A Performance Assessment of Express on the iPSC/2 and iPSC/860 Hypercube Computers

Ishfaq Ahmad
Department of Computer Science
Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong
email: *iahmad@cs.ust.hk*

Min-You Wu
Department of Computer Science
State University of New York, Buffalo, NY

Jaehyung Yang and Arif Ghafoor
School of Electrical Engineering
Purdue University, West Lafayette, IN 47907

Abstract

This paper describes the performance evaluation of Express programming environment on the iPSC/2 and iPSC/860 hypercube computers. Express allows parallel programs to be developed in a completely portable fashion and is available on most commercially available parallel computers as well as networks of workstations. We have developed a set of benchmarks to make a comprehensive performance assessment of the frequently used communication primitives of Express on hypercube computers. A comparison with the equivalent iPSC/2 and iPSC/860 primitives is also carried out. In addition, we have developed and benchmarked a suite of real applications including three different versions of Gaussian elimination, fast Fourier transform, and the N-body problem. These results help us evaluate the performance of Express and judge its merits against the extra overhead incurred due to portability. The results also enable us to make a comparison of two generations of hypercube systems. Furthermore, algorithm developers can benefit from these results by using the times required by the basic communication primitives on both hypercubes with and without Express, and can automate performance estimation by using these timings.

1 Introduction

During the past few years, we have witnessed an unprecedented growth of parallel computing hardware platforms. This is because a variety of architectures have been designed exploiting advancements in processors technology, low overhead switches, faster communication channels, and rich interconnection network topologies. With the advent of the High Performance Computing and Communication initiative and the quest for achieving a teraflop computing speed, this trend is likely to continue [12]. However, the advances in software technology for parallel computers have been outpaced by the spectacular progress in hardware. Consequently, programming of parallel computers still remains a tedious task. Parallel computing is not likely to be of common use until considerable efforts are invested in designing parallel programming environments, CASE (Computer Aided Software Engineering) tools, parallelizing compilers, debuggers, and performance-monitoring tools. The lack of desired progress in software for parallel computers might be attributed to the fact that there is no unified programming model and that parallelism can be achieved with a variety of paradigms. There also exists the problem of portability - it is distressing to repeat the implementation work on a new machine. A few noteworthy efforts have been made to deal with such issues. These include PVM [32], Linda [10] PICL [21] and Express [29]. These systems allow parallel programs to be developed using C or Fortran by including their message-passing library routines.

Express from Parasoft Corporation is a software programming environment for writing parallel programs for MIMD multiprocessors. Programs can be written using SPMD or pure MIMD paradigms for shared-memory and message passing multiprocessors as well as distributed multicomputers. Express provides a communication system for communicating processes, mechanisms for data sharing, reading files, debugging tools, and performance analyzing tools. An important feature of Express is that these functionalities are carried out in a user transparent manner. In addition, it has features such as automatic domain decomposition library which can map physical domain of the problem onto the underlying topology of the parallel or distributed computing system. The performance evaluation tools, using text and graphics, can be effectively used to analyze the run-time performance of the program.

Express has been implemented on a variety of machines including Cray, Alliant, BBN Butterfly, nCUBE, Symult, Intel iPSC/2 and iPSC/860 hypercubes, Intel Paragon, IBM370/3090, Thinking Machine's CM-5, KSR-1, and transputer arrays. The network version of Express allows a network of workstations to be used as a virtual parallel machine. It has been implemented on DEC, HP, IBM/RS6000, SGI, Sun and PC's. Reportedly, over 1000 sites world-wide use Express. From user's stand point, Express is completely portable. That is, a program written for one machine can run on any other machine that has Express without any sort of modification in the source code. The flexibility of Express makes it an attractive tool set for developing and running parallel programs on different platforms. Languages supported by Express include C and Fortran.

A key measure of the usefulness of programming tools like Express is the execution speed of the basic primitives which are frequently used in programs. Since Express provides portability with a rich set of utility functions, the cost is larger overhead incurred due to the translation of high level primitives to the local machine level functions. We have evaluated the performance of some of the basic primitives of Express, running on the iPSC/2 and iPSC/860. This performance is influenced not only by the implementation of Express primitives but also by the underlying hardware of the two systems. These performance results are important for a number of reasons. They help us evaluate the performance of Express and assess its merits against the extra overhead incurred due to portability. In addition, we are able to make an in-depth comparison of two generations of hypercube systems. Furthermore, algorithm developers can benefit from these results by understanding the overheads incurred by basic communication primitives on both the iPSC/2 and iPSC/860 hypercubes with and without Express, and can automate performance estimation by using these timings. These results can also enable simulator developers to use more accurate and realistic timing values.

The rest of the paper is organized as follows. In the next section, we provide a brief history of Express. Section 3 summarizes some of the basic tools and utilities of Express. Section 4 describes the summary of iPSC/2 and iPSC/860 computer systems used in our experimentation.The results of basic communication tests including one-to-one communication, exchange, and broadcast are given in Section 5 while the results of global communication are given in Section 6. In Section 7, we present the results of the ring, all-to-all broadcast and complete exchange communication patterns. The execution times of application benchmark suite are presented in Section 8 and Section 9 concludes this paper.

2 Background and Overview of Express

The history of Express can be traced back to the Caltech/JPL machines, including the first hypercubes (the Cosmic Cube and Mark II), developed in the early eighties [30]. Initially, a so called "operating system" known as CrOS (Crystalline Operating system) consisting of a few message passing primitives was the only support that was available to derive those machines [19]. The need for efficient solutions of a number of scientific problems led to the incorporation of a number of message passing primitives, such as broadcast and concatenation, into CrOS. The advanced version of this operating system, called CrOS III, was implemented on the Mark III hypercube [11], [23]. The main features of CrOS III were synchronous communication among nodes with direct communication links. The programming models in these system were of host-node style where the user writes a node program which runs on the computing nodes of the parallel system and a host program for controlling the node program and performing I/O. To remove from the user the burden of writing host programs, a system known as "c3po" was developed which provided a shell based generic host program. A more sophisticated version of the system, called Cubix, allowed opening, reading and writing files and interfacing with the user in a host-free programming style. Later, to support multi-tasking, remote task creation, scheduling and a number of other house-keeping functions, an operating system known as "Multitasking, object-oriented, operating system", (MOOOS, commonly known as MOOSE.) was built which employed light weight processes communicating through box/pipe constructs. To support more efficient concurrent communication, a system called "crystal router" was developed in which the messages would be buffered on their nodes of origin until a synchronization point was reached when a single system call sent every message to its destination in one pass. This system reduced the latencies and allowed messages to be routed to nodes other than those to which direct connections existed. Later, the group of researchers who developed these systems started Parasoft Corporation by the end of 1987. Express, one of the commercial product of Parasoft Corporation is an integrated software package developed as a result of their research efforts at Caltech.

As illustrated in Figure 1, the implementation of Express consists of three layers. The lowest level consists of utilities for controlling the hardware, such as allocating processors, loading programs, etc. The medium level provides support for problem partitioning and communication between the nodes, and between the node and the control processor. The highest level contains the facilities for node programs to perform I/O and utilities for access to the host operating system. Each of the levels described above is logically distinct building only on those below it. As a result, Express is portable to a variety of systems using a "top-down" approach in which higher levels are built upon existing lower levels. The network version of Express for workstation based system is based on TCP/IP protocols.

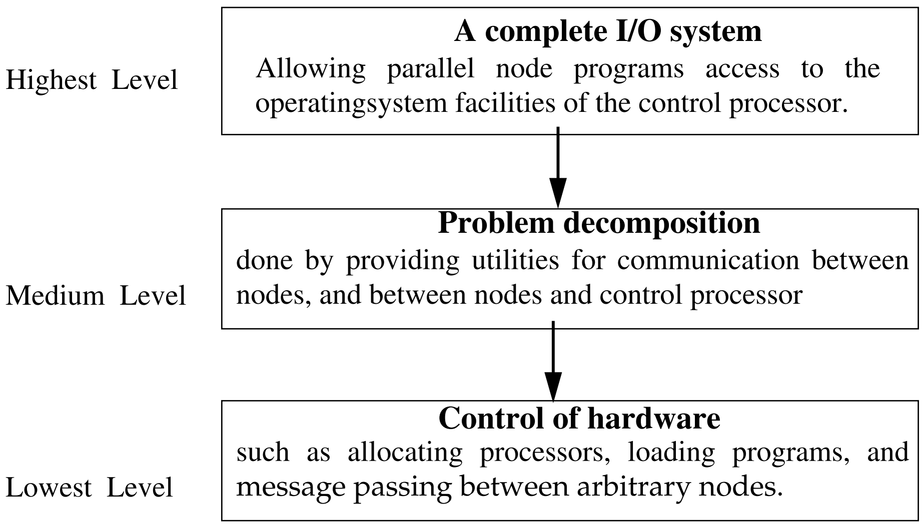

Figure 1 Three layers of Express implementation.

3 The Express Utilities

Express allows parallel programs to develop in the following four different styles: 1) The conventional *host-node* style in which the control part of the program appears in the host program and the computation intensive part appears in node program, 2) the *cubix* environment in which only node program is written and it interfaces to the outside world through graphics and text server, 3) the *profile* environment which can be added to both host–node and Cubix programming models, and 4) the *plotix* environment which provides extra graphics features to analyze program profiling. These programming models can be used by inserting appropriate Express routines in standard C or Fortran programs.

As illustrated in Figure 2, Express provides a number of utilities which can be classified as follows.

Communication: The communication facilities include blocking and non-blocking communication among nodes, exchange, broadcast and reading and writing a vector. Global communication includes concatenation, global reduction operations, synchronization, etc.

Processor Control: Allocation and de–allocation of processors, loading program into all nodes or into selected nodes. Facilities are also there to map different programs into different nodes.

Decomposition Tools: Decomposition tools allow the user to set up a virtual network topology on top of the physical topology. This enables the system to map user domain coordinates to processor number, distribute data to processors or distribute processors on user domain.

I/O Routines: The Cubix programming model of Express allows concurrent I/O directly to the nodes of the parallel system without requiring any assistance from a central control program.

Graphics: Graphics tools under Express are useful in analyzing the activity and performance of parallel programs. This can be done by not only merging the graphics data from different processors but also by having multiple output devices.

Debugging: Express also provides a source level debugger. The commands of the debugger can be directed to all or only a selected set of nodes.

Performance Analyses: For performance analysis, Express provides a number of tools. The *CTOOL* allows analysis of relative amount of time spent in computation, communication and I/O. The *ETOOL* shows the relationship between various computing elements. The *XTOOL* allows to understand the way CPU time is spent on each processor.

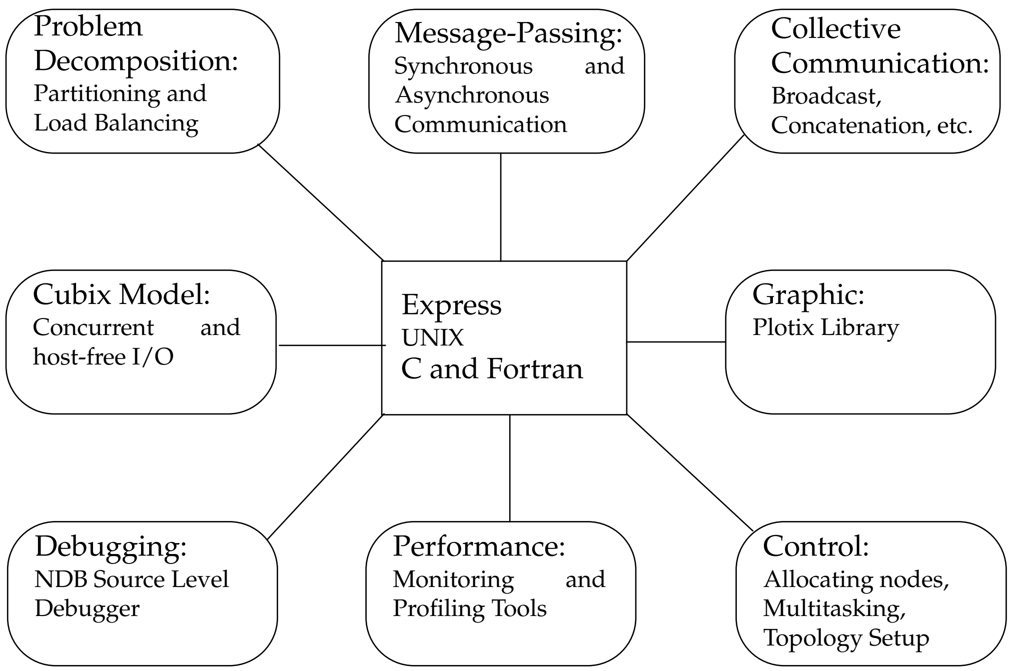

Figure 2 Express utilities

User commands include system administration such as configuration, reboot and reload, user's usage of system resources, and a source level debugger (communication, execution and events). Express allows explicit control over resources. For example, in a workstation environment, hosts and links among them can be added to the system configuration interactively. In addition, the Express toolkit contains a number of other tools including *INISGHT* for graphically displaying a sequential algorithm to demonstrate algorithm structure and potential parallelism, *FTOOL* for analysis of a program including variable usage analysis and flow structure, *ASPAR* which is a semi-automatic parallelizer for converting sequential *C* and *Fortran* programs to parallel form.

