Chapter in Wiley Encyclopedia of Electrical and Electronics Engineering

Jack J. Dongarra, Graham E. Fagg, Rolf Hempel, and David W. Walker

October 13, 1999

Keywords: Parallel computing, message passing libraries, MPI, PVM, active messages

1 Introduction

A parallel computer is a set of processors, connected via some interconnection network, working together to solve a computational problem. The major hardware features of a parallel computer are the processing units themselves and the interconnection network that ties them together. The effective use of this hardware requires a software infrastructure that provides an appropriate level of abstraction and functionality, and that allows the applications running on the processors to use the interconnection network.

In this document, we review the development of the software infrastructure for parallel computers, focusing in particular on the message passing systems that are commonly used for communicating processor data over the interconnection network. We shall start with a brief introduction to parallel programming and message passing, followed by a detailed examination of the most influential message passing systems. The chapter will conclude with an examination of MPI, the standard for message passing software.

2 Parallel Programming Model

The von Neumann machine model assumes a processor capable of executing sequences of instructions. In addition to various arithmetic operations, an instruction can specify the address of a datum to be read or written in memory and/or the address of the next instruction to be executed. While it is possible to program a computer in terms of this basic model using machine language, this method is for most purposes prohibitively complex, as millions of memory locations must be tracked and the execution of thousands of machine instructions must be organized. In addition, this approach is not portable between different computers. Hence, modular design techniques are applied, whereby complex programs are constructed from simple components, and components are structured in terms of higher-level abstractions such as data structures, iterative loops, and procedures. Abstractions, such as procedures, make the exploitation of modularity easier by allowing objects to be manipulated without concern for their internal structure. High-level languages, such as Fortran, Pascal, C, and Ada, allow designs expressed in terms of these abstractions to be translated automatically into executable code.

Parallel programming introduces additional sources of software complexity most of which arise from providing multiple processors with access to memory. When more than one processor is able to write concurrently to the same memory location a race condition may arise, resulting in a non-deterministic program. This can result in a program being incorrect, and requires access to memory to be synchronized. An incorrect program can also arise when one processor attempts to read a memory location before its value has been set by another processor. Again, program correctness requires that access to memory be synchronized.

Ideally a parallel application should keep the processors of a parallel computer running all the time. In practice inefficiencies arise that degrade performance. Synchronization between processors in a parallel program leads to inefficiency as some processors may need to wait to access memory. Also, in parallel computers with a deep memory hierarchy (i.e., several layers of memory with different access times) the time taken to access remote memory may reduce the performance of an application. Applications that contain little inherent parallelism, or that are highly inhomogeneous resulting in an unequal distribution of work among the processors, will also perform poorly on a parallel computer. To be efficient parallel software seeks to exploit a high degree of concurrency, while at the same time achieving good load balance, avoiding unnecessary synchronization, and making optimal use of hierarchical memory by keeping accesses local. Parallel software that achieves these aims will be efficient on large numbers of processors, and is said to be highly scalable.

2.1 Processes and Communication

We consider next the question of which abstractions are appropriate and useful in a parallel programming model. Clearly, mechanisms are needed that allow explicit discussion of concurrency and locality and that facilitate development of scalable and modular programs. Also needed are abstractions that are simple to work with and that match the architectural model.

A computation consists of a set of processes that may communicate data by sending and receiving messages. This message passing model of parallel computation is most commonly used on distributed memory parallel computers (or multicomputers). In this model, a process encapsulates a program and local memory. It is important to realize the distinction between a process (an abstract unit of computation) and a processor (a piece of physical hardware). In addition to reading to and writing from local memory, a process can send and receive messages by making calls to a library of message passing routines. The coordinated exchange of messages has the effect of synchronizing processes. This can be achieved by the synchronous exchange of messages in which the sending operation does not terminate until the receive operation has begun. A different form of synchronization occurs when a message is sent asynchronously but the receiving process must wait (or "block") until the data arrives.

Processes can be mapped to physical processors in various ways; the mapping employed does not affect the semantics of a program. In particular, multiple processes may be mapped to a single processor. The message passing model provides a mechanism for talking about locality; data contained in the local memory of a process are "close" and other data are "remote."

We now examine some other properties of the message passing programming model: performance, mapping independence, and modularity.

2.1.1 Performance.

Sequential programming abstractions such as procedures and data structures are effective because they can be mapped simply and efficiently to the von Neumann computer. Processes and messages have a similarly direct mapping to the multicomputer. A process represents a piece of code that can be executed sequentially, on a single processor. Two processes need to communicate if a data dependency exists between them, i.e., if the computation of one process depends on data held in the local memory of the other process. If the two communicating processes are mapped to different processors interprocessor communication is required. If they are mapped to the same processor, some more efficient mechanism can be used, such as direct sharing of memory.

To achieve good performance for a parallel program the computation must be equally shared between the processes, and the need for communication between processes must be minimized by preserving good locality of reference. Performance can also be enhanced by overlapping communication with computation through the use of non-blocking communication. If the distribution of work over the computational domain is fixed throughout the computation then work can be assigned to processes statically at the start of the program. If the work load changes dynamically during computation some processes will have more work to do than others. In such cases it may be necessary to perform dynamic load balancing to re-distribute the work evenly among the processes.

2.1.2 Mapping Independence.

The computation and communication performed by processes is independent of process location, thus the result of a program does not depend on where the processes execute. Hence, algorithms can be designed and implemented without concern for the number of processors on which they will execute. In fact, algorithms are frequently designed that create many more processes than processors. This is a straightforward way of achieving scalability. As the number of processors increases, the number of processes per processor is reduced, but the algorithm itself need not be modified. The creation of more processes than processors can also serve to mask communication delays by providing other computation that can be performed simultaneously with communication to access remote data.

2.1.3 Modularity.

In modular program design, various components of a program are developed separately as independent modules and then combined to obtain a complete program. Interactions between modules are restricted to well-defined interfaces. Hence, module implementations can be changed without modifying other components, and the properties of a program can be determined from the specifications for its modules and the code that connects these modules together. When successfully applied, modular design reduces program complexity and facilitates code reuse.

2.2 Other Programming Models

2.2.1 Shared Memory

In the shared-memory programming model, processes share a common address space, which they read and write asynchronously. Various mechanisms such as locks and semaphores may be used to coordinate concurrent accesses to the shared memory, thus avoiding implicit race conditions. An advantage of this model from the programmer's point of view is that the notion of data "ownership" is lacking, and hence there is no need to specify explicitly the communication of data from producers to consumers. Although extra mechanisms known as monitors may be required to handle changes in temporary control of data spaces, this model can simplify program development. However, understanding and managing locality can be more difficult than on distributed memory systems. It can also be more difficult to write deterministic programs.

Shared-memory computers can be relatively easy to program, but it is difficult for applications to maintain efficiency when scaling up to numbers of processors in excess of a hundred or more, without the use of good tuning tools. On the other hand, distributed-memory computers which do require explicit passing of messages may have an advantage over shared-memory machines when it comes to a high degree of parallelism such as in applications that require the coordination of thousands of processes.

2.2.2 Data Parallelism

Another commonly used parallel programming model, data parallelism, calls for exploitation of the concurrency that derives from the application of the same operation to multiple elements of a data structure, for example, "add 2 to all elements of this array," or "increase the salary of all employees with 5 years service." A data-parallel program consists of a sequence of such operations. As each operation on each data element can be thought of as an independent task, the natural granularity of a data-parallel computation is small, and the "locality" of the data has to be well known and exploited to achieve good performance. Hence, data-parallel compilers often require the programmer to provide information about how data is to be distributed over processors, in other words, how data is to be partitioned among processes. The compiler can then translate the data-parallel program into a Single Program Multiple Data (SPMD) formulation, thereby generating communication code automatically as in the case of High Performance Fortran (HPF) and Fortran 90. It should be noted that many of the earlier parallel computer systems were data-parallel or 'array' based in nature and that the mapping of data and locality were exactly known and specified by the user at compile time, i.e. scaler, array, matrix, or tiled data mappings.

