate with any other process, not just processes on neighboring nodes. Furthermore there is no distinction between neighboring nodes and others - all nodes are represented by a number, and only with considerable difficulty can a programmer even determine whether two nodes are nearest neighbors or not. The communication routines (see Table 4) appear to have a channel parameter *chan,* but this is misleading - the channels used here are simply software channels used to allocate storage for parameters and data needed by the succeeding communications.

MIMD Communication: NX1 communication is true MIMD communication: one, some or all nodes may participate in a communication operation. Certain global patterns, such as global synchronization or global sums, are represented by NX1 operations that require all nodes to participate simultaneously. But the vast majority of NX1 operations require only a sender and a receiver.

Non-Blocking Communication: The blocking routines - *sendw* and *recvw* - do not return until they have completed their operations. Thus *sendw* returns after it has sent a message, and *recvw* returns only after a requested message has been received. Note that this latter situation will lead to a lockout if no sender sends an appropriate message. The non-blocking routines are *send* and *recv.* These return prior to completion of the operation, basically right away. It is necessary to later call *status(chan)* to find out if the requested operation has completed. Until it does complete, the corresponding data buffers are not safe to use.

Asynchronous Communication: It is not even necessary that senders and receivers of data match their communication requests. NX1 provides for communication that may be either synchronous or asynchronous. Asynchronous communication utilizes the *probe* call which a receiver may use to determine if there are appropriate messages waiting. The receiver is free to read (receive) these messages at a later time, or indeed to ignore them.

Typed Messages: A further innovation of NX1 deals with the fact that often one finds a proliferation of messages arriving at a process, some not in the order that the receiver would like to have them in. NX1 requires that each message has a user-assigned *type* - an integer value. Messages are received only by type. The communication calls for receiving messages allow only the specification of a message type, or message type wildcard, but not a source node or process number. As a result, type becomes an important characteristic that permeates all communication and can be used to discipline programming style. In the context of asynchronous communication, where the user may request to read messages out of order, type is invaluable for selecting messages appropriately. While type is a powerful aid to programming complex applications, it is not a restriction in any sense. A user may choose to ignore type altogether of course by assigning the same type number to ALL messages. Similarly a user who wants to receive messages by source node number and pid could achieve this by encoding the source node and pid in the type integer.

Unknown Message Lengths: It frequently happens, especially in the course of asynchronous communication, that a receiver may be unsure about the expected length of incoming messages. The recv and recvw routines both require a buffer *mesg* and its length *len* into which the message will be stored. However after receipt of the message, an output parameter *cnt* records the actual number of bytes received - if *cnt* is greater than *len*, part of the message will be lost.

Non-Busy Waits: Because multiple processes can run on a node, deadlocks or inefficiencies could occur if processes awaiting messages remained active. The *flick* routine was provided to allow control to transfer to other processes at such times.

Perhaps the only place in NX1 where the underlying hardware appears in some way, is that all data packets must have a size of at most 16,384 bytes.

Table 4: iPSC1 NX1 System Calls:

Node Routines:	
chan = copen(pid)	open a virtual channel
send(chan,type,mesg,len,node,pid)	send a type message to pid on node.
recv(ci,type,msg,len,&cnt,&node,&pid)	receive message of type
length = probe(chan,type)	are there messages of type?
status(chan)	is a channel free yet?
flick()	non-busy wait.
sendw(chan,type,mesg,len,node,pid)	blocking send to pid on node.
recvw(ci,type,msg,len,&cnt,&node,&pid)	blocking receive of type
syslog(pid,string)	print a message on the host
Host routines:	
sendmsg(chan,type,mesg,len,node,pid)	blocking send to pid on node.
recvmsg(ci,&type,msg,len,&cnt,&node,&pid)	blocking receive

One peculiarity of NX1 is that the host call *recvmsg* is more restrictive than the node version. The host is not allowed to receive messages by type. Thus effectively the host is forced to receive whatever arrives first. Furthermore only blocking communication calls are supported at the host. These features were fixed in NX2 which provides full symmetry between host and node operations.