4 Test Environment

For performance evaluation, we used two commercially available parallel computers from Intel, the iPSC/2 at Cornell Theory Center and iPSC/860 at Caltech's Concurrent Supercomputing Center. Both systems are based on hypercube interconnection network topology. They can have up to 128 (in powers of 2) nodes. Both hypercubes are controlled from a host computer, called 'System Resource Manager' or SRM which runs System V UNIX. The operating system on nodes is NX (Node eXecutive) developed by Intel. The iPSC/2 system is based on 32-bit 80386 microprocessor while the iPSC/860 is based on 64-bit i860 RISC microprocessor. Details of the iPSC/2 and iPSC/860 node architectures can be found in [17] and [26], respectively. The summary of some of the architectural features of the computers we used is given in Table 1.

Table I: Configurations of iPSC/2 and iPSC/860 used in the experiments.

	iPSC/2	iPSC/860
Number of nodes	32	32
Node CPU	80386/387	i860
Clock Frequency	16 MHz	40 MHz
Cache Memory/node	64 Kb	8 Kb
Main Memory/node	4 Mb	8 Mb
Express Version	3.0	3.1
iPSC Node O/S	NX/2 (rel. 2.2)	NX/2 (rel. 3.2)

In both systems, each node has a hardware communication module, called DCM (direct connect module) which connects a node to the interconnection network [27]. Each DCM router can support up to eight channels. Channels are bit-serial and full duplex. Physically, each channel consists of four wires that connects each of the nearest neighbor nodes. One pair of wires, consisting of a data and strobe wires is used in only one direction and the other pair only in the other opposite direction. Seven out of eight channels per node are used for communication with the direct neighbor node and one channel is reserved for performing special I/O. Thus, the largest configuration that can be supported by the DCM is a 7 dimensional hypercube consisting of 128 nodes. The interprocessor communication is based on circuit switching protocol, that is, when two nodes want to communicate, a dedicated path is set up between them by the DCM, and messages travel through this path without any intervention. The intermediate nodes in the path are determined by the e-cube algorithm [27]. Each node maintains one outgoing and one incoming 4 Kbyte FIFO buffer. The flushing and filling of the buffer is done by a DMA controller on the iPSC/2 and by the i860 processor itself on the iPSC/860.

It is worth mentioning that there were different versions of Express available on the iPSC/2 and iPSC/860. On the iPSC/2, it was version 3.0 while on the iPSC/860, it was version 3.1. The objective of including the results of iPSC/2 is twofold. First, as mentioned above, we are interested in comparing the performance of iPSC/2 with iPSC/860. Second, by comparing the two versions of Express, as we show later, a considerable improvement can be noticed.

In the following sections, we will present the results of our experiments. These experiments include the timings for various Express primitives on both the iPSC/2 and iPSC/860 as well as the equivalent primitives provided by these two machines. Since Express uses the available compiler in the local machine, the execution time of the computational part of a program written with Express does not incur extra overhead and is governed by the speed of the available processor. Thus, we are not interested in benchmarking the basic computational operations. Some performance results showing the computational operations on the iPSC/2 and iPSC/860 can be found in [13] and [16]. All of our programs have been written in Fortran. For communications tests, every point in each test is the result of taking the average of a large number of repetitions. For small data sizes, we have used 1000 repetitions to improve the accuracy, while for larger messages, the number of repetitions has been varied from 100 to 200. The node clock has a resolution of 1 microsecond.

5 Basic Communication

The basic communication tests include one-to-one communication among nearest neighbors, multi-hop communication, exchange and broadcast. The results of our benchmark programs for these tests are provided in the following section.

5.1 One-to-One Communication

For measuring the communication speed between two nodes, we performed the standard echo test. In the echo test, the communication time between sending and receiving processors is measured by starting a clock at the sending processor and then invoking the send and receive routines to send out a message and wait for a reply. On the destination processor, receive and send routines are used to echo this message back to the sending processor. This process is repeated n number of times and the clock is then stopped. The communication time is then taken as elapsed time divided by $2n$. Before describing the result of node-to-node communication, we briefly explain the communication mechanisms in iPSC systems and the related Express and NX primitives.

The message passing between two nodes consists of a number of steps. On the sending node, these steps include performing the message passing subroutine, buffer allocation, copying the user data to the buffer area, constructing the routing path, obtaining the communication channel and actual transmission time. On the receiving side, these steps include obtaining the buffers for message receipt, copying the data to the user area. If the message is to be forwarded, new channel has to be obtained. It is well known that in both iPSC/2 and iPSC/860 hypercubes, there are two types of protocols used for sending messages [14], [18]. For messages of length smaller than or equal to 100 bytes, the sending node does not check to see if there is buffer space available at the receiving node. For messages of length greater than 100 bytes, first a short message is sent to the destination node to request that a buffer be allocated for the coming message. If the receiving node is able to allocate a buffer large enough to store the data, it sends an acknowledgement message. The sending node sends the data after receiving this request.

From a user's stand point, there are three types of methods for sending and receiving messages. In blocking send/receive, the sending node returns after sending the message into the network. The sending buffer can be reused in sending a new message. The receiving node does not return until the data has been copied into the receiver buffer. On the other hand, in non-blocking send/receive, the sending and receiving nodes do not wait and return immediately. The communication is done in the background in an asynchronous fashion. This allows overlapped communication and computation because the main processor can proceed with regular computation while the communication processor is performing communication. However, the buffer cannot be used until the communication operation is finished. For the echo test, we used blocking communication.

The Fortran calls for the blocking communication in NX are as follows:

csend(MessageType, SendBuffer, Length, DestNode, Pid),

crecv(MessageType, RecBuffer, Length)

where *MessageType* is a message type, *SendBuffer* is the pointer to the data structure which is to be sent, *Length* is the size of the message in bytes, *DestNod*e is the node number of the receiving node, *RecBuffer* is the pointer to the data structure where the message will be stored and *Pid* is a process identifier which receives the message.

The equivalent Express calls are as follows:

kxwrit(SendBuffer, Length, Destination Node, MessageType),

kxread(RecBuffer, Length, SourceNode, MessageType)

One noticeable difference between Express calls and equivalent NX calls is that in kxread, one needs to specify the source node as well. This can of course be avoided by using *NOCARE* macro. Also, *Pid* does not need to be specified in Express calls.

The time for one-to-one communication can be described by the following equation.

$$T_{comm} = T_{startup} + B \times T_{transmission} + h \times d$$

where B is the message length in number of bytes, $T_{startup}$ is the latency incurred due to the time required to set up the communication request, $T_{transmission}$ is the transmission time for one byte, d is the hop distance between the sending and receiving nodes and h is the extra time for each traversing each hop. These parameters for both iPSC/2 and iPSC/860, with Express and NX, are shown in Table II and Table III for $0 < B \leq 100$ and $B > 100$, respectively.

From Table II, it can be seen that the start-up latency, $T_{startup}$, for sending small messages on the iPSC/860 is 62.5 microseconds using NX and 80.5 microseconds using Express- a difference of 18 microseconds. We also observe that the difference in $T_{transmission}$ of Express and NX is not very large on both the iPSC/2 and iPSC/860. Comparing the performance of both iPSC's

Table II: Communication parameters for node-to-node message passing with $0 < B \leq 100$.

	Exp.-iPSC/860	NX-iPSC/860	Exp.-iPSC2	NX-iPSC/2
$T_{startup}$	80.05	62.5	592.0	324.3
$T_{transmission}$	0.431	0.421	0.553	0.552
h	10.12	10.0	10.33	10.14

Table III: Communication parameters for node-to-node message passing with $B > 100$.

	Exp.-iPSC/860	NX-iPSC/860	Exp.-iPSC2	NX-iPSC/2
$T_{startup}$	160.5	147.0	1065.0	585.0
$T_{transmission}$	0.394	0.394	0.462	0.394
h	30.38	29.29	30.99	30.52

using NX, we notice that the major difference lies in $T_{startup}$ which has been reduced by a factor of more than 5 for the iPSC/860 over iPSC/2. The $T_{transmission}$ has also improved from 0.552 to 0.421 - an improvement of about 24%, while the factor h has improved from 10.14 to 10.0 - an improvement of about 1.4%. When we compare the performance of Express on both hypercubes, we notice that the extra overhead of Express on the iPSC/860 (compared to NX) is far less than the extra overhead on the iPSC/2. For example, the difference between $T_{startup}$ using NX and Express on the iPSC/860 is only 17.55 microseconds while this difference on the iPSC/2 is 267.7 microseconds. Using Express on the iPSC/860, additional 0.12 microseconds account for each hop distance between two nodes while on the iPSC/2, this extra overhead is 0.19 microseconds.

As given in Table III, for message of more than 100 bytes on the iPSC/860, the difference in $T_{startup}$ using NX and Express is 13.5 microseconds. This difference on the iPSC/2 is 480 microseconds which is quite large. $T_{transmission}$ on the iPSC/860 is the same for NX and Express. However, on the iPSC/2, $T_{transmission}$ for Express is about 17% high. The difference in h for Express and NX on both systems is negligible. When we compare the performance of NX on the

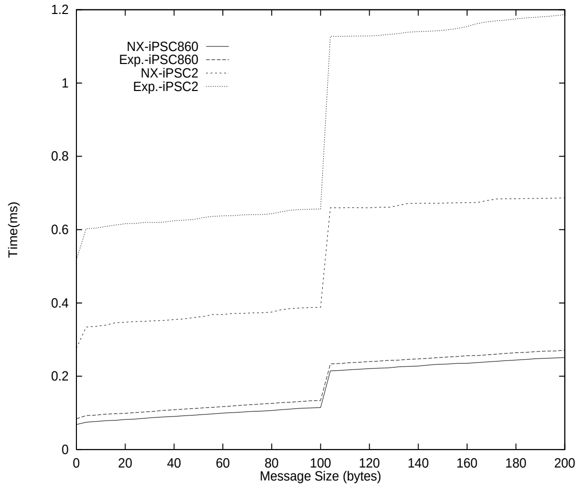

Figure 3(a): Times for one-to-one communication for small size messages with Express and NX primitives on the iPSC/2 and iPSC/860.

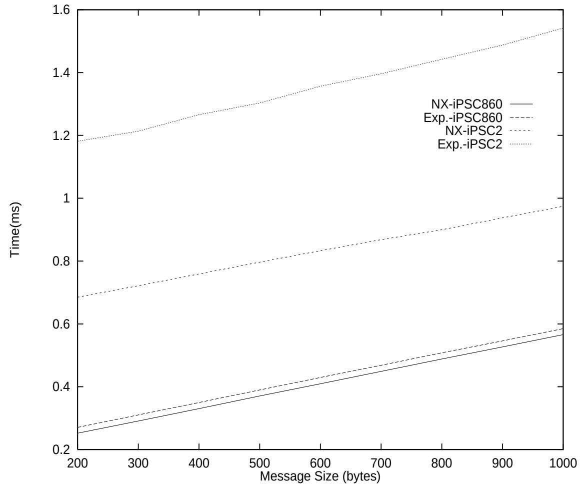

Figure 3(b): Times for one-to-one communication for medium size messages with Express and NX primitives on the iPSC/2 and iPSC/860.

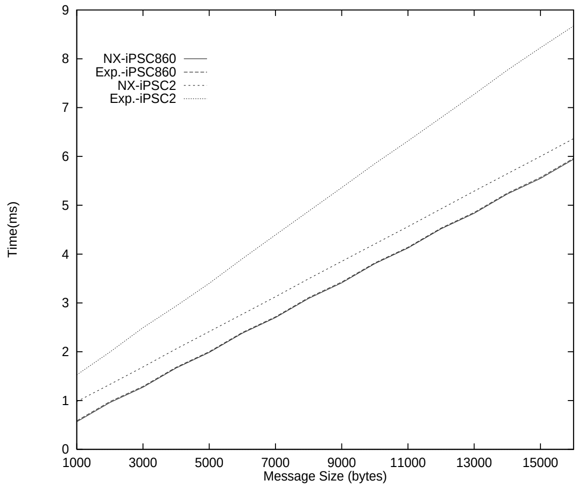

Figure 3(c): Times for one-to-one communication for large size messages with Express and NX primitives on the iPSC/2 and iPSC/860.

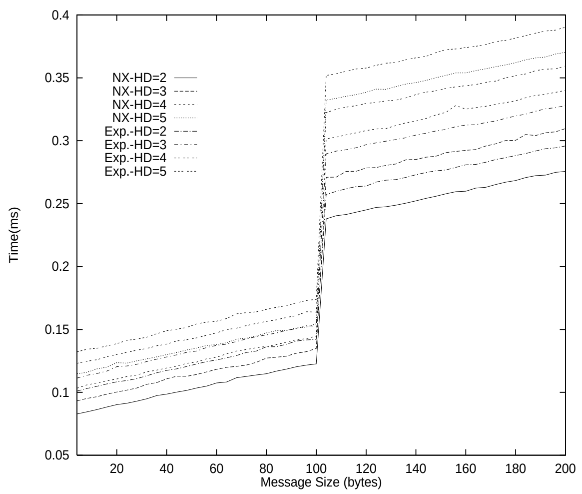

Figure 3(d): Times for one-to-one communication between nodes at various hop distances (HD) with Express and NX primitives on the iPSC/860.