3 Message passing

The most basic programming paradigm assumes a single sequential process. The programmer has a simplified view of the target machine as a single processor, that can access a certain amount of memory, and writes a single program to run on that processor. The paradigm may in fact be implemented in various ways, perhaps in a time-sharing environment where other processes share the processor and memory. The programmer, however, wants to remain above such implementation-dependent details, in the sense that the program or the underlying algorithm could, in principle, be ported to any sequential computer.

The programming model of communicating sequential processes [22], developed in the 1970s, is the basis of the message passing paradigm for parallel computing. Several instances of the sequential paradigm are considered together. That is, the programmer imagines several processes, each with its own memory space, and writes a program to be executed by each process. Since the process-local computations contribute to the solution of a global problem, at certain points during execution the processes have to exchange data. This data transfer, also called message passing, enables a process to use previous results of other processes, and thus to cooperate with them in the global problem solution.

Message passing is probably the most widely used parallel programming model today. The main point of this paradigm is that the processes exchange data only by sending messages to each other. Thus the concept of a shared memory space or of processes directly accessing each other's memory is outside the scope of the paradigm. Some mixed model systems do exist, such as Linda[4], which is a shared space that is accessed using operations similar to message passing.

Each process in a message passing application is identified by a unique name which is used to identify the sender and receiver of messages. A message consists of the data being sent, also called the message payload, and any necessary system information required to route it to its destination. This data is sometimes referred to as the header as it is often transferred in the beginning of a message, or the message envelope, in analogy to the use of envelope information in delivering a letter. The message passing system has no interest in the contents of the message payload, it is only concerned with moving it. In general, the following information must be given to the message passing system to specify the message transfer.

- Which process is sending the message?
- Where is the data in the sending process?
- What kind of data is being sent?
- How much data is there?
- Which process(es) is/are receiving the message?
- Where should the data be stored in the receiving process?
- How much data is the receiving process prepared to accept?
- How is the data differentiated from other data sent between the same processes, i.e. is it tagged?
- Is this information of a higher priority than other data and thus can be read out of order?

In general, the sending and receiving processes will cooperate in providing this information through some kind of message passing interface, the details of which can vary from system to system. Technically, this interface in most cases is realized by calls to library functions, which, for example, send or receive a message, or broadcast some data to a whole group of processes. Some of the envelope information provided by the sending processor may be attached to the message as it travels through the system, and is made available to the receiving process, either by being returned as arguments in a receive function call or by additional query function calls.

The message passing model does not preclude the dynamic creation of processes, the execution of multiple processes per processor, or the execution of different programs by different processes. However, in practice most message passing systems create a fixed number of identical processes at program startup and do not allow processes to be created or destroyed during program execution. These systems are said to implement a single program multiple data (SPMD) programming model[12] because each process executes the same program but operates on different data. The SPMD model is sufficient for a wide range of parallel programming problems but does hinder some parallel algorithm developments.

Message passing is a programming paradigm used widely on parallel computers, especially scalable parallel computers with distributed memory, and on Networks of Workstations (NOWs). Since it provides the most explicit way of programming a parallel computer with physically distributed memory, the application programmer can limit the communication traffic on the interconnect system to the minimum required by the underlying parallel algorithm. The resulting application performance is, therefore, close to the theoretical optimum for a wide range of hardware designs and configurations. The superior efficiency of this very basic parallel programming model has been the main reason for its success.

The message passing programming paradigm is not limited to hardware with distributed memory. It has been implemented on many parallel machines with shared physical memory, often with very good communication performance. Although on those systems applications can also be programmed efficiently by directly using the shared memory, the advantage of the message passing paradigm is that it guarantees near-optimal performance both on shared and distributed memory systems, thus providing the highest level of portability to application programs.

Since the days of the first parallel computers, many message passing programming interfaces have been developed. In particular, most hardware vendors developed their own variant, thus limiting the portability of application programs. More recently, several public-domain systems have demonstrated that a message passing system with a uniform programming interface can be efficiently implemented.

4 Message passing research and experimental systems

Experiments with message passing programming started in the early 1980s and by the end of the decade many programming interfaces had been developed and used by application programmers. The reason for this diversity was that, in the absence of a standard, each parallel computer hardware vendor defined their own interface. On the one hand, this provided the basis for experiments with many different approaches, but on the other hand it inhibited application program portability between different machines.

Several research groups solved the lack of portability by defining vendor-independent application programming interfaces (in the following called *portability APIs*) and by implementing them on various machines. Some of the more widely used examples are Express [28], P4 [1, 2], PARMACS [3, 20, 21], and PVM [34]. A program using such an API was portable between computers. The remaining problem, however, was the diversity of portability APIs, making applications and library software incompatible if they used different APIs. In section 4.2 we will describe some portability APIs in more detail.

Experience gained with portability APIs was very important in the Message Passing Interface (MPI) standardization process. In contrast to the vendor-specific libraries, their design could only be successful if they provided sufficient abstraction from hardware details of existing machines. Also, wide-spread use was only possible if they supported the requirements of a wide range of application areas. Portability APIs quickly became popular among application programmers. The main reason was the independence from hardware platforms, which was of particular importance since early parallel machines were often produced by small and short-lived companies.

The availability of the same message passing API on NOWs and high-performance parallel systems also allowed a new means to develop software for expensive and thus not widely available parallel systems. Users could develop their applications on NOWs which they usually owned or had easy access to, and when they had a proven application, they could move the code to the larger machines with little or no modification. This allowed users at smaller institutions to develop applications that could be run first time at the newly appearing supercomputer centers without wasting limited machine access time on development.

4.1 History of vendor-specific message passing systems

How the facilities and functionality of modern message passing systems evolved has been influenced by a vast number of research projects and commercial implementations by vendors of MPP machines. We will briefly discuss some of these systems in terms of which features (we now think of as standard) they introduced and which they omitted.

Since the introduction of the first distributed memory MPP, the Caltech Hypercube, the design of the computing nodes and their interconnection structure has changed many times, often repeating in the light of advances in physical hardware technology. This has directly affected the scope of message passing in allowing more flexibility in terms of addressing, types of messages that can be passed and the level of "services" available upon each node, such as dynamic processes, debugging support, parallel IO/disk operations etc.

4.1.1 Caltech Hypercube and the CROS operating environment

The Caltech Hypercube (circa 1984) was a d-dimensional hypercube-structured system with a computational node at the end of each vertex, and a single host to control the machine (known as an Intermediate Host). The system was programmed in either C or Fortran77, and communication was based on a subroutine library known as the Crystalline Operating System (CrOS)[24].

The communications library assigned addresses to tasks depending on which node they were physically located, and processes could only communicate to neighbors or the intermediate host (a total of d+1 links). The CrOS terminology for a link between two nodes was that of a *channel* through which 8 byte message packets could be sent. The system only supported collective operations (broadcast) to/from the intermediate host.

Seitz[30] indicated that the hardware structure of the Caltech Hypercube was a "difficult target for programming any but the most highly regular problems". This led to the development of the *Distributed Process* environment, a kernel-based small operating system which allowed greater control and flexibility than CrOS. In particular, the user developing message passing programs no longer had to think about a fixed mapping between nodes (running programs) that communicate in a fixed way via channels (i.e. hardware links), but could now think in terms of processes connected via *virtual communication channels* which provided the basic address independence from hardware locations that is now commonplace.

Within two years Dally and Seitz[6] suggested the design of a special VLSI routing chip that would allow automatic routing for messages between not-adjacent nodes as in the virtual communication channels. Until then messages had to be forwarded using the store-and-forward technique at each hop in *software*, resulting in transfer times proportional to the number of hops. The new hardware routing system was one of the first worm-hole routing systems which allowed a route to be set up through which subsequent message packets could then follow the head/lead packet with minimal overhead. This deadlock-free network allowed programmers of the system some freedom in coding as they no longer had to explicitly code around possible deadlock in the message passing system.

Thus the Hypercube not only demonstrated efficiency when programming down to the "bare metal", but it also later showed the advantages of virtual addresses, virtual communication channels, and more kernel-like support at each node, trends which were to continue.