3.4. The Intel iPSC2 Computer

3.4.1. Intel iPSC2 Programming: NX2

NX2 is the current Intel operating system for the iPSC/860 and Paragon computers [4,5], see Table 5. The NX2 operating system introduced some small changes to the NX1 system, and added extra capabilities such as interrupt driven communication. Again NX2 represents each processor by an integer node number in the range $[0, P-1]$, but now the host is denoted by the integer P. As in NX1, each processor may have several processes, distinguished by a *pid*, and each *process* may communicate with any other. Communication may be either blocking or non-blocking, synchronous or asynchronous. Each message has a user-assigned *type* and messages are received by type request only.

Message Identifiers: NX2 introduced the concept of a Message Identifier (MID), an integer associated with any incomplete non-blocking operation. Both *send* and *recv* (now renamed *isend* and *irecv*) return an MID. Three routines are then available for later processing of the message: *msgdone(mid)* reports whether the operation has completed; *msgwait(mid)* returns only when the operation has completed; and *msgcancel(mid)* cancels the pending message if it has not already completed. These operations allow for a programming style in which senders communicate data as soon as possible, while receivers postpone receipt until they are fully ready to use data, performing useful computations in the interim. This allows some messages to be overlapped with computation, which in turn leads to the possibility of message passing algorithms that exhibit no efficiency losses due to communication.

Interrupt Driven Communication: NX2 also supports interrupt driven versions of communication routines. This allows a receiver to be interrupted upon receipt of a message at its process, in a fashion similar to handling of signals in UNIX.

Receive Info Calls: On receipt of messages, the source information such as *node, pid, type* and *length* are not returned by the receive calls as happened in NX1. Instead a set of four information routines is provided for that purpose. Calls to these routines return information on the most recently concluded communication.

Wildcard Arguments: NX2 includes a wide assortment of wildcards that may be used in communication routines - wildcards are represented typically by negative node or type integers. For example, writing to node -1 denotes writing to all nodes - i.e. a broadcast. It is also possible to write to complete subcubes using a similar notation. A request to receive a type -1 message is interpreted to allow receipt of any message type whatever, and again there are other conventions that restrict to certain subsets of types. While many of the more specialized node and type conventions are ad hoc, are not portable to other systems, and are potentially dangerous, several such as those we have mentioned, are extremely useful.

Global Operations: NX2 supports a full range of global operations such as synchronizations and associative commutative arithmetic operations. Broadcasts as well as reductions are included. Each of these operations is carefully coded using optimal mappings of binary trees to the architecture.

Configuration Operations: While not discussed previously, each of the hypercube message passing systems supports a set of configuration operations. These may be used to learn the number of nodes in use, the node and pid of the current process, and similar information.

Table 5: iPSC2 Communication Routines

Configuration Routines:	
numnodes()	return *number of nodes*
mynode()	return *node number* of my node
myhost()	return *node number* of my host
setpid()	used by host to set a *pid*
mypid()	return *pid* of host or node
Synchronous Routines:	
csend(type,msg,len,node,pid)	blocking send to *pid* on *node*
cprobe(type)	wait for a message of type *type*
crecv(type,buf,len)	blocking read of type *type*
infocount()	returns actual *length* of last message
infotype()	returns *type* of last message
infonode()	returns *source node* of last message
infopid()	returns *source pid* of last message
Asynchronous Routines:	
isend(type,msg,len,node,pid)	send a *type* message to *pid* on *node*
irecv(type,buf,len)	read message of type *type*
iprobe(type)	is there a message of type *type*
msgwait(mid)	await completion of message with ID *mid*
msgdone(mid)	is message with ID *mid* completed
msgcancel(mid)	cancel message with ID *mid*
flick()	non-busy wait
Global Routines:	
gsync()	globally synchronize all nodes
gopf(x,len,work,f)	global associative operation *f(x,work)*
gdsum(x,len,work)	global sum, elements of *x(len)* independent

3.5. nCUBE Hypercubes

The nCUBE series of MPP systems (nCUBE1 and nCUBE2) are unique in having remained faithful to a single design from the outset. The systems are also the only MIMD hypercubes still being manufactured. nCUBE supports a 13 dimensional hypercube. Each node has 14 built-in bi-directional communication channels - 13 to hypercube neighbors and one being an I/O channel. nCUBE nodes have always been rugged custom-designed scalar processors, and the simplicity of design has led to both very long mean-time to failure and good performance for non-vectorizable or non floating point applications. In the nCUBE2, messages travel at about 2.75MB/sec and message startup time is about 150μs (there is an extra 2μs overhead for each intermediate node traversed, up to a maximum therefore of 26μs)

