Evaluation of Distributed Communication Systems

Ian Parsons*
Department of Computing Science
University of Alberta
ian@cs.ualberta.ca

Abstract

Selecting a distributed communication system is a balancing act. Ease of use, efficiency of the final product, and future needs are not mutually exclusive considerations. Several questions spring to mind immediately. What are the requirements of the communication system? What is available to use? What are the requirements of the user? Software engineers desire software to have clean interfaces and to make any internal details inaccessible to the other components. Performance users want the software to be scalable, efficient, and easy to use, read, and debug. This paper examines four communication systems (ISIS, NMP, PVM, and Concert/C) from these two points of view.

Several programs designed to test the performance of the communication system are used as examples for comparing features. These programs are neither definitive test programs nor do they have complicated communication structures. Rather, they are used to highlight potential problem areas and implementation differences.

All four of these communication systems are being evaluated for the Enterprise project. Enterprise is a programming environment for developing and running distributed parallel programs on a network of workstations. Enterprise, using templates and a precompiler, constructs a software layer around the application. The user is shielded from the tiresome low-level details of hand crafting the distributed communication portion of the application. Other software tools within Enterprise manage and monitor the distributed program.

For two of the communication systems (ISIS and NMP) the code was created using Enterprise. The other two systems (PVM and Concert/C) are compared using handcrafted code. A comparision between the Enterprise code and handcrafted code is done using NMP.

1 Introduction

Choosing a general communication kernel for distributed computing is difficult. Frequently, one aspect of a particular package is attractive or sufficiently well done while other parts are deficient or difficult to use. Using an existing kernel upholds sound software engineering tenets: reuse of existing components and retention of the expertise gained from the system. However, the existing kernels have insufficient scope to create truly distributed programs. This is not to say that existing systems are poorly done. Rather, the problem they are attempting to solve is complex. As more and more of the problem is addressed, the existing

*The IBM contact for this paper is Jacob Slonim, Centre for Advanced Studies, IBM Canada Ltd., 81/894, 895 Don Mills Road, North York, Ontario M3C 1W3.

kernels fail to deliver the necessary capabilities.

Distributed memory parallel computing (DMPC) is developing along lines similar to the evolution of early computing. Early computing used machine or assembly language to construct a program. The program was usually not portable and was difficult to modify by other users. High level languages, along with the automatic and efficient code writing of compilers, have proven their worth. Currently, a DMPC programmer uses communication library calls to hand construct programs that are somewhat portable but still difficult to modify.

A DMPC programming environment, to follow the familiar sequential path, needs to supply the user with templates for parallelization. It should shield the casual user from the tedious details of low-level communication while providing efficient automatic code generation. Debugging, heterogeneity, visualization, fault tolerance, and process migration of programs must be supported. These concepts are different from the familiar sequential programming environment. At the root of the parallel programming environment lies the communication kernel.

Today DMPC has relatively few programmers compared to those involved with sequential programming. There is a sizeable investment in developing DMPC skills. Presenting the casual user with a familiar sequential programming environment is a first step. Shielding the tedious details permits the user to concentrate on developing the parallel algorithm.

This paper looks at four communication kernels and their potential use with Enterprise [9] – a DMPC programming environment. The four systems were chosen based on their approach to a communication kernel. Enterprise has been implemented on top of two communication kernels (ISIS and NMP) with the other two as potential candidates. This comparison looks at program performance, ease of use, and scalability. None of the four systems evaluated is currently suitable for a fully developed DMPC programming environment.

2 Communication Kernels

Four communication kernels were selected. Three are traditional software libraries that supply the user with functions to use when passing messages. Each library tackles the problem from a different perspective. The fourth kernel extends the familiar sequential C language. The user's code is compiled by the kernel prior to conventional compiling.

Software libraries are traditional methods for adding new capabilities to a programming language. The user makes function calls to perform these tasks. The language remains unchanged.

NMP takes the minimalist approach – essentially one thin layer above sockets. Typically, the user creates a library of routines that buffers the programmer from NMP. PVM concentrates on providing support for heterogeneous architectures and creation of a virtual parallel machine. PVM has an interface that is intuitive to learn. ISIS, with its distributed cluster of servers, concentrates on scalability, fault tolerance, and logical process groups. The success of each of these approaches lies in its implementation.

The fourth kernel, Concert/C, takes a different approach. It has added extensions to the C language and uses a compiler to create the user's applications. This approach deeply buries the communication primitives in the resultant programs. The user defines a series of function interfaces while Concert/C inserts all the necessary low-level details. Conceptually, there is no difference in semantic behaviour for the user between a sequential function call or one that is located on a different processor.

