Low-Latency Message Communication Support for the AP1000

Toshiyuki Shimizu, Takeshi Horie, and Hiroaki Ishihata
Fujitsu Laboratories Limited
1015 Kamikodanaka, Nakahara-ku,
Kawasaki 211, Japan
toshi@flab.fujitsu.co.jp

ABSTRACT

Low-latency communication is the key to achieving a high-performance parallel computer. In using state-of-the-art processors, we must take cache memory into account. This paper presents an architecture for low-latency message communication and implementation, and performance evaluation.

We developed a message controller (MSC) to support low-latency message passing communication for the AP1000, to minimize message handling overhead. MSC sends messages directly from cache memory and automatically receives messages in the circular buffer. We designed communication functions between cells and evaluated communication performance by running benchmark programs such as the Pingpong benchmark, the LINPACK benchmark, the SLALOM benchmark, and a solver using the scaled conjugate gradient method.

Keywords: message handling, message-passing, cache memory, low-latency, parallel computer.

1 Introduction

Key issues in implementing a high-performance parallel computer include the utilization of high-speed processing elements, incorporation of a low-latency, high-throughput interprocessor communication facility, and minimization of message handling time.

In this paper, we present the communications scheme designed for the AP1000, a 2-D toroidal mesh MIMD computer. Our technique involves sending messages directly from the sender's cache memory to the receiver's buffer. This communication method can easily be incorporated into ordinary message passing models, and minimizes message handling overhead.

After reviewing previous work and problems encountered, we introduce our scheme in detail, describe our hardware and software implementations, and conclude by presenting performance results.

Review of present communications techniques

Many distributed memory parallel processors (DMPP) have been proposed. The Cosmic Cube[1], MarkIII[2], and J-Machine[3] have been developed for research purposes, and the iPSC[4], uCUBE, and Transputer have been developed as commercial products. In a larger parallel computer, the DMPP is easily expandable and is cost-effective. However, it is not easy to obtain good results for many applications. Some algorithms require much communication to execute in parallel when there are many processing elements.

To run a program efficiently, communication time should be minimized. Minimizing communication latency and widening communication throughput are critical issues in developing a DMPP. A large system that has hundreds or thousands of processor elements attempting to communicate with each other has a large message handling overhead for small messages. Much research has been done to widen communication throughput[5], and to minimize communication latency[6, 7] and message handling overhead[3, 8, 9, 10, 11].

Kernhani and Kleinrock avoided the interaction of intermediate processors between sender and receiver by incorporating virtual cut-through[6]. Dally proposed wormhole routing to avoid buffering messages at processors between the sender and receiver[7]. Dally also proposed virtual channels to increase communication throughput and prevent deadlock[5].

As network latency decreases and throughput increases, the time consumed by the sending and receiving processors becomes dominant. The iWarp supports systolic communication, which is very efficient at sending and receiving when messages are well scheduled[11].
However, it is usually difficult to write tightly scheduled programs because predicting and scheduling messages is difficult.

Nikhil and Papadopoulos introduced the *T processor for fast execution of data flow computation[9]. A synchronization processor is attached to support remote data (token) access and process scheduling. The functions for handling tokens can apply to efficient message passing functions. Since the token length is limited, messages would be partitioned in many tokens and processed. The amount of the data actually transferred would be large, which may cause network congestion. The total throughput of data movement is limited.

Hsu and Banerjee developed a message passing co-processor (MPC) to accelerate message communication by supporting process scheduling, message buffer management, and buffer copying[10]. To use the MPC, an application program has to request buffers for messages to be sent and to release buffers that hold unneeded received messages. These restrictions can eliminate copying.

A message-driven processor has well-designed mechanisms for receiving messages[3, 8]. Although these mechanisms were designed to handle messages produced by object-oriented programs, such techniques may also be applied to achieve efficient message passing.

However, all the above designs fail to take advantage of possible speedups offered by communicating directly to-and-from cache memory. All state-of-the-art RISC processors include cache memory, and we must realize that data which is to be sent from a processor is often already cached.

Communications on the AP1000

We introduce a new technique, which we call "line-sending," by which messages may be sent from a processor directly from its cache memory, bypassing DMA and main memory access. This technique eliminates time needed to copy data from user memory to system memory, flush the cache, process interrupts, and initialize the DMA controller. In the AP1000, the line-sending function is implemented in a single VLSI chip, the message controller (MSC).