4.1.2 The Meiko Computing Surface CS-1 and Occam

This was an Inmos Transputer T800-based system that could be programmed using either Inmos's Occam language or the later CSTools environment, which is mentioned later when discussing the more powerful second generation CS-2 machines. The transputer was a 32 bit microprocessor with hardware support for intercommunication and very fast task switching capabilities. The interesting point here is that the initial systems could be programmed using Occam-2, a language based on the Communicating Sequential Processes (CSP) specification previously discussed. In this language, processes communicated via abstract links known as channels using only synchronous blocking sends and receives in the form of:

```
proc1                     proc2
chan1 !data               chan1 ? data
```

where *!* denotes a send and *?* denotes a receive. In many ways, programming such a system was as rigid as that of the Caltech Hypercube, with the added advantage that if the program deadlocked due to communication mis-match, it was due to an *incorrect* program. For example, the following program would always fail:

```
proc1                     proc2
chan1 ?data               chan1 ?data
chan1 !data               chan1 !data
```

Due to the close relationship between Occam and CSP it was possible to prove correctness of code before execution and this was the basis of many early European ESPRIT projects. As other message passing systems allowed users to make mistakes and produce hard to debug applications, some advocates of Occam proposed ways that would allow its use in more general ways, such as language extensions in the form of Occam2-1/2 and Occam3, together with portable compilers aimed at workstations such as the Kent retargetable Occam compiler project[38].

4.1.3 NX from the Intel iPSC1 to the Paragon

¿From the hardware point of view, the original iPSC1 (circa 1986) was a seven-dimensional hypercube, much like the Caltech Hypercube. Its software environment NX1, however, showed more similarity with the "Distributed Processes" environment than with CrOS.

The NX1 operating system was based on the Caltech Reactive Kernel which provided hiding of the underlying communication topology (processes were identified by a simple integer from 0 to P-1, where P was the number of processes per partition), multiple processes per node, any-to-any message passing, asynchronous messaging (i.e. the sender and receiver did not have to be active at the same time for a communication to complete) and non-blocking (i.e. no need to wait for completion), in short what is now thought of as a typical set of features that define a message passing environment.

This additional flexibility also increased the complexity of the message passing library. Users needed additional routines to inquire about location information, and messages needed to be addressed correctly (with additional arguments in the subroutine calls). On the one hand, program developers now had to identify messages on an individual basis as there was no longer a fixed order of transmission or receipt, but on the other hand the pattern of communications was no longer dictated by topology. To assist developers, messages could be tagged or *typed.* That is, a user-assigned integer could be associated with a message which would be used by the receiver to distinguish messages. Unfortunately, the original system did not permit the filtering of messages by sender identity and type at the same time, although the type could be set to the sender's ID to allow for receive-from-sender semantics.

An additional new complication for the developer was that messages of different length could be received without the receiving process knowing which one was first and hence knowing how much buffer memory to allocate. Thus the user had to indicate for each message how large the receive buffer was. Only after the receive completed could the user find out how much data was received, up to the maximum allowed of 16Kbytes. If the buffer offered was too small, the excess message data was discarded (*truncated*).

The later Intel machines implemented an improved version of NX, known as NX2. In the case of the Paragon, this was implemented upon an OSF/1 Unix microkernel. A number of improvements were added such as interrupt-driven communication which allowed an application to perform computation and be woken up when a message arrived instead of having to poll for it intermittently (leading to decreased cache performance and possible page faults as OS calls were invoked to check for messages).

Other changes included the inclusion of message identifiers *(mids)* that allowed simple identification of non-blocking operations. When a non-blocking operation was initiated, a mid would be issued and the user could check for completion of this mid later, thus allowing the underlying hardware communications processors to overlap communication while the compute processors continued. Semantic changes included allowing the use of wild cards (typically negative integers) to denote groups of processes. That is, receiving a message of type -1 would denote receive from anybody of any type, i.e. whatever was received next. Sending to a node of type -1 would denote sending to all processes on a partition. Up to this point, communication had been point-to-point, i.e. from a single sender to a single receiver. The new broadcast functions allowed the construction of global operations such as global synchronizations (barriers) and some arithmetical reduction operations.

Although the design of NX inspired many of the features of current message passing systems, it also had a number of shortcomings, such as lack of more comprehensive group communication functions (to assist certain types of calculations), lack of message identification and filtering at the receiver's end (only one type compared to up to three used on later systems), and initially a high software overhead compared to simpler protocols such as active messages which had direct access to hardware[29].

4.1.4 The German SUPRENUM project

SUPRENUM was a German project to develop supercomputing expertise in Europe by developing a supercomputer and the required software infrastructure. Although only five machines were delivered, this 5 Gflop MPP system and the follow up ESPRIT GENESIS project led to major advances in network interconnection, node design, operating systems, languages, and benchmarking systems.

Two separate systems were used to program the machine, one was the PEACE operating system which through a complex compiler allowed message passing statements, similar in syntax to read/write statements, and Fortran 90 array extensions to each program running on each node. The other was a version of PARMACS which is discussed below in more detail. On top of the PARMACS system, high-level libraries such as GMD's COMLIB, a grid based tool, were created to assist users in using problem domain specific numeric solvers. Other examples included LiSS, a parallel multi-grid solver for partial differential equations. The use of PARMACS is important, as the software designers not only concentrated on improving the range of group operations available to users but also realized that portability of code was important especially in the light of the short life span of each MPP.

4.1.5 IBM Scalable Power Series and the External User Interface (EUI)

The IBM Scalable Power (SP) Series of systems, starting with the IBM 9076 SP1 and the later SP2 machines, consisted of tightly-coupled distributed memory sets of RS/6000 RISC processors interconnected by a high-speed switch. The systems were based on experience gained from the Vulcan hardware project and the Viper operating environment. The message passing library designed to program these systems was known as the IBM external user interface (EUI) and consisted of four main components: task management, message passing (point-to-point), task groups, and collective operations. The last two items were of particular importance to later systems such as MPI.

Point-to-point communication under EUI was performed by sending messages to tasks directly in the same style as on the Intel iPSC systems, with addresses from 0 to N-1, where N was the number of processes making up a parallel *job*. The point-to-point system supported typed messages for both blocking and non-blocking messages. Unlike NX, messages could be selected by the receiver on both message type and source (sender) including the use of wild cards. To assist in handling non-blocking messages, the user could check the status of a particular transfer as well as wait for completion of a named operation, any of a range of operations, or all pending outstanding transfers.

EUI allowed the construction of conceptual collections of processes into logical groups that could be addressed by a single group ID. This allowed users to avoid having to list (sometimes large numbers of) processes explicitly when passing messages in structured ways repeatedly. The use of collective operations on these groups avoided the use of many point-to-point calls and allowed the system to perform these as efficiently as possible on the given hardware. The range of operations included barriers, data shifts, broadcasts, gather, scatter, a generalized combine and an associative reduction. All the collective operations required all members of each group to partake in a blocking fashion.

During the design of EUI the IBM designers faced many issues relating to how the interface should look and operate. For example, asynchronous returns from non-blocking functions were an example of where a two-part status lookup was required. If the user interface to a non-blocking receive were as follows:

```
void recvf (&data, sizeofbuffer, &size-received)
```

the size-received variable could not be filled in by the system until the non-blocking operation had completed, which might be while the thread that made this call was in a different program module. Thus it was necessary to get a status handle which could be queried after the operation had completed and the memory storage of which was handled by the messaging system:

```
recvf (&data, ... &status)
...
wait (for above recv to complete)
/* status is now safe to examine, as the wait operation
   has completed any status data structures. */
```

The EUI project was not the first to introduce this two step strategy, this also occurred between NX1 and NX2, but the overall design used by IBM was the basis of that used later by the MPI forum.

4.1.6 Meiko CS-2 and the CSTools environment

The Meiko CS-2 grew out of many of the lessons learned from the SUPRENUM project, especially in terms of Meiko's ELAN message handling hardware co-processor and the coupling of a compute engine (i860 and later Sparc-based processors) with optional vector floating point units at each node. Each node ran a copy of the Solaris Unix microkernel which mapped the Elan memory into User space and thus provided access directly to the communications hardware. Therefore, a system call in protected mode could be avoided, which led to drastically increased performance compared with previous designs. An example for such a system which had suffered from poor performance due to software overheads in accessing hardware had been the NX library running under OSF/1 on the Intel Paragon.