A brief background discussion of the four communication kernels is presented.

2.1 NMP

Network Multi-Processor (NMP) is a communication system developed at the University of Alberta in the 1980s (Marsland et al. [10]). It is currently used in the teaching of undergraduate courses. It is a simple layer over the UNIX socket communication primitives. As such, it provides some of the requirements for a DMPC programming environment.

NMP provides a set of library routines that launch processes, establish communication ports between processes, transmit messages over established ports, and convert between big-endian and little-endian architectures.[1] NMP can work with heterogeneous architectures but all the responsibilities for correct translation lie with the user. All processes are launched prior to the program starting run. Physical machines and all communication patterns are established in the user-specified configuration file. Processes are launched using the Internet services daemon (**inetd**) and the **nmpserver** process.

To send a message to another processor involves writing to a logical identifier which NMP maps to a physical device. The number of connections a process may have is limited by both the operating system and NMP. (It currently stands at 64.) These connections are made during the startup and may be disconnected but are not reconnected. Unfortunately, dynamic addition of processes is not robust and unlikely to be fixed. Thus, the scalability of NMP is compromised.

A message is sent using either the TCP/IP or UDP protocols. The tested version used TCP/IP. Since the message is sent via a socket, an arbitrary message can easily fill the system buffer. When the buffer is filled, the sender is blocked until the buffer is consumed by the receiver. However, if the receiver is waiting on another port or is trying to send on the same port, deadlock is achieved. This provides a good learning experience for students but an unnecessary technical complication when developing parallel algorithms.

Because it is a thin layer over the socket primitives, NMP has very little overhead and potentially can extract the best performance for an application. Its use of the **inetd** daemon to launch other processes reduces the consumption of the root processor resources (process slots). However, NMP is deficient in that deadlock is easily accomplished and not easily detected.

2.2 PVM

Parallel Virtual Machines (PVM) is a communication system that is becoming widespread on many machine architectures (Geist and Sunderam [5]). PVM has many desirable features that make it a good candidate for DMPC: simplicity, robustness, and portability. A daemon (**pvmd3**) coordinates communication and process launching on each host machine that makes up the virtual machine. PVM coexists with other communication systems. There is support for heterogeneity at the simple variable level with the user writing translating functions to read or write more complex data (arrays or structures) from and to the transmission buffer. There is a mechanism for dynamic grouping of processes. A message to a particular process also can be typed with an integer value (a user interpreted value).

Processes are separated logically from their physical machine addresses. Scalability is limited by PVM 3.0 to 4,096 hosts and 262,143

[1] Big-endian or little-endian refer to the byte ordering for numeric values.

tasks per host, which is sufficient for most applications. However, the operating system may also impose limits. For example, the workstations used for the experimental work have a limit of 138 processes or tasks allowed at one time and can only connect 63 hosts into one PVM.

PVM uses UDP for communication between the **pvmd3** daemons composing the virtual machines. The user messages are sent using TCP/IP.

2.3 ISIS

ISIS [8] is based on a distributed network of servers that globally constitute the kernel. A process connects to a server as a member of an existing or new group. A group name is an address to which other processes may send messages. If a server crashes (not unusual), the connected programs immediately fail. There is no chance of transferring to another server.

ISIS is truly scalable. The number and location of connected processes is irrelevant to the kernel. This is accomplished by using global logical names to define groups. A group may have one or more members. Messages are sent to groups. A process may leave or join a group at any time. The flaw in having a global view is that different processes may use similar names for completely different group functions. An example would be a process that collects results. A logical name would be **collector**. If one collector expects to read a numerical matrix and another collector expects to read a character string then the incompatibility of message content makes for interesting program behaviour. This naming conflict can occur with two copies of the same program running simultaneously.

Since ISIS is built upon UDP packet transfer, there is a considerable effort expended assembling and maintaining the temporal ordering of message lists. Fortunately, this is hidden from the user.

A message is similar to writing to a device. The buffer sent to the remote process is composed of base types (int, double, char, etc.) and vectors of base types. The user can either cast structures as one of these base vector types or decompose the structure to its base components. The recipient of a message can forward it to another process without knowing the details of the contents. An example where this would be useful is a generic *manager* process that passes messages to idle individual processes of another group.

Process launching is the responsibility of the user. ISIS provides no mechanism for this. Enterprise was initially built on top of ISIS but serious implementation problems in ISIS forced it to be abandoned. The details are explained in more depth in [14]. The ISIS versions tested are 2.2.5 and 2.2.7.

2.4 Concert/C

Concert/C (Auerbach et al. [2, 6]) is an extension to the C language. The use of new keywords conflicts with the Enterprise intention to be independent of the communication kernel. The user must be aware of the keyword conflict and the new compiler restrictions (ANSI C only).