3.5.1. nCUBE PSE

The nCUBE Parallel Software Environment (PSE) provides a set of communication primitives [6] similar to those of Intel NX. Communication routines use a process id that combines both the processor location and the process number on that processor into a 32-bit integer. Messages carry an integer type, allowed in the range [0,32,767]. The basic communication calls are *nwrite* and *nread.* Here *nwrite* is non-blocking, but *nread* always blocks. However an extra routine *ntest* allows a receiver to check for a suitable message, so that *nread* is typically only called when it is known already that it will return with a message. PSE supports global communication calls, and in addition has support for grid-based computation. In the latter case it is possible to layout data as a multi dimensional distributed array, and PSE provides explicit calls to exchange boundary edges of subarrays as required by many algorithms.

3.6. Cost/Benefit Aspects of Different Message Passing Environments

In order to evaluate the costs associated with the increasing sophistication of the message passing interfaces, it is interesting to compare the messaging performance of the systems discussed above. For this purpose, we will introduce the time to transfer k words between neighboring processors. To a good approximation this may be represented on all systems by a linear function:

$$T(k) = \alpha + \beta k \ .$$

This time should be compared to the time γ required for a multiply on that system. The value of γ characterizes the computational speed of a system, while the ratios α / γ, β / γ characterize the communication efficiency of a system.

Table 6: Comparison of Communication Efficiencies						
Machine	**MPI**	$\alpha\ \mu s$	$\beta\ \mu s$	$\gamma\ \mu s$	α / γ	β / γ
Caltech Hypercube	CROS	92	5.0	30.0	3	.17
Intel iPSC1	NX1	5,500	2.8	30.0	183	.09
Intel iPSC2	NX2	500	2.0.	10.0	50	.20
Intel iPSC/860	NX2	470	0.77.	0.025	18,800	30.80
Intel Paragon	NX2	60	0.05.	0.015	4,000	3.33
KSR1	TCGMSG	160.	0.26.	0.025	6,400	10.40
nCUBE2	Vertex	210.	0.58.	0.5	420	1.16

Table 6 records measurements we have made for several representative systems. It is clear from Table 6 that the Caltech Hypercube is by far the best balanced of these systems. It had the lowest latency and the second fastest transfer rate of any system, relative to floating point performance. It is also clear that current machines are doing much more poorly than the early systems, a reflection of the fact that processor speedups have far outpaced communication speedups. The only mitigating factor is that the amount of memory per node has also been increasing rapidly, a factor that tends to diminish the relative amount of communication needed in most computations (perimeter to area ratio).

4. Other Message Passing Systems

There is insufficient space to cover all of the interesting message passing systems developed todate. However we will use this opportunity to note here several of these because of particularly interesting features.

4.1. IBM EUI

The IBM External User Interface (EUI) is the message passing system for the IBM SP MPP computer series [7]. EUI is also designed to run on a loosely coupled set of workstations such as the RS/6000. EUI supports both blocking and non-blocking I/O, and the usual forms of collective communication. The number of tasks in a job is fixed.

An interesting feature is support for *groups* of tasks - a group being an ordered subset of the tasks in a user's partition, which is known as the *allgrp*. Groups can be created dynamically either by specifying a list of member tasks to the *mp_group* routine, or by partitioning another group into several disjoint parts using *mp_partition*. Collective communication routines work on groups rather than the full partition. Typical applications of groups include creating row and column groups in grid based applications, and allowing collective communications to be performed on single rows or columns. The collective communication routines include barrier, shift, broadcast, scatter, gather, reduction, and parallel prefix. EUI supports both blocking and non-blocking barriers.

EUI messages carry a user defined type and messages may be received by source and/or type, with wildcards allowed. The facilities available are very similar to those seen in Intel NX2 so we do not discuss these further. EUI will in the future support a concept of *channel* similar to that on the Caltech Hypercube. A channel connects a specific pair of processes and is optimized to provide very efficient communication between those processes. Separate channel communication operations use the channel number rather than process information for send and receive operations. The real payoff for channel communication will be for repeated communication patterns - channel creation will be expensive, but once created a channel can preserve buffers that will be used in later repetitive communication on that channel.

