# CoMet: A Synthetic Benchmark for Message-Passing Architectures [*][†]

Nalini Ganapati, Steve W. Otto, and Jonathan Walpole
Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
20000 NW Walker Rd, PO Box 91000
Portland, Oregon, USA 97291-1000
otto,walpole@cse.ogi.edu

February 20, 1994

**Abstract**

Rapid advances in hardware technology have led to wide diversity in parallel computer architectures. This diversity makes it difficult to evaluate or compare the performance of different parallel computers. Existing benchmarks tend either to be too architecture-specific, or too high-level. Both problems can result in benchmarks that not only provide insufficient information on the performance characteristics of the computer being tested, but are also difficult to port. New benchmarking approaches are needed for new architectural classes, particularly distributed-memory, message-passing computers.

This paper focuses on benchmarking distributed-memory message-passing computers. A synthetic benchmark called CoMet (COmmunication METrics), is presented. CoMet is based on common communication patterns found in parallel scientific algorithms. This paper presents the CoMet design, and describes an implementation of CoMet on the Intel iPSC/860. CoMet is freely available by anonymous FTP from Oregon Graduate Institute.

## 1 Introduction

Rapidly changing technology has resulted in a proliferation of supercomputers with diverse architectures and programming models. This diversity makes it difficult to compare supercomputers, or to evaluate their suitability for running specific classes of applications. Evaluating a computer is a function of numerous issues that arise from the interplay of the application, system software, programming language, degree of optimization, and hardware architecture. Consequently, benchmarks designed for one class of architectures and applications can be inappropriate for evaluating computers and applications in a different class. This paper addresses the problem of benchmarking distributed-memory, message-passing multiprocessors, with emphasis on numerical, scientific applications.

Because of the difficulty in isolating performance effects, the general trend in evaluating supercomputers is to construct benchmarks from complete kernel and application programs. The idea is to predict the performance of a specific class of applications on the target computer without having to port the entire application. Instead, smaller benchmark programs with behavior similar to

[*]Submitted to *Concurrency: Practice and Experience*

[†]This work was partially supported by grants from Intel Supercomputing Systems Division and the Oregon Advanced Computing Institute (OACIS).

the application are ported and timed. The accuracy and usefulness of such benchmarks depend on several factors. First, the benchmark must be portable. That is, porting it to a new architecture should require few changes to the benchmark and should be considerably less time consuming than porting entire application programs.

Second, the benchmark should be representative of many aspects of the target application's behavior. To aid in porting existing applications and in developing new ones, the benchmark should also provide detailed information regarding the performance of the target computer for each aspect.

Existing benchmarks fall short on both these counts when used to evaluate distributed-memory message-passing computers. Regarding portability, time is wasted initially because benchmark programs must be rewritten in message-passing style. Later in the porting process, the optimization effort required to make the message-passing version of the benchmark efficient either further increases the cost of porting the benchmark, or it is ignored, making the benchmark results less accurate. Furthermore, since benchmarks for shared memory architectures are not designed around the concept of message-passing, they fail to capture the performance effects of various communication patterns commonly found in message-passing applications.

The following sections present a synthetic benchmark, called CoMet (COmmunication METrics), designed specifically for evaluating distributed-memory, message-passing computers. Unlike other benchmarks, such as Genesis [1, 20], the NAS Parallel Benchmarks [2], MPLinpack [10], and CPEP [24], which are based on application programs, CoMet is a synthetic benchmark containing code to measure the communication characteristics of the system. As well as measuring basic communication costs, CoMet also includes communication patterns found in typical scientific and engineering applications. Rather than reducing the results of the benchmark to a single number, they are presented in the form of graphs and communication bandwidth and latency figures. This approach provides detailed information that enables users to make realistic predictions of the performance of their own application programs.

The remainder of the paper is organized as follows. Section 2 discusses the overall design of CoMet and describes each of the benchmark kernels. An implementation of CoMet on the Intel iPSC/860[1] is described in section 3. Section 4 presents and discusses the results of running CoMet on the iPSC/860. Section 5 surveys related work, and section 6 concludes the paper.

## 2 The CoMet Benchmark

CoMet consists of two levels of benchmark "kernels." The low-level kernels measure the machine's basic communication capabilities by timing variable-size, message exchanges among nodes. These kernels comprise a basic "echo" benchmark to measure unidirectional message exchange, a "pairwise exchange" benchmark to measure bidirectional exchange, plus "broadcast," "global sum," and "global synchronization" benchmarks. The low level kernels also include "contention" and "overlap" kernels to measures the effect of load on the machine's basic communication performance. The "contention" kernel is based on running the echo kernel in the presence of artificially generated communication loads. The "overlap" kernel examines the machines ability to overlap communication and computation.

The remaining kernels measure the efficiency with which the machine can support certain patterns of communication. The communication patterns are generated by performing operations on a two-dimensional block-decomposed matrix. The benchmark includes kernels that update guard

[1]iPSC/860 is a registered trademark of Intel Corporation

strips, shift and transpose the matrix, and broadcast rows and columns. These kernels are intended to be representative of the behavior of scientific computations on distributed-memory architectures. Clearly, CoMet could be extended to include more communication patterns.

All of the kernels assume a distributed-memory message-passing architecture, and are written entirely in C. The individual kernels are described in detail in section 2.2.

## 2.1 Methodology

CoMet is based on a single program, multiple data (SPMD) model of parallel programming. Hence, one instance of the benchmark is loaded on each of the nodes in the system. Each processor proceeds independently between synchronization points. CoMet also supports an optional host program for machines that have a front-end machine and require a separate host program to load the individual node programs. The host program has the three functions listed below.

1. Allocate the required number of nodes.
2. Load the benchmark programs on the allocated nodes.
3. Assign each node a unique number in the range 0..P-1, where P is the number of nodes allocated.

The time measurements in CoMet are based on elapsed, wall-clock time measured on a dedicated system. This approach avoids complications due to external system load and architecture-specific measures of CPU time. Furthermore, CoMet is structured such that all timing measurements can be made on a single node. This approach avoids the need for globally synchronized clocks that are generally not available on distributed architectures. Finally, architectures with low clock resolution are handled by using a repeat loop to execute each kernel multiple times for each time measurement. The use of a call to a dummy function inside the loop, with the increment variable as a parameter, prevents optimizing compilers from removing this loop. The overhead of the repeat loop and dummy function calls is measured separately and subtracted from the total time in order to derive the final measurement.

CoMet is designed to work over a wide range of architectures, from loosely coupled networks of workstations to hypercube and tree-based communication networks [28]. However, it is unrealistic to expect the benchmark kernels to be completely architecture independent. Since CoMet contains two-dimensional matrix operations, it is important to map the matrix to the processing nodes in a manner appropriate to the architecture. CoMet defines a two-dimensional matrix of processors that captures nearest-neighbor mappings. This matrix is referred to as the adjacency matrix for the machine. This approach allows a two-dimensional application matrix to be block-decomposed such that adjacent blocks of the application matrix are located on nearest-neighbor nodes in the hardware architecture. The adjacency matrix can be specified by hand or in other ways. In our implementation on the iPSC/860 it is specified using Gray codes [17].

Although there have been several attempts at message-passing standards, such as PARMACS [6], PICL [19], and MPI [16], none are yet established as standards. CoMet implements communications using high-level macros and functions to describe the different patterns. The current macros are written in NX/2, but can be ported to other systems. The locally synchronous[2] communication

[2] In the locally synchronous model [16], a send blocks until the application buffer is copied into system space and is available for reuse. Similarly, a receive blocks until the contents of the message are completely copied into the process's application buffer.

macros used in CoMet are *blocked_send*, *blocked_recv*, and *blocked_broadcast*. The macros for asynchronous communication are *unblocked_send* and *unblocked_recv*, and are used in conjunction with the *wait_for_send_to_complete* and *wait_for_recv_to_complete* macros. The *unblocked_probe_msg* macro is provided to detect the arrival of typed messages. Finally, the *barrier* macro is provided to allow synchronization among nodes at the start of the benchmark kernels. Supporting communication in this way allows communication library functions to be invoked from within the benchmark kernels in a portable manner, but with minimal overhead.

CoMet also provides a set of support functions and macros for tasks such as initialization, clean-up, timing, and topology description (the number of nodes, the mapping of nodes to the adjacency matrix, the identification of nearest neighbors, and the measurement of inter-node distances, etc). For more details see [18].

Finally, our approach to benchmark integrity is to specify explicitly the functions and macros that are allowed to be modified by CoMet users. Users are not supposed to alter any other parts of the benchmark source code. CoMet does, however, permit the use of any level of compiler optimization. The motivation for this approach is that special compiler techniques that optimize communication arising from common matrix manipulations are likely to be of widespread benefit to applications.

## 2.2 Benchmark Kernels

### 2.2.1 Basic Communication Kernels

CoMet's basic communication benchmarks are organized into a number of kernels, each of which transfers varying length messages between system nodes. The first kernel, **echo**, measures the cost of unidirectional message transfer. The second, **pairwise exchange**, measures the cost of bi-directional message exchange[3]. The next three kernels, **broadcast**, **global sum**, and **barrier** measure a machine's support for global communication and synchronization. The **echo with contention** kernel measures the effects of contention on communication performance, and the **overlap** kernel measures the extent to which computation and communication can be overlapped. Each of these kernels is described below.