Concert/C provides all the necessary libraries for interprocess communication. Messages are marshalled, sent, received, and demarshalled into the user's variables. If necessary, memory space is allocated. However, it then becomes the user's responsibility to deallocate.

One excellent feature of this package is the specific definition of the interface between communicating processes. An interface is presented as a function call with specific parameters defined by the user. This means that a process receiving a message does so by providing the function. This function or *port* may be passed to other processes, providing a clean and unique interface address for interprocess communication.

For the purposes of this paper, only the asynchronous send and receive functions were used for message passing other than in the initialization phase. This is necessary since Enterprise employs *lazy synchronization.* Lazy synchronization allows the sending process to continue program execution after the message send until a point is reached where data is needed from the reply message. The process then blocks until the message is received.

Concert/C synchronous functions calls were not examined in any detail. One feature that is important when making distributed function calls is the notion of parameter direction. Each variable in a function parameter list can have an `in` or `out` direction. This is new to sequential programmers. An `in` parameter corresponds to a value in an incoming message to the function while an `out` parameter is placed in the return message. Reducing message size makes for efficient resource utilization.

One aspect of Concert/C should be pointed out. The design of Concert/C is such that a variety of underlying communication mechanisms are available. The test results presented in this paper are based on the Sun RPC communication mechanism only.

The version of Concert/C used in this study is Alpha 2.

3 Desirable Properties

Any communication kernel (CM) used for DMPC should address the following properties: scalability, heterogeneity, process naming, dynamic process allocation, ease of use or learning, and performance. The CM does not have to necessarily provide the property but it must be addressed in the design.

Table 1 shows the four communication kernels with a rating for each property. A neutral rating means the property is partially acceptable. A negative rating implies the system is deficient in that area while a positive rating indicates the property is acceptable. These ratings are subjective and reflect my opinion after working with the systems.

3.1 Scalability

The implementation of a CM should not be limited by constraints imposed by the operating system or network. A process that wants to connect to one process should also be able to connect to 1,000 or more processes. This does not necessarily mean *direct* communication links (although that would be ideal) but rather that the user is shielded from such mundane details. The performance penalty for this scalability should not be significant.

NMP is limited by the number of file descriptors available for an individual process. The limitation is a function of the operating System. PVM has a limit of 4,096 hosts in one virtual machine. The SunOS kernel used for the experiments allows only 64 file descriptors per process and 138 process slots per workstation. Thus, PVM is limited to the total number of tasks per workstation and the total number of hosts while NMP is limited to having any one process connected to 61 others (stdin, stdout, and stderr are present). NMP is not limited by the number of hosts.

ISIS can have an arbitrary number of processes connected since it uses three communication ports for message passing. Those ports, however, must be globally visible to all connected ISIS servers. It is possible to have different ISIS servers operating with different triples of port numbers but they are invisible to each other.

The Enterprise implementation of ISIS uses **rsh** to launch processes. The operating system puts a limitation on the number of processes

Property	ISIS	NMP	PVM	Concert/C
Scalable	+	=	+	+
Heterogeneity	–	–	+	+
Process Naming	+	–	+	+
Dynamic Processes	+	–	+	+
Learning Curve	=	+	+	–
Fault Tolerance	+	–	–	–
Performance	–	+	=	+=

(+ means positive, – means negative, = is neutral.)

Table 1: Summary of the communication kernel properties.

that could be launched. ISIS itself is not limited.

Concert/C currently uses **rsh** to launch processes from within a program. This puts a limit on the number of processes due to the operating system. Interprocess communication is limited by the amount of virtual memory available. Communication ports are allocated as needed and messages may be multiplexed on a given port.

3.2 Heterogeneity

Heterogeneity between different architectures must be addressed. It cannot be expected that a single type of architecture is used when parallel programs are run. Ideally, since the CM is already running on the different platforms it should take the responsibility for heterogeneity.

PVM takes the responsibility as far as base types (integer, floating-point, character, byte) and vectors of base types. If the user wants to send some structure or composite message, the structure must be broken down to base types by the user. This is relatively simple to do if the DMPC programming environment uses a pre-compiler to wrap the user's code with the necessary communication primitives. A pre-compiler has access to the symbol table and can write the necessary code fragments to do the packing and unpacking from the message buffer. PVM also has a mechanism for the user to bypass the conversion process if desired.

NMP does not provide any more than macros that convert between big-endian and little-endian architectures. ISIS does not provide services for heterogeneity.

Concert/C converts all variables to the *universal* format it uses for messages. The receiver would then translate from the universal format to the local format. The user has little to say about whether this is to be used. The user must define or *annotate* [6] the interface. This provides enough information to allow the messages to be marshalled and demarshalled in the correct format.