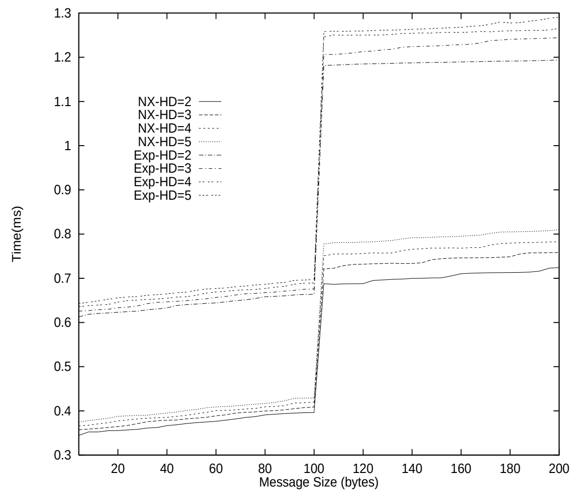

Figure 3(e): Times for one-to-one communication between nodes at various hop distances (HD) with Express and NX primitives on the iPSC/2.

iPSC/860 and iPSC/2, we observe that $T_{transmission}$ is the same on both systems while $T_{startup}$ on the iPSC/860 has improved by a factor of 4. When we compare the parameters given Table III with those of Table II, we can conclude that communication performance of iPSC/860 has increased by a larger factor for messages of less than 100 bytes compared to larger messages of more than 100 bytes.

Figure 3 (a) shows the plots of times required for one-to-one communication between directly connected processors for message sizes ranging from 0 to 200 bytes. Inspection of these plots reveal that Express is slower than NX by a factor of about 14 to 20% on the iPSC/860 and 68 to 80% on the iPSC/2. NX on the iPSC/2 is 3.3 to 4 times slower than the iPSC/860. Figure 3 (b) and Figure 3 (c) show the result for message sizes of 200 to 1000 bytes and 1000 to 16000 bytes, respectively. Relatively, the overhead of using Express is negligible on the iPSC/860 but is 30 to 46% on the iPSC/2. Spacing between the plots of NX is constant on the iPSC/860 but it increases on the iPSC/2 with an increase in message size.

Figure 3 (d) illustrates plots of times required for communication between nodes that are at 2, 3, 4 and 5 hop distances apart on the iPSC/860. The results for the iPSC/2 are shown in Figure 3 (e). Using NX on the iPSC/2 and iPSC/860, the spacing of curves is 10 microseconds for messages of length up to 100 bytes and about 30 microseconds for messages of length greater than 100. As described earlier, the amount of spacing is slightly larger for Express (1 to 2%).

The exact figures for the times to send messages of 0, 4, 104, 200, 1000 and 16000 are given in Table IV. As can be seen from these results, the difference in the times of Express and NX on

Table IV: Times required to send messages of different sizes to a neighboring node (μs).

Bytes	Exp.-iPSC860	NX-iPSC860	Exp.-iPSC2	NX-iPSC2
0	83	68	518	276
4	92	74	603	336
100	134	115	657	389
104	232	217	1135	660
200	270	255	1187	692
1000	584	566	1532	977
16000	5958	5941	8672	6364

the iPSC/860 is about 15 to 20 microseconds. On the iPSC/2, this difference starts with 242 microseconds (for 0 byte) and keeps on increasing with message size. For very long messages such as 16000, this difference can be as high as 0.3 milliseconds.

5.2 The Exchange Operation

The exchange function simultaneously sends a message to a node and receives a reply. The advantage of this function is that data can be read from and sent to the same or different nodes in one step. Another advantage is that data transmission in reading and writing can be overlapped. Moreover, the user is freed from any worry about which node should be read first and which should write. The NX primitive for the exchange operation is

status = csendrecv(OType, OBuffer, OLen, Odest, PId, IType, IBuffer, ILen)

This function causes *Olen* bytes of data to be sent to the node denoted by *Odest* in a message of *OType* and *ILen* bytes of data to be read from message of type *Itype*. The data is placed in memory at *IBuffer*. The process is blocked until the return message is received. The corresponding Express call is

status = kxchan(IBuffer, ILen, ISrc, IType, OBuffer, OLen, Odest, OType)

All of the arguments in *kxchan* are the same as *csendrecv* except that the order of arguments for sending and receiving messages is reversed. Furthermore, *kxchan* requires the source node as an argument, *ISrc*. A call to *kxchan* must be used with calls to *kxchan* in nodes denoted by *Idest* and *Isrc*, and the message types in these nodes must also be compatible. The variables *Isrc* and *Idest* can be the same. If the call is successful, status contains the number of bytes read or written.

The plots showing the times of exchange operation on the iPSC/860 and iPSC/2 for message sizes ranging from 0 to 200 bytes are given in Figure 4(a). The plots for 200 to 1000 bytes and from 1000 to 16000 bytes are shown in Figure 4(b), and Figure 4(c), respectively. On the iPSC/860, it takes about 140 microseconds to exchange a message of 0 byte with a neighboring node with NX. With Express, this time is 152 microseconds. On the iPSC/2, the exchange time is 671 and 1357 microseconds with NX and Express, respectively. In the range of 0 to 100 bytes, Express is slower than NX by a factor of 1 to 3%. Thus the exchange primitive of Express is more efficient than the simple one way send and receive. For messages of size greater than 100, Express is slower than NX by a factor of 15 to 22%. On the iPSC/2, Express is 70 to 80% slower than NX. The exchange operation using NX is 2 to 4 times faster on the iPSC/860 as compared to the iPSC/2 for small messages. Clearly, for very large messages, the relative performance differences become less dominant.

In order to see the impact of hop distance on the exchange operation, we also benchmarked the exchange operation by varying the distance between nodes. The plots showing these results for the iPSC/860 and iPSC/2 are shown in Figure 4(d) and Figure 4(e), respectively. The message size in both cases has been varied from 200 to 1000 bytes. The impact of hop distance with Express appears to be less as compared to NX on both on the iPSC/860 and iPSC/2. The spacing between plots of various hop distances using NX on the iPSC/860 is about 50 microseconds while the spacing between Express plots is about 30 microseconds. On the iPSC/2, this spacing is 60 to 70 microseconds and 30 to 35 microseconds for NX and Express, respectively.

5.3 The Broadcast Operation

Broadcast is performed to do broadcasting operations among the nodes and to and from the host node. It is used in various linear algebra algorithms, matrix-vector multiplication, matrix-ma-

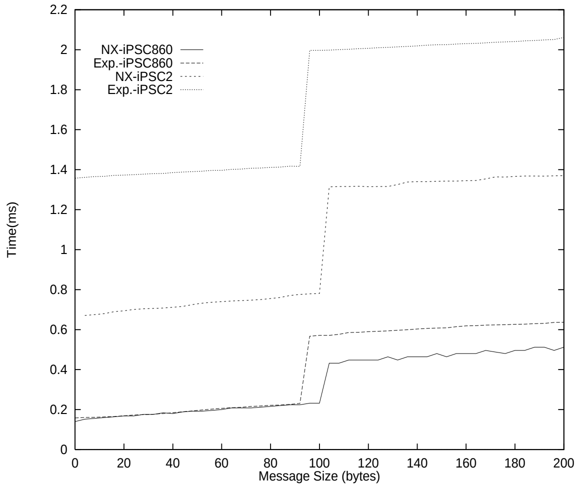

Figure 4(a): Times for exchange operation between nearest neighbors for small size messages on the iPSC/2 and iPSC/860.

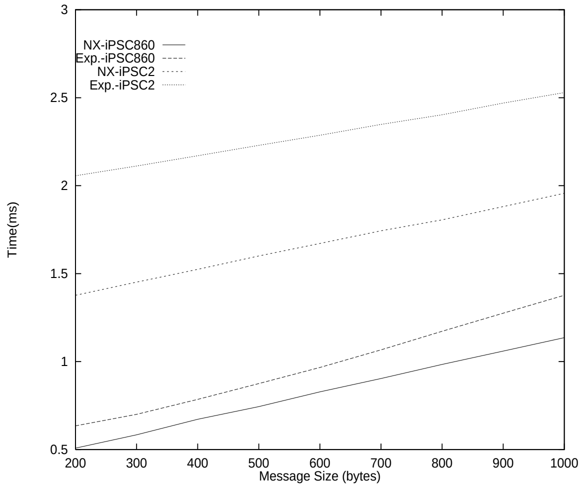

Figure 4(b): Times for exchange operation between nearest neighbors for medium size on the iPSC/2 and iPSC/860.

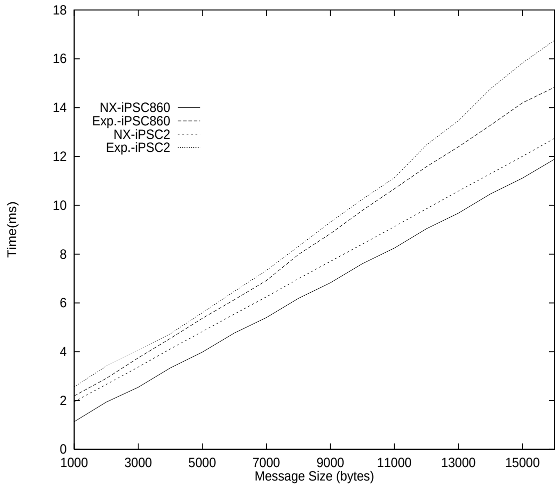

Figure 4(c): Times for exchange operation between nearest neighbors for large size messages on the iPSC/2 and iPSC/860.

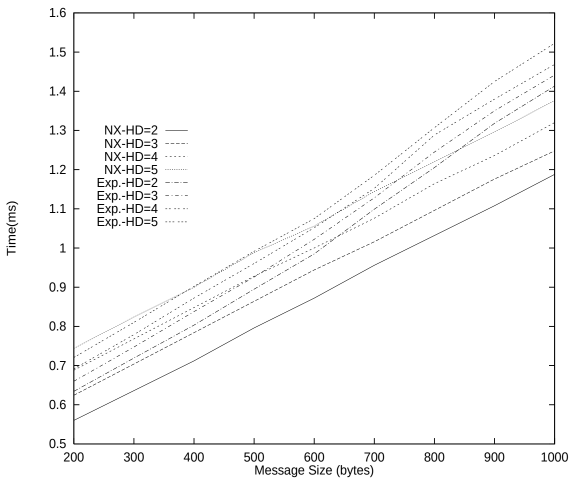

Figure 4(d): Times for exchange operation between nodes at various hamming distances on the iPSC/860.

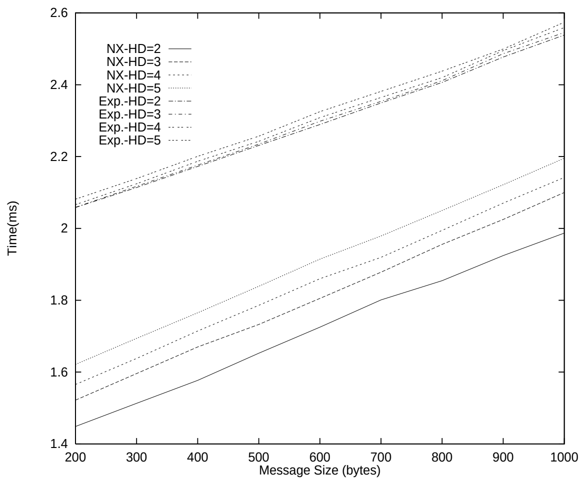

Figure 4(e): Times for exchange operation between nodes at various hamming distances on the iPSC/2.

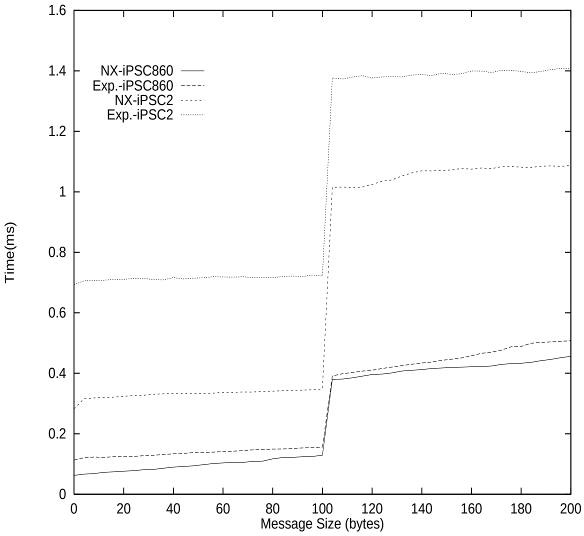

Figure 5(a): Times for broadcasting to 4 processors on the iPSC/2 and iPSC/860 (0 to 200 bytes).

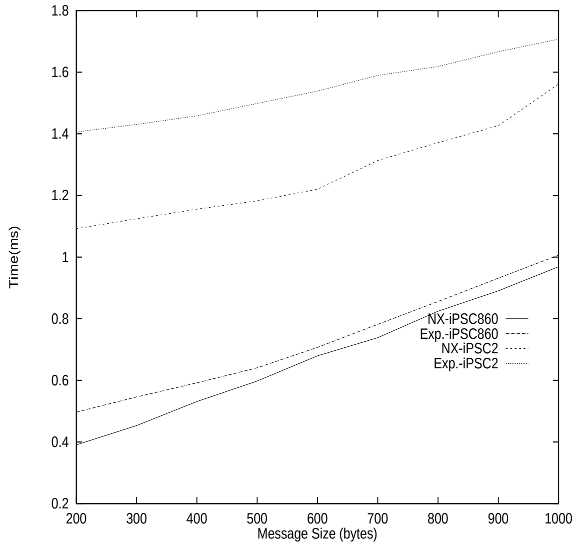

Figure 5(b): Times for broadcasting to 4 processors on the iPSC/2 and iPSC/860 (200 to 1000 bytes).