The system was programmed with the previously developed CSTools environment which used the concept of named communication channels known as *transports* through which messages could be passed. Hence users could code in terms of these named links rather than in terms of the actual end-points. Once a transport had been instantiated, repeated communication through it incurred very little system overheads. Blocking, non-blocking, synchronous, and asynchronous point-to-point operations were all supported.

4.1.7 Thinking Machines CM5 and the CMMD Active Message Layer (AML)

The Connection Machine 5 from Thinking Machines (TMC CM5) was very different from previous Connection Machine designs, being a true distributed memory MIMD system as opposed to the previous SIMD systems. The machine featured two interconnection networks, and Sparc-based processing nodes, each with four vector units for pipelined arithmetic operations. The programming environment consisted of the CMOST operating system, the CMMD message passing library and various array-style compilers, the most popular of which, known as CMF, supported a F90-like SIMD programming style.

The CMMD message passing system was unique in that it offered users access to routines from the lowest level, the Active Message Layer (AML), a point-to-point library, channels and a cooperative functions library. The four systems will be reviewed briefly here:

The Active Message Layer was the lowest level of operation and manipulated the communications hardware directly. The layer provided three basic operations. Active messages are like a lightweight RPC call. The sender sends the address of a function to be invoked at a remote node together with its arguments in a single 72 byte packet. The second operation was a data transport mechanism which only wrote data into a remote memory at a set location (much like Cray's shmput operations). The final operation was a receive port data structure which assisted in handling multiple data packets.

The CMMD point-to-point library was built on top of the AML and provided the common list of operations including blocking, non-blocking, synchronous, and asynchronous point-to-point operations with selection upon source, message type or wild cards. Interestingly, the blocking calls were quicker than the non-blocking calls as they avoided system level copying of message data. Another interesting feature was the inclusion of a joint send and receive operation, that allowed for simpler coding of stencil operations and boundary exchanges in domain decomposition problems. An important advantage of this combined function was that it could be implemented more efficiently than the two separate operations. For example, previously two operations were required to exchange values:

```
proc A                  proc B
Send (B, data)          make copy of data edge into data'
Recv (B, data)          Recv (A, data)
                        Send (A, data')
```

As opposed to:

```
Task A                              Task B
CMMD_send_and_receive (B, data)     CMMD_send_and_receive (A, data)
```

Note that the first version is made more complex by the need to copy data values to prevent them from being overwritten before they are copied into the message layer, a common complication found in many wavefront calculations.

The CMMD Virtual Channels, like the transport operations on the Meiko CS-2, allowed multiple communications to occur with minimal overhead once they had been initiated. The interface was very basic, with calls to open, close, and status checking of channels. A write operation was provided, although the receiver would have to check the status to find out if its data had arrived, and then reset the channel to allow for more data to be sent.

The CMMD Cooperative Communications included broadcast, reduce, synchronize (barrier), and scan functions. Unlike other systems, they used a separate interconnection network for better performance. Another more recent architecture to also use separate hardware to enhance performance of global operations are the SGI Cray Research T3D/E machines that utilize a special signaling line for handling barrier synchronization.

4.1.8 Summary of vendor-specific message passing systems

This section has shown how several different MPP systems provided a wide range of message passing features that primarily took advantage of different hardware facilities. Although there appeared a standard subset of functions expected by users, the detailed interface definitions were often tailored to the special hardware of each particular vendor. Another interesting point was the demand for communication functions at a higher abstraction level, as application programmers repeatedly had to code the same communication patterns, for example, as they occur on regular process grids. Vendors thus introduced better facilities than just simple send and receive primitives to support these higher level abstractions.

4.2 Portable message passing systems

The lack of application code portability between the parallel systems of different vendors prompted the development of portable message passing interfaces *(portability APIs)* by various research groups. While portability of user codes among hardware platforms of different vendors was a common feature of those interfaces, to some extent they catered for different problem-specific domains. These two aspects, portability and application diversity, made the portability APIs an important step on the way towards the message passing standard. This section presents some of the more influential developments.

4.2.1 The m4 and p4 macros

The *p4*[1, 2] system grew out of a set of Fortran macros that was developed at Argonne National Laboratory (ANL) for use on a HEP shared memory computer. The original macros, called MonMacs, were processed at compile time by the Unix m4 preprocessing utility, and offered the user a set of monitor functions used to provide locks on critical sections of code that accessed shared data. The use of macros avoided an additional set of stack operations, that function calls would have made necessary.

The authors of the system co-wrote the book "Parallel Programs for Parallel Processors" which gave the system its final name of *p4*. The system was used as a basis for several specialized versions, such as TCGMSG for Chemistry problems, and the GMD macros for solving problems on regular grids. This later version was used as the basis for the very successful PARMACS system developed by Rolf Hempel at GMD in Germany.

The final p4 system was based on procedure calls and supported C as well as Fortran on a very wide range of systems including both distributed-memory and shared-memory systems. The programming paradigm was that of processes that formed administrative clusters which intercommunicated by either locks or explicit message passing. The system provided user access to buffers, so that an experienced user could avoid additional buffering by the system. There were no non-locally blocking or asynchronous calls, i.e., the calls returned when the data was sent, and the user did not have to *probe*, *test* or *wait* for completion before reusing buffers. A globally blocking point-to-point call was also included *p4sendr()* which waited for an explicit acknowledgment from the receiver before returning (hence the *r* on the end of the call name to signify a *rendezvous*). The p4 system also included a wide range of collective operations, with the additional option to construct user-defined operations with the *p4_global_op()* call.

Although the *p4* system was very efficient, it was not as popular as other message passing system and was later used as a layering scheme to support other projects such as Chameleon[15], BlockComm and later the MPICH implementation of MPI[16, 17].

4.2.2 The Express environment

Express[28] was initially based on the Crystalline Operating System's message passing library from Caltech, with special emphasis on performance. Highly-tuned versions of the software were available for certain MPP systems. There was some limited support for dynamic process creation.

The system later grew into a complete application development package which supported dynamic load balancing, parallel IO, and comprehensive collective libraries for common topologies such as rings, grids, tori etc. These topology features were made accessible through the *exgrid*()* family of calls. If used correctly, a complete message passing application could be created by using *exlayout()* to specify distribute data and *exdist()* to move the data, leaving the user with no explicit send and receive calls, and thus avoiding mis-ordered calls and complex handling of boundary conditions when implementing numeric solvers.

4.2.3 Zipcode

Zipcode[31] was an experimental portable message passing system that emphasized support for parallel library development. As message passing applications became more complex, for example through the use of third-party libraries, insulating the message traffic in the application code from the communication inside a library became a non-trivial problem, in particular if wild cards were used in receiving messages.

Zipcode introduced safe communication spaces, so that separate program units (libraries) could have their own spaces within which to communicate. This was achieved by making the addressing of processes relative to static process groups combined with a message context, a system-supplied additional message tag, that was opaque to the user. The process groups (with relative addresses known as ranks) were bound to contexts by processes known as mailers.

Zipcode also allowed for topology information to be associated with process groups, so that communication addresses could be specified relative to the current position in topological terms. For example, one could send a message to the next process in a ring, rather than having to look up an absolute address and then specifying that value as an address in the send operation.

4.2.4 Linda

Linda[4] was based on an associative shared virtual memory system, or *Tuple Space*. Rather than sending messages from one process to another, processes created, consumed, duplicated or evaluated data objects known as *tuples*. Creating a tuple was performed by calling out(), which was then passed into a common shared space which all other processes had access to. Reading a tuple was performed by either rd() or in(), where rd() just read (duplicated) the tuple and in() consumed the tuple and removed it from the shared space. Matching of tuples was performed by specifying a template of possible fields in the tuple when calling either rd() or in(). eval() created tuples asynchronously, with each field making the tuple evaluated in parallel (or at least in a non-deterministic order much like in the Occam ALT construct).

Linda allowed for very simple but powerful parallel computer programs by simplifying addressing (no explicit addressing of processes, only the data they handled) and by removing the coupling between processes. In Linda, one could not specify the consumer of a tuple, and the consumer might not even had existed when the tuple was created.