3.3 Process Naming

The CM must provide a separation between logical and physical machine addresses. The user should be able to send a message to a logical address. Independently, the CM would translate the address to an actual physical address. This permits portability of the code and dynamic processor allocation. It also implies that when multiple copies of the same parallel program are running, logical names are globally unique.

NMP uses logical unit addresses that are unique to the program. Since communication is established prior to passing control to the user's program, multiple copies of the program can be run without interference.

PVM assigns an identifier (TID) to each process that joins the parallel virtual machine. This TID is global and unique to the current user's PVM. This is similar to the familiar sequential environment. Messages are sent to this TID without regard to the physical location of the processor. Multiple copies of the same program can run without fear of interfering with each other.

ISIS uses global logical group names that are supplied by the user. If the user is not imaginative in picking names, there will be a conflict with messages sent to the group name. An example of this was given earlier in Section 2.3. The DMPC programming environment must supply dynamic group names to eliminate such naming conflicts.

Concert/C maps function ports to processes on specific processors. The concept of a process is avoided. PVM has a similar behaviour since a process can receive or send a particular message type from a specific process. The message type is just an indicator and the message contents are not enforced. However, PVM relaxes either restriction if the user wishes. Concert/C does not permit this relaxation at all. Ports are typed as to contents and assigned to specific processes.

3.4 Dynamic Process Connection

Processes can be added or deleted dynamically. Having a static communication structure is not useful unless the user intends to use specific processors regardless of their dynamic state. With one or more networks of workstations available, the program should be allowed to increase or decrease the number of processors currently working on the task, depending on the dynamic state of the network and processors.

NMP is the only CM tested that uses a static communication structure. Processes may leave the program prior to the program finishing but adding processes is supported only under limited conditions.

Dynamic process allocation is necessary if process migration and fault tolerance are wanted. In these cases, the CM must provide a clean interface for disconnecting and reconnecting to the parallel program. This becomes complicated if checkpointing and migration are used for dynamic load balancing. The CM and operating system extensively cooperate with the DMPC environment in order to cleanly migrate process without user intervention. A process halted in the middle of a message transfer or system call is difficult to restart correctly on the same platform, much less on a different platform. The CM cannot be relied upon to provide all the service since messages are not always sent regularly or frequently. This is a very difficult problem to implement in the general case.

PVM, Concert/C, and ISIS all support dynamic process connection. However, at this point, none of them provide the facilities for process migration. In the future, developers of these products should address this problem.

3.5 Learning Curve

How easy is it to use the four CMs when developing distributed applications? This subjective metric has two components. First, what needs to be learned to create a distributed program? Second, how complicated is the code to write.

NMP and PVM are very similar. There are relatively few function calls with clearly defined interfaces. A large part of these two systems is hidden from the user. A process must connect to the other processes or virtual machine. Data is written to a buffer and sent out on a port to another process. A process must explicitly exit the program if an error is not to be propagated to the other processes. A process may poll a port associated with another process to see if any messages have arrived.

At first glance, ISIS appears to be as simple to use. ISIS provides all of above functions except that ISIS does not provide facilities for launching processes. The user is responsible for this. However, ISIS provides the user with many more choices than the previous two CMs. For example, ISIS provides several different message sending primitives for the user. These primitives allow messages to be received in the order they are sent out or not.

NMP has no concept of message types. The connection is just a byte stream. PVM has the user pack and unpack the buffer with specific data types and the user can send or receive messages with a tag value accompanying the message. ISIS messages are packaged by a format statement indicating to ISIS how to pack or unpack the data. None of these packages have explicit enforcing of the structure of the message. Programmers must be careful when matching the send and receive portions of their code. A DMPC programming environment should enforce the matching of message types for the user.

Concert/C has the user precisely define the interface for the port. This includes parameter directions (in or out), type definitions, and marshalling information (how big, how many). Once the effort is made to define the interface, Concert/C transparently packages and sends the message off to a properly initialized port. Exchanging of message ports is usually done either at startup or via public posting in a file. The startup exchange is done by a master/slave relationship between the processes. Arbitrary connections are easiest through a common file.

More thought must go into designing a Concert/C application than the other three. The implementation details are more complicated in that message definition and port connection concepts are mixed together. This differs from the more traditional approach of consider connection details first and then worry about message contents. Concert/C has the user worry about the definition of the port and the distribution of the port addresses. The low-level connection details are hidden from the user.