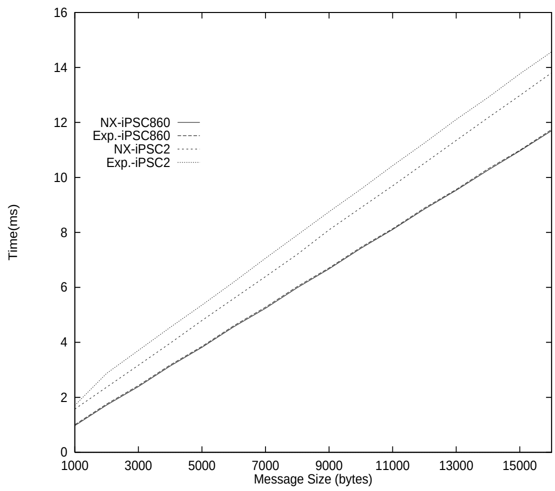

Figure 5(c): Times for broadcasting from one processor to 4 processors on the iPSC/2 and iPSC/860 (1000 to 16000 bytes).

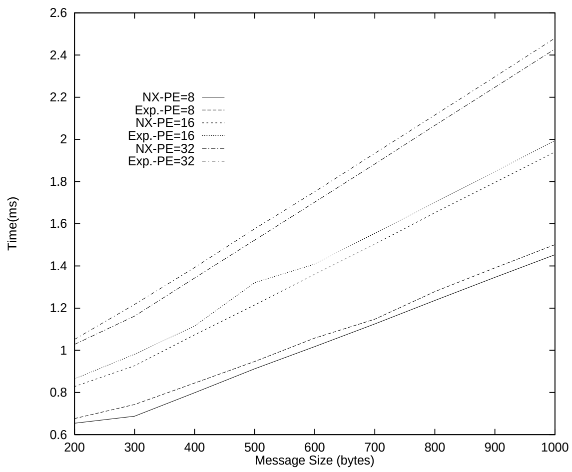

Figure 5(d): Broadcasting from one processor to 8, 16 and 32 processors on the iPSC/860 (200 to 1000 bytes).

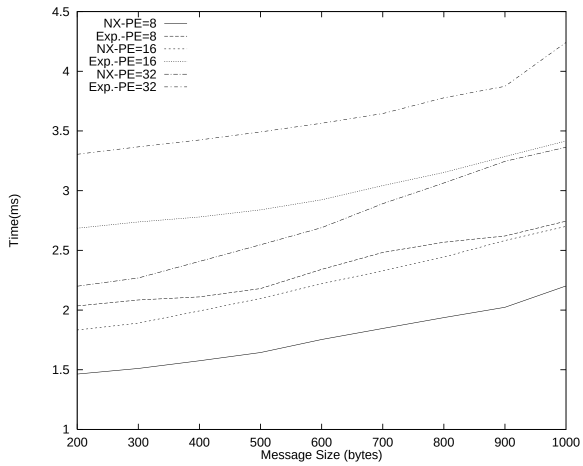

Figure 5(e): Broadcasting from one processor to 8, 16 and 32 processors on the iPSC/2 (200 to 1000 bytes).

trix multiplication, LU-factorization, [24] etc. The Express primitive for the broadcast operation is as follows:

status = kxbrod(Buffer, Origin, Size, NumNodes, NodeList, Type)

This function starts from node *Origin* which broadcasts *Size* bytes of data in *Buffer*. The number of nodes to which the date is broadcasted is indicated by *NumNodes*. The argument *NodeList* is an array of node numbers which should receive the broadcast. A call to *kxbrod* must have corresponding calls to *kxbrod* in all of the target nodes.

On the other hand, in NX, routines *csend* and *crecv* are used for broadcast also. If the argument *DestNode* in the call for *csend* routine (see above in section 5.1) can be give as -1, the message is sent to all the nodes in the attached subcube.Message can also be sent to a subcube composed of a set of nodes surrounding the source node [22]. Compared to this, Express has the extra advantage of specifying arbitrarily any node in the *NodeList* argument which should receive the message.

Figure 5(a) to Figure 5(c) show the timings of broadcasting for various message sizes. The number of nodes in this case has been chosen as 4. The inspection of these figures indicate that, for small messages, Express is about 10% slower than NX on the iPSC/860 and about 80% slower on the iPSC/2. For medium messages, compared to NX, the Express is again about 10% slower on the iPSC/860 and about 30% on the iPSC/2. For very large messages, the difference is less than 1% on the iPSC/860 and less than 11% on the iPSC/2. For small and medium messages, the iPSC/860 is about 4 times faster than the iPSC/2.

To analyze the impact of the number of nodes on the performance of broadcast operation, we provide timings for 8, 16 and 32 nodes as well. These results are given in Figure 5(d) and Figure 5(e) for iPSC/860 and iPSC/2, respectively. For higher number of nodes, compared to NX, Express is about 7 to 10% slower on the iPSC/860 and about 20 to 28% slower on the iPSC/2. The spacing between the plots of NX on both systems increases as the message size is increased. Likewise, the spacing between the plots of Express increases accordingly.

6 Global Communication

In global communication, all the nodes participate in some operation. The reduction operations and concatenation operations are two examples that require global communication. These operations are describe below.

6.1 Reduction Operations

Global operations for producing reduction across all the nodes are frequently used in parallel programs. A reduction operation takes as input a value in each processor and outputs a single value in every processor. There can be many types of reduction operations such as *add*, *prod*, *max*, *min*, *and*, *or*, *xor*, etc. While NX provides different calls for each of such operations, Express has a unified function for performing reduction operations. This is described below.

status = kxcomb(Buffer, Function, Size, NumItems, NumNodes, NodeList, Type)

Where *NumItems* of data, each of size *Size* bytes and taken from *Buffer* are individually combined using the user-supplied reduction routine *Function*. The argument *NumNodes* is the number of nodes participating in the operation and *NodeList* is the array containing the id's of these nodes.

In NX, the global combining operations are separate for each operation but have the following common format:

G*x*Operation(Buffer, Size, Work)

where *x* can be *i, l, s, d* or *l* if the type of data is integer, long integer, single precision floating point, double precision floating point or logical, respectively, and Operation is the name of global operation, such as *SUM* (addition), *PROD* (multiplication), *AND* (logical AND), etc.

Buffer is the input vector or scalar to be used in the operation and contains the final result when the operation is complete. *Size* is the number of elements in *Buffer* and *Work* is a Temporary Array which is used to receive intermediate results. Express has an extra facility of specifying the number of nodes and their id's which need to participate in the global operation, while in NX all the nodes have to participate. In our study, only sum, product and maximum operations are tested. The format of calls for these operations is given below.

gssum(Buffer, Size, Temp),

gsprod(Buffer, Size, Temp),

gshigh(Buffer, Size, Temp)

The timing results for the global sum operation are given in Figure 6(a) to Figure 6(c) for different data sizes. The plots using 8, 16 and 32 nodes on the iPSC/860 and iPSC/2 are shown in Figure 6(d) and Figure 6(e). For these global reduction operations, we used single precision floating point data where each word is of 4 bytes. The results indicate that the performance degradation with Express as compared to NX is dependent on the data size. For example, as can be seen from Figure 6(a), Figure 6(b) and (c), the spacing of curves increases if the data size is increased on both iPSC/860 and iPSC/2. On the iPSC/860, Express is about 30% slower than NX for small data size and about 50 to 60% for large data sizes. On the iPSC/2, Express is more than 100% slower than NX. For very large messages, the performance degradation of Express is even more severe. The results shown in Figure 6(e) reveal that Express becomes more slow if the number of processors is increased.

The curves for global maximum and multiplication show the similar patterns and hence are not included in this paper. However, the exact timing results for a selected set of data points for global maximum and multiplication as well as global sum are provided in Tables V to VII.The data set includes times to perform these operations on 1, 4, 16 and 26 words. The results are repeated for 8, 16 and 32 processors. It can be seen from these tables that global maximum takes slightly more time to compute as compared to global sum while global multiplication takes even longer in most of the cases.

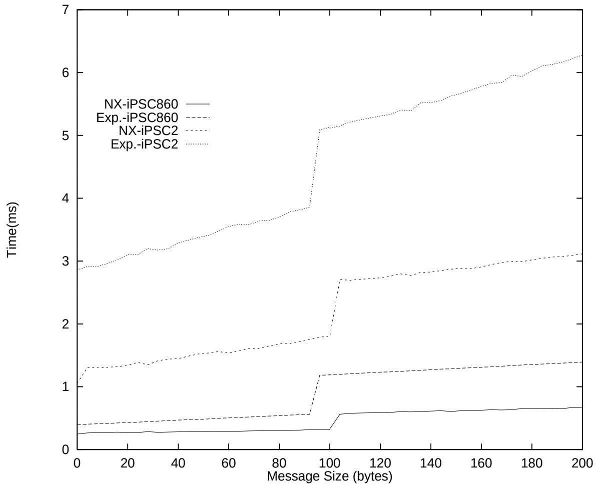

Figure 6(a): Times for global sum across 4 processors on the iPSC/2 and iPSC/860 (0 to 200 bytes).

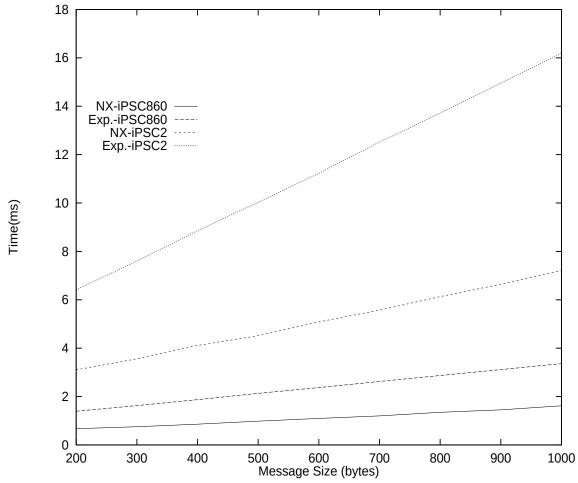

Figure 6(b) Times for global sum across 4 processors on the iPSC/2 and iPSC/860 (200 to 1000 bytes).

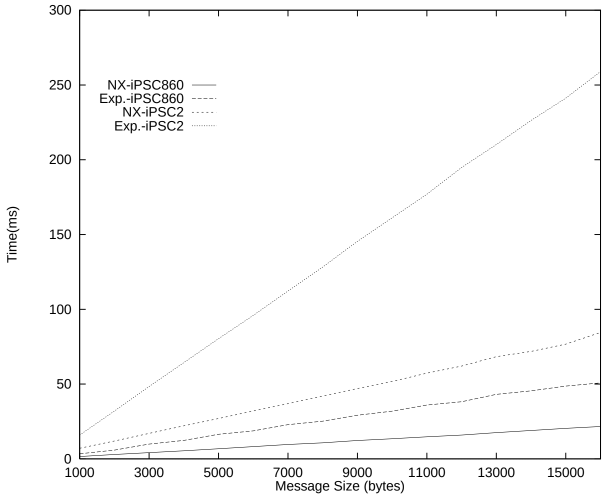

Figure 6(c): Times for global sum across 4 processors on the iPSC/2 and iPSC/860 (1000 to 16000 bytes).

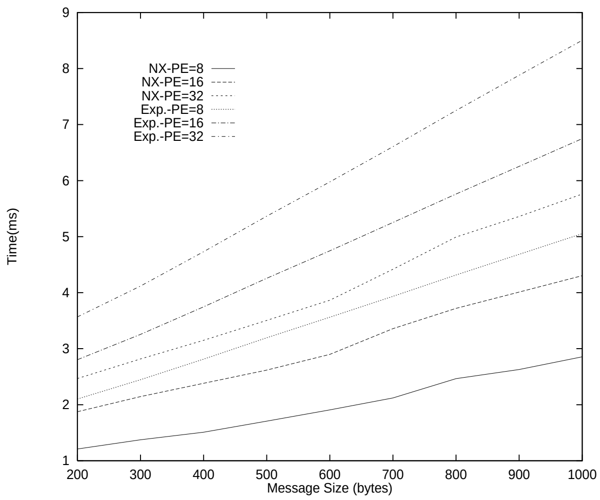

Figure 6(d): Times for global sum on the iPSC/860 using 8, 16 and 32 processors (200 to 1000).

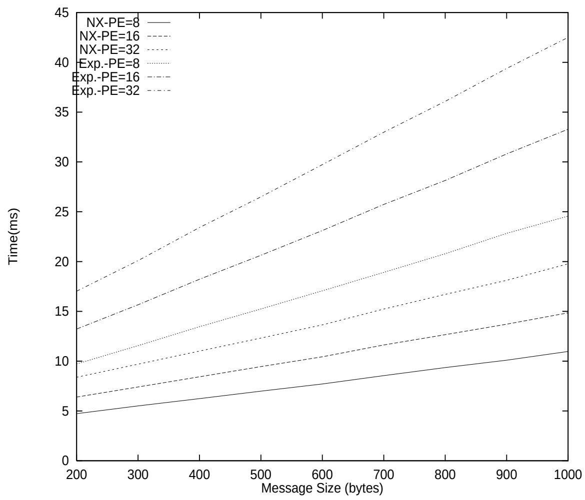

Figure 6(e): Times for global sum on the iPSC/2 using 8, 16 and 32 processors (200 to 1000).

Table V: Times for global sum (milliseconds).