For example in a conventional message passing system the following steps are required to pass a message from one process to another:

```
proc A                              proc B
find address of 'B' (addrB)         find address of 'A' (addrA)
Send (outdata, addrB, messagetag)   Recv (indata, addrA, messagetag)
```

Where in Linda one could have:

```
proc A
out (messagetag, outdata)
exit
```

Some time later:

```
proc B
in (messagetag, ?indata)
{process data}
exit
```

In this case the tuple that contained 'messagetag' as the first field would be consumed by process B (the '?' specifies a wild card) upon which its data would be placed in the indata memory buffer. Neither process ever overlaps temporarily or knows of each others 'address'.

Although initial implementations were slower than native libraries, later versions that utilized compile time analysis of data fields used by application programs allowed the run-time system to select tuned low-level services that implemented the tuple space management and matching operations. On a number of tests[7], some versions of Linda performed conparably to both networked message passing systems such as PVM as well as native vendor message passing libraries on a number of MPPs for medium to large message payloads.

4.2.5 The Parallel Virtual Machine (PVM)

PVM was a system for managing and coordinating message passing programs that was developed as a research project by researchers at Emory University, the University of Tennessee, Knoxville, and Oak Ridge National Laboratory [34, 13]. PVM allowed message passing across diverse heterogeneous collections of machines which could be considered a single computational resource (the virtual machine). It was unusual in that it supported dynamic functionality as much as possible, unlike most systems that were aimed at a static view of the world.

In PVM the hosts that made the virtual machine could be added, removed or just plainly fail while the rest of the system continued. This was also true for processes, called tasks, which could be created (spawned) on the fly, and which in turn could spawn more tasks or kill any other PVM tasks at will. This flexibility added greatly to PVM's use on networked machines which were prone to changes of configuration unlike most MPPs which were very static in nature.

All tasks in PVM had a unique address known as a *Task IDentifier* or TID. Any task could send a message to any other task once its address was known. Messages were typed (using *tags*), allowing receivers to filter using wild cards as well as source addresses. In the final version of PVM (3.4) this was extended to also include user allocated system contexts.

All messages were initially built in a PVM message buffer which would be sent to the receiving task. This allowed for the use of XDR (supporting heterogeneous configurations of hosts) and the mixing of many different data types in a single message. The receiver would then have to unpack messages to access the data contained within. This, together with the notion that sends never blocked, due to the system providing all necessary buffering, helped users avoid deadlocks.

In the following example, the message passed between two processes is made up of a number of double precession data elements, preceded by an integer containing the number of data elements:

```
Sender Task
pvm_initsend(PvmDataDefault)
pvm_pkint (&num_elements, 1, 1)
pvm_pkdouble(data, num_elements, 1)
pvm_send (receiverTID, msgTAG)

Receiver Task
bufid = pvm_recv (senderTID, msgTAG)
pvm_unpkint(&num_elements, 1, 1)
pvm_unpkint(data, num_elements, 1)
```

The additional overheads caused by buffer management on MPP systems could be considerable. This was resolved with the addition of pack send/receive calls *pvm_psend() / pvm_precv()* which provided performance very close to that of many vendor libraries on a number of MPPs.

PVM also provided group collective operations, with a separate task (*PVMGS*) maintaining association between group names and lists of member TIDs. Following usual PVM practice, these groups were dynamic, although the ability to freeze groups for improved performance was added in PVM version 3.3.

A number of versions of PVM were placed in the public domain, and the authors of PVM encouraged feedback as a mechanism for improving their software and assessing the needs of parallel application developers. Thus, in the early 1990s, PVM became quite widely used, particularly on networks of workstations, and when the MPI standardization process was begun in 1992, PVM was the most popular message passing system in use.

4.2.6 PARMACS

The PARMACS (PARallel MACroS)[20, 3] were initially a macro-based message passing system developed from the ANL & GMD macro system, which itself was a grid based version of the earlier MonMacs. Later versions of the system were purely library-based and supported both C and Fortran base languages. The PARMACS were designed to be the most portable and efficient general purpose message passing system for distributed memory parallel systems prior to any MPI standard implementations. The system supported synchronous and asynchronous message passing as well as message probe tests and an extensive range of collective group operations. Messages could be selected by the sender using both source address as well as message type. In an aid to efficiency, the system would only perform data conversion and buffering if the processors executing the sending and receiving processes used different data representations.

PARMACS supported virtual topologies, a mapping between actual process addresses and a virtual torus grid with up to four dimensions, or arbitrary graph structures specified by vertex and neighbor lists. The topology was defined by the *PMPTOR()* call in the master process, which then created the computation processes with a subsequent *PMPCRTE()* call. Addresses in PARMACS were then calculated from each process' mapping within this virtual topology and could be accessed by calls to the Torus Query function *PMQTOR()* or the generic Cartesian lookup function *PMLKUP()*. Thus communication between processes could be programmed using completely relative addressing, such as "send message to my predecessor" (or successor). This allowed the readability of programs to be greatly enhanced and reduced much of the code overhead normally associated with explicitly calculating addresses and handling boundary conditions that occur in many message passing codes.

As the PARMACS were ported efficiently to many MPPs systems, it became a target for many tools that operated at an even higher level of abstraction. Examples were the GMD Communications Library (Comlib), which was designed to support partial differential equation solvers for multidimensional spaces, and the SLAP Linear Algebra package which contained parallel solvers for linear equations and eigenvalue problems.

5 The Message Passing Interface standard - MPI

One of the main advantages of software standards is portability. A corollary of this is that it becomes worthwhile to invest effort in efficient implementations of the standard and to develop higher-level software based on it. Thus software standards create scope for the development of high quality, commercial software. However, standardization does have a potential downside – if the standard is not well thought out it may enshrine deficiencies, omit important functionality, or quickly become out-dated. A standard that frequently needs to be updated does not inspire confidence and is likely to be discarded.

5.1 MPI-1

MPI arose from a desire to ensure that efforts in the United States and Europe towards the standardization of message passing libraries did not diverge. Attempts in Europe to standardize message passing date to a SERC[1] workshop in March 1990 on software standards for MIMD computers. By the early 1990s two approaches were contenders in Europe for a message passing standard. One of them, named UBIK ASI[2], had its roots in Occam, reflecting the fact that a large fraction of European parallel computers were transputer-based. UBIK ASI featured named channels for message passing, and arose out of the ESPRIT II GP-MIMD project[2]. The second approach was PARMACS[2], which was initially developed by Rolf Hempel of GMD and Jim Patterson of ANL. From 1990 onwards PARMACS was further developed by Hempel and colleagues in Europe within the EU-funded ESPRIT project Genesis[2]. PARMACS was adopted as the programming model of the PPPE and RAPS projects[2]. At ANL, PARMACS evolved essentially independently to become the P4 project[2].

A strong impetus for a message passing standard was provided by the Commission of the European Community which placed an emphasis on uniform programming interfaces for all projects in the ESPRIT III program. From 1991, the ESPRIT Special Interest Group on Language Standardization for Distributed Memory MIMD Computers had sought to develop a standard in Europe. This led to the SHAPES project[2], which attempted to reconcile the UBIK ASI and PARMACS programming models. It turned out that the two approaches were too different to be merged without creating an unwieldy interface. UBIK ASI continued to be used in the GP-MIMD project, whereas PARMACS became one of the foundations for MPI.

In the United States, a number of differing distributed memory machines were in use by the early 1990s, and several research groups had developed

[1]Science and Engineering Research Council (Great Britain)

[2]Some of these designations have become accepted as proper names and are not usually resolved any more. The GP in GP-MIMD stands for *General Purpose*, PARMACS for *PARallel MACroS*, PPPE for *Portable Parallel Programming Environment*, RAPS for *Real Applications on Parallel Systems*, and the name P4 is derived from the book title *Portable Programs for Parallel Processors* by E. Lusk et al.

message-passing systems that sought to provide portability across these platforms. The public domain software PVM (Parallel Virtual Machine), developed at Oak Ridge National Laboratory, the University of Tennessee at Knoxville, and Emory University, was in wide use for message passing in the United States and in Europe. Express was commercially marketed by Parasoft, Inc. and was also popular. Before 1992, there was no organized attempt to adopt a message passing standard in the United States.