Creating the correct code by hand for one of the algorithms discussed in this paper took about one day of additional effort using NMP or PVM. This was after the sequential portions of the code were written and debugged. The Concert/C applications took longer because of two items. First, the CM was alpha software and problems that arose had to be checked carefully to locate where the fault lay. Second, the language extension aspect required some experimentation – unlike using a software library.

Of course, all these problems vanish when a programming environment like Enterprise, which inserts the necessary code for distributed communication is used. The applications discussed in this paper took about twenty minutes to implement, after the sequential portions were written, using Enterprise. The ISIS and NMP applications were created using Enterprise. Hand-generated NMP applications were also created to check the efficiency of the Enterprise code.

3.6 Performance

The implementation of the CM should provide efficient message passing. That is, the overhead of all the above items should not seriously impair the primary function of the CM

– guaranteed fast delivery of the user's messages. The remainder of this paper looks at the performance of three algorithms that were implemented using the four CMs.

4 Evaluation

Three algorithms were implemented using the four kernels – block matrix multiplication, chaotic Gauss-Seidel, and alpha-beta search of random trees. These programs are not to be considered definitive test cases. However, the performance and implementation details are informative.

Five versions were written. Two versions were created by Enterprise (ISIS and NMP) and three versions were hand coded (NMP, PVM, and Concert/C). The two NMP versions were done to examine the cost of the Enterprise manager and the effectiveness of the automatic code generation.

Enterprise uses a manager process that collects and forwards messages to replicated processes. The manager will forward a message to an *idle* process if available. The hand coded versions do not have a manager. In this case, messages are sent out in sequence to the replicated processes. This works only if there are not many messages sent out and the workload is evenly distributed. Alpha-beta search satisfies neither of these conditions and the Enterprise version of the algorithm shows better performance.

All experiments were run on a quiet network of twenty Sun IPCs with local swap space, 12 MBytes of main memory running SunOS 4.1.3. The network is a 10 MBS Ethernet. All results are the best of at least five runs.

Speedup is calculated as the ratio of the user time for the sequential algorithm against the elapsed time for the parallel algorithm. Timing results were obtained using the **getrusage()** and **gettimeofday()** Unix functions. The processor clocks are synchronized by the **xntpd** time daemon.[2]

4.1 Block Matrix Multiplication

One parallel technique for matrix multiplication is to split the two matrices into compatible blocks and then multiply these smaller blocks independently. The product of this multiplication is then added to the existing answer matrix.

Two matrices $A \in R^{M \times N}$ and $B \in R^{N \times O}$ can be divided into blocks of submatrices to solve

$$C = AB$$

where $C \in R^{M \times O}$ by using the formula

$$C_{ij} = \sum_{k=1}^{N/b_N} A_{ik} B_{kj}$$

where $i = 1 : M/b_M$, $j = 1 : O/b_O$ and b_M, b_N, and b_O are the number of processing blocks of each dimension [7]. The advantage in a parallel environment is that these blocks of submatrices can be multiplied in parallel by a conventional sequential method. The addition or merger of the product of these two matrices can be done by another processor. The cost of communication of the blocks is balanced against the computational price of the matrix multiplication.

The program is a pipeline of three processes: a make-work process which divides up the two matrices into blocks and distributes the work, a process which multiplies the two blocks, and a collector process. The collector process merges the submatrix into the larger global matrix, performing the necessary additions. No process expects a reply to any message. The second process is replicated to parallelize the program.

[2]**Xntpd** is a complete implementation of the NTP version 3 specification as defined in RFC 1305. David L. Mills. University of Delaware, March 1992.

Matrix Size	Replication: four Enterprise ISIS	NMP	Hand-coded NMP	PVM	Concert/C	Replication: ten Enterprise ISIS	NMP	Hand-coded NMP	PVM	Concert/C
100	0.5	0.6	0.6	0.7	0.7	0.4	0.3	0.3	0.6	0.7
200	1.5	1.6	3.1	2.4	3.1	1.4	2.1	1.1	1.5	4.4
300	1.8	2.4	3.1	3.1	3.2	1.6	2.8	2.8	3.1	3.4
400	1.9	2.5	3.3	3.4	3.4	1.7	2.9	4.3	3.3	5.5
500	1.9	2.5	3.3	3.5	3.3	1.7	3.1	5.5	3.1	6.8
600	2.0	2.6	3.5	3.6	3.4	1.6	3.1	6.5	3.3	7.5

Table 2: Speedup for block matrix multiplication using the four communication kernels.

The number of messages is a function of the size of the two matrices. The dimension divided by the block size gives the number of messages. The block size used is a 100×100 matrix which yields a message size of 160,032 bytes sent to the worker and half that size sent to the collector process. To multiply two matrices of this size uses between one and two seconds of processor time to calculate, giving a good balance between message passing and processing. Larger block sizes showed a slower consumption of outstanding messages, while smaller block sizes resulted in processor underutilization.