Our message controller also manages the reception of messages from the network directly into a circular buffer, which we call "buffer-receiving." This is performed as soon as the data arrives, and does not require a processor interruption. Also, because the MSC has direct access to main memory, the usual DMA controller is not used in receiving messages. Incoming messages are immediately stored by the MSC in a circular buffer allocated in the user space. This buffer may be accessed by the user directly.

In the next section, we present an analysis of the performance of our communications techniques.

2 Analysis of message communication

This section summarizes the message passing mechanism, and discusses the overhead for sending and receiving messages.

2.1 Conventional message passing mechanism

Message communication in conventional machines is shown in Figure 1.

The steps indicated in the figure are as follows:

A: Allocates the buffer for saving messages
B: Blocked by the network congestion
C: Copies a message to or from the system message buffer
D: Transfers data to or from devices by DMA
E: Enqueues the messages in a trap routine that will invoke the DMA transfer
F: Finds the origin of the interrupt
I: Initiates DMA
P: Checks parameters and calls system service routines
S: Checks the status and reports it
T: Checks the transfer status
X: Extracts a specified message from the receive buffer

We will describe the timing diagrams. A user program on a sender calls the library to send a message. Parameter checking is done by the library routine (P), buffer\(^1\) for the sent message to be stored in is allocated (A), the message is copied to the allocated buffer (C), message to be transferred is enqueued (E), the DMA controller is initialized and is started (I), and control is returned to the user program (S).

---

\(^1\)Send buffer is usually an uncached area to eliminate flushing after copy.
After DMA is started, the message arrives at the receiver and a trap occurs (F), the receiver allocates buffer space\(^2\) for the message (A) and starts the DMA controller (I). While (F), (A), and (I) are processed by the receiver, the network is stopped, then sending DMA is blocked (B). When DMA transfer is done, status checking and initialization for another transfer is processed (T). A user program on a receiver calls the library to receive a message. After parameter checking is done by the library routine (P) and DMA transfer is finished (T), the CPU finds the message in the buffer (X), and copies it to the user buffer (C), and checks status and returns control to the user program (S).

Even if we use machines with a high-performance network, messages are blocked by F, A, and I. *Network latency*, the time from the sender starts transferring to the receiver receives all the data, is \(D + F + A + I + \text{network delay}\).

If the receiving DMA is started before message arrival, F, A, and I can be preprocessed. However, without examining the message size, a large message buffer should be used for DMA.

**Latency in the sender**, in which the user program on the sender is blocked, is \(P + A + C + E + I + S\). **Latency in the receiver**, in which the user program on the receiver is blocked, is \(P + T + X + C + S\). These are the best cases. If a message has not arrived, the user program is suspended.

**Overall latency** is the most important, it is the time from when the sender calls to the time the receiver can access the message. Latency in the sender and receiver is important because it shows how long the user program is blocked by message passing.

All symbols in Figure 1 show the message handling overhead, except D. Should be reduced or eliminated for faster execution of parallel programs. Data copying, C, can be eliminated if a pointer to the message buffer is returned and the user can access the buffer area. The iPSC[4] provides isend() and irecv() to avoid copying.

The MPC[10] processes the message buffer management, interrupt handling, and buffer copying to reduce the overall latency. The MPC succeeds in releasing the CPU from processing message handling and accelerates copying and interrupt handling. Stage F is eliminated, and stage A and I are preprocessed. Latency in the sender and receiver is thus reduced. However, although a similar sequence of message copying and interrupt handling is executed by the MPC, the overall latency is still long.

\(^2\)This buffer is usually an uncached area to eliminate invalidation before use.

The MDP[8] allows access to the incoming message. This mechanism reduces overall latency and allows pipelining execution. Since the MDP uses a send instruction to send messages, the timing of sending is different from Figure 1. Data can be sent immediately. Message handling routines should be carefully designed since interruption is disabled while messages are sent. The MDP does not apparently to utilize cache memory.

### 2.2 Line sending and buffer receiving

To perform message passing on a processor with cache memory, some problems should be considered.

- Message buffer for receiving should not be cached or should be invalidated before receiving. That is because the received message is not accessed if the data had been in cache memory. Old data might be read from cache memory.

- To use a copyback cache, flushing or copying the data to uncached area should be done before invoking DMA for sending. This is because the newest data is not always in main memory where DMA access can take place.

Accessing uncached data is very slow, and invalidating cache memory is expensive. Flushing the cache or copying to an uncachable area causes a problem with overhead.

![Figure 2: Message passing performed by the MSC](image_url)

We propose a message transfer scheme using line sending and buffer receiving. This method utilizes cache memory and eliminates unneeded operations. Figure 2 illustrates message passing. LS is the line sending and the other symbols are the same as in Figure 1. LS is faster than memory copying because it can move data in a block with one operation, but the copy function needs a load and store operation using a register.