Processors	Words	Exp.-iPSC860	NX-iPSC860	Exp.-iPSC2	NX-iPSC2
4	1	0.404	0.265	2.912	1.303
	2	0.410	0.273	2.917	1.305
	4	0.424	0.277	3.026	1.321
	26	1.098	0.562	5.148	2.706
8	1	0.604	0.422	4.244	1.869
	2	0.613	0.427	4.332	1.906
	4	0.636	0.431	4.435	1.948
	26	1.819	1.045	7.724	3.935
16	1	0.797	0.597	5.660	2.475
	2	0.811	0.604	5.767	2.513
	4	0.843	0.609	5.942	2.605
	26	2.447	1.722	10.466	5.293
32	1	1.033	0.766	7.091	3.143
	2	1.053	0.774	7.207	3.188
	4	1.091	0.788	7.448	3.303
	26	3.138	2.305	13.354	6.936

Table VI: Times for global maximum (milliseconds).

Processors	Words	Exp.-iPSC860	NX-iPSC860	Exp.-iPSC2	NX-iPSC2
4	1	0.409	0.291	2.876	1.286
	2	0.415	0.293	2.906	1.309
	4	0.428	0.296	3.015	1.331
	26	1.101	0.586	5.122	2.695
8	1	0.606	0.445	4.239	1.872
	2	0.616	0.448	4.313	1.911
	4	0.638	0.465	4.471	2.050
	26	1.829	1.107	7.703	4.063
16	1	0.809	0.606	5.685	2.506
	2	0.822	0.619	5.762	2.563
	4	0.854	0.633	5.948	2.660
	26	2.462	1.770	10.528	5.369
32	1	1.011	0.783	7.057	3.156
	2	1.031	0.789	7.167	3.218
	4	1.073	0.801	7.390	3.344
	26	3.113	2.375	13.315	7.163

Table VII: Times for global multiplication (milliseconds).

Processors	Words	Exp.-iPSC860	NX-iPSC860	Exp.-iPSC2	NX-iPSC2
4	1	0.409	0.274	2.858	1.296
	2	0.414	0.278	2.918	1.297
	4	0.426	0.278	2.964	1.331
	26	1.197	0.562	5.268	2.655
8	1	0.607	0.444	4.237	1.869
	2	0.617	0.446	4.329	1.902
	4	0.638	0.458	4.446	1.957
	26	1.823	1.101	7.868	4.007
16	1	0.811	0.598	5.662	2.489
	2	0.824	0.606	5.756	2.520
	4	0.854	0.610	5.955	2.622
	26	2.457	1.722	10.580	5.385
32	1	1.083	1.138	7.081	3.150
	2	1.159	1.289	7.222	3.197
	4	1.305	1.326	7.460	3.326
	26	4.677	4.084	13.666	7.092

6.2 Concatenation

The concatenation operation is used to globally collect and concatenate a contribution from each node in node number order. If each processor's contribution is *N* elements and there are *P* processors, then the global concatenation operation computes a concatenation of the local list with *N* times *P* elements in each of the processors. The resultant list is left in each processor. The length of contribution from each node can be specified.

On the iPSC/2 and iPSC/860, the following primitive is used for concatenation:

gcolx(MyBuffer, Sizes, ResBuffer)

Where *MyBuffer* refers to the input buffer in each node, *Sizes* is an array containing the length of input buffer from each node and *ResBuffer* refers to the output buffer which contains the final results. The equivalent Express primitive is as follows:

status = kxconc(MyBuffer, MySize, ResBuffer, RecSize, Sizes, NumNodes, NodeList, Type)

where each node contributes *MySize* bytes of data from the array *MyBuffer* to be placed in each node's *RecBuffer*. The individual blocks of data are sorted into order of increasing processor number and placed in *ResBuffer*, separated by *ResSize*, The argument Sizes contains the amount of data contributed by each node. The rest of the arguments are provided as usual.

Our timings for the concatenation operation using 4 processors are presented in Figure 7(a) to Figure 7(e). Like other communication primitive, there is a sharp jump when the amount of data from each node exceeds 100 bytes. On the iPSC/860, Express is found to be rather slower: about 45% for data size less than 100 bytes and 60 to 80% for data size between 104 to 200 bytes. On

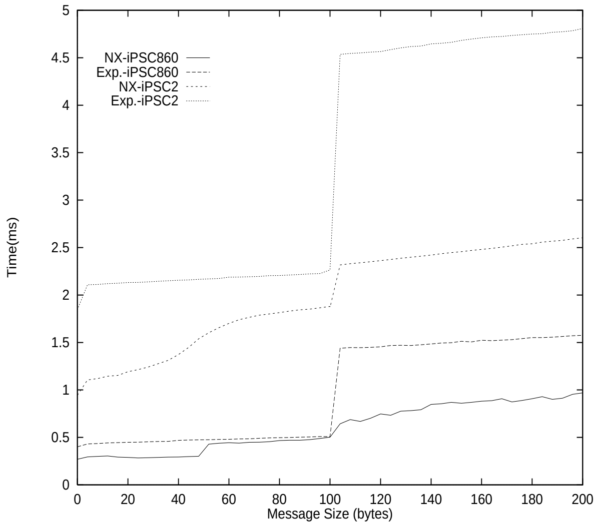

Figure 7(a): Times for concatenation operation across 4 processors on the iPSC/2 and iPSC/860 (0 to 200 bytes).

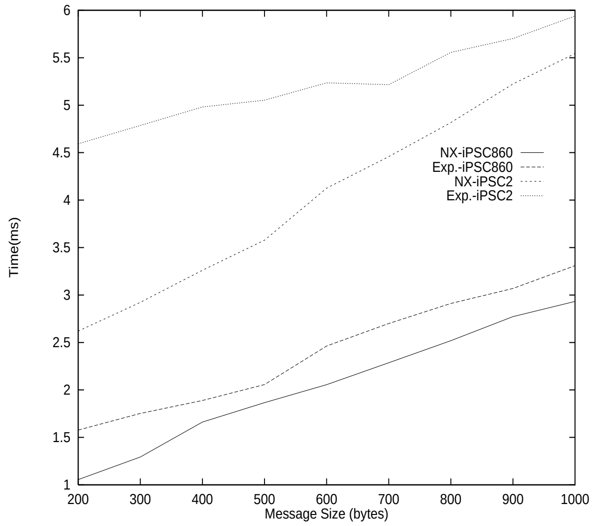

Figure 7(b): Times for concatenation operation across 4 processors on the iPSC/2 and iPSC/860 (200 to 1000 bytes).

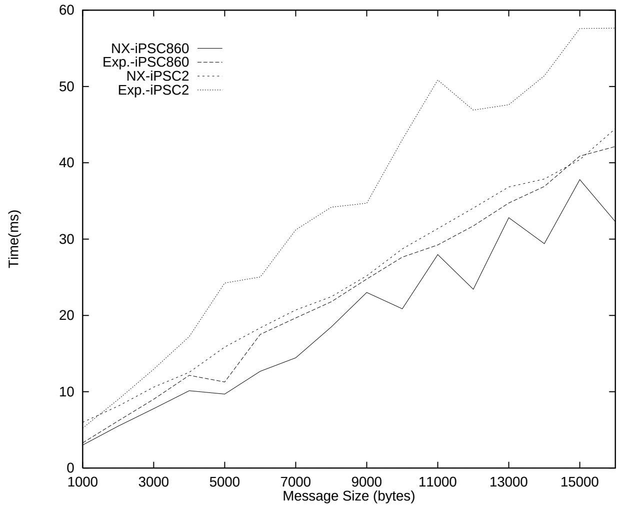

Figure 7(c): Times for concatenation operation across 4 processors on the iPSC/2 and iPSC/860 (1000 to 16000 bytes).

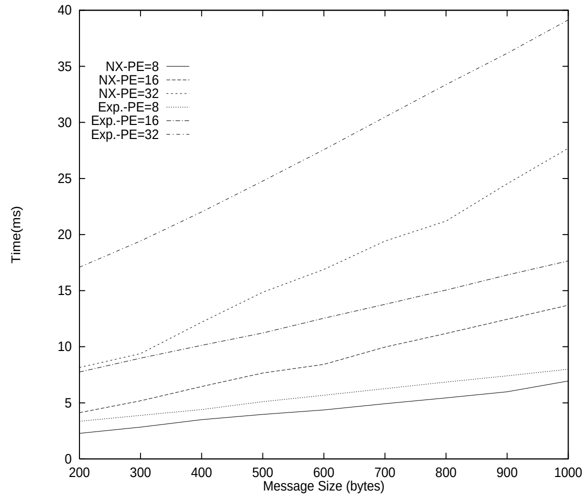

Figure 7(d): Times for concatentation operation across 8, 16 and 32 processors on the iPSC/2 (200 to 1000 bytes).

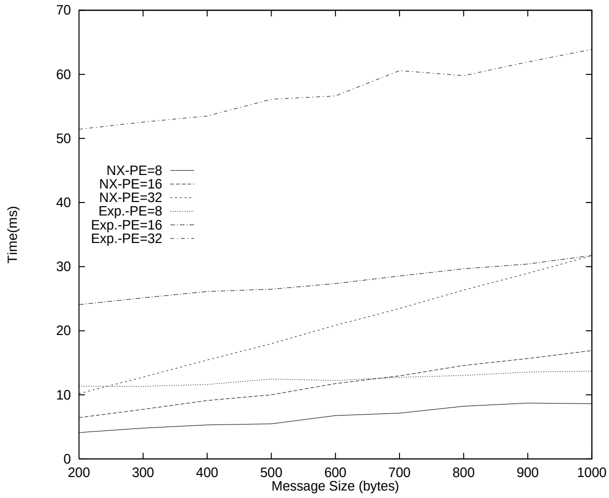

Figure 7(e): Times for concatentation operation across 8, 16 and 32 processors on the iPSC/860 (200 to 1000 bytes).

the iPSC/2, Express is 60 to 80% slower. NX on the iPSC/2 is more than 2 times slower than iPSC/860 while Express on the iPSC/2 is about 3 times slower than iPSC/860. However, these factors narrow down for very large data sizes. As can be observed from Figure 7(d), the plots of NX and Express are linear and are equally spaced. The difference in the performance of Express and NX increases if the number of processors is increased. On the iPSC/2, Express is 100% slower if concatenation is done across 32 processors with very large data size.

7 Other Communication Patterns

In addition to the basic communication primitives and global operations, we are interested in analyzing the performance of Express with some additional useful communication patterns. We have conducted benchmark experiments on three such patterns, namely the ring communication, all-to-all broadcast and complete exchange. These tests and their timing results are explained below.

7.1 The Ring Communication

In the ring communication test, each node takes a message and circulates it through each of the other nodes in the network. This communication pattern is generated in a straight forward manner. Each node finds its “forward” and “backward” neighbor with whom it communicates. Each node then writes the message to its “forward” neighbor and reads from the “backward” neighbor. This is repeated N-1 times where N is the total number of nodes. In general, the efficiency of this communication pattern depends on the underlying network topology. Determination of the “forward” and “backward” neighbors can be done using express’s *kxgrid* utilities which can convert any physical topology to a virtual topology. On the hypercube, “forward” and “backward” neighbors can be easily found using gray codes for implementing the ring pattern using NX or Express primitives.

Figures 8(a) to 8(e) show the results of timing the ring communication pattern. To communicate 4 bytes in a ring on the iPSC/860, it takes 740 microseconds using Express and 470 microseconds using NX. On the iPSC/2, it takes 3.13 and 1.88 milli seconds. To communicate 104 bytes on the iPSC/860, it takes 2.35 and 1.88 and milli seconds on the iPSC/860 and 6.20 and 4.56 milli seconds on the iPSC/2. The relative difference in the performance of Express and NX is about 50 to 58% on the iPSC/860 and 65 to 75% on the iPSC/2. The absolute difference in the timings of Express and NX increases if the message size or the number of processors is increased. Nevertheless, the relative degradation in the performance decreases.

7.2 All-to-All Broadcast

The all-to-all communication pattern is implemented by repeating the regular broadcast primitives N times for N nodes with changing the source node in each iteration.

Using NX primitives, the implementation can be done as follows:

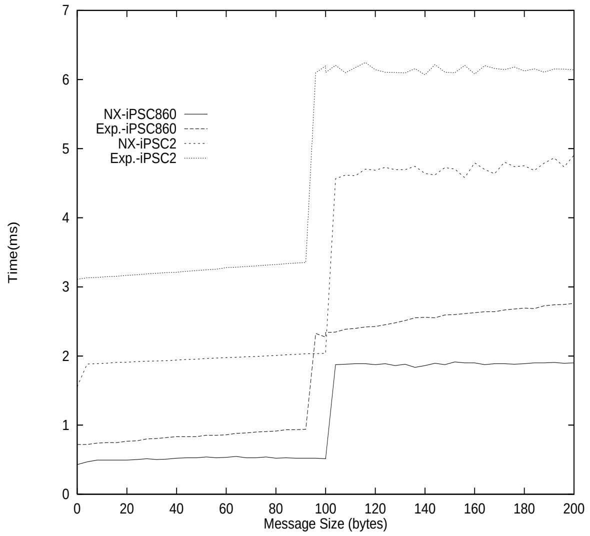

Figure 8(a): Ring test across 4 processors on the iPSC/2 and iPSC/860 (0 to 200 bytes).