**Echo** The echo kernel (Figure 1) is based on a simple, uni-directional transfer of messages between nodes. A test node sends a message to another node and waits for a reply message. Time measurements are taken only on the initiating node. Therefore, the time for the uni-directional transfer is assumed to be half of the measured time. Echo generates a set of results for different message sizes and various inter-node distances.

**Pairwise Exchange** The pairwise exchange kernel (figure 2) is an extension of the echo kernel that measures the time to exchange messages between two nodes. Two nodes simultaneously call send and then receive in order to exchange messages. Timing is performed on one node only. This kernel assumes buffered message passing semantics for the simultaneous sends. Note that on systems that do not support such semantics, this kernel can deadlock.

**Broadcast** The broadcast kernel (figure 3) measures the time taken to broadcast a message to all other nodes. In the absence of a broadcast primitive or library support, this kernel can be

[3] This kernel has the potential to deadlock on machines with non-buffered message-passing semantics.

```
begin
    for ($hop$ = 0 to $hop$ = maximum hops possible)
    begin
        for ($message\ size$ = 0 to $message\ size$ = over a page size)
        begin
            $t1$ = get time;
            if ($test\ node$) then
                send message to another node;
                receive message from other node;
            else
                if ($communicating\ node$)
                    receive message from test node;
                    send message back to test node;
                end if
            end if
            $t2$ = get time;
            echo time on test node = $t2 - t1$;
        end for
    end for
end
```

Figure 1: Pseudo-code for **echo**

```
begin
    for ($hop$ = 0 to $hop$ = maximum hops possible)
    begin
        for ($message\ size$ = 0 to $message\ size$ = over a page size)
        begin
            $t1$ = get time;
            if ($test\ node$) then
                send message to communicating node;
                receive message from communicating node;
            else
                if ($communicating\ node$)
                    send message to test node;
                    receive message from test node;
                end if
            end if
            $t2$ = get time;
            pairwise exchange time on test node = $t2 - t1$;
        end for
    end for
end
```

Figure 2: Pseudo-code for **pairwise exchange**

```
begin
    for (message size = 0 to message size = over a page size)
    begin
        t1 = get time;
        if (test node) then
            broadcast message to all nodes;
        else
        begin
            receive message broadcasted;
        end if
        barrier();end if
        t2 = get time;
        broadcast time on test node = t2 - t1;
    end for
end
```

Figure 3: Pseudo-code for **broadcast**

```
begin
    double precision : vector;

    for (vector length = 0 to vector length = max_vec_length)
    begin
        t1 = get time;
        vector := global sum of vector across all nodes;
        t2 = get time;
        global sum time on test node = t2 - t1;
    end
end
```

Figure 4: Pseudo-code for **global reduction**

implemented using a series of sends. As with the other kernels, broadcast generates results for various message sizes.

**Global Reduction** The global reduction kernel (figure 4) is related to the broadcast kernel in the sense that it requires each node to send a value to all other nodes. However, global reduction also requires each node to calculate the global sum of the individual elements of a double precision vector stored on each node. At the end of the global reduction, all nodes contain the answer. The global reduction kernel is executed for varying vector lengths.

**Global Synchronization** The global synchronization kernel measures the minimum time required to complete a global barrier synchronization. Timing it is difficult because different nodes may be at different stages of execution when it is first called. Hence, attempting to measure the best-case time for a global synchronization makes more sense than measuring the worst-case time. The global synchronization kernel accomplishes this by repeatedly calling and timing global synchronizations. The time for the first barrier is not reported by the benchmark.

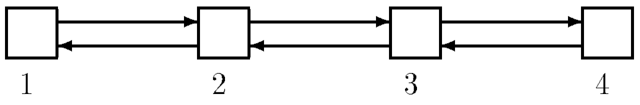

Figure 5: The logical topology for the contention kernel. The arrows denote communication links and the boxes denote nodes. Nodes 2 and 3 run the **echo** kernel while nodes 1 and 4 provide cross traffic.

**Contention** All the kernels described above are intended to be run on idle systems. In order to study the effects of system load on communication performance, the contention kernel (Figure 5,6) introduces measured loads into the system while running the echo kernel. It chooses four nodes such that two can introduce load (by continuously exchanging messages) on the communication links between the other two that are running the **echo** kernel. The choice of these four nodes is architecture dependent. For example, on an eight-node iPSC/860 hypercube, nodes 0 and 7 would be chosen to continuously exchange messages, i.e., they would correspond to nodes 1 and 4 respectively in figure 5. These messages would pass through nodes 1 and 3, which would be running the echo kernel in order to gather measurements, i.e., hypercube nodes 1 and 3 would correspond to nodes 2 and 3 respectively in figure 5. The degree of load introduced is varied by transferring messages of different sizes.

**Overlap** The overlap kernel (figure 7 and 8) is designed to measure the extent to which communication and computation can be overlapped on the target architecture. First, it runs the pairwise exchange kernel with a DAXPY ([11]) computational part using synchronous communication. Then it repeats the pairwise exchange kernel, overlapping asynchronous communication with the DAXPY computational part. The amount of computation is varied by changing the lengths of the vectors for the DAXPY, and the amount of communication is varied by changing the message size. The kernel uses different message sizes and vector lengths to explore the extent to which overlap can be achieved.

### 2.2.2 Matrix-Related Kernels

Although scientific computation gives rise to a variety of communication patterns, matrix manipulation is central to many scientific applications. Therefore, the kernels in CoMet's higher layer are modeled on common matrix manipulation operations. The **update guard**, **shift**, **transpose**, and **row/column broadcast** kernels are based on a two dimensional matrix that is block-decomposed, with a one-element guard-wrapper[4] over a set of acquired nodes. A setup module is used to initialize the matrix with values that are un-writable by the user. Similarly, the values in the resultant matrix are checked by a verification module. All matrix operations are invoked as macros or functions that can be in-lined, and all but the setup and verification modules can be re-implemented by the user, as desired.

[4]The guard wrapper conceptually surrounds the part of the matrix local to the node. It is used to store the neighboring values of the matrix that are resident on neighboring nodes. The motivation behind the guard wrapper lies in the fact that a node can access some of the non-local values of the global matrix by simply referring to the values in this guard wrapper - see Figure 9.

```
begin
    if ($node$ == 1 or $node$ == 4) then
        forever loop
            if ($node$ == 1) then
                send message to fourth node;
            end if
            if ($node$ == 4) then
                receive message from first node;
            end if
        end for
    else
        if ($node$ == 2 or $node$ == 3) then
            $t1$ = get time;
            if ($node$ == 2) then
                send message to third node;
                receive message from third node;
            else
                if ($node$ == 3)
                    receive message from second node;
                    send message back to second node;
                end if
            end if
            $t2$ = get time;
            echo time with contention (on node 2) = $t2$ - $t1$;
        end if
    end if
end
```

Figure 6: Pseudo-code for **echo with contention**

```
begin (synchronous version)
    $t1$ = get time;
    if ($test\ node$) then
        send message to other node;
        receive message from other node;
    else
        if ($communicating\ node$) then
            send message to test node;
            receive message from test node;
        end if
    endif
    perform DAXPY computation (vector length);
    $t2$ = get time;
    time without overlap on test node = $t2$ - $t1$;
end (synchronous version)
```

Figure 7: Pseudo-code for synchronous portion of the **overlap** kernel

```
begin (asynchronous version)
    t1 = get time;
    if (test node) then
        post a send message to other node;
        post a receive message from other node;
    else
        if (communicating node) then
            post a send message to test node;
            post a receive message from test node;
        end if
    endif
    perform DAXPY computation;
    wait for sends and receives to complete;
    t2 = get time;
    time with overlap on test node = t2 - t1;
end (asynchronous version)
```

Figure 8: Pseudo-code for asynchronous portion of the **overlap** kernel

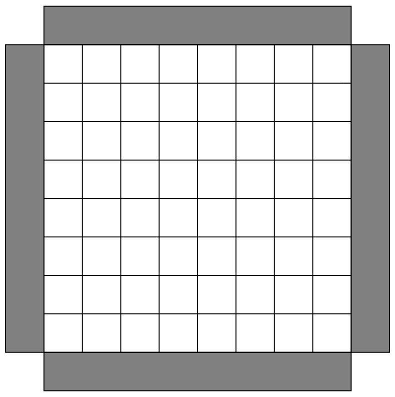

Figure 9: The view of the block-distributed matrix on a node. The shaded portions refer to the guard wrapper, while the inner values of the matrix refer to the parts of the matrix local to the node.

**Update Guard** The update guard kernel (figure 11) is implemented as a function that takes as parameters a pointer to the local matrix and its dimension, and updates the non-local elements in the guard wrapper by communicating with neighboring nodes. The identities of the neighboring nodes are specified in the global values *pred*, *succ*, *top* and *bottom* for each node by dereferencing the adjacency matrix. The northern portion of the guard wrapper thus contains values from the *top* node, the southern portion of the wrapper contains values from the *bottom* node, and the eastern and western portions of the wrapper contain values from the nodes identified by *pred* and *succ* respectively (see figure 10).