The pipeline configuration multiplied two square matrices of size 100, 200, 300, 400, 500, and 600 together using four and ten replication factors. Table 2 contains the results. The heavy contention on the network resources by this algorithm is shown by the relatively poor performance by PVM and the two versions of Enterprise. In these cases, the PVM daemons and the Enterprise managers need network resources addtional to the actual messages generated by the algorithm. Collisions and other conflicts limit the potential speedup.

The Concert/C version is better than the handcoded NMP version since Concert/C is able to simulate a multi-threaded process. A message is queued for sending and control is returned to the process immediately. NMP must wait until the entire message is placed in the system buffer for the socket before returning control. In this case, the system buffer is limited to a maximum of 52 kbytes, which is considerably smaller than the actual message size. PVM was faster with a smaller number of messages than NMP for the similar reasons.

ISIS has a serious implementation flaw which contributed to its lacklustre performance in this case. All the messages sent from the creator-of-work to the manager of the replicated processes for forwarding to a specific worker were retained by the manager despite the use of the ISIS message deallocation routines to free the consumed messages. The size of the problem became severely limited since the manager processor would run out of virtual memory. None of the other kernels showed this limitation. NMP and Concert/C do not need any network resources other than to send the actual messages. In this case, maximal throughput is achieved.

5 Chaotic Gauss-Seidel

The Gauss-Seidel algorithm, found in many matrix algebra texts, is an iterative procedure used to solve the family of equations

$$Ax = b$$

Problem	Replication: four					Replication: ten				
	Enterprise		Hand-coded			Enterprise		Hand-coded		
Size	ISIS	NMP	NMP	PVM	Concert/C	ISIS	NMP	NMP	PVM	Concert/C
1000	2.7	3.0	2.9	3.1	2.3	4.8	4.0	2.0	2.4	1.4
2000	0.7	2.5	3.4	3.6	1.7	4.8	7.6	6.0	7.7	4.4

Table 3: Parallel speedup for chaotic Gauss-Seidel using the four communication kernels.

where A is the $N \times N$ coefficient matrix, x is the $1 \times N$ vector of unknowns, and b is the $1 \times N$ vector of constants [1].

Baudet [3] suggested various ways of parallelizing this algorithm. He proposed that less synchronicity between parallel components will result in better speedup since processors do not have to wait for slower processors. The drawback is the non-determinism of the answer. This is readily apparent from the experiments. The answers were not the same from run to run, but they were similar and met the user-specified tolerance.

To parallelize this algorithm, the answer vector is divided into independent blocks, each processed by one processor. Hence, the number of blocks is the degree of parallelism. Relaxing the requirement that all blocks sent out must return before the next round of work improves performance. Idle processors are utilized rather than waiting for the slowest processor to finish. Increasing the number of blocks increases the observed non-determinism between runs.

The program is divided into three sections. The first section (one process) receives input from the user and then apportions the work and distributes it to the replicated worker processors. The third section is a single process that simulates the shared memory (the answer vector). This process is initialized by the startup process. The startup process exits after scheduling all the work.

Fixed replication is used since all the work segments must be running simultaneously in order for the program to finish. All blocks must be processed at least once to get the error estimate below the user-supplied tolerance. A processor receives a block of the solution vector to work on only when it starts up. There must be a one-to-one correspondence between processors and blocks.

The parallel results for 1,000×1,000 and 2,000×2,000 matrix sizes using four and ten replicated workers are in Table 3. The larger problem sizes show the good performance of both the NMP versions and PVM. Concert/C and ISIS have poorer performances partly due to the many-to-one direction of the communication graph. The poor speedup reported for the large problem size using four workers for ISIS is due in part to the workstations generating a large number of page faults. The ISIS binaries are large (+350 kbytes) and have significant dynamic memory requirements. This, coupled with the 8 Mbyte matrix segment stored at each workstation, caused many page faults on the 12 Mbyte machines. The time for page faults is included when calculating the speedups. The other CMs did not produce such large binaries or have sufficient memory requirements to generate significant numbers of page faults.

ISIS did not have problems with message retention since only the startup messages were forwarded. All other messages were direct sent to the intended receiver.

	Replication: four			Replication: sixteen		
Search	Enterprise	Hand-coded		Enterprise	Hand-coded	
Depth	NMP	PVM	Concert/C	NMP	PVM	Concert/C
6	2.9	2.8	2.0	7.5	5.4	4.7
7	1.2	1.0	0.7	2.9	2.2	1.7
8	2.6	2.3	1.6	6.6	4.5	3.9
9	1.4	1.2	0.9	3.6	2.8	2.0

Table 4: Parallel speedup for alpha-beta search using three communication kernels. The tree had a branching factor of 40 with the granularity of parallelism set at 3.