4.2. Meiko CS System

The Meiko Computing Surface (CS) operating system was developed for the Meiko line of transputer, i860 and Sparc based MPP systems. These systems are typical message systems, and the most recent product, the CS-2, is among the most powerful MPP systems developed todate.

CS communication, outlined in Table 7, differs dramatically from other systems in that it is based on a communications name space [8]. Processes create virtual communication objects called *transports*. Transports become useful only when registered with a global name server - implemented by the function *csnregname*. Once a transport has a registered name, other processors that know of the name can look up the name and, if successful, are returned a *netid* to be used in further communications. As in the Intel case, blocking, non-blocking, synchronous and asynchronous communications are all supported. All of these communications are performed to a previously opened transport, which in turn is expected to be connected to other processes.

Table 7: Meiko CS Communication Routines

Configuration Routines:	
csgetinfo(numnodes,mynode,mypid)	return *numnodes, mynode, mypid*
Transport	communication object, access via global *netid*
csnopen(index,tr)	create a new transport *tr* with ID *index*
csnclose(tr)	close out a transport
csnregname(tr,name)	establish global name for transport *tr*
csnderegname(tr)	remove any global name for transport *tr*
csnlookupname(netid,name,block)	return *netid* of transport called *name*
csngetid(tr)	return *netid* of transport *tr*
csngetnode(netid)	return *node* given a *netid*
csngetnet(netid)	return *network number* given a *netid*
csngettransport(netid)	return *transport number* given a *netid*
Synchronous Routines:	
csntx(tr,netid,flag,msg,len)	blocking send to transport *netid*
csnrx(tr,netid,buf,len)	blocking read - returns *netid*
Asynchronous Routines:	
csntxb(tr,netid,flag,msg,len,mid)	non-blocking send to transport *netid*
csnrxnb(tr,buf,len,mid)	non-blocking read - assign *mid* as an ID
csntest(t,,,,mid,.)	test if a non-blocking message *mid* is complete
csncancel(tr,,,mid)	cancel message *mid*

4.3. SUPRENUM

The German SUPRENUM computer is a 256-node system based on a 2-level hierarchical design. At the bottom level, 16 processors connected by a bus form a cluster. A full system consists of 16 clusters interconnected by a grid of horizontal and vertical busses. Despite this complexity, SUPRENUM communication concepts are quite similar to other systems. An unusual aspect is that SUPRENUM supports extensions to Fortran for task control and to assist in communication operations. For example, SUPRENUM uses language extensions similar to Fortran I/O lists to call communication functions. This allows compilers to optimize communication - for example by concatenating multiple arrays into a single one, which can greatly reduce latency overheads. The disadvantage is that SUPRENUM message passing programs are then not completely portable.

SUPRENUM is unique in providing a sophisticated high-level library interface to the communication system [9]. The library supports a range of 2D and 3D grid-oriented operations that largely shield a numerical user from dealing with the communication system directly. In addition to providing powerful programming tools, such systems deliver the possibility of substantial program portability across architectures that support the common set of primitives. In fact these grid communication routines have been ported to other systems as well.

4.4. Thinking Machines CMMD

The Thinking Machines Corporation (TMC) Connection Machine CM-5 supports two distinct programming models. The simplest and most elegant is CMF - Connection Machine Fortran - which provides a Fortran 90 style SIMD programming system [10]. This is clearly the method of choice for those applications that are representable as SIMD processes in an efficient way. For truly MIMD applications it is necessary to write message passing programs, which are similar in style to Intel NX programs. The message passing system for the CM-5 is known as CMMD [11,12]. A further complication is caused by the complexity of the nodes each of which contains a Sparc scalar processor and four vector nodes. To utilize the latter, node programs must be written in a vector language, either in a flavor of Fortran 90 or by explicitly calling vector routines from Fortran or C. In addition to standard send/recv communication, CMMD supports a second channel-based communication mode. In this case, once a channel has been allocated, further repeated communication calls on the same channel have very low overhead. As an example, message latency is about 35μs for send/recv communication, but only 14μs for channel communication between nearest neighbors. Typical long-message data rates are of order 20 MB/sec.

CMMD supports the full range of messaging capabilities familiar from other systems - blocking, non-blocking, synchronous and asynchronous sends and receives, probe operators and collective communication routines. The hardware has separate communication channels for point to point and for collective communication. Interrupt driven communication is also available, at an increased cost of about 20μs per interrupt.