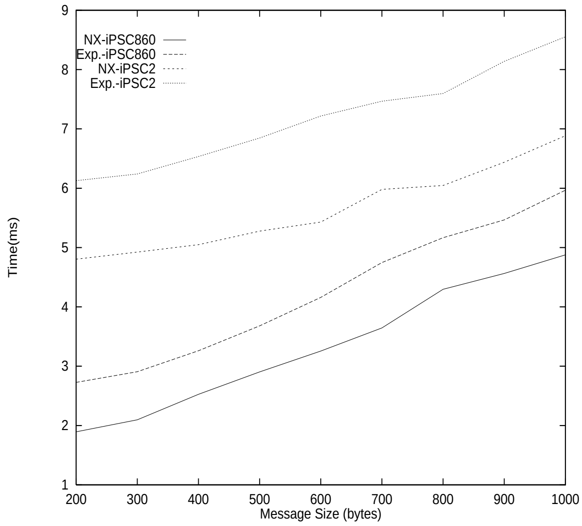

Figure 8(b): Ring test across 4 processors on the iPSC/2 and iPSC/860 (200 to 1000 bytes).

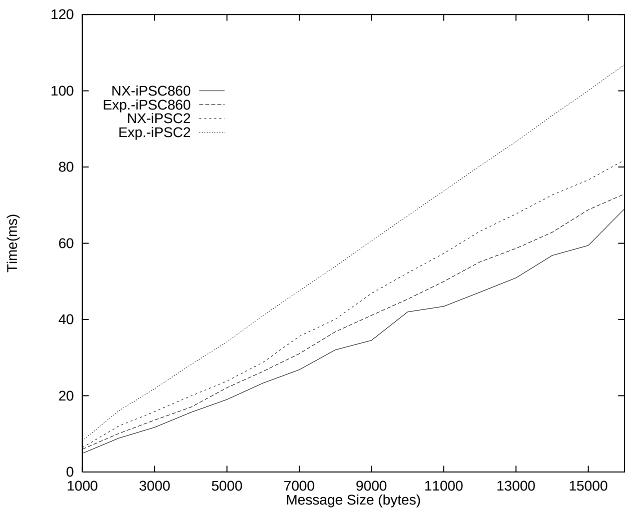

Figure 8(c): Ring test across 4 processors on the iPSC/2 and iPSC/860 (1000 to 16000 bytes).

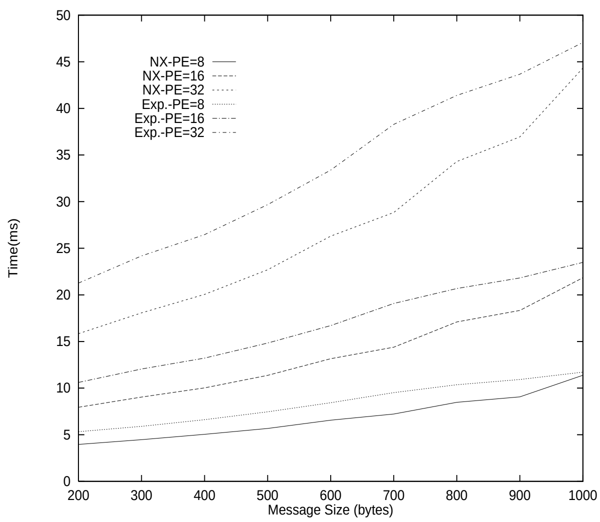

Figure 8(d): Ring test across 8, 16 and 32 processors on iPSC/860 (200 to 1000 bytes).

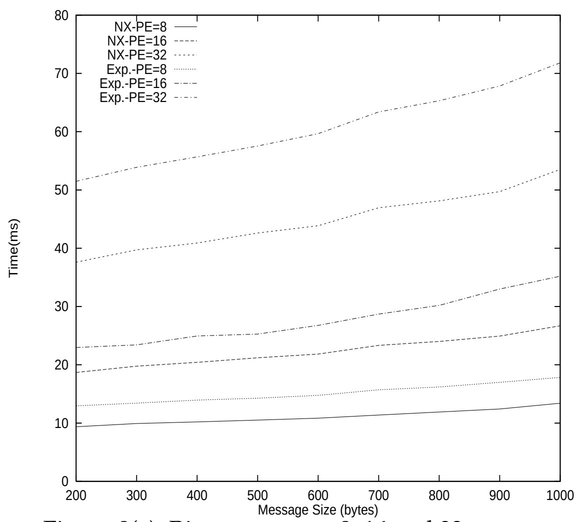

Figure 8(e): Ring test across 8, 16 and 32 processors on iPSC/2 (200 to 1000 bytes).

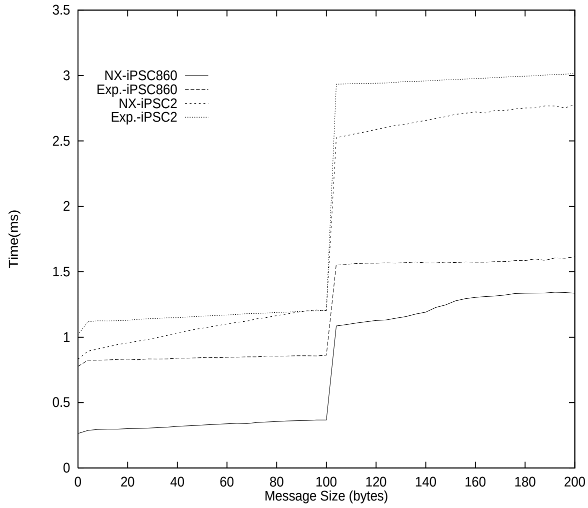

Figure 9(a): All-to-all broadcast using 4 processors on the iPSC/2 and iPSC/860 (0 to 200 bytes).

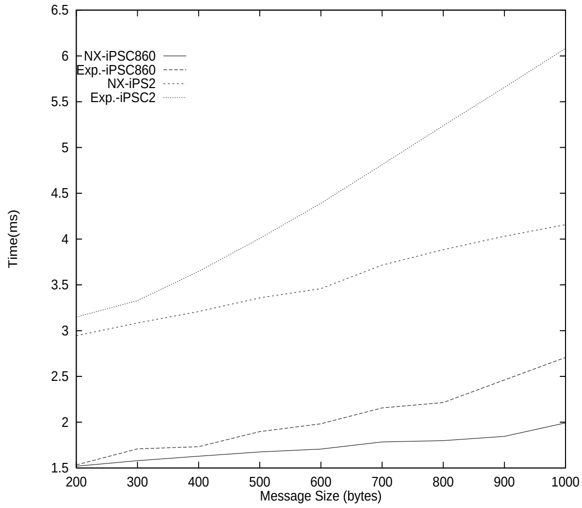

Figure 9(b): All-to-all broadcast using 4 processors on the iPSC/2 and iPSC/860 (200 to 1000 bytes).

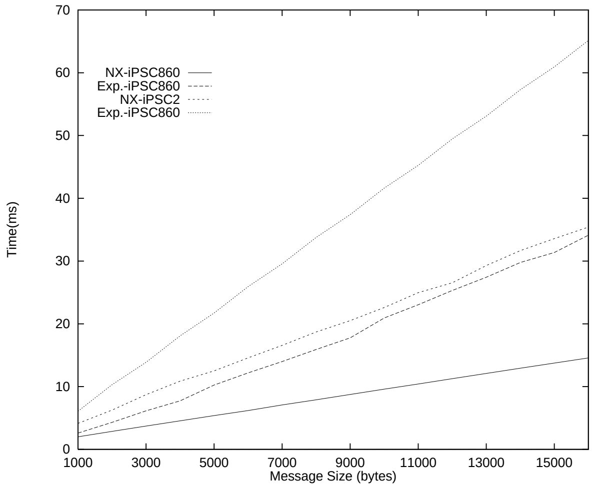

Figure 9(c): All-to-all broadcast using 4 processors on the iPSC/2 and iPSC/860 (1000 to 16000 bytes).

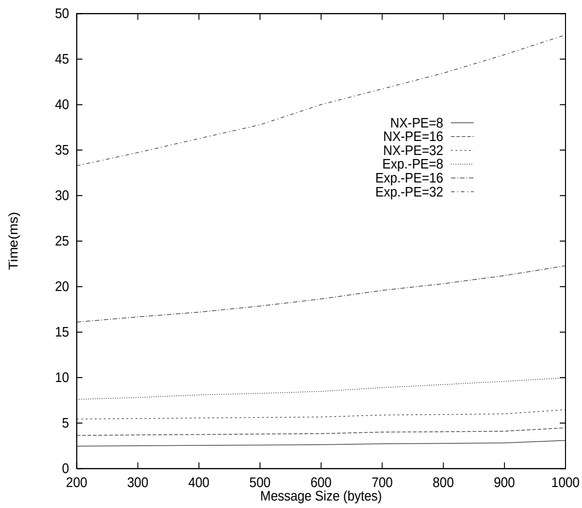

Figure 9(d): All-to-all broadcast with 8, 16 and 32 processors on the iPSC/860 (200 to 1000 bytes).

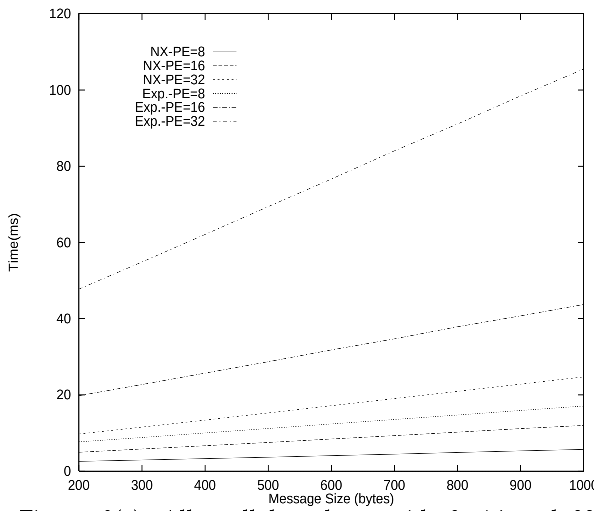

Figure 9(e): All-to-all broadcast with 8, 16 and 32 processors on the iPSC/2 (200 to 1000 bytes).

```
MsgType = thisNode
call csend(Type, SendBuffer, Size, AllNodes, PID)
do j= 0, NumNode - 1
        if (j .NE. thisNode) then
                MsgType = j
                call crecv(MsgType, RecBuffer, Size)
        end if
end do
```

Using Express primitives, the code which is more compact, is as follows:

```
do j = 0, NumNodes - 1
        MsgType = j
        Origin = j
        status = kxbrod(Buffer, Origin, Size, AllNodes, MsgType)
end do
```

The results are provided in Figures 9(a) to 9(e). Overall, the communication cost of all-to-all broadcast test is quite high but is less than that of the ring test. Express is almost 100% slower than NX for small messages. The performance degradation also appears to increase with an increase in the message size and number of processors used.

7.3 The Complete Exchange

In the complete exchange communication pattern, which is also known as all-to-all personalized communication, each of the *N* nodes sends a different block of data to each of the remaining *N*-1 nodes. This communication pattern is equivalent to a complete directed graph. It is used in a number of algorithms including matrix transpose, matrix-vector multiply, 2-dimensional FFTs, distributed table look-ups [15], etc. The time required to carry out the complete exchange is an important measure of the power of a distributed-memory parallel computer system since it is the densest communication requirement that can be implemented on a network. A simple implementation is given in the following although more sophisticated versions can be found in [15], [24], and [25].

```
do j = 0, NumNodes - 1
        if (thisNode .EQ. j) then
                do k = 0, NumNodes - 1
                        if (thisNode .NE. k) then
                        call crecv(MsgType, RecBuffer, Size)
                        end if
                end do
        else
                Dest = j
                call csend(MsgType, SendBuffer, Size, Dest, PId)
end do
```

The Express version is similar to the above except that calls to *csend* and *crecv* are replaced by *kxread* and *kxwrit*.

The plots of our timings are presented in Figure 10(a) to 10(e). As can be seen, overall timings for the complete exchange are higher than those of the ring and all-to-all broadcast tests. For small messages, Express is slower than NX only by a factor of 25 to 35% on the iPSC/860 and 70 to 85%

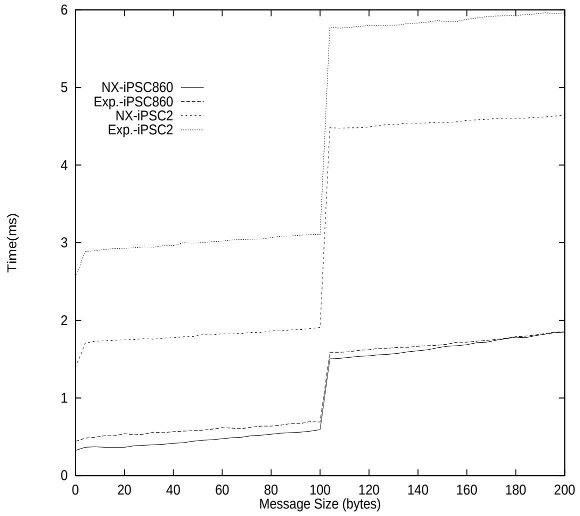

Figure 10(a): Complete exchange across 4 processors on the iPSC/2 and iPSC/860 (0 to 200 bytes).