**Shift Matrix** The shift matrix kernel (figure 12) shifts a distributed matrix by N/P elements, where N/P is the number of matrix elements divided by the number of processors in the direction being shifted. This is a commonly used matrix operation. The kernel implements the local portions of the matrix, with guard wrappers, as an array. To simplify buffer management, the kernel shifts

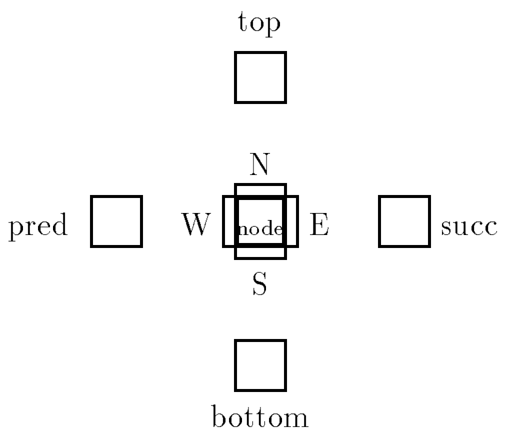

Figure 10: View of neighboring nodes with respect to a node and its guard wrapper.

```
begin
    t1 = get time;
    read northern guard wrapper values into a vector;
    send vector to top node;
    read southern guard wrapper values into a vector;
    send vector to bottom node;
    read eastern guard wrapper values into a vector;
    send vector to succ node;
    read western guard wrapper values into a vector;
    send to pred node;
    receive vectors from top, bottom, pred and succ nodes;
    update southern, northern, eastern and western guard wrapper values;
    t2 = get time;
    update time = t2 - t1;
end
```

Figure 11: Pseudo-code for **update_guard(***local matrix*, *local dimension***)**

```
begin
    t1 = get time;
    if (direction is NORTH) then
        send local matrix to top node;
    else
        if (direction is EAST) then
            send local matrix to succ node;
        end if
    end if
    receive matrix into local buffers;
    t2 = get time;
    shift matrix time = t2 - t1;
end
```

Figure 12: Pseudo-code for **shift_matrix(***local matrix*, *local dimension*, *direction***)**

```
begin
    t1 = get time;
    identify transpose node;
    send local matrix to transpose node;
    receive matrix into local buffers;
    transpose local matrix;
    t2 = get time;
    transpose time = t2 - t1;
end
```

Figure 13: Pseudo-code for **transpose_matrix(***local matrix*, *local dimension***)**

the whole local matrix along with its guard wrapper. In addition to taking a pointer to the local matrix and the local matrix dimension as parameters, the shift matrix kernel also allows the shift direction to be specified as a parameter. The enumerated variable NORTH is used as the direction to be shifted if the matrix has to be shifted upward and the enumerated variable EAST is used if the matrix has to be shifted sideways. Following the shift, the shifted part of the matrix resides on each node in the original local matrix data structure.

**Transpose Matrix** The transpose matrix kernel (figure 13) is included because it is an important application in its own right and yields a completely different communication pattern from the previous kernels. The kernel involves two stages: an exchange of local matrices among nodes, followed by an internal transpose of the local matrix. The target nodes for exchanging data are determined from the adjacency matrix. Transpose could also have been implemented using the divide and conquer approach, which would have led to a completely different communication pattern. We did not implement this in CoMet. However, it would be interesting to compare the results of the two approaches. The transpose kernel only implements the simple transpose algorithm for which the sub-matrix on each processor is square.

**Row and Column Broadcast** The row broadcast and column broadcast kernels (figure 14) measure the speed with which a machine can broadcast a single row and a single column of a block-

```
begin
    t1 = get time;
    if (node possesses part of the row) then
        identify nodes to which the part of the row has to be broadcast;
        loop for all identified nodes
            send local part of the row to identified node;
        end loop
    else
        receive broadcast row;
    end if
    t2 = get time;
    time to broadcast row = t2 - t1;
end
```

Figure 14: Pseudo-code for **row_broadcast**(*local matrix*, *local dimension*, *row*)

distributed matrix to all other processors. This kernel is important for LU decomposition. The kernels take as input the (global) index of the row or column to be broadcast. If a node possesses part of the specified row or column, it multi-casts it to all other nodes in a direction orthogonal to the row or column. That is, rows are multi-cast in the vertical direction of the matrix, columns are multi-cast in the horizontal direction of the matrix. Nodes that do not contain a piece of the specified row or column wait to receive part of the row or column. The time that is reported by CoMet represents the rate at which broadcasts can be initiated. Other useful measures not currently reported by this kernel are: the machine-wide latency in completing the broadcast; the times for the initiators and receivers of the broadcasts could be made distinct.

## 3 Implementing CoMet on the Intel iPSC/860

In this section, we describe an implementation of CoMet on the Intel iPSC/860.

### 3.1 Architectural Overview of the Intel iPSC/860

The iPSC/860 is a distributed-memory multiprocessor based on a hypercube interconnect and Intel i860 processors. Each node is connected to the hypercube communication network via a direct connect module (DCM). DCMs support 8 bit-serial and bidirectional 2.8 MB/sec channels to connect to direct neighbor nodes. Since one channel in the DCM is reserved for special I/O, the maximum size of the iPSC/860 is 128 nodes. Communication paths between any two nodes are established dynamically using free channels and an e-cube algorithm [25]. These paths are freed once the communication request has been completed. Each node on the iPSC/860 hypercube runs the NX/2 node operating system. NX/2 [26, 21] performs process management and message passing and allows only one application to execute per node at a time.

### 3.2 Communication Protocols in NX/2

The communication protocol used by NX/2 is based on buffers. The system buffers all incoming and outgoing messages such that no rendezvous is necessary for any two communicating processes. Each NX/2 instance has a number of 100-byte buffers reserved for every other node that can communicate with that node. NX/2 (on node A) also keeps track of bounds on the number of free buffers (reserved for node A) on all other nodes. Applications identify the memory area containing the message to be communicated by passing a pointer to it as a parameter to the message passing primitive. In the blocking version of the send primitive, the application is then blocked from further execution until the message has been successfully copied into the system buffers.

For messages smaller than or equal to 100 bytes, NX/2 on the sending node tries to determine (locally) if there is a free buffer reserved for it on the receiving node. If it knows that such a free buffer exists, NX/2 decrements its count of free buffers available for it on the receiving node and sends the data. If it isn't sure that there is an available buffer (the counts are merely bounds), NX/2 enters a non-local protocol (handshake) with the receiving node that will wait until a buffer becomes available. The system piggy-backs buffer statuses on messages. In this way, the free buffer counts are eventually incremented.

For larger messages, allocation is performed dynamically in a circuit-switched manner. A short message is sent requesting that a buffer, equal in size to the length of the message, be allocated. The sending node waits for an acknowledgement before sending the data.

Three different semantics for sending and receiving data are provided:

1. blocking send/receive (locally synchronous),
2. non-blocking send/receive (asynchronous) and,
3. interrupt-driven send/receive (also asynchronous).

Using the blocking primitives, the sender is blocked until the message has been copied into the system buffers and the receiver is blocked until the message data has been copied into the receive buffer in its address space. The C version of the calls are

**csend(message_type, buffer, buffer_size, id_of_receive_node, process_id)**,
**crecv(message_type, buffer,buffer_size)**.

At present NX/2 supports only a single process executing at a time on each node. The parameter **process_id** is therefore unused and is to be specified as an integer 0 in the calls.

For the non-blocking primitives, the sender and receiver initiate the communication with a call and return immediately with an identifier for the message being sent/received. Before re-using the contents of either send or receive buffers, applications must check for completion of the non-blocking call. This checking is accomplished using **msgwait**.

**message_id = isend(message_type, buffer, buffer_size, id_of_receive_node,**
**process_id)**,
**message_id = irecv(message_type, buffer, buffer_size)**,
**msgwait(message_id)**

The last type of communication semantics is implemented using:

**hsend(message_type, buffer, buffer_size, id_of_receive_node, process_id,**
**handler)**,
**hrecv(message_type, buffer, buffer_size, handler)**.

These calls return as soon as possible, however, instead of requiring the application to explicitly check for completion, NX/2 directly invokes a handler function specified with the call.

In addition to the basic message passing primitives, the programmer is also provided with system calls to search for specific messages identified by their message types ("probing"). Again, these calls are implemented in both blocking and non-blocking versions. In the non-blocking form, the call returns a flag to indicate the result of the search for the message of a specific type. The blocking probe is,

**cprobe(message_type),**

while the non-blocking form is,

**flag = iprobe(message_type).**

## 3.3 Mapping CoMet Primitives

To port CoMet to the iPSC/860, we

1. mapped all the macro calls to iPSC/860 message passing primitives,
2. set all the macro variables,
3. mapped the hardware hypercube topology onto a two-dimensional grid,
4. wrote the functions to implement the matrix manipulation routines, and
5. wrote a shell script to extract and organize all relevant output data into a form acceptable to gnuplot.[5]

The iPSC/860 does not require any special initialization in order to start CoMet other than loading the various kernels onto the nodes. Only one process is initiated on each node and since the nodes are numbered contiguously starting at 0, by the operating system itself, porting CoMet to the iPSC/860 was relatively straightforward. CoMet's **get_node_id** and **get_num_nodes** support functions were mapped directly onto the NX/2 primitives **mynode()** and **numnodes()** respectively. Table 1 lists the mapping of CoMet macros to iPSC/860 primitives and values.