TMC has been especially effective in reducing messaging latency. The key has to been to implement the CMMD system in terms of a simple low overhead communication mechanism called the Active Message Layer (AML) [12,13]. An AML message is a message consisting of a function pointer (i.e. an address in the program address space), along with a set of arguments. Upon receipt it causes the specified function to be activated with the supplied arguments. Because the CM5 supports only the SPMD programming paradigm, all nodes run the same text program, and so function pointers are exchangeable between nodes. A special case of an active message is where the function simply deposits the argument data to a specified memory location - known as a Data Deposit. Data sent in this ways incurs a latency of only 8µs, and travels at bandwidths as high as 25MB/sec. Injection of an Active Message into the network requires only about 1µs.

4.5. Virtual Shared Memory

Several MPP systems such as the Myrias SPS-2, the Evans and Sutherland ES-1, and the Kendall Square KSR1 and KSR2, implement virtual shared memory (VSM) on a distributed memory system [14]. In these systems the user no longer needs to employ a message passing library for communication. Instead, all communication is handled by direct virtual memory loads and stores. This is an enormous convenience to the user, and greatly reduces the effort needed to port a program from a sequential machine. There are significant overheads involved in VSM and in addition one always has to remember that the underlying architecture is a distributed memory system. Thus certain easily programmed VSM algorithms may result in hopelessly inefficient code. Experience with the KSR1 indicates that VSM can do almost as well as message passing for regular grid applications, although it does not do as well for highly irregular problems.

5. Portability Platforms and Heterogeneous Environments

Portability platforms are message passing environments developed by non-vendors, with a view towards using the same environment on several different MPP architectures. Heterogeneous computing refers to all situations where two or more different computers cooperate on a task. Heterogeneous computing has become increasingly important due to the need to tie together disparate resources. Because portability platforms run on several architectures, they automatically tend to be well suited to heterogeneous computing as well. For this reason we treat portability platforms and heterogeneous computing environments together in this section.

5.1. RPROC

The RPROC system, developed by McBryan in 1982 [15,16], anticipated many features of current heterogeneous message passing systems - particularly heterogeneous use, active messages, mixed data packets and automatic data conversion. The system was designed to interconnect computers which had different operating systems, a common problem at that time. Furthermore because important systems (such as CRAY) did not support TCP/IP, it was designed to assume only the most minimal communication facilities.

The only assumption in RPROC was that an unreliable file transfer mechanism was available between interconnected systems. All RPROC messages were delivered as a pair of transferred files. The first file delivered the message, while the second file, called a signal file, signaled that the message file was complete. Thus even if the file transfer failed half-way through, the receiving program would not assume the message was complete until it saw the signal file. These file transfer features were invisible to the user and indeed could be replaced by a different message transfer protocol (such as TCP/IP) if desired. Typical methods used for file transfer included rcp, ftp, and decnet.

RPROC messages were packed in a fashion similar to that proposed for MPI. Packing of messages was essential because of the overhead of file access. Messages consisted of a header followed by one or more arrays of data, each preceded by a count. The header provided information on the source of data, and in particular the machine type. Message creation, message packing and message transmission were separate logical operations. Having created a message header, it could be packed with many data items in succession by calls to user-defined packing routines. Having completed a buffer, the message was then sent to the destination. On the receiving end, a matching receive routine located and accepted the message, and called a user supplied routine to decode it. Several pairs of compatible send/receive routines were available. For example, in some cases the send did nothing, whereas the receive fetched the message.

For packing messages, a set of routines was supplied that read and write arrays of each of the basic datatypes. These routines utilized the header data to determine if data conversion was required, and if so each data item was automatically converted to the correct form for the local machine while being read. The conversion was not always done on the receiving machine, but instead the decision on whether to convert on send or receive was based on machine performance. For example, when communicating from a SUN or VAX to a CRAY, the conversions were always performed on the CRAY, using highly efficient vectorized code, see [16] for examples. As a result data conversion was a negligible part of communication cost to CRAY or other fast machines. For debugging purposes, an ascii format (i.e. formatted) data transfer was available in addition to the binary transfer mode.