Line sending and buffer receiving have the following advantages:
• Line sending sends data directly from cache memory in a similar manner to cache flushing. This scheme eliminates copying, DMA initialization, and interrupt processing. A, E, and I for sending are eliminated, and C is replaced by LS.

• Line sending is not buffered, and data is transferred immediately.

• Receiving DMA is always enabled, so a message is transferred on arrival. Received messages are stored in the large circular buffer. An interrupt is not needed to control DMA. The sender is not blocked because newly arrived data is immediately processed. F, A, and I for receiving are eliminated.

• Since all operations are done in user mode, no operating system service calls are needed. Traps only occur when the network is busy.

Line sending and buffer receiving promise good performance which can utilize cache memory and eliminate overhead, A, C, E, I, F, X, and T, completely. However, there are the following problems:

• Line sending is controlled by the CPU, and other calculation cannot be done while messages are being sent.

• Line sending may produce a trap when the network is busy, this depends on network utilization of the application program.

• Message passing is done by cache line size and is aligned. Handling a misaligned message is difficult.

• Receiving by the ring buffer is complicated compared with ordinary approaches. To look for the desired message and treat wraparound message seems to be difficult and expensive.

We will show that the above problems can be made negligible in Section 6.

3 MSC
This section discusses the hardware design which implements line sending and buffer receiving. To implement line sending and buffer receiving, the message handling hardware (MSC) incorporates a cache controller and a DMA controller. Figure 3 shows the MSC configuration.

Memory, other devices, and the MSC are connected by a local bus called LBUS. The CPU and cache memory are connected to the MSC. During normal operation, the MSC works as a direct-mapped cache memory controller with a copyback policy. The line size is four words. Wbuf is a write buffer that is used to write data to memory or devices.

3.1 Line sending
A line sending function sends a cache line message like that of a cache flush. DMA is activated automatically even when there is no specified data in the cache. The arrows in Figure 4 indicate when there is data movement: HIT when there is data in the cache, and MISS when the cache is invalid. The figure also shows a normal cache flush (FLUSH) for comparison.

Initiating line sending similar to cache memory control commands is implemented by a simple store instruction. The 6 MSBs of the address are decoded as commands. The function for line send and line send with end bit are assigned to 0x3b and 0x3c. This operation is different from the sending by program mode, which reads data from memory to the register and writes data to the device. Line sending doesn’t load and store data but specifies the address of data. Cache line data is sent by a store instruction.

Figure 3: MSC configuration

Figure 4: Data movement during line sending

When an CPU initiates a line sending command, the MSC checks the tag memory to see if the data is in the
cache or in main memory. It simultaneously checks the output network device status to see if the device can accept the data, and checks that the local bus is ready, and that no other DMA is activated to write the same network device. If all devices are ready and data is in the cache, data is read from the cache and written to the Wbuf (HIT). If data is not in the cache, DMA to one cache line is invoked, and the data is read from main memory and written to the device (MISS). When line send with end bit is specified, the data ends the message.

The CPU is blocked until the status has been checked and data has been moved from the cache to Wbuf or DMA is initiated. The CPU does not wait for all the data to be written to the device. To implement this mechanism, the devices must have a FIFO deeper than the cache line size. Because it is not known when a whole line is sent to the network, MSC checks the FIFO status to see whether there is room for the message.

If devices are busy, a data access exception is reported to the CPU, and a trap occurs. This trap enables the CPU to retry line sending and avoid deadlock.

Line sending will not invalidate the cache entry or write back to main memory. Because this data may be accessed again, invalidation might lose the newest data, and flushing is expensive. Programs often modify a part of the messages and send again.

3.2 Buffer receiving

Buffer receiving (Figure 5) corresponds to any asynchronous data transfer request. Hardware monitors the ring buffer for overflow, notifies the CPU of such an error by an interrupt, or stops transferring until data reading produces a receive area in the buffer. The accessed area is released and returned to the MSC as a receive buffer by modifying the register.

Data movement is done in 4-word aligned blocks. This is the same as the cache line size so that the LBUS can move quickly.

The BASE, write pointer (WTP), and read pointer (RDP) are registers (Figure 3). The BASE holds the buffer address. A pair of pointers to the circular buffer, WTP and RDP, points to the data that has been received from other processors but which has not yet been read by the CPU. The WTP points to the address where the next received data is to be stored. It is updated by the hardware after each data is stored. The RDP points to the address that the CPU is currently accessing. The CPU writes any value to the RDP to dispose of data which is no longer needed.