Both the benchmark macros and iPSC/860 message passing primitives are based on buffered communication, making it straightforward to map the macros to NX/2 message passing calls. The macros **blocked_send** and **blocked_recv** were mapped onto **csend()** and **crecv()**, and the macros **unblocked_send** and **unblocked_recv** were mapped onto **isend()** and **irecv()** respectively. CoMet's **blocked_broadcast** macro was implemented using the **csend()** primitive with the input parameter identifying the node the message is intended for, set to -1. The **wait_for_send_to_complete** and **wait_for_recv_to_complete** macros were both mapped onto the system call **msgwait()**. The macro implementing the search for specific messages **unblocked_probe_msg** was mapped onto **iprobe()** and the macro **barrier** was mapped onto **gsynch()**.

To measure elapsed times for the various benchmark kernels, the macro **get_time** was mapped onto the system call **dclock()** which returns double precision time in seconds since the machine was booted. The elapsed times for the benchmark were found to converge when averaged over about 1000 runs. Hence, to obtain accurate measurements, the macro **max_times** was set to 1000. Finally, the macro **page_size** was set to the page size of the i860 microprocessor, which is 4096.

[5] Gnuplot is a freely-available, plotting package available from the Free Software Foundation.

| | |
|---|---|
| page_size | 4096 |
| max_times | 1000 |
| max_vec_len | 200000 |
| get_time() | dclock() |
| blocked_broadcast(msgtyp, msg, msg_size) | csend(msgtyp, msg, msg_size, -1, 0) |
| blocked_send(msgtyp, msg, msg_size, node) | csend(msgtyp, msg, msg_size, node, 0) |
| blocked_recv(msgtyp, msg, msg_size) | crecv(msgtyp, msg, msg_size) |
| unblocked_probe_msg(msgtyp) | iprobe(msgtyp) |
| barrier() | gsync() |
| global_sum(num, dummy) | gdsum(&num, 1, &dummy) |
| unblocked_send(msgtyp, msg, msg_size, node) | isend(msgtyp, msg, msg_size, node, 0) |
| unblocked_recv(msgtyp, msg, msg_size) | irecv(msgtyp, msg, msg_size) |
| wait_for_send_to_complete(id) | msgwait(id) |
| wait_for_recv_to_complete(id) | msgwait(id) |

Table 1: Mapping of macros. The left-hand column contains CoMet macros, the right-hand column contains the corresponding NX/2 implementation.

To measure the overlap of computation and communication, the macro specifying maximum vector length for the DAXPY part, **max_vec_len**, was set, somewhat arbitrarily, to 200000 based on the memory available in the system and to observe significant overlaps of communication with computation. The message lengths in the communication part of the kernel ranged from 0 to about 85000, to observe the behavior of small to large messages in the benchmark kernels.

To map the hypercube topology onto a two-dimensional mesh, a Gray code was used [17]. The elements in the mesh represent nodes in the system such that adjacent elements in the matrix are also physical neighbors. The dimensions of the mesh are determined by variables **proc_dim1** and **proc_dim2**. If the number of nodes allocated to the benchmark is a perfect square, the two dimensions are set equal to the square root of the number of nodes allocated. Otherwise, **proc_dim1** is twice **proc_dim2** and is set equal to the square root of twice the number of processors allocated to the benchmark. The function **get_hop_node()** which returns the identity of nodes $n$ hops away, is defined using bit arithmetic that characterizes the hypercube configuration [17].

The **update_guard**, **shift_matrix**, **transpose**, **row_broadcast** and **col_broadcast** kernels were written using a combination of **csend**, **crecv**, **isend** and **irecv**. For **update_guard**, the northern, southern, eastern and western values were assembled into separate buffers, and the four assembled buffers were sent to the neighboring nodes **top**, **bottom**, **pred** and **succ** respectively. To update its own guard wrappers, each node receives messages from the four neighboring nodes and copies them into its respective guard wrappers. This function is implemented such that the assembly of buffers is overlapped with communication (see Figure 15).

In the **shift_matrix** kernel, the entire local matrix is sent to a neighboring node using **csend()**, and the shifted matrix is received into the same buffer using **crecv()**. The **transpose** kernel is implemented in a straightforward manner, with the identity of the node to exchange messages with being obtained by dereferencing the processor mesh with the **node_row** and **node_col** transposed. After receiving the local matrix from another node, each node performs an internal transpose to complete the operation. Finally, the **row_broadcast** and **col_broadcast** kernels were implemented using **csend** and **crecv** (see Figure 16).

```
begin
    assemble north guard wrapper into buffer;
    initiate send to north neighbor top;
    assemble south guard wrapper into buffer;
    initiate send to south neighbor bottom;
    assemble east guard wrapper into buffer;
    initiate send to east neighbor succ;
    assemble west guard wrapper into buffer;
    initiate send to west guard wrapper pred;
    initiate receive from bottom to fill north guard wrapper;
    initiate receive from top to fill south guard wrapper;
    initiate receive from pred to fill west guard wrapper;
    initiate receive from succ to fill east guard wrapper;
    wait for all sends and receives to complete;
end
```

Figure 15: Pseudo code for **update_guard**.

```
row = global_row/local_dim;
local_row = global_row%local_dim;
ptr = local_matrix+(local_dim+2)*(++local_row)+1;
if (node_row == row) {
        for(i=0; i<proc_dim2; i++) {
                if (i != node_row)
                        csend(ROW_CAST, ptr, local_dim*sizeof(double),
                        (i*proc_dim1+node_col), 0);
                }
                for (i=1; i<proc_dim2; i++)
                        crecv(GOTIT, null, 0);
        }
        else {
                crecv(ROW_CAST, ptr, local_dim*sizeof(double));
                csend(GOTIT, null, 0, infonode(), 0);
        }
```

Figure 16: C code fragment for **row_broadcast**

## 4 Results and Analysis

CoMet was implemented and executed on 2, 4, 8 and 16 nodes of the iPSC/860. The results obtained from running the benchmark are presented in the form of graphs together with a brief analysis.

### 4.1 Basic Communication Kernels

The fundamental characteristics associated with message-passing machines are the start-up time and time taken per byte of data. These characteristics can be derived using CoMet by running the **echo** kernel and plotting the elapsed times for different message sizes. If $t_{start-up}$ is the message latency for a 0-byte message and $t_{per-byte}$ is the transfer time for one byte, the time taken to echo a message of $N$ bytes from a neighboring node one hop away, is modeled by,

$$t(N) = t_{start-up} + Nt_{per-byte}.$$

For messages echoed between nodes that are more than one hop apart, say $n$ communication links away, an additional overhead proportional to the number of hops is expected. This behavior is reflected in figure 17. The time taken for a message of $N$ bytes to be echoed from a node $n$ hops away thus becomes,

$$t(N) = t_{start-up} + Nt_{per-byte} + (n-1)h,$$

where $h$ is the incurred overhead per hop. Bomans and Roose [6], Dunigan [13, 14], Berrendorf and Helin [4] have all used the above empirical formulae in interpreting the communication characteristics of the iPSC/860. However, the echo graph also shows discontinuities at around 100, 2048 and 4096 bytes. The first discontinuity is easily explained because the operating system uses a different protocol for messages of size smaller than or equal to 100 bytes. The other two discontinuities are more difficult to explain. Messages of size 4096 bytes are guaranteed to lie across page boundaries which could potentially lead to additional overhead. However, it is interesting to note that message transfers get *faster*, not slower, at around 2048 and 4096 bytes. We are unable to explain this behavior.

One can usefully fit the data in the regions between the discontinuities. A linear least-squares fit can be used to deduce $t_{start-up}$ of around $79\mu s$ for transmitting messages less than 100 bytes and $156\mu s$ for messages greater than 100 bytes. Similarly, $t_{per-byte}$ is around $0.63\mu s$ for messages less than 100 bytes and $0.41\mu s$ for messages greater than 100 bytes, and the overhead $h$ is $25\mu s$ for messages greater than 100 bytes. Adding hops seems to add only a small overhead to the total transmission time. Hence, wire time does not account for much of the transmission time.

The graph in Figure 18 presents the same results as before, but as bandwidth figures. The peak bandwidth for the echo kernel between adjacent nodes is 2.5 MB/s. Half of the peak bandwidth is achieved with message sizes of 486 bytes and ninety percent of the peak bandwidth is achieved with message sizes around 5000 bytes. The message sizes that produce 50% and 90% peak bandwidth utilization will be used later to generate load in the contention kernel.

The results for the broadcast kernel plot the elapsed times against message size — see Figure 19. The different curves correspond to different machine sizes. Again, a discontinuity is observed at a message size of 100 bytes due to the change of protocol. For message sizes larger than 100 bytes, the time taken to broadcast messages to all other nodes increases linearly with respect to message size. As expected, the broadcast times for different machine sizes increases linearly with the log of the number of processors.