5.1 Alpha-Beta Search

A discussion of sequential alpha-beta search of combinatorial spaces can be found in Nilsson [13].

One way to parallelize this search is to search all the child nodes connected to a node in parallel [11]. This particular algorithm distributes work while traversing the leftmost branch of the tree. This leftmost branch is called the *principal variation* of the tree. One feature of the leftmost branch is that all nodes on this branch must be examined while other branches of the tree have the potential to be pruned from the search space. Because of synchronization and load imbalances in the search, the potential speedup for random trees is limited to $O(\sqrt{N_p})$ where N_p is the number of processors [4]. (Idle processors waiting for work are expensive.) Careful ordering of the sibling nodes gives good speedups since the limiting bounds found early in the search will result in maximum pruning of future branches. Small branching factors imply only a few processors are necessary to achieve maximal potential speedup since there is only a limited amount of work to do in parallel.

The alpha-beta search is divided into two processes. The first is the master process. It searches the principal variation and sends work to the slave processes. The algorithm for the slaves is any standard alpha-beta algorithm. When the granularity of the recursion reaches the user's threshold, this routine calls one of the individual slave processes to sequentially search a branch of the tree rooted by one of the child nodes. After all the work has been posted and received back, the waiting master sets the score (alpha or beta) for that node and the recursion unwinds. Unwinding the recursion increases the amount of work at each level being done by each slave process. Each level further up the tree should have better information available to prune any unsearched branches if the ordering is favourable.

The program used in these experiments is a simulation tool for testing different search algorithms [12] on random trees with a constant branching factor. The program inputs are the search depth, branching factor, and the probability factors. Each branch is ordered by a probability factor that controls where the eventual answer will be found. Giving a large probability factor to one branch increases the likelihood of finding the correct answer within that particular branch. Changing the probability distribution changes the search pattern.

The granularity factor is the level at which the master process would switch to parallel mode. For example, a granularity of three means the master would recursively traverse down the tree for three levels to make a call to the conventional alpha-beta process (on another machine) ending the recursion. Then, unwinding the recursion, the master process calls, in parallel, the alpha-beta search for all the sibling nodes. The lower the number the sooner the master node stops the recursion; less parallelism is used. The experiments shown in Table 4 had the granularity fixed at three.

The experiment varied the depth of search and number of processors for a tree of width forty with a bias to having the solution being found in the left hand part of the tree. Table 4 contains the experimental results for four and sixteen workers.

The algorithm shows an interesting performance characteristic when the depth of search is increased. Even depth searches have better results than the odd depth searches. The number of nodes searched in even depth searches is nearly four times the number searched in the seqential version. The odd depth searches only examine 1.5 times the number of nodes. Closer examination of the individual work segments reveal that the odd depth searches generate minimal work (nodes searched) implying the search is benefitting from maximal cutoff. In this case, communication costs will dominate timing results. The even depth searches distribute work to the processors; the communication costs have reduced significance. Despite doing more work, the parallel version performs much better than the sequential version for even depth.

The ISIS version is not shown since there is a marked performance degradation when problems of this size are solved. There is a threshold that after 50 - 100 messages, the time for message delivery takes a very sharp jump from 0.2 seconds to 1.2 seconds [14]. A second reason was a restriction in the maximum number of processes (13) that could be launched for an application. This limitation was a function of the operating system in use at that time.

The Enterprise version of NMP shows the best performance with increasing search depth, since the manager passed work onto idle processors. The PVM and Concert/C versions just cycled around the list of workers, whether they were idle or not. However, when a process returned a value, new work could be waiting immediately. For small number of workers the managed pool of workers and the uniformly distributed work have essentially the same values. For large pools, the manager is effective.

6 Conclusions

Three algorithms were implemented using four communication kernels. None of the kernels are ideal for a DMPC programming environment. The reasons are based on both the design of the CM and the implementation.

NMP, as the minimal kernel, generally gives the best performance. There is very little overhead involved with this package. It is robust and delivers exactly what it promises. The one drawback to using NMP is the lack of dynamic process connections. NMP does not directly support heterogeneity but a software layer could be constructed. The scalability is limited by the operating system constraints, not by internal NMP constraints.

PVM has imposed limits for true scalability, which are beyond the needs for most applications. However, the operating system for a specific processor may impose stricter limits. The interface provides sufficient support for heterogeneity. Dynamic process addition is easily done using PVM. The latest addition of dynamic groups shows the careful and steady growth of this product.