When data arrives from a network device, the MSC checks that the WTP value does not exceed the RDP. If the WTP and RDP hold the same value, no more data can be received, and the MSC suspends transfer or notifies the CPU by generating an interrupt signal.

The WTP is initialized to the top of the buffer address (base address), and RDP is initialized to the bottom address (Limit: base address + buffer size - 1). The buffer size can be between 64 to 512 kilobytes.

A received message is not written to the cache memory directly. That is because it may cause the replacement of the cached data which is used before the message.

4 Software implementation

As described in Section 3, the MSC only supports simple mechanisms. A software library provides a more general interface for message passing. We developed basic send and receive functions, `send()` and `recv()`. We show how alignment problems are handled.

We also describe global operation functions such as global sum, minimum, and maximum. Global operations require a lot of small message communication (4 bytes or 8 bytes). To get good performance, specially formatted messages are used.

It is important that the operating system we use schedules processes only when the process calls the receive message function in order to reduce process switches. Since any other process will run concurrently in send and receive functions, we don’t have to disable interrupts.

This section explains how these routines are implemented.

4.1 Message format for send and receive

Table 1 lists the message format for ordinary message transfer.

---

The address is always aligned on a cache line, and the base address must be aligned according to the buffer size.
Table 1: Message format

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Routing header</td>
<td>4 Bytes</td>
</tr>
<tr>
<td>4</td>
<td>Message length</td>
<td>4 Bytes</td>
</tr>
<tr>
<td>8</td>
<td>Alignment of message start</td>
<td>4 Bytes</td>
</tr>
<tr>
<td>12</td>
<td>Cell IDx of sender</td>
<td>4 Bytes</td>
</tr>
<tr>
<td>16</td>
<td>Cell IDy of sender</td>
<td>4 Bytes</td>
</tr>
<tr>
<td>24</td>
<td>Message ID for trace</td>
<td>4 Bytes</td>
</tr>
<tr>
<td>28</td>
<td>Message type</td>
<td>4 Bytes</td>
</tr>
<tr>
<td>32</td>
<td>Message body</td>
<td>Any</td>
</tr>
</tbody>
</table>

The routing header contains a hardware header to specify the destination cell. We assume that only the header word is used by the hardware. The other words are used by software.

Cell IDx and Cell IDy identify the source processor with the relative distance. Alignment of message specifies the message start alignment from offset 32. Message length, which holds the message size, and alignment of message, enable the handling of byte alignment messages. Since cache line size is 16 bytes, alignment varies from 0 to 15. There is no restriction on the message length.

4.2 Send message function

send() sends a message specified by the address (p) and size to a cell specified by cid. The message type can be specified. The code is as follows:

```c
send(cid, type, p, size)
{
    Set message info to the header in buffer q:
    LSEND(q);
    LSEND(q+16);
    do size/cache_line_size-1 times {
        LSEND(p);
        Increment p;
    }
    LSENDE(p);
}
```

LSEND(p) and LSENDE(p) invoke line send functions with and without an end bit. These are written as follows:

```c
st %r0,[%r1+%r2] ;LSEND(p), %r1=p,%r2=0xec000000
st %r0,[%r1+%r2] ;LSENDE(p),%r1=p,%r2=0xf0000000
```

4.3 Receive message function

recv() waits until a message is received from a specified cell. It avoids unneeded copies and can return control if there is no appropriate message. The code is as follows:

```c
recv(cid, type)
{
    Return NULL if no message has been received;
    Pick up a header from top of the ring buf.(RDP);
    forever {
        if header is marked 'USED'{
            Invalidate this message area;
            Increment RDP to skip this message;
        } else if message is found {
            Mark a header 'USED';
            Wait until the whole message is received;
            Copy data only if wraparound occurred;
            Return the message address;
        } else if no more messages{
            Return NULL;
        } else{
            Pick up the next message header;
        }
    }
```

Figure 6: Ring buffer operation

Figure 6 shows how a ring buffer is used and wraparound data is managed. When recv() is called to find message C, the buffer is in the initial state (a). Message A is accessed first, but, since message A is marked 'USED', message area A is invalidated and the RDP is incremented to the beginning of message B. Message B is not 'USED' but does not match, so message C is picked up. This is a match, but has wraparound, so the remainder of the message is appended to the end of the buffer (limit). Message C is marked 'USED' and is returned with the message address (msg. ptr.). When

...
function recvs() exits. The buffer status is as shown in (b).