Jack Dongarra and Tony Hey were among the first to recognize the need for coordination to prevent European and US researchers selecting different message passing standards. In the Summer of 1992 Jack Dongarra, Rolf Hempel, Tony Hey and David Walker became involved in an effort to define a common message passing standard. This process began with the Workshop on Standards for Message passing in a Distributed Memory Environment which took place April 29–30, 1992, in Williamsburg, Virginia. This workshop, which was organized by Jack Dongarra and David Walker and sponsored by the Center for Research on Parallel Computation (CRPC) at Rice University, discussed a number of current message passing interfaces and examined what the essential features of a message passing standard should be (see [36] for a summary). At the workshop, Ken Kennedy of Rice University encouraged the message passing community to set up a working group in order to carry the standardization process forward.

In October 1992, a preliminary draft proposal, now known as MPI-0, was introduced by Dongarra, Hempel, Hey, and Walker and circulated for comment within the working group. This draft was presented to the Working Group in November 1992 during the SuperComputing Conference in Minneapolis. MPI-0 embodied the main features identified at the Williamsburg workshop as being necessary in a message passing standard, but it lacked collective communication routines, and was not thread-safe in its management of resources. However, MPI-0 brought to the forefront a number of important issues and served as the initial catalyst for subsequent discussion. A revised version was published in February 1993 [8]. At the Minneapolis meeting, it was decided to place the standardization process on a more formal footing. The procedures and organization of the HPF Forum were adopted to create the MPI Forum. Subcommittees were formed for the major component areas of the standard and an email reflector was established for each. The ambitious goal of producing a draft MPI standard by the Fall of 1993 was set. In addition, Ewing Lusk and William Gropp of ANL made a commitment to implement the standard.

The original goals of the MPI Forum were to develop a widely-used, practical, portable, efficient, and extensible standard for message passing, or stated more precisely:

- Design a programming interface that can be used by software development projects at various levels, ranging from end-user applications to highly-optimized parallel libraries and run-time systems of data-parallel compilers.

- Allow for efficient implementations. For example, it should be possible to avoid repeated copying of volume data, to overlap computation and communication, and to offload work to a communication co-processor, if available.

- Define an interface not too different from current practice, such as Express, NX, PARMACS, PVM, p4 etc.

- Define an interface that can be implemented on many vendor's platforms with no significant changes in the underlying communication and system software.

- Allow for implementations in heterogeneous environments.

- Allow convenient C and Fortran bindings for the interface (later also C++), while making the semantics of the interface language independent.

- Provide a reliable communication interface. The user should not have to cope with communication failures.

- Design a thread-safe interface.

It should be noted that some of the requirements were commonly regarded as counter to each other, such as efficiency through fewer internal copy steps versus heterogeneous environment support. The goal of the MPI Forum was to design the interface such that all requirements could be fulfilled at the same time, whereas previous generations of systems such as PVM opted for portability (across heterogeneous systems), with less than optimal performance.

The MPI standard was intended for use by all programmers of portable message passing codes in Fortran, C, and C++. This included individual application programmers, developers of software designed to run on parallel machines, and creators of environments and tools. In order to be attractive to this wide audience, the standard had to provide a simple, easy-to-use interface for the basic user, while not precluding experienced software developers from exploiting the full efficiency of advanced machines.

To finish this ambitious standardization project within a short time frame, the MPI Forum met every six weeks throughout the first nine months of 1993 in Dallas, Texas. During this period, a draft specification for what is now termed MPI-1 was produced. Nearly 150 people were on the email discussion lists while about 60 were actively involved in the MPI Forum. In addition, an average of 35 people attended each MPI Forum meeting in Dallas. Membership of the MPI Forum was open to all members of the high performance computing community and included people from academia, the hardware and software industries, and the US national laboratories. Both European and US researchers participated.

The first two or three meetings were somewhat chaotic, with many proposals quickly considered and rejected, or retained for future consideration. The fact that the forum had members from a broad range of backgrounds and interests meant that some issues were intensely debated, but this was generally beneficial as it ensured that the final MPI specification provided the required functionality without being too large and unwieldy. The major manufacturers of parallel computers showed their support by sending their best technical people, who ensured that the standard was efficiently implementable on their machines. The involvement of most of the major manufacturers of parallel computers prevented most machines from being disadvantaged by the specification. Thus, some features of the MPI-1 specification were included so that certain machines would be able to take advantage of them – for example the ready communication mode was largely included to permit use of the force type facility on Intel machines. Later meetings were calmer and more constructive as attendees got used to working together effectively. Between January and September 1993, seven meetings of the MPI Forum took place, each lasting about 2 days. During this time a great deal of discussion took place via the email reflectors.

A draft specification of MPI-1 was presented to the whole high performance computing community at a meeting at the Supercomputing '93 conference in Portland, Oregon. A comment period followed lasting until February 1994 during which people were invited to comment on the specification. In January 1994, a meeting of the MPI Forum was arranged at INRIA Sophia Antipolis in France to give the European HPC community an opportunity to get more involved. This gave rise to some important contributions – the *pack* and *unpack* functions for sending noncontiguous or mixed-datatype messages were added as a result of this meeting to provide compatibility with the PVM and PARMACS message passing libraries. MPI's pack and unpack routines also allow a message to be received in several parts, so that the receipt of a later part may depend on the content of an earlier part. The buffered send mode was also added as a result of input from this meeting, again to allow compatibility with existing message-passing libraries. Other improvements stemming from the Sophia Antipolis meeting were the provision of portable timers, a more sophisticated treatment of error codes and classes, and a guaranteed range for message tags.

Following the European MPI meeting in January 1994, the MPI specification was largely complete. A further meeting of the MPI Forum was held in late February 1994 at the University of Tennessee, Knoxville, to incorporate new features suggested at the European meeting. Version 1.0 of the MPI specification, now known as MPI-1, was then released through the Internet on May 5, 1994.

Over the next 12 months, some minor errors were found in the MPI-1 specification, and several points required clarification. Obviously, changing an existing standard should only be considered as a last resort, and the Forum was very scrupulous in doing so. Fortunately, most changes referred to quite subtle MPI features and affected few, if any, existing application programs. The MPI Forum released a revised version 1.1 of the MPI-1 specification on June 12, 1995. This formally ended the first phase of the MPI standardization process, although a further revision was published as version 1.2 in July 1997.

¿From mid-1993 onwards, an up-to-date version of the current MPI draft specification was implemented, maintained, and distributed by Ewing Lusk and William Gropp of ANL. This reference implementation, now known as MPICH [17], played an important role in the MPI standardization process. It quickly revealed any problems or inconsistencies in the MPI specification as it evolved. It demonstrated that efficient implementations of MPI were possible. It provided an implementation of MPI that could be quickly ported to a wide range of parallel computing platforms, and it formed the basis of many hardware vendors' custom implementations of MPI. At one point in the development of MPI, it was proposed to denote a subset of the MPI routines as being the essential ones that vendors should implement. The existence of MPICH demonstrated that this "subset MPI" was not necessary, and the idea was dropped.

5.2 Technical details of the MPI-1 standard

The MPI-1 standard includes chapters with the following topics:

- Point-to-point communication (Chapters 2 and 3)
- Collective operations (Chapter 4)
- Process groups (Chapter 5)
- Communication domains (Chapter 5)
- Process topologies (Chapter 6)
- Environmental Management and inquiry (Chapter 7)
- Profiling interface (Chapter 8)
- Bindings for Fortran and C

The basic communication mechanism of MPI-1 is the transmission of data between a pair of processes, one side sending the other side receiving. Many variants of this point-to-point communication are supported, including blocking and non-blocking functions, buffered send operations, and persistent (channel-like) communication requests.

Message data in MPI-1 is strongly typed, to allow for communication in heterogeneous environments, where type information is used by the MPI-1 implementation for automatic data representation conversion. Integer tags attached to messages allow messages to be selected on the receiving side, either by specifying a particular value or a wild card. Message selection by specifying the source process of the message is also available.