The performance of PVM is good. The overhead of heterogeneity and maintenance of the virtual parallel machine contribute extra overhead compared with NMP. PVM is a more complex and robust product than NMP. Its widespread acceptance and portability across many different architectures speaks well of its design and the demand for such a product.

ISIS is the most scalable of the kernels presented. Its design is excellent but its implementation over several versions has shown several serious implementation flaws. Failure to deallocate consumed messages and arbitrary overwriting of stack variables are just two that have persisted through several releases. These implementation faults contributed to the poor performance for the various algorithms discussed in this paper. ISIS is more complicated to learn than NMP. The greater complexity of ISIS is due to it offering more choices for the user. This provides more flexibility but takes longer to master. Heterogeneity is not addressed by ISIS. There is support for fault tolerance.

Concert/C extends the C language, which is not compatible with the current Enterprise model. Heterogeneity support, strong typing of message ports, and no serious limitations to scalability make Concert/C a good candidate for a communication kernel for DMPC. As the current version is an alpha release, its use with Enterprise will be at the research level. The performance will improve with the maturity of the product.

Using a generic manager process to forward work to idle processors is effective if there are idle processes. Evenly distributed workloads (block matrix multiplication, chaotic Gauss-Seidel) show that the manager process actually contributes to poorer performance. In the case of alpha-beta search, the utilization of idle workers improves performance.

These different communication kernels demonstrate both the size and complexity of the problems facing DMPC. They address different aspects of the problem with varying success. It is clear that much more work is necessary before a complete DMPC programming environment is possible.

7 Acknowledgments

This work is supported by a CAS PhD. Fellowship. I would like to thank the Department of Computing Science Instructional Support Group at the University of Alberta for allowing the use of the undergraduate computing facilities. I would also like to thank the support groups at ISIS, PVM, and Concert/C for their quick responses to my questions.

About the author

Ian Parsons is in the PhD. program at the University of Alberta. His main interest is in programming environments for distributed parallel applications. He can be reached at the address Department of Computing Science, Room 615 General Services Building, University of Alberta, Edmonton, Alberta, T6G 2H1.

References

[1] Howard Anton and Chris Rorres. *Elementary Linear Algebra with Applications.* John Wiley and Sons, Toronto, 1987.

[2] J. Aurebach, M. Kennedy, J. Russell, and S. Yemini. Interprocess communication in Concert/C. Technical Report RC 17341, IBM T.J. Watson Research Centre, Yorktown Heights, 1992.

[3] G.M. Baudet. *The Design and Analysis of Algorithms for Asynchronous Multiprocessors.* PhD thesis, Department of Computing Sciences, Carnegie-Mellon University, 1978.

[4] J. Fishburn. *Analysis of speedup in distributed algorithms.* PhD thesis, Computer Sciences Department, University of Wisconsin-Madison, 1981.

[5] G.A. Geist and V.S. Sunderam. Network-Based Concurrent Computing on the PVM System. *Concurrency: Practice and Experience*, 4(4):293–311, June 1992.

[6] Arthur P. Goldberg. *Concert/C Tutorial: An Introduction to a Language for Distributed C Programming*. IBM T.J. Watson Research Center, Yorktown Heights, NY, March 1993.

[7] Gene H. Golub and Charles F. Van Loan. *Matrix Computations: Second Edition*. Johns Hopkins University Press, Baltimore, Maryland, 1989.

[8] Isis Distributed Systems, Inc., Ithica, N.Y. *The Isis Distributed Toolkit: Version 3.0, User Reference Manual*, 1992.

[9] Greg Lobe, Paul Lu, Stan Melax, Ian Parsons, Jonathan Schaeffer, Carol Smith, and Duane Szafron. The Enterprise Model for Developing Distributed Applications. Technical Report TR 92-20, Department of Computing Science, University of Alberta, 1992.

[10] T.A. Marsland, T. Breitkreutz, and S. Sutphen. A Network Multi-processor for Experiments in Parallelism. *Concurrency: Practice and Experience*, 3(1):203–219, 1991.

[11] T.A. Marsland and M.S. Campbell. Parallel search of strongly ordered game trees. *Computing Surveys*, 14:533–551, 1982.

[12] T.A. Marsland, Alexander Reinefeld, and Jonathan Schaeffer. Low overhead alternatives to SSS*. *Artificial Intelligence*, 31(1):185–199, 1987.

[13] Nils J. Nilsson. *Problem-solving methods in Artificial Intelligence*. McGraw-Hill, Toronto, 1971.

[14] Ian Parsons. An Appraisal of the Enterprise Model. Master's thesis, Department of Computing Sciences, University of Alberta, 1993.