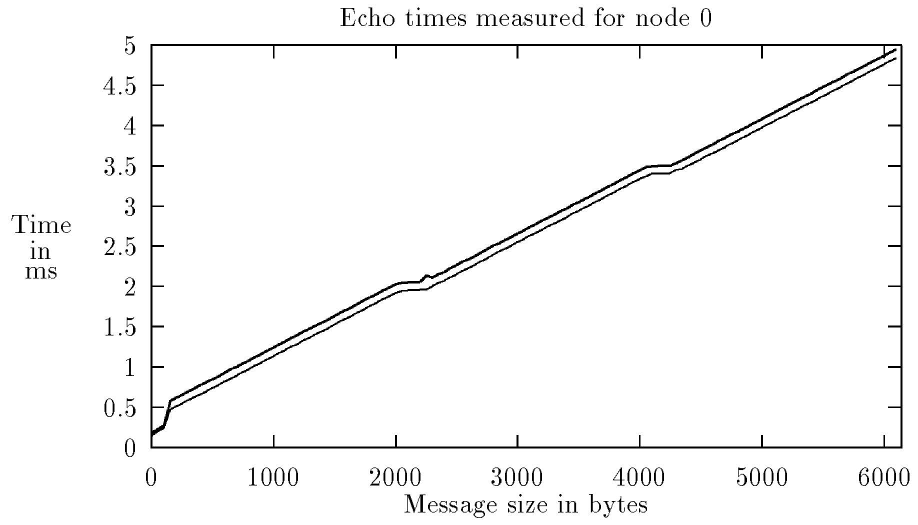

Figure 17: Echo kernel on 16 nodes. The lower line represents the echo times between adjacent nodes and the upper line represents the echo times between nodes 4 hops apart. The time for a single send/recv is obtained by halving these numbers.

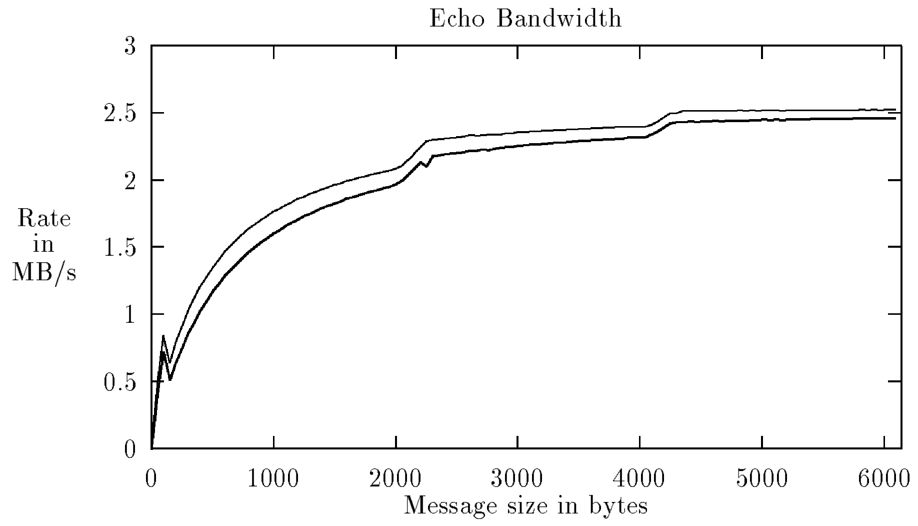

Figure 18: Echo kernel (bandwidth) on 16 nodes. The upper line represents the bandwidth achieved between adjacent nodes and the lower line represents the bandwidth achieved between nodes 4 hops apart.

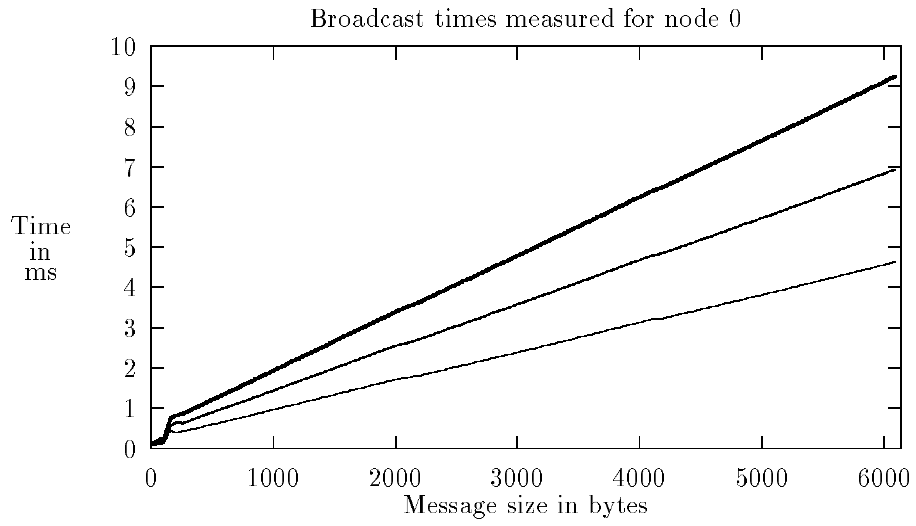

Figure 19: Broadcast kernel. The lowest line represents the broadcast times for 4 nodes, the middle line represents the times for 8 nodes and the upper line represents the times for 16 nodes.

The results obtained from the pairwise exchange kernel are interpreted by plotting the elapsed time against message size (see figure 20). Empirically, this curve can again be interpreted by the equation,

$$t(N) = t_{start-up} + Nt_{per-byte} + (n-1)h.$$

The bidirectional nature of the communication channels in the iPSC/860 is reflected in the fact that pairwise exchange is faster than the results obtained by running the **echo** kernel for the same message size. The graph also shows points at which the pairwise exchange completes considerably faster than expected. We hypothesize that this behavior occurs when the communicating nodes are synchronized such that the receive is posted before the message arrives.

All of the results discussed above reflect a contentionless environment. Figure 21 presents the elapsed times for the echo test between nodes 1 and 3 under 50% and 90% loads between node 0 and 7. Predictably, figure 21 shows that echo runs slower as the load between nodes 0 and 7 increases.

For barrier synchronization, the log of the number of nodes is plotted against the minimum time required for a barrier synchronization (see figure 22). For global reduction, the time taken for a global sum of a vector of doubles is plotted against the length of the vector on each node (see figure 23). The different curves correspond to different machine sizes.

Interestingly, the global reduction times increase almost linearly with the log of the number of processors. This illustrates that the vector, global sum primitive available follows a logarithmic, tree reduction scheme. Other algorithms are competitive with this, depending on various parameters such as the vector length. See Littlefield for a study of alternative algorithms [23].

Finally, the last results (Figures 24, 25, 26 and 27), pertain to the overlap of computation and communication. The fact that the iPSC/860 has separate communication and computation hardware means that communication and computation can be partially overlapped. The figures display results

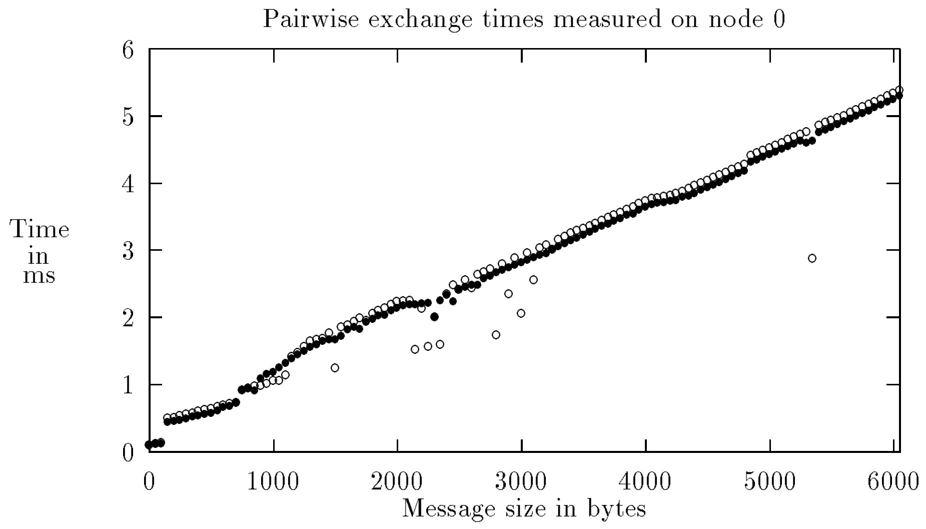

Figure 20: Pairwise exchange kernel on 16 nodes. The shaded circles represent elapsed times for a pairwise exchange between adjacent nodes and the empty circles represent elapsed times between 4-hop nodes.