RPROC messages were actually active messages. Each message specified in its header a routine to be executed on the receiving process. On receipt, the routine was called with the associated incoming data as arguments. RPROC was asynchronous and could be either blocking or non-blocking, allowing extensive opportunities for overlap of communication and computation.

RPROC ran on over 10 architectures. It was used extensively at many sites to build heterogeneous applications, and also as an effective way to harness otherwise inaccessible resources. For example it was used at several sites to allow FPS Array Processors attached to VAX or IBM computers to be accessed from UNIX machines. It was also used to allow a single SUN workstation to harness three CRAY computers at Los Alamos National Laboratory. A 200,000 line C program running on the SUN performed all its compute-intensive operations in parallel on the CRAY machines, at a time when CRAY C compilers did not yet exist. Finally it was used in an early (1983) long-distance experiment in which computers in New York and Los Alamos cooperated on the same CFD computation.

5.2. P4 Macros

The P4 system developed at Argonne National Laboratory [17,18] was the earliest attempt to develop a portability platform. The original system, known as MonMacs, was a set of M4 preprocessor monitor macros for the HEP shared memory MPP. Later developments added support for other shared memory systems, added message passing and finally support for mixed systems such as loosely coupled shared memory clusters. The system has spun off several other well-known systems as by products, including the PARMACS system from GMD, an object-oriented C++ version for the Sequent and the Argonne TCGMSG message passing system.

P4 is unique in its comprehensive treatment of both shared and distributed memory systems. It is also extremely efficient, because it avoids introducing function call overheads through the use of macros (a critical issue on shared memory systems). P4 provides built-in facilities for process generation using a "procgroup" file. One notable restriction is that P4 message passing is entirely blocking.

5.3. PARMACS

PARMACS is a macro based message passing system developed initially from the P4 macros [19]. Execution starts with a single host process which can spawn node processes using the *remote_create()* macro which reads a machine dependent input file that specifies the programs to run on nodes, and the pid to assign to each. PARMACS supports both synchronous and asynchronous send routines. In the synchronous case, the sender is suspended until the receiver has completely received the message. Messages carry an integer type parameter and may be received by process id of the sender and/or type. Standard forms of message probe and collective communication are also supported.

PARMACS supports heterogeneous computing and the *msg_format* macro is provided for use before a send to specify the contents of a buffer. This information is used on the receiving end to automatically convert the data as needed. A buffer can contain any number of arrays of arbitrary lengths of seven different data types ranging from 2-byte integers to double precision complex.

One of the most interesting features of PARMACS is the extensive support for application topologies. The *torus* macro maps rings and process grids of 2 or 3 dimensions, while the *graph* macro maps arbitrary process configurations defined by a graph. The *torus* macro actually creates the input file for the *remote_create* macro. This allows the topological mapping to be optimized to the hardware. Special macros are then provided so that processes can locate their position in grid or graph coordinates. Macros are also supplied to map between tori of different sizes and dimensions, the latter case allowing one to switch to planes in a 3D torus for example. PARMACS has strongly influenced the topological aspects provided for in MPI.

5.4. EXPRESS

The EXPRESS system grew directly from the Crystalline Operating System developed at Caltech [2,20]. In fact EXPRESS is a product of the Parasoft company which is a Caltech spin-off. Initially Parasoft was oriented to implementing Express on a range of architectures and attempting to get highest performance from each. It was quite successful, often reducing message latency by factors of four or more.

More recently, Parasoft has emphasized useability and attempted to hide many of the details inherent in EXPRESS. This led to the development of mapping and communication libraries for rings, grids, tori and so on, each optimized to a specific hardware platform. EXPRESS has also begun to tackle the problem of performing parallel I/O as well as dynamic load balancing [20]. These latter two are issues that are almost universally ignored by message passing systems, including MPI.

5.5. PVM

PVM - Parallel Virtual Machine - represents an extremely successful example of a message passing environment for heterogeneous computing. PVM, developed at Oak Ridge National Laboratory, uses a simple send/receive library to control the interaction of an arbitrary number of possibly remote computers [21]. Early versions of PVM used TCP/IP sockets to implement all communication, but recent versions also provide more efficient implementations for use within an MPP. Conseqwuently PVM has evolved to become a portability platform. In fact it is reasonable to guess that PVM is the most widely used of all message passing environments because of its generality and its applicability to networks of workstations. Because the PVM library routines are so similar to others we have seen above, we refer here to the detailed paper describing the system [21].