MPI-1 incorporates the notion of process groups and insulated communication domains, called communicators, that were first introduced by the Zipcode interface (cf. Section 4.2.3). Process groups can have a topological structure, very similar to the topology support in PARMACS (cf. Section 4.2.6). Building on the concept of process groups, MPI-1 provides the following collective communication functions:

- Barrier synchronization across all group members (Section 4.4).
- Global communication functions. They include
  - Broadcast from one member to all members of a group (Section 4.6).
  - Gather data from all group members to one member (Section 4.7).
  - Scatter data from one member to all members of a group (Section 4.8).
  - A variation on gather, called all-gather, where all members of the group receive the result (Section 4.9).
  - Scatter/gather data from all members to all members of a group (also called complete exchange or all-to-all) (Section 4.10).
- Global reduction operations such as sum, max, min, or user-defined functions. The result can either be returned to only one group member or to all members (all-reduce) (Section 4.11).
- A combined reduction and scatter operation (Section 4.11.5).
- Scan across all members of a group (also called prefix operation) (Section 4.12).

MPI-1 is suitable for use by fully general Multiple Program, Multiple Data (MPMD) programs, where each process follows a distinct execution path through its own code. It is also suitable for codes written in the more restricted style of Single Program, Multiple Data (SPMD, cf. Section 3), where all processes follow a distinct execution path through the same program. Although support for threads is not required, the interface has been designed so as not to preclude their use. MPI-1 provides many features intended to improve performance on scalable parallel computers with specialized interprocessor communication hardware. These features have been used by hardware vendors to produce native, high-performance implementations. At the same time, implementations of MPI-1 on top of standard Unix interprocessor communication protocols provide portability to workstation clusters and heterogeneous networks of workstations.

5.3 MPI-2

When MPI-1 was published, the MPI Forum already had plans for a continuation. Some important topics, which could not be handled within the short time frame of MPI-1, were postponed to this later phase. About one year passed, however, before the MPI-2 meeting series started. During this interval, several MPI-1 implementations were released, and feedback from MPI users became available.

In March 1995, the first MPI-2 Forum meeting took place. Ewing Lusk of ANL was the convenor of all sixteen MPI-2 meetings, and the location was moved to Chicago. Otherwise, the procedural rules of the forum were kept the same as for MPI-1.

The main activity of the MPI-2 Forum was the standardization of issues which had been left open during MPI-1. The main topics were:

- Parallel I/O
- Dynamic process management,
- Single-sided communication,
- Extensions to collective operations,
- New language bindings.

Again, sub-committees were created for each topic. Discussions at sub-committee meetings and over the email reflectors led to proposals that were then presented and voted on at the plenary meetings.

In spite of the similarities, there were important differences between the two MPI phases. One of the MPI-1 guidelines had been to keep the standard as close as possible to "current practice". Most features had been tested in several existing message-passing interfaces, and even the more exotic ones were available at least in one experimental implementation. Several topics were passed to MPI-2 mainly because in 1993 they seemed too immature for standardization.

In many cases during MPI-2, the borderline between current practice and research was passed, and new features were included in the standard without any experience with available implementations. As a consequence, there was a great diversity of ideas brought to the forum, and at times it was difficult to keep the focus on the standardization aspect and away from computer science research. In the end, many of the more research-oriented ideas were put into the "Journal of Development" instead of the standard itself.

Another important difference was that during MPI-2 complementing implementation activities only covered parts of the standard. Examples are the IBM implementation of single-sided operations, a preliminary version of the ROMIO I/O software from ANL, and some process management functions in the LAM package of the Ohio Supercomputer Center. Therefore, there was limited information available as to the practical applicability of the interface specification.

The different character of the two MPI forums is also reflected by the number of meetings required to finalize the standard document – sixteen for MPI-2, compared with only seven for MPI-1.

Parallel I/O played a special role in MPI-2. When the new forum was created, parallel I/O was dropped from its initial list of topics, although it had been the MPI extension most strongly requested by application programmers. The reason was that with the multi-agency Scalable I/O Project and the MPI-IO project by IBM and NASA Ames, there were already activities going on which the MPI-2 forum did not want to duplicate. After more than a year, however, MPI-IO was integrated officially into the MPI-2 activity. At that point, their interface definition document was taken by the forum as a first draft which then went through the same procedure of readings and voting sessions as any other proposal. In the end, a revised version became the official I/O chapter of the MPI-2 standard. Later, the Scalable I/O Project adopted the MPI-2 standard as their application programming I/O interface.

The MPI-2 forum published the results of their deliberations in July 1997. Documents and related information can be retrieved through the official MPI web site `http://www.mpi-forum.org/`. A book publication of an annotated reference manual for MPI-2 [19] appeared in 1998 analogous to its MPI-1 counterpart [32] combined with a revision [33] of the latter document.

In contrast to MPI-1, implementations have emerged rather slowly for MPI-2. Apart from MPI-2 being larger than MPI-1, the main reason was the lack of a public domain version which hardware vendors could use as a starting point for their optimized implementations. One difficulty in the design of a portable implementation was that MPI-2 goes beyond message-passing in several areas and requires interaction with highly machine-dependent system services such as I/O, scheduling and resource allocation. Argonne National Laboratory did not make a firm commitment to provide a full MPI-2 extension of their highly successful MPICH software, but some implementers were waiting for it to become available anyway. As first steps, ANL published an implementation of the parallel I/O chapter, known as ROMIO [35], and a C++ binding as extensions to MPICH. As a consequence, the parallel I/O chapter was the first part of MPI-2 available on a wide variety of parallel platforms, and it has since been integrated into several proprietary MPI implementations.

Even two years after the publication of the MPI-2 standard, hardware vendors were spending less effort on the development of their own MPI-2 implementation than they did for MPI-1. In addition to their reluctance to write all the necessary software themselves, there were at least two other reasons. One reason was that the status of the MPI-1 interface of most vendors was far from being optimized. Many aspects were still handled by generic software which did not exploit special hardware features. For example, many MPI implementations still copied non-contiguous messages to contiguous memory first before sending them, instead of combining the compression and send operations into a single step. Also, the process topology functions in most cases did not provide an optimized mapping between processes and processors in heterogeneous network configurations. With limited personnel resources, many hardware vendors regarded these and other MPI-1 optimizations more important than an early availability of MPI-2.

The other reason for the slow take-up of MPI-2 was that few application programmers requested MPI-2 functionality beyond parallel I/O. On the other hand, since programmers could not use an interface that was not available, to some extent this was a chicken-and-egg problem. It remains to be seen how successful MPI-2 features such as single-sided communication and dynamic process management will turn out to be, once the full interface is available on a broad range of platforms.

6 Lessons learned

¿From the very beginning, the organizers of MPI recognized the importance of having as many interested parties as possible represented in the forum, and in particular a majority of the hardware vendors of parallel systems. In the technical discussions, care was taken to avoid interface features that would be difficult to implement on some existing or conceivable future hardware platform. As a result, most vendors, if not all, support MPI as their primary message passing interface, and other interfaces continue to be available mainly for reasons of compatibility with legacy codes.

The impression that some interested parties tried to impose a standard onto the rest of the community never arose, since the MPI Forum adopted the rules of complete openness and democracy.

It would have been very difficult to make one of the existing message passing interfaces the universal standard. From the technical point of view, no interface before MPI fulfilled the functionality requirements of the whole range of potential users, from novice programmers to parallel library writers, and allowed good performance to be achieved on all target platforms. At least as important was the political aspect since choosing an existing interface would have created opposition by vendors and users who preferred other choices. The adopted approach of combining the proven features of existing interfaces into something new was the only way to gather the required support.

The universality of MPI led to a complexity which many critics initially regarded as a disadvantage. Two developments helped to overcome this impression. The MPICH implementation of the full standard showed that it could be implemented without loss of efficiency, and that it could be ported to a new platform with reasonable effort. The publication of introductory textbooks [16, 26] helped application programmers focus their attention on the MPI constructs needed for their work. Before, the typical way to learn how to use MPI had been to study the standard document, so beginners were immediately confronted with the full complexity of the interface. The textbooks demonstrated that simple example programs were as easy to write with MPI as with other message passing interfaces.

The de facto status of the MPI standard was never a limiting factor to its success. Since there were no formal rules imposed by a standardization body, the organization was very flexible and effective. For MPI, it was accepted that the early publication of the standards document would make minor corrections necessary later, and in this case it happened after about one year. Experience has shown that the negative impact of "changing a standard" was small compared with the benefit of a timely result. It may be true, however, that de facto standards are particularly successful within the technical computing community, at which MPI was targeted.

MPI is firmly established in the high-performance computing community and supported by all relevant hardware and software vendors. At least for now, there appears to be no interest to make MPI an official international standard. The general expectation is that the potential gain would not justify the effort required.