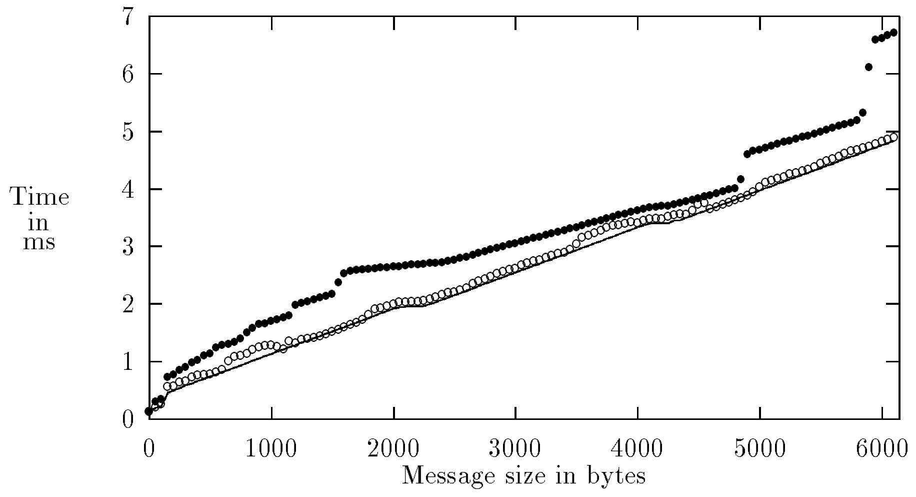

Figure 21: Contention kernel on 8 nodes. The empty circles represent echo times under contention from 50% load and the filled circles represent echo times under 90% load

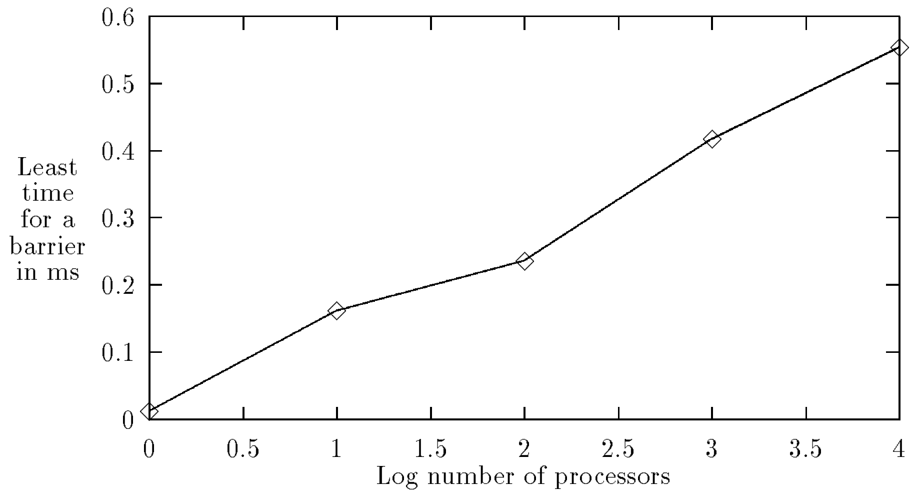

Figure 22: Barrier synchronization

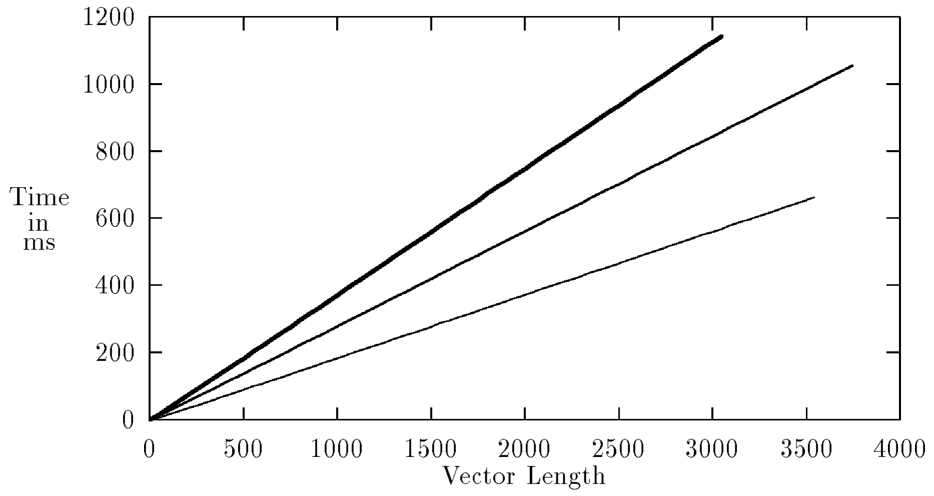

Figure 23: Vector global sum. The lowest line gives the times when the kernel was run on 4 nodes, the middle line represents the times on 8 nodes and the upper line gives the times on 16 nodes.

for message sizes varying from 0 to 85000 bytes and for vector lengths varying from 0 to 200000 double precision units. The asynchronous version of the kernel (the one that allows overlap) shows an improvement over the synchronous version in all cases except for very small message sizes where the overhead involved in setting up the asynchronous calls outweighs the saving obtained by overlapping communication. Performance improvements of over 50% are observed when overlapping a pairwise transfer of large messages with a double precision computation involving large vectors.

Interestingly, as messages get larger the asynchronous version of the kernel is faster than the synchronous version even when there is little computation. This behavior occurs because messages are copied directly into user space, avoiding a memory copy (see figure 27). The results for the overlap kernel also reflect the same unusual behavior as the pairwise exchange kernel. That is, at some points, the kernel completes much faster than expected (see figure 25). Again, we suspect that this behavior may occur when events are synchronized such that the receive is posted prior to message arrival.

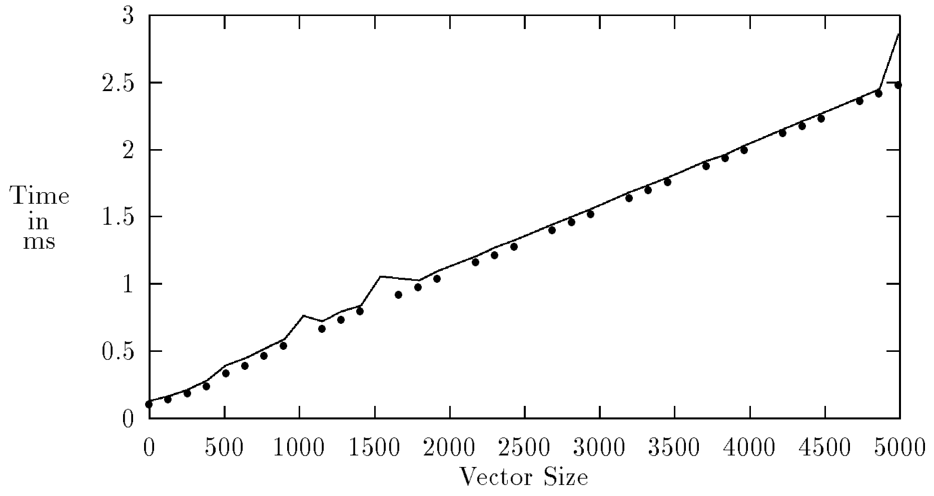

Figure 24: Overlap kernel at message size 0. The vector size refers to the size of the vector input to the DAXPY. The dots represent the times for the non-overlapped version of the kernel and the line represents the times for the overlapped version.

## 4.2 Matrix-Related Kernels

Since it was possible to preserve adjacency while decomposing the matrix onto the iPSC/860 nodes, most of the matrix operations are transfers between adjacent nodes. Therefore, it is not surprising to find a linear relationship between the times for the **update_guard** (see figure 28), **row_broadcast** and **col_broadcast** (see figure 30) kernels and matrix sizes. The relationship between the times for the **shift_matrix** (see figure 29) and **transpose** (see figure 31) kernels and the matrix size is quadratic. This behavior is also expected. Note that matrix size in the graphs refers to the matrix order (number of rows), not the number of elements. Finally, the difference between the results

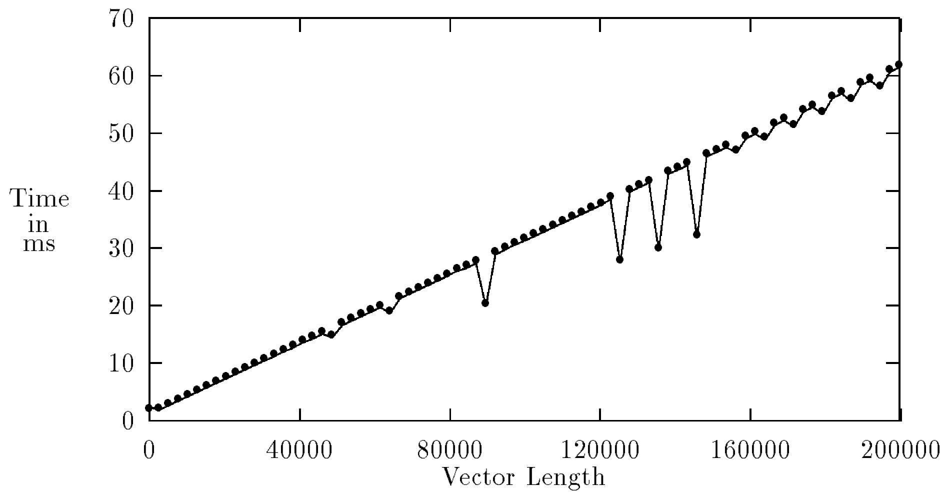

Figure 25: Overlap kernel at message size 2536. For this message size, the overlapped times begin to beat the non-overlapped times.