PVM differs from most message passing systems by supporting dynamic creation of processes. Because of the intended heterogeneity, PVM uses strongly typed constructs for buffering. The system is remarkably small, ignoring features such as collective communication or process topologies found in other systems. However this simplicity of design is actually a key to the current success of PVM. PVM supplies routines to register a collection of cooperating processes, to initiate and terminate tasks, to synchronize with other PVM tasks and to obtain configuration information. Synchronization is accomplished using either barriers or signals.

Blocking and non-blocking asynchronous sends and a non-blocking receive are supported between tasks. Messages can be selected by source and/or tag. Dynamic process groups are supported, and broadcasts and barriers use the group name as a qualifier. A form of context switching is supported that allows libraries to safely interact with applications without danger of message interference. PVM v3 allows native MPP message passing calls to be used to implement PVM communication. PVM latency is typically of order milliseconds, but with the new native messaging implementations, substantial improvement can be expected.

5.6. Zipcode

Zipcode is another portability platform that has been used effectively as an experimental laboratory for message passing [22]. During its development many new concepts have been tested, and the result has been a considerable influence on MPI design. Zipcode made a special effort to tackle the problem of providing a development environment for parallel libraries. Standard message passing systems are not adequate in this regard because of the danger of confusion between internal library messages and user messages. For example there is no mechanism to restrict message types that a user can employ. Indeed by using a wildcard a user can potentially intercept any message in most systems, effectively introducing errors or deadlock in an associated library.

Zipcode overcame the difficulties of other systems by providing a safe communication space for libraries, and by providing collective operations to operate on only a subset of all processes. Zipcode introduced three concepts to provide for these features. Process Groups define ordered sets of processes, and within process groups, process names are determined relative to rank in the group. Similarly groups are used to define collective communications. Zipcode groups are static. Contexts provide the ability to separate universes of messages. Contexts can be thought of as a second system-supplied message tag. Zipcode does not allow communication between different contexts. Mailers bind process groups and contexts together to form a safe communication space within a group. Zipcode also provides for concepts of process group topology by allowing for different forms of message selection - for example, mail can be selected by source using grid coordinates in 1, 2 or 3 dimensional grids. Zipcode influence on the design of MPI is particularly apparent in the area of process groups and contexts.

5.7. LINDA

Linda [23] is a communication system that is in a different category from the other message passing systems we have discussed above. Linda is an associative, virtual shared memory system. The associative memory is called Tuple Space. Linda's operations applied to Tuple Space provide the process management, synchronization, and communication functionality required for MIMD programming. Data objects are known as tuples and Linda basically provides four operations that operate on tuples: - *out, eval, in, rd..* Here *out* generates a tuple serially, *eval* generates a tuple asynchronously (used to create tuples in parallel), *rd* reads a tuple and *in* withdraws a tuple destructively. The in operator provides a template for matching tuples in tuple space. If a match occurs then the tuple is withdrawn, effecting communication.

The most interesting versions of Linda - C-Linda and Fortran-Linda - are actually language extensions to standard C and Fortran. These allow shared memory programs to be written in a very modest extension of a standard sequential language, yet run on systems that actually have only a distributed memory. All of the required message passing is then generated automatically by the Linda translator.

Linda is best suited to those problems that involve many processes per node, ill-defined communication, extensive asynchrony and global communication. Surprisingly, recent work at Yale [24] shows that Linda can be fairly competitive with standard message passing systems even for typical numerical applications.

5.8. MPI

MPI is a new message passing standard that has evolved from a series of meetings held between November 1992 and January 1994 by the MPI Committee [25]. The committee consists of members from about 40 institutions and includes almost all of the MPP vendors, as well as universities and government laboratories worldwide that are involved in parallel computing.

The MPI standard is intended to be comprehensive enough to encompass the major features of all of the vendor systems, while at the same time being efficient enough to allow vendors to move to native MPI implementations on their platform as an eventual replacement for their current systems. MPI draws on features represented in all of the other systems described above. However it is not necessarily inclusive - some features have been omitted where they were felt to be inefficient or inappropriate.