Many people regarded MPI and PVM as competing to become the message passing standard even though Jack Dongarra and Al Geist, two of the principal designers of PVM, were key players in the development of MPI. However, MPI and PVM were designed for different uses. PVM was originally intended for use on networks of workstations, and to address issues such as heterogeneity, fault tolerance, interoperability, and resource management – its message passing capabilities were not very sophisticated. The design of MPI focused on message passing capabilities, and it was intended to attain high performance on tightly-coupled, homogeneous parallel architectures. For several years MPI and PVM developed in tandem, and each benefited by a cross-fertilization of ideas.

It is debatable whether PVM would have become the message passing standard in the absence of MPI. Unlike MPI, PVM was a research project with a finite lifetime. Its development and support was undertaken by a small group of researchers, and although it was a highly successful project, it seems unlikely that it would ever have developed the same range of functionality and sophistication that the MPI Forum deemed necessary in a message passing standard. It should be noted that the consultative process through which MPI was created was important not only in generating a well thought-out specification, but also in giving MPI a degree of credibility that other message passing libraries, developed by research groups or hardware vendors, did not have. (For details, see [18].)

A major strength of PVM was its resource management capabilities, some of which have been incorporated into MPI-2. MPI has been successful in delivering high performance on tightly-coupled parallel systems, but PVM is still in use on networks of workstations (NOWs). The message passing aspects of PVM were frozen after Version 3.4, and the PVM project has since branched out into other research areas that build upon the PVM work, such as CUMULVS [14] and HARNESS [9] which are concerned with distributed, heterogeneous environments for NOWs.

References

[1] R. Butler and E. Lusk, *User's guide to the p4 parallel programming system*, Technical report, Argonne National Laboratory (1992).

[2] R. Butler and E. Lusk, Monitors, messages, and clusters: the p4 parallel programming system, *Parallel Computing* **20(4)** (1994) 547–564.

[3] R. Calkin, R. Hempel, H.-C. Hoppe and P. Wypior, Portable programming with the PARMACS message passing library, *Parallel Computing* **20(4)** (1994) 615–632.

[4] N.J. Carriero, D. Gelernter, T.G. Mattson, and A.H. Sherman, The Linda alternative to message-passing systems, *Parallel Computing* **20(4)** (1994) 633-655.

[5] L. Dagum and R. Menon, OpenMP: an industry-standard API for shared-memory programming, *IEEE Computational Science & Engineering*, **5(1)** (1998) 46–55.

[6] William J. Dally and Charles L. Seitz, The Torus Routing Chip, *Distributed Computing* **1**, pp. 187-196, 1986.

[7] A. Deshpande and M.H. Schultz, Efficient parallel programming with Linda, *Supercomputing '92*, Minneapolis, MN (1992), 238–244.

[8] J. Dongarra, R. Hempel, A. Hey and D. Walker, *A proposal for a user-level, message passing interface in a distributed memory environment*, Technical report ORNL/TM-12231, Oak Ridge National Laboratory (February 1993).

[9] J. Dongarra, A. Geist, J. Kohl, P. Papadopoulos and V. Sunderam, HARNESS: Heterogeneous adaptable reconfigurable networked systems, in: *proceedings 1998 conference on High Performance Distributed Computing* (1998).

[10] Geoffrey C. Fox and Steve W. Otto, Concurrent Processing for Scientific Calculations, *Physics Today* **37(5)**, pp. 70-73, May 1984.

[11] G. C. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tsang and M. Wu, *Fortran D language specification*, Technical report TR 900079. Department of Computer Science, Rice University (1991).

[12] M. Flynn, Some computer organizations and their effectiveness, *IEEE Trans. Comput.*, C-21 948-960.

[13] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam, *PVM: a users' guide and tutorial for networked parallel computing*, Scientific and engineering computation series, (MIT Press, Cambridge, MA, 1994).

[14] A. Geist, J. Kohl and P. Papadopoulos, CUMULVS: providing fault-tolerance, visualization, and steering of parallel applications, *International Journal of Supercomputer Applications and High Performance Computing*, **11(3)** (1997) 224-236.

[15] W. Gropp and B. Smith, Parallel programming tools user's manual, Technical report, Argonne National Laboratory, 1993.

[16] W. Gropp, E. Lusk and A. Skjellum, *Using MPI* (MIT Press, Cambridge, MA, 1994).

[17] W. Gropp, E. Lusk, N. Doss and A. Skjellum, A high performance, portable implementation of the MPI specification of the MPI message passing standard, *Parallel Computing*, **22** (1996) 789–828.

[18] W. Gropp and E. Lusk, Why are PVM and MPI so different?, in: *Recent advances in parallel virtual machine and message passing interface, Proceedings of the 4th European PVM/MPI Users' Group Meeting*, (Lecture notes in Computer Science, Springer, November 1997).

[19] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, W. Nitzberg, W. Saphir, M. Snir, *MPI: the complete reference - 2nd Edition, Volume 2 - The MPI-2 extensions*, Scientific and engineering computation series, (MIT Press, Cambridge, MA, September 1998).

[20] R. Hempel, H.-C. Hoppe and A. Supalov, *PARMACS 6.0 library interface specification*, GMD internal report, Sankt Augustin (December 1992).

[21] R. Hempel, A. Hey, O. McBryan and D. Walker, editors, Special issue: message passing interfaces, *Parallel Computing* **20(4)** (1994).

[22] C. A. R. Hoare, Communicating sequential processes, *Commun. ACM*, **21(8)** (1978) 666–677.

[23] High Performance Fortran Forum, Special issue: High Performance Fortran language specification (version 1.0), *Scientific Programming*, **2(1/2)** (1993) 1–170.

[24] A. Kolawa and B.Zimmerman, *CrOS III manual*, Caltech C3P-253, 1986.

[25] The MPI Forum, Special issue: MPI: a message passing interface standard, *The International Journal of Supercomputer Applications and High Performance Computing*, **8(3/4)** (1994).

[26] P. Pacheco, *Parallel programming with MPI* (Morgan Kaufmann, San Francisco, 1997).

[27] Parallel Computing Forum, PCF parallel Fortran extensions, *Fortran Forum*, **10(3)** (1991) 1–57.

[28] Parasoft Corporation, *Express C user's guide*, (Pasadena, CA, 1990).

[29] P. Pierce, The NX message passing interface, *Parallel Computing* **20(4)** (1994) 463–480.

[30] Charles L. Seitz, The Cosmic Cube, *Communications of the ACM* **28(1)**, (1985) 22–33.

[31] A. Skjellum, S.G. Smith, N.E. Doss, A.P. Leung and M. Morari, The design and evolution of Zipcode, *Parallel Computing* **20(4)** (1994) 565–596.

[32] M. Snir, S. Otto, S. Huss-Lederman, D. Walker and J. Dongarra, *MPI: the complete reference*, Scientific and engineering computation series, (MIT Press, Cambridge, MA, 1996).

[33] M. Snir, S. Otto, S. Huss-Lederman, D. Walker and J. Dongarra, *MPI: the complete reference - 2nd Edition, Volume 1 - The MPI core*, Scientific and engineering computation series, (MIT Press, Cambridge, MA, September 1998).

[34] V. Sunderam, PVM: a framework for distributed computing, *Concurrency: Practice and Experience* **2(4)** (1990) 315–339.

[35] R. Thakur, E. Lusk and W. Gropp, *Users' guide for ROMIO: a high performance, portable MPI-IO implementation*, Technical Memorandum ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory (July 1998).

[36] D. Walker, *Standards for message passing in a distributed memory environment*, Technical report ORNL/TM-12147, Oak Ridge National Laboratory (August 1992).

[37] D. Walker, The design of a message passing interface for distributed memory concurrent computers, *Parallel Computing* **20(4)** (1994) 657–673.

[38] P.H. Welsh and D.C. Wood, The Kent Retargetable occam Compiler, *Parallel Processing Developements - Proc. of WoTUG***19**, pp 143-166, IOS Press, Netherlands, 1996.

[39] H. Zima, P. Brezany, B. Chapman and A. Schwald, Programming in Vienna Fortran, *Scientific Programming*, **1(1)** (1992) 31–50.