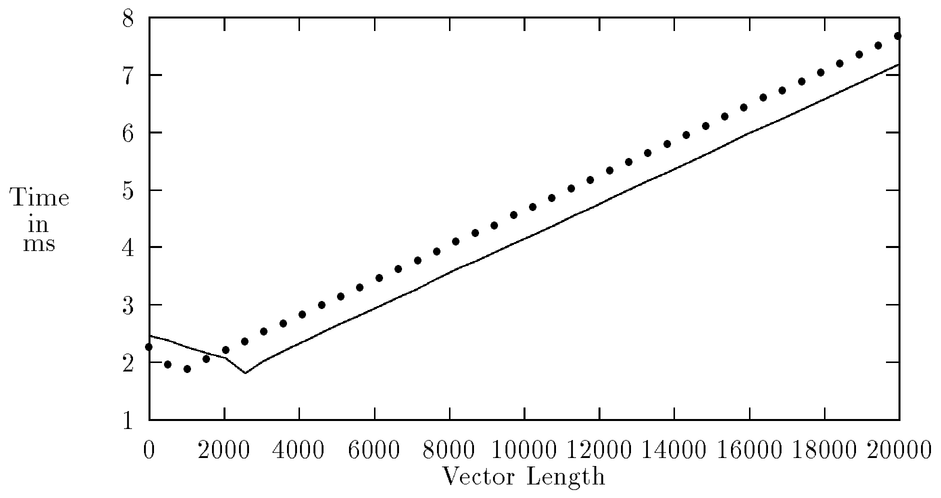

Figure 26: Overlap kernel at message size 2536. This figure is a zoomed-in view of the previous figure for low message sizes.

for the row and column broadcast in figure 30 is due to the additional time required to assemble columns into a buffer before broadcasting it (in C — had we used Fortran, the curves would have been reversed).

## 5 Related Work

To provide meaningful performance information for distributed-memory computers, benchmarks based on complete applications or application components have been developed. These benchmarks can be categorized as either synthetic, kernel, or application-based benchmarks.

Synthetic benchmarks consist of code segments that reflect frequently used program constructs and basic machine functions. Existing benchmarks in this category include Whetstones [32], Dhrystones [32, 31], and BeLinda [22]. The simplicity of benchmarks at this level usually makes them easy to port from one architecture to another. However, relating the results to performance predictions for real applications requires a detailed understanding of the application's use of the benchmark primitives.

Kernel benchmarks consist of larger code segments extracted from real applications. They represent frequently used algorithms that are thought to contribute most to application execution times. Example kernel benchmarks include Livermore Loops [15], NAS kernels [5] and LINPACK routines [9, 32]. These benchmarks are a compromise between synthetic benchmarks and full-blown application benchmarks: they are relatively easy to port and support fairly detailed information about application performance.

Application-based benchmarks are implementations of real-world applications, usually in the area of scientific computing. Well-known benchmarks in this category include the SPEC [8], Perfect [7, 5] and Euroben benchmark suites [29]. These benchmarks accurately reflect the performance characteristics of a machine with respect to specific classes of real-world applications. However, the porting process can be difficult and the amount of work put into porting the benchmarks can have a major impact on results.

In addition to these benchmarks, micro-measurements and global performance formulae have been used to estimate supercomputer performance [27]. These approaches provide a more detailed view of the underlying hardware characteristics than the higher-level benchmarks described above.

According to the classification presented above, CoMet is a synthetic benchmark. That is, its components are considerably smaller than application based benchmarks. However, the high level of specificity of CoMet's matrix related kernels to scientific applications makes it more application-specific than other synthetic benchmarks. It is this compromise between synthetic and application levels that distinguishes CoMet from other benchmarks such as Genesis. The primary motivation for this compromise is (a) distributed-memory multiprocessors are predominantly used for scientific applications that make extensive use of matrix manipulations, and (b) it is extremely difficult to port complete application-based benchmarks across the diverse range of distributed-memory multiprocessor architectures [7].

## 6 Conclusions

This paper has described a synthetic benchmark for message-passing architectures. The benchmark, called CoMet, consists of kernels to measure a machine's basic communication characteristics under light and heavy load. CoMet also contains some higher-level kernels that measure a machine's performance on common matrix manipulations.

In addition to outlining CoMet's key components in general terms, we described a specific implementation of CoMet on the Intel iPSC/860 and discussed the results obtained.

An obvious area for future work is to port CoMet to other architectures, particularly those far removed from the iPSC/860. Ports to PVM [3, 12] and MPI [16] would be interesting and useful.

The benchmark itself could also be extended to include kernels that measure characteristics of interrupt-driven communication such as the hsend/hrecv of the iPSC/860 or "active" messages such as those proposed by von Eicken, et al. [30]. At the application level, CoMet should be made more comprehensive by including more communication patterns from scientific computing. Finally, the performance of many applications also depends on the performance of secondary storage accesses. CoMet could be extended to characterize the interaction of a machine's distributed-memory system with its Input/Output subsystem.

The CoMet benchmark program is publicly available. The C source, together with a makefile, user's guide and a shell script to execute the benchmark and organize the results are available by anonymous ftp from `cse.ogi.edu` in `pub/dsrg/CoMet`.

## 7 Acknowledgements

We thank Joe Brandenburg of Intel SSD, Jon Inouye, and Ravi Konuru for useful discussions. This work was partially supported by grants from Intel Supercomputing Systems Division and the Oregon Advanced Computing Institute.

## 8 Bibliography

## References

[1] Cliff Addison, Vladimir Getov, Anthony Hey, Roger Hockney, and Ivan Wolton. The Genesis distributed-memory benchmarks. *Proc. of Parallel Processors - Benchmarking and Assessment*, March 1992.

[2] D. H. Bailey, E. Barszcz, L. Dagum, and H. D. Simon. NAS parallel benchmark results. In *Proceedings - Supercomputing '92*, 1992.

[3] A. L. Beguelin, J. J. Dongarra, A. Geist, R. J. Manchek, and V. S. Sunderam. Heterogeneous network computing. In *Sixth SIAM Conference on Parallel Processing*, 1993.

[4] Rudolf Berrendorf and Jukka Helin. Evaluating the basic performance of the Intel iPSC/860 parallel computer. *Concurrency: Practice and Experience*, 4(3):223–40, May 1992.

[5] M. Berry, G.Cybenko, and J. Larson. Scientific benchmark characterisations. *Parallel Computing*, 17, 1991.

[6] L. Bomans, D.Roose, and R. Hempel. The Argonne/GMD macros in FORTRAN for portable parallel programmning and their implementation on the Intel iPSC/2. *Parallel Computing*, 15:119–32, 1990.

[7] George Cybenko. Supercomputer performance trends and the perfect benchmarks. Technical Report 1093, CSRD, University of Illinois at Urbana-Champaign, April 1991.

[8] Kaivalya M. Dixit. The SPEC benchmarks. *Parallel Computing*, 17, 1991.

[9] J. Dongarra. Performance of various computers using standard linear equations software in a fortran environment. Technical Report CS-89-85, Computer Science Department, University of Tennessee, Knoxville, TN, 1989.

[10] J. Dongarra, J. Bunch, C. Moler, and G. W. Stewart. *LINPACK User's Guide*. SIAM, Philadelphia, PA, 1979.

[11] J. Dongarra and D. Sorensen. Linear algebra on high performance computers. In *Proceedings of Parallel Computing*, 1986.

[12] Jack Dongarra, Al Geist, Robert Manchek, and Vaidy Sunderam. Integrated PVM framework supports heterogeneous network computing. *Computers in Physics*, April 1993.

[13] T. H. Dunigan. Performance of the Intel iPSC/860 and Ncube 6400 hypercubes. *Parallel Computing*, 17:1285–1302, 1991.

[14] Thomas Dunigan. Communication performance of the intel touchstone delta mesh. Technical Report ORNL/TM-11983, Oak Ridge National Laboratory, January 1992.

[15] J. T. Feo. An analysis of the computational and parallel complexity of the Livermore Loops. *Parallel Computing 7*, 1988.

[16] The Message Passing Interface Forum. Document for a standard message-passing interface. Technical Report CS-93-214, University of Tennessee, November 1993.

[17] G. Fox, M. Johnson, G.Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving problems on Concurrent Processors*, volume 1. Prentice Hall, Inc., 1988.

[18] Nalini Ganapati. CoMet: A synthetic benchmark for message-passing architectures. Master's thesis, Oregon Graduate Institute of Science & Technology, July 1993.

[19] G. A. Geist. A user's guide to PICL: a portable instrumented communication library. Technical Report ORNL/TM-11616, Oak Ridge National Laboratory, September 1990.

[20] Anthony J. G. Hey. The Genesis distributed memory benchmarks. *Parallel Computing*, 17, December 1991.

[21] Intel Corporation. *iPSC/860 Programmer's Manual*, 1991.

[22] S. Kambhatla, J. Inouye, and J.Walpole. Experiences with BeLinda: A Synthetic Linda Benchmark for Parallel Computing Platforms. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume II, pages 155–162, St. Charles, Illinois, August 1990.

[23] R. Littlefield. Modeling node bandwidth limits and their effect on vector combining algorithms. Technical Report PNL-SA-20425, Batelle Pacific Northwest Laboratory, 1992.

[24] P. Messina, C. Baillie, E. Felten, P. Hipes, R. Williams, A. Alagar, A. Kamrath, R. Leary, W. Pfeiffer, J. Rogers, and D. Walker. Benchmarking advanced architecture computers. *Concurrency: Practice and Experience*, 2(3):195–255, September 1990.

[25] Steve Nugent. The iPSC/2 direct-connect technology. In *3rd conference on Hypercube Concurrent Computers and Applications*, New York, NY, 1988. ACM Press.