MPI generalizes the concept of message buffer by allowing elementary data types, contiguous arrays, strided blocks, indexed arrays of blocks and general structures. Datatypes are constructed recursively. MPI generalizes the tag or type field by introducing a second context field which is system allocated (and for which wildcards are not allowed) to define families of messages. MPI provides groups of processes, as well as group management routines. All communication occurs within groups. Groups and contexts are combined in the concept of a communicator. The source/destination in send/receive operations is the rank in the communication group. Finally MPI supports application-oriented process topologies, and has built-in support for grids and graphs. MPI provides no mechanisms for process management, remote memory transfers, active messages, threads or virtual shared memory. However MPI has tried to remain compatible, for example by being thread safe.

References

1. M. Flynn, "*Some Computer Organizations and Their Effectiveness*", IEEE Transaction on Computer C-21 pp 948-60.

2. A. Kolawa, B. Zimmerman, *"CrOS III Manual"*, Caltech C3P-253, 1986.

3. J. Seizovic, *"The Reactive Kernel"*, Caltech CS-TR-88-10, Oct. 1988.

4. P. Pierce, *"The NX/2 Operating System"*, Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, January 1988.

5. P. Pierce, *"The NX Message Passing Interface"*, later in this volume.

6. M. Schmidt-Voigt, *"Efficient Parallel Communication with the nCUBE 2S Processor"*, later in this volume.

7. V. Bala, J. Bruck, R. Bryant, R. Cypher, P. de Jong, P. Elustondo, D. Frye, A. Ho, C.-T. Ho, G. Irwin, S. Kipnis, R. Lawrence, M. Snir, *"The IBM External User Interface for Scalable Parallel Systems"*, later in this volume.

8. E. Barton, J. Cownie, M. McLaren, *"Message Passing on the Meiko CS-2"*, later in this proceedings.

9. K. Solchenbach, U. Trottenberg, *"SUPRENUM - System Essentials and Grid Applications"*, Parallel Computing, 7, North Holland, 1988.

10. Thinking Machines Corporation, *Connection Machine CM-5 Technical Summary*. Nov 1993.

11. Thinking Machines Corporation, *CMMD Reference Manual V 3.0*, May 1993.

12. L. Tucker and A. Mainwaring, *"CMMD: Active Messages on the CM-5"*, later in this volume.

13. T. Eicken, D. Culler, S. Goldstein and K. Schauser *"Active messages; A mechanism for Integrated Communication and Computation"* In Proceedings of the Nineteenth International Symposium on Computer Architecture. ACM Press, May 1992.

14. O. McBryan, *"Software Issues at the User Interface,"* in Frontiers of Supercomputing II: A National Reassessment, ed. W.L. Thompson, University of Colorado CS Dept. Tech Report CU-CS-527-91 and MIT Press, 1994, to appear.

15. O. McBryan, Los Alamos National Laboratory Annual Report, 1983.

16. O. McBryan, *"Using Supercomputers as Attached Processors"*, in New Computing Environments: Microcomputers in Large-Scale Scientific Computing, ed. A. Wouk, SIAM, Philadelphia, 1987.

17. R. Butler and E. Lusk, *"User's Guide to the P4 Programming System"*, Technical Report TM-ANL-92/17, Argonne National Laboratory, 1992.

18. R.M. Butler, E.L. Lusk, *"Monitors, Messages, and Clusters: the p4 Parallel Programming System"*, later in this volume.

19. R. Calkin, R. Hempel, H.-C. Hoppe, P. Wypior, *"Portable Programming with the PARMACS Message-Passing Library"*, later in this volume.

20. J. Flower, A. Kolawa, *"Express is not just a message passing system"*, later in this volume.

21. V.S. Sunderam, G.A. Geist, J. Dongarra, R. Manchek, *"The PVM Concurrent Computing System: Evolution, Experiences and Trends"*, later in this volume.

22. A. Skjellum, S.G. Smith, N.E. Doss, A.P. Leung, M. Morari, *"The Design and Evolution of Zipcode"*, later in this volume.

23. N. Carriero, D. Gelertner, T. Mattson, A. Sherman, *"The Linda alternative to message-passing systems"*, later in this volume.

24. C. Douglas, T. Mattson, M. Schultz, *"Parallel Programming Systems for Workstation Clusters"*, Yale CS Dept Research Report YALEU/DCS/TR-975, Aug 1993.

25. D. Walker, *"The design of a standard message passing interface for distributed memory concurrent computers"*, later in this volume.