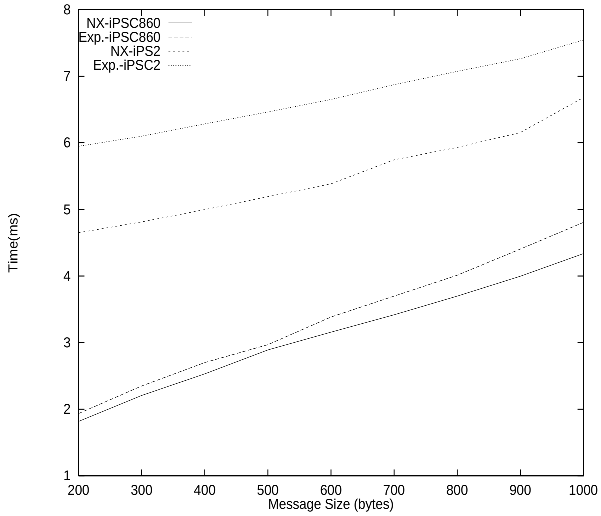

Figure 10(b): Complete exchange across 4 processors on the iPSC/2 and iPSC/860 (200 to 1000 bytes).

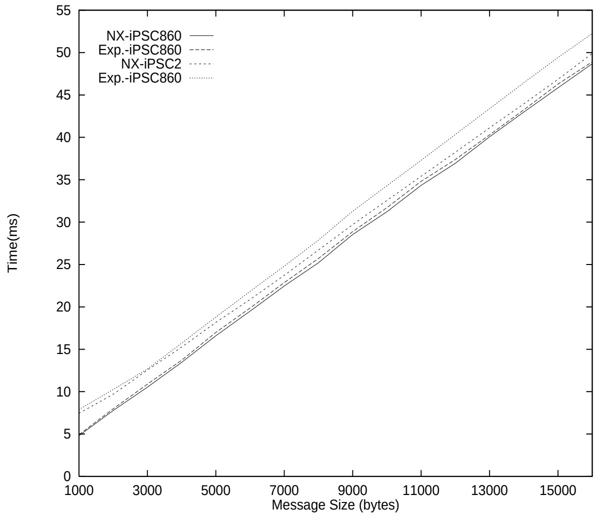

Figure 10(c): Complete exchange across 4 processors on the iPSC/2 and iPSC/860 (1000 to 16000 bytes).

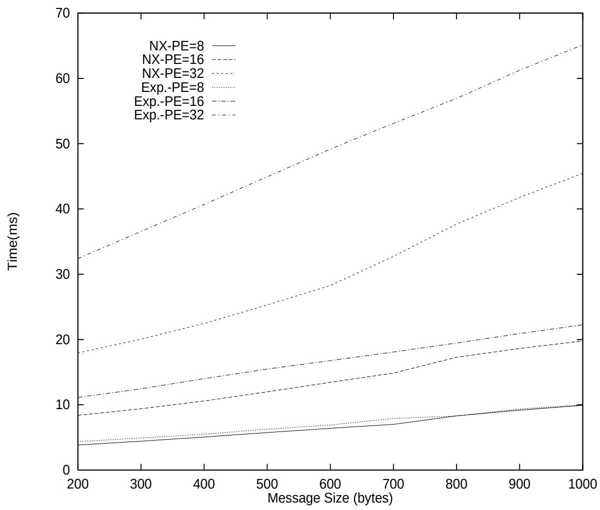

Figure 10(d): Complete exchange across 8, 16, and 32 processors on the iPSC/860 (200 to 1000 bytes).

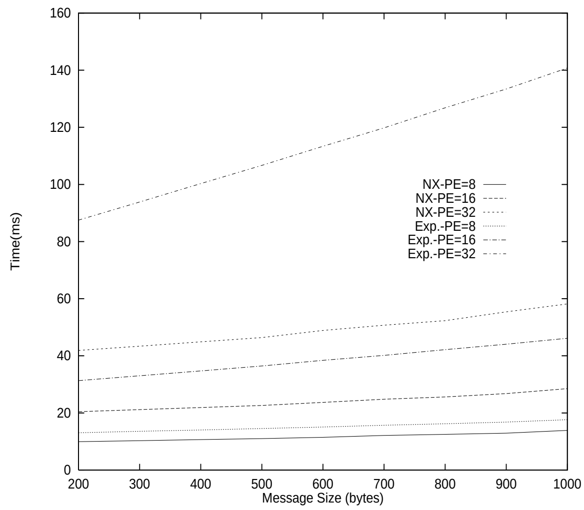

Figure 10(e): Complete exchange across 8, 16, and 32 processors on the iPSC/2 (200 to 1000 bytes).

on the iPSC/2. The plots of Express and NX are equally spaced for most message sizes, implying that the extra overhead of Express is constant. For messages of 1000 to 16000 bytes, Express is marginally slower than NX on the iPSC/860 and NX on the iPSC/2 is 2 to 3 times slower than iPSC/860.

8 Evaluation with an Application Benchmark Suite

In order to compare the performance of Express and NX systems with real applications, we have implemented an application benchmark suite. The applications in the benchmark suite include three different versions of Gaussian elimination, fast Fourier transform and the N-body problem.These algorithms has been coded using Express and NX primitives on the iPSC/860. We have obtained the execution times of these programs on 1, 2, 4, 8, 16 and 32 nodes. The execution time of a program is taken as the maximum of all the nodes. The results are given in the following sections.

8.1 Gaussian Elimination (Row-block Partitioning)

The three versions of Gaussian elimination for solving linear equations are based on partial pivoting algorithm. We have used different techniques to partition the data across processors. As a result, the algorithms used in the three versions are quite different. In this section, we present the results of the first version of Gaussian elimination. In this version, the data has been partitioned in blocks of rows, that is, an equal number of contiguous rows of the coefficient matrix are assigned to each processor. The program consists of three routines for generating the random data, performing Gaussian elimination and performing backward substitution. Only the Gaussian elimination routine is timed.

Table VIII: Timings (seconds) for row-partitioned Gaussian elimination using Express (and NX) on the iPSC/860.

Matrix \ PEs	1	2	4	8	16	32
256×256	5.94	3.75 (3.64)	2.31 (2.22)	1.81 (1.47)	2.11 (1.06)	3.46 (0.93)
384×384	24.71	12.99 (12.77)	6.79 (6.64)	4.72 (4.02)	4.44 (2.59)	6.87 (2.15)
512×512	59.63	31.13 (30.08)	15.65 (15.49)	9.20 (8.55)	8.50 (5.55)	11.38 (4.10)
640×640	120.87	62.62 (61.14)	31.06 (30.97)	17.49 (16.20)	13.96 (9.80)	17.37 (6.89)
768×768	219.03	108.24 (106.52)	54.71 (54.18)	28.66 (27.91)	20.77 (16.74)	24.69 (10.82)
896×896	347.10	182.77 (181.91)	89.06 (88.53)	46.53 (44.75)	31.19 (24.60)	34.01 (15.77)
1024×1024	520.07	261.94 (260.36)	131.08 (130.42)	66.21 (63.89)	43.23 (34.96)	45.56 (21.99)

The execution times of Gaussian elimination with row-block partitioning using Express with various matrix sizes and number of processors are shown in Table VIII. The corresponding execution times using NX are shown in the parenthesis. This table also indicates the times for serial execution of Gaussian elimination using one processor. As can be seen, the execution times for the Express version exhibit gain in speedup if the number of processors are increased from 1 to 16. The speed up is better if the matrix size is large. However, the execution times start increasing if 32 processors are used. On the other hand, the NX version still yields some speedup with 32 processors. The difference in the performance of Express and NX can be explained by the ratio of execution times of Express over NX. These ratios for various matrix sizes and number of processors are shown in Figure 11(a). One can observe that for small number of processors, there is not much difference in the performance of Express and NX. Express performs poorly with large number of processors. One can also observe that Express performs better for larger data sizes. The poor performance of Express with large number of processors is due to a number of factors. First, the algorithm used in this implementation makes an extensive use of broadcast operation for sending the pivot row to other processors. Second, no optimizations are made in communication calls to exploit the hypercube topology. As a result, the algorithm uses a number of global operations which are quite slow with Express. Finally, for small matrices and large number of processors, the grain sizes become small. Then the impact of large overhead in Express makes it significantly slower than NX.

8.2 Gaussian Elimination (Column-block Partitioning)

In this version of Gaussian elimination, the data is partitioned across processors in terms of blocks of columns. Table IX shows the executions times for this algorithm. It can be noticed that

Table IX: Timings (seconds) for column-partitioned Gaussian elimination using Express (and NX) on the iPSC/860.

Matrix \ PEs	2	4	8	16	32
256×256	7.03 (6.26)	3.64 (3.18)	2.78 (1.84)	3.79 (1.20)	5.35 (1.13)
384×384	23.92 (23.07)	11.57 (6.78)	6.75 (5.03)	7.64 (3.17)	12.30 (2.61)
512×512	47.55 (45.97)	27.48 (23.02)	14.26 (11.89)	13.19 (6.61)	20.25 (5.12)
640×640	110.18 (102.59)	56.49 (51.54)	28.62 (25.79)	20.75 (12.14)	30.30 (8.28)
768×768	182.78 (178.63)	98.72 (90.62)	50.24 (46.86)	30.96 (24.17)	42.69 (12.96)
896×896	290.50 (287.37)	151.39 (147.94)	79.62 (74.66)	49.90 (37.77)	57.52 (19.22)
1024×1024	408.08 (404.49)	226.55 (222.81)	118.61 (110.42)	65.52 (51.09)	80.61 (31.20)

the execution times in this case are significantly greater than those of the row-block partitioning algorithms. This is due to the fact that in the row-block partitioning algorithm, the determination of the pivoting row is done in parallel. On the other hand, column-block partitioning algorithm performs this step serially. The ratios of execution times of Express version over NX are shown in

Figure 11(b). Again, Express is shown to perform well for large matrices if the number of processors is between 2 and 16, but performs poorly for 32 processors. For smaller matrices, Express is 3 to 4 times slower than NX.

8.3 Gaussian Elimination (Column-scatter Partitioning)

In this version, the data is partitioned using cyclic distribution of the columns of the coefficient matrix. The results are given in Table X. The results indicate that this algorithm performs better than column-block partitioning but worse than row-block partitioning. This is due to the fact that column-scatter partitioning can balance load well, and as a result improve performance. However, it requires excessive exchanges of messages which can delay the execution. When the matrix size is small and the number of processors is large, the benefit of load balancing is small compared to extra cost of communication. The ratios of execution times of column-scatter partitioning algorithm with Express over NX are given in Figure 11(c), indicating that the degradation in performance using Express with this algorithm is better than the column-block partitioning algorithm but is comparable with row-block partitioning.

Table X: Timings (seconds) for column-scattered Gaussian elimination using Express (and NX) on the iPSC/860.

Matrix \ PEs	2	4	8	16	32
256×256	4.39 (4.07)	2.59 (2.21)	2.66 (1.44)	3.76 (1.25)	6.34 (1.22)
384×384	14.50 (13.92)	7.68 (6.68)	5.93 (3.79)	7.52 (3.00)	12.26 (2.76)
512×512	33.90 (31.98)	17.73 (16.23)	11.23 (8.90)	12.87 (5.81)	20.10 (5.04)
640×640	65.61 (63.27)	34.13 (32.10)	19.80 (16.14)	19.96 (9.83)	30.01 (8.14)
768×768	112.71 (110.67)	58.34 (55.59)	34.85 (28.61)	29.16 (15.42)	42.14 (12.46)
896×896	176.13 (172.06)	92.06 (88.42)	52.07 (45.01)	41.02 (22.98)	56.66 (17.46)
1024×1024	264.72 (260.42)	137.02 (131.85)	75.46 (63.98)	57.84 (33.26)	79.37 (23.91)

8.4 Fast Fourier Transform

For the fast Fourier transform, we used the algorithm given in [20] with modification. For N points, the algorithms's complexity is $N(\log N)$. The two dimensional matrix is partitioned across nodes in a row orientation style. The algorithm works by transforming the rows first, then transposing the rows with columns, transforming the columns and so on. We applied vector communication and reduced repeated computation.

Table XI shows the execution times for inputs ranging in size from 2k to 64k using various number of processors. The performance differences between Express and NX are found to be very small. As indicated in Figure 11(d), Express is only about 5 to 10% slower than NX for most of the cases. This is due to the fact that FFT has a very regular communication pattern. Due to smaller number of communication operations, the problem granularity is large and the effect of extra overhead of Express is substantially low.

Table XI: Timings (seconds) for fast Fourier transform using Express (and NX) on the iPSC/860.

Points \ PEs	1	2	4	8	16	32
64k	______	18.989 (18.507)	12.271 (12.036)	8.768 (8.688)	6.956 (6.939)	6.078 (5.987)
32k	14.314	8.822 (8.614)	5.738 (5.658)	4.115 (4.107)	3.325 (3.262)	2.904 (2.818)
16k	6.397	4.079 (3.988)	2.667 (2.647)	1.932 (1.915)	1.576 (1.522)	1.367 (1.314)
8k	2.940	1.887 (1.864)	1.243 (1.238)	0.911 (0.893)	0.739 (0.711)	0.645 (0.615)
4k	1.391	0.862 (0.857)	0.577 (0.568)	0.424 (0.411)	0.338 (0.328)	0.299 (0.285)
2k	0.635	0.395 (0.394)	0.265 (0.262)	0.197 (0.190)	0.160 (0.152)	0.140 (0.134)

8.5 The N-body Problem

The program for the N-body problem has been written using the algorithm reported in [20]. The algorithm used in this program is the simple $O(N^2)$ algorithm and not the more optimized $O(N(\log N))$ approach. The execution times for this problem are shown in Table XII using various number of bodies and processors while the ratios of execution times of Express to NX are shown in Figure 11(e). As can be seen from the table, this problem yields good speedup using either programming model. The difference in the performance of Express and NX is insignificant if small number of processors are used. For 32 processors, the performance difference is large for small problem sizes and but decreases as the problem size is increased.