[26] Paul Pierce. The NX/2 operating system. In *Proceedings of the 3rd conference on Hypercube Concurrent Computers and Applications*, 1988.

[27] Willi Schonauer and Harmut Hafner. Performance estimates of supercomputers. *Parallel Computing*, 17, 1991.

[28] Thinking Machines Corporation. *Connection Machine CM-5 Technical Summary*, October 1991.

[29] Aad J. van der Steen. The benchmark of the Euroben group. *Parallel Computing 17*, December 1991.

[30] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. *Communications of ACM*, pages 256–66, July 1992.

[31] R. Weicker. Dhrystone: A synthetic systems programming benchmark. *Communications of ACM*, 1984.

[32] R. P. Weicker. A detailed look at some popular benchmarks. *Parallel Computing*, 17, 1991.

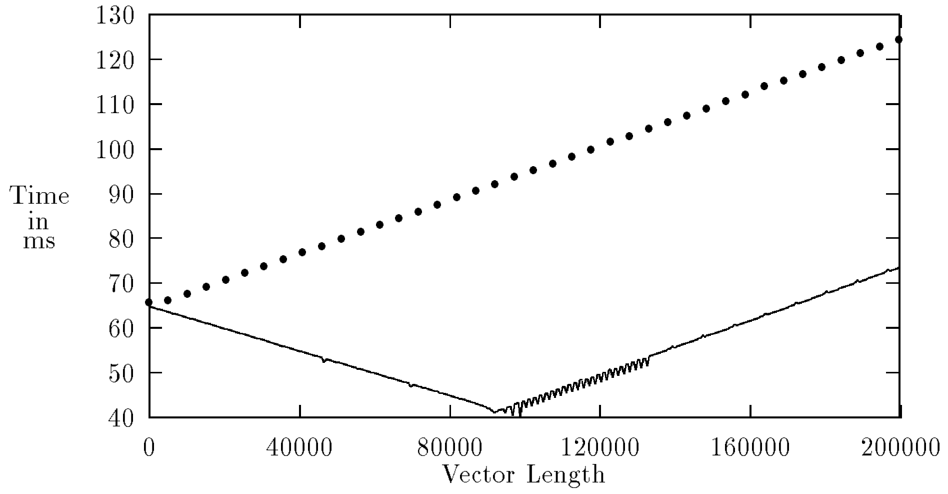

Figure 27: Overlap kernel at message size 85000.

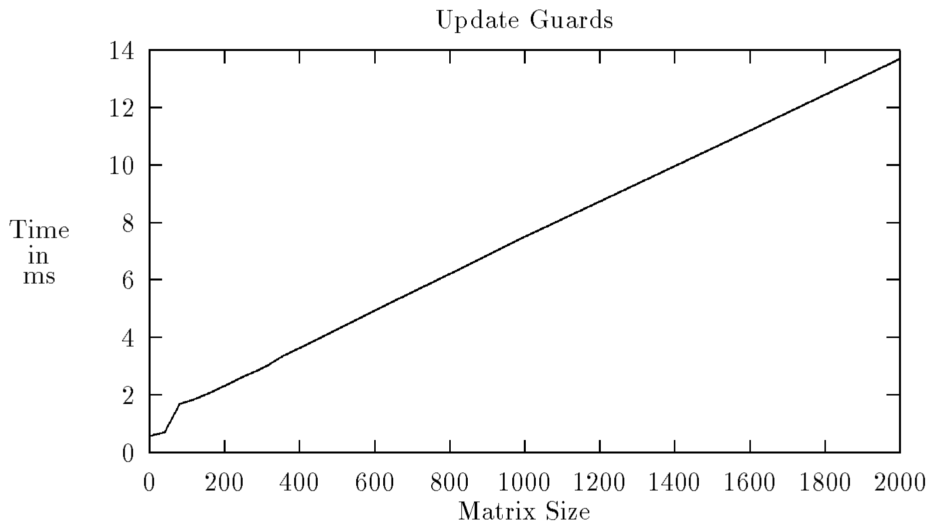

Figure 28: Update guard wrapper on 16 nodes.

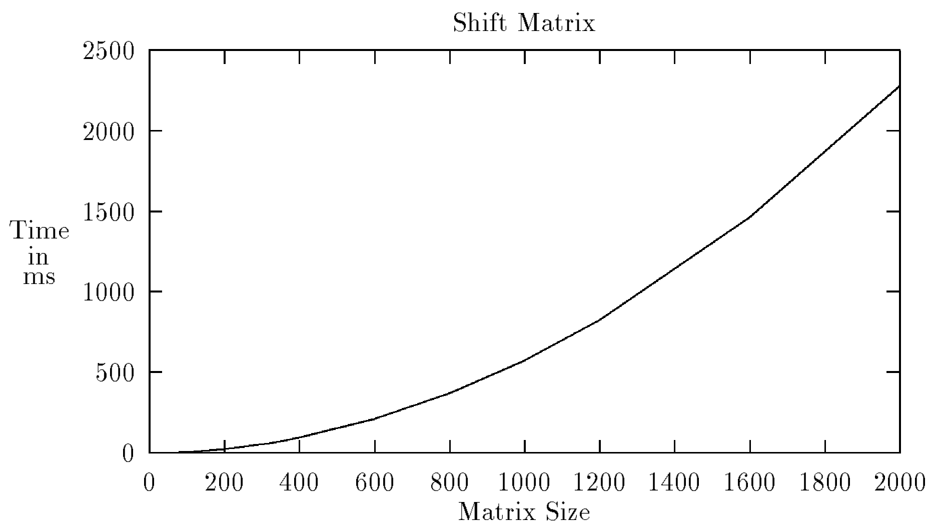

Figure 29: Shift matrix on 16 nodes. The quadratic growth is due to the fact that the amount of data shifted grows as the square of the matrix size.

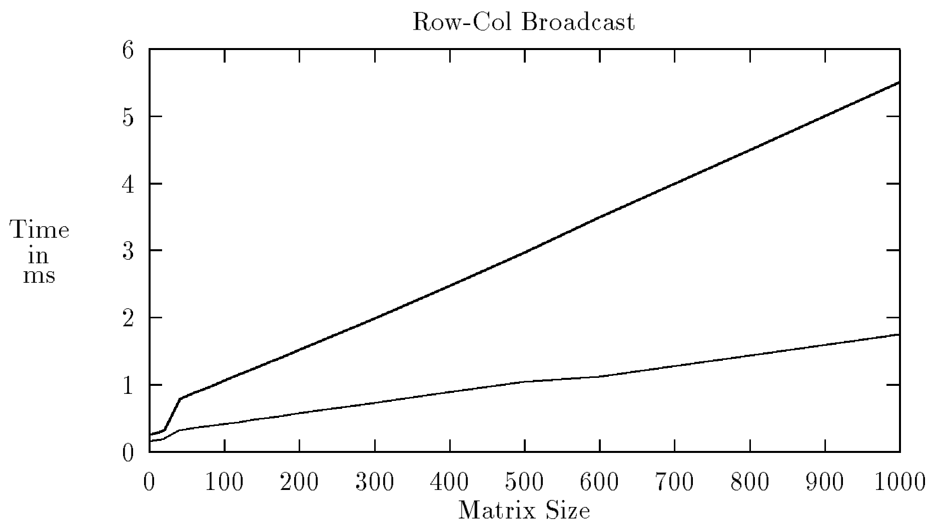

Figure 30: Row and Column Broadcast on 16 nodes. The lower curve is the row multi-cast time, the upper curve is the column multi-cast time. The difference is due to the necessity of marshalling the data in the column case.

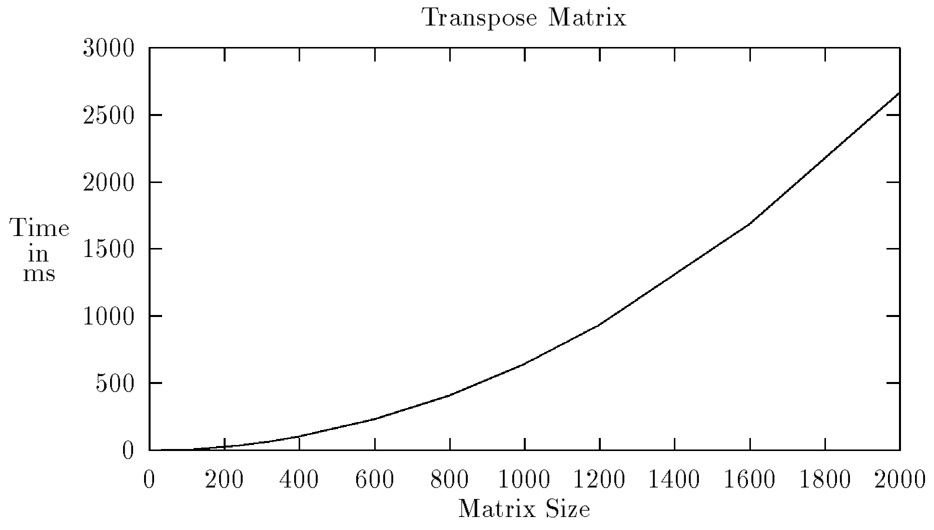

Figure 31: Transpose matrix on 16 nodes.