Table XII: Timings (seconds) for the N-body problem using Express (and NX) on the iPSC/860.

Bodies \ PEs	1	2	4	8	16	32
16k	______	1401.2 (1342.9)	694.64 (674.42)	350.30 (339.83)	177.67 (170.04)	89.88 (85.34)
8k	657.88	342.19 (331.46)	171.23 (167.28)	86.99 (84.11)	44.47 (42.34)	24.01 (22.42)
4k	166.51	86.50 (84.16)	44.00 (42.51)	22.27 (21.58)	12.29 (11.43)	8.19 (6.32)
2k	42.71	22.99 (22.25)	11.36 (11.20)	5.90 (5.42)	3.35 (2.89)	2.23 (1.61)
1k	10.96	6.20 (5.69)	2.97 (2.97)	1.69 (1.49)	0.95 (0.81)	0.67 (0.53)
512	2.78	1.52 (1.49)	0.87 (0.85)	0.48 (0.41)	0.31 (0.26)	0.32 (0.20)

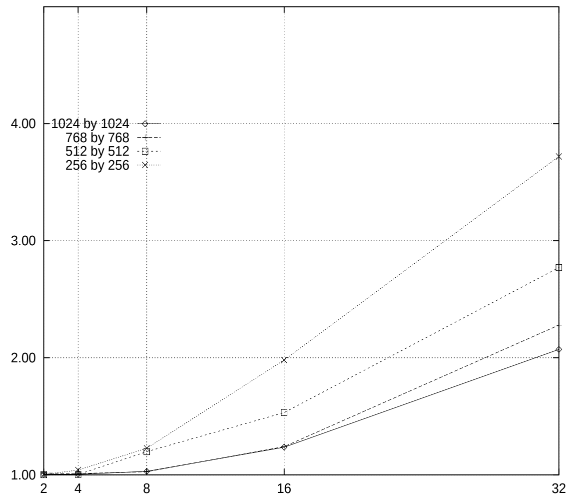

Figure 11(a): Ratios of execution times of Express to NX for row-block partitioned Gaussian elimination.

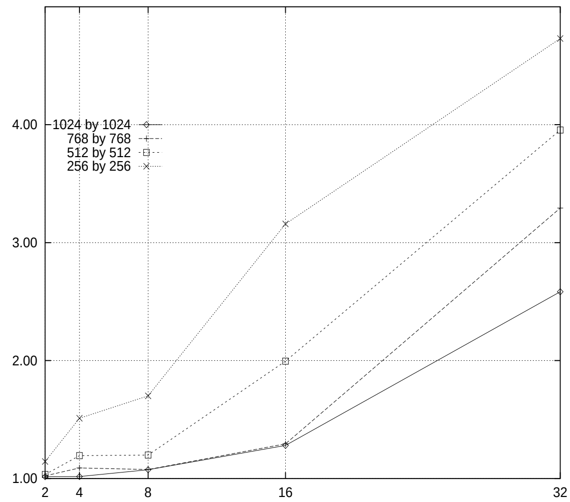

Figure 11(b): Ratios of execution times of Express to NX for column-block partitioned Gaussian elimination.

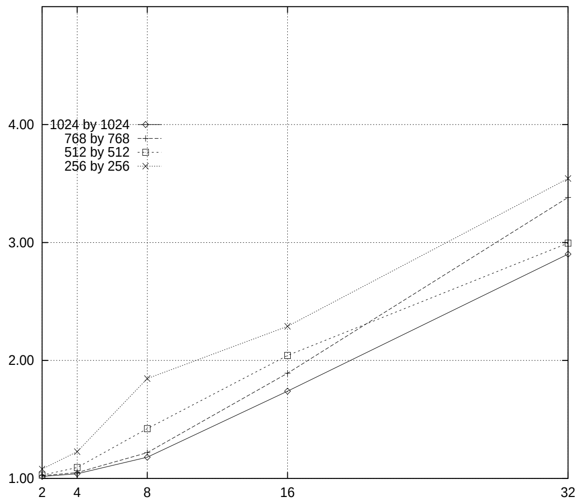

Figure 11(c): Ratios of execution times of Express to NX for column-scattered Gaussian elimination.

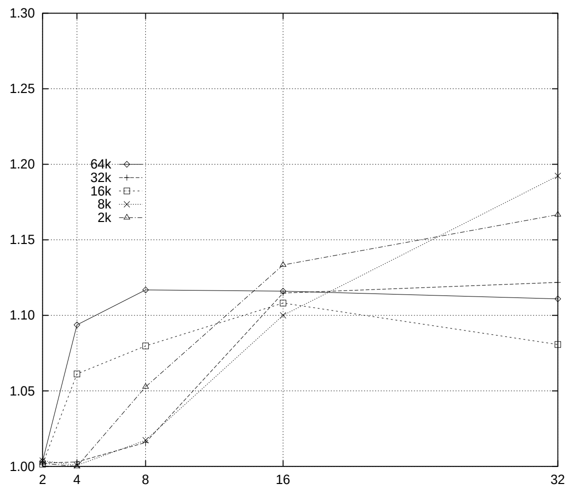

Figure 11(d): Ratios of execution times of Express to NX for fast Fourier transform.

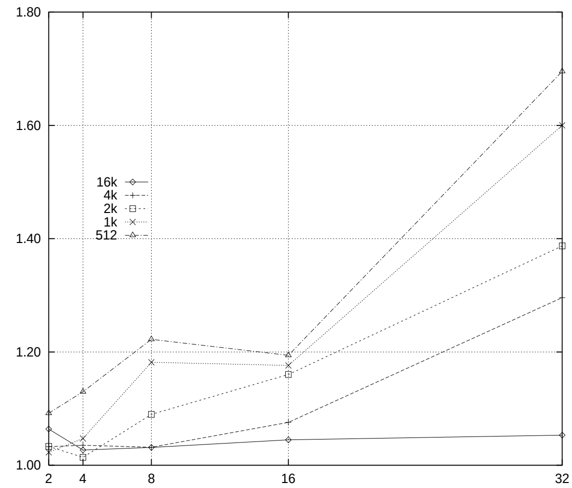

Figure 11(e): Ratios of execution times of Express to NX for N-body problem.

9 Conclusions

In this paper we studied the performance of Express on the iPSC/2 and iPSC/860. This is done by benchmarking various Express Primitives and their equivalent iPSC/2 and iPSC/860 primitives.

For one-to-one communication, the difference in the times of Express and NX on the iPSC/860 is negligible and is only about 15 to 20 microseconds. On the iPSC/2, Express with the older version was found to be 68 to 80% slower than NX. For very long messages, the difference can be as high as 0.3 milliseconds. As expected, the difference in the transmission times of Express and NX is not very large and the main difference lies in latencies. For sending messages to nodes at hop distances greater than 1, it takes 10 microseconds per hop distance for messages of length up to 100 bytes and about 30 microseconds for messages of length greater than 100. This time is almost the same with Express. Using NX, the iPSC/2 is 3.3 to 4 times slower than the iPSC/860.

The exchange primitive of Express is quite efficient than the simple one way send and receive. In the range of 0 to 100 bytes, Express is only 1 to 3% slower than NX. For messages of size greater than 100, Express is slower than NX by a factor of 15 to 22%. For very large messages, the relative performance differences become very small. On the iPSC/2, the exchange operation with Express is 70 to 80% slower than NX. The broadcast operation of Express for small messages is 8 to 10% slower than NX on the iPSC/860 and about 80% slower on the iPSC/2. For medium messages, compared to NX, the Express is again about 8 to 10% slower on the iPSC/860 and 28 to 30% on the iPSC/2. For very large messages, the difference is less than 1% on the iPSC/860 and less than 11% on the iPSC/2.

The concatenation operation and global operations such as addition, and multiplication, are rather slow with Express. On the iPSC/860, Express is about 30% slower than NX for small data size and about 50 to 60% for large data sizes. On the iPSC/2, Express is more than 100% slower than NX. The global operations on the iPSC/2 are found to be more than 2 times slower than iPSC/860. For the concatenation operation, Express is slower than NX by a factor of about 45% for data sizes less than 100 bytes and 60 to 80% for data sizes between 104 to 200 bytes. However, the relative performance degradation decreases for very large data sizes. The ring communication using Express is about 50 to 58% slower than NX on the iPSC/860 and 65 to 75% on the iPSC/2. The all-to-all broadcast operation is quite slower with Express. The complete exchange operation is comparatively faster using Express and is slower than NX only by a factor of 25 to 35% on the iPSC/860.

For most of the primitives, the relative performance of Express, compared to NX, decreases if data size is increased and the extra overhead tends to become negligible. However, Express tends to become slower with an increase in the number of processors.

In general, when programming with small granularity, Express shows significantly worse performance, but with large granularity, Express and NX will perform almost the same. Considering its functionality and advantage of portability, the overhead of Express appears not too large and is affordable.

Acknowledgements

We would like to express our thanks to Cornell Theory Center and Caltech's Concurrent Supercomputing Consortium for allowing us to use their machines. We also thank Jonathan Sewell and Chuck Koelbel for providing feedback on the implementation of test programs. This work was partly supported by Center for Research on Parallel Computation, U.S.A. under cooperative agreement no. CCR-8809165 and by the UPGC of Hong Kong under grant no. DAG92/93.EG06.

References

[10] S. Ahuja, N. Carriero and D. Gelernter, "Linda and Friends," *IEEE Computer*, no. 8, August 1986,

[11] I. Angus, G. Fox, J. Kim, D. Walker, *Solving Problems on Concurrent Processors, Volume II*, Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[12] G. Bell, "Ultracomputers - A Teraflop Before its Time," Communications of the ACM, pp. 27-47, Vol. 35, No. 8, August 1992.

[13] R. Berrendorf and J. Helin, "Evaluating the Basic Performance of the Intel iPSC/860 Parallel Computer," Concurrency: Practice and Experuience, vol. 4(3), May 1992, pp. 223-240.

[14] S. Bokhari, "Communication Overhead on the Intel iPSC/860 Hypercube," ICASE Interim Report 10 182055, NASA Langley Research Center, Hampton, VA, May 1990.

[15] S. Bokhari, "Multiphase Complete Exchange on a Circuit Switched Hypercube," Tech. Report 91-5, ICASE, NASA Langly Research Center, Hampton, Virginia, January 1991.

[16] L. Bomans and D. Roose, "Benchmarking the iPSC/2 Hypercube Multiprocessor," *Concurrency: Practice and Experience*, vol. 1(1), Sept. 1989, pp. 3-18.

[17] P. Close, "The iPSC/2 Node Architecture," *The Third Conference on Hypercube Concurrent Computers and Applications*, Vol. I, 1988, pp. 43-50.

[18] T. H. Dunigan, "Performance of the Intel iPSC/860 and Ncube 6400 Hypercubes," *Parallel Computing*, No. 17, 1991, North Holland, pp. 1285-1302.

[19] J. Flower and A. Kolawa, "A Packet History of Message Passing Systems," *Parasoft Corporation*, 1992.

[20] G.C. Fox, M.A. Johnson, G.A. Lyzenga, S.W. Otto, J.A. Salmon, and D.A. Walker, *Solving Problems on Concurrent Processors*, Volum I, Prentice Hall, 1988.

[21] G.A. Geist, M.T. Heath, B.W. Peyton and P.H. Worley, *A User's Guide to Picl, a Portable Instrumented Communication Library*, Technical Report ORNL/TM-11616, Oak Ridge National Laboratory, Oak Ridge, TN, Oct 1991.

[22] Intel Corporation, *iPSC/2 and iPSC/860 Programmers Reference Manual,* June 1990.

[23] A. Kolawa and S. Otto, "Performance of the MARK II and Intel Hypercube," Hypercube Multiproc-

essors 1986, M.T. Heath, ed., SIAM, Philadelphia, 1986, pp. 161-180.

[24] S. L. Johnson and C.T. Ho, "Optimal Broadcasting and Personalized Communication in hypercubes," *IEEE Tran. on Computers*, C-38, No. 9, Sept. 1989, pp. 1249-1268.

[25] M.H. Lee and S.R. Seidel, "Concurrent Communication on the Intel iPSC/2," Tech. Report CS-TR 90-03, Dept. of Computer Science, Michigan Technical University, July 1990.

[26] S.A. Moyer, "Performance of the iPSC/860 Node Architecture," Tech. Report IPC-TR-91-007, Institute of Parallel Computations, University of Virginia, May 1991.

[27] S.F. Nugent, "The iPSC/2 Direct-Connect Communications Technology," *The Third Conference on Hypercube Concurrent Computers and Applications*, Vol. I, 1988, pp. 51-60.

[28] Parasoft Corporation, *Express, Fortran User's Guide,* 1990.

[29] Parasoft Corporation, *Express Introductory Guide* version 3.2, 1992.

[30] C.L. Seitz, "The Cosmic Cube." Communications of the ACM, No. 28, 1985, pp. 22-33.

[31] T. Schmiermund and S.R. Seidel, "A Communication Model for the Intel iPSC/2," Tech. Report CS-TR 90-02, Dept. of Computer Science, Michigan Technical University, April 1990.

[32] V. Sunderam, "PVM: A Framework for Parallel Distributed Computing," Concurrency: Practice & Experience, Vol. 3, No. 4, Dec 1990, pp. 315-339.

[33] J. Yang, I. Ahmad and A. Ghafoor, "Estimation of Execution Times on Heterogeneous Supercomputer Architectures," *Int'l Conference on Parallel Procesing*, 1993, pp. I-219 - I-226