### 4.4 Global operations

A parallel computer often uses global operations such as sum, maximum, and minimum. The functions are briefly explained below:

- **dmax()** gets the cell ID and maximum value in double-precision data for all the cells.
- **dsum()** sums the double-precision data for all cells.

A call to global function `func()`, such as `dmax()` and `dsum()`, in a cell must be complemented by a call to `func()` in all other cells. All functions perform binary tree communication to get information from all cells.

Line sending and buffer receiving are used to implement the above, using the special formatted messages listed in Table 2.

Table 2: Message format for global operations

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Routing header (with func.)</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>Index of DATA (cid)</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>DATA (float, int, double)</td>
<td>8</td>
</tr>
</tbody>
</table>

By using this format, messages can be packed within one cache line to get good performance. The lower nine bits of the routing header indicate which global operation should be done.

### 5 AP1000 architecture

This section overviews the hardware architecture of the AP1000[12]. Figure 7 shows the AP1000 system configuration. Each processing element, or cell, is connected by three independent networks, the T-net, B-net, and S-net. The total number of cells varies from 16 to 1024.

To obtain high-throughput and low-latency communication, we choose a two-dimensional torus topology for the interprocessor communication, T-net. We developed a routing controller (RTC), for the T-net which is designed to run a routing scheme that incorporates the structured buffer pool algorithm into wormhole routing[13]. This scheme provides low-latency, high-throughput communication, and avoids deadlock. The RTC handles the flow control and routing in communication between cells. The RTC connects four neighbors with a 16-bit data bus and uses a transfer rate of 25 megabytes per second per channel. The RTC also provides broadcast facilities on the T-net.

Figure 7: AP1000 system configuration

In the cell processor, network devices and the MSC are connected by a local bus, the LBUS. The network devices have a FIFO 8 words deep. The device decodes the FIFO status for the MSC.

We use a SPARC IU and FPU for the cell processor, which operates at 25 MHz. The IU and FPU are connected to a simple direct-mapped cache. The cache has a capacity of 128 kilobytes with a line size of 16 bytes. The cache controller in the MSC controls general-purpose SRAM by copyback.

The local memory for the processing elements is managed by the DRAM controller called DRAMC. The DRAMC controls 16 megabytes of memory. It takes 160 ns for word write access and 400 ns for block write access. Because wait cycles are inserted for read accesses, it takes 400 ns for word read access and 640 ns for block read access.

### 6 Performance results

To evaluate the line sending and buffer receiving, we ran several benchmark programs. The Pingpong benchmark[14] evaluates the overall latency of message communication. From the results of global functions we can evaluate the effect of the special format messages. We can see how much effect communication speedup has on larger applications.

#### 6.1 Pingpong benchmark

To determine the effects of message handling, the time taken to send various lengths of messages from a master cell to a slave cell and back was measured. In the experiment, cell (0,0) sends a message to cell (4, 1), then cell(4, 1) returns the message to cell (0, 0). Since wormhole routing is used, distance does not affect the result. Half of this time is used for sending the message.
It is the overall latency discussed in Section 2. The other cells are idle and have no effect on the communication cells.

Figure 8 shows the time required to handle a message versus the message size in bytes. Normal refers to the timing of the ordinary DMA transfer functions. All functions used here return the address of the received message. There is no copy overhead on receiving. BR refers to timing with buffer receiving, recv(). LS refers to line sending, send(), while LS+BR refers to both BR and LS.

Table 3 summarizes the overhead of transferring modes. The symbols have the same meaning in Section 2, and $S_a = A + E + C + I, N_d = \text{Network delay}$, and $S_r = F + A + I + T$. Table 3 also shows the Pingpong time when data size is small (10 bytes) and large (10 kilobytes).

For small data sizes, copying time is negligible. The difference between BR and LS+BR is the setup time for send, and is about 28 $\mu$s (61.3 - 32.8). The result of Normal and LS for sending of small data sizes is limited by the setup time for receiving. The setup time for receiving is about 78 $\mu$s (111 - 32.8).

![Figure 8: Pingpong performance](image)

In LS mode, receiving interrupts degrade the performance for smaller-sized messages. BR avoids interrupt overhead when receiving messages, and performance is good for smaller-sized messages. When sending is performed by DMA, setup time degrades the performance for smaller-sized messages.

For large data transfer, copying time for sending is the dominating factor. In the LS+BR mode, memory copying is eliminated, so performance is good even when messages are large.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Over all latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>10 B</td>
</tr>
<tr>
<td></td>
<td>10 KB</td>
</tr>
<tr>
<td>LS+BR</td>
<td>32.8</td>
</tr>
<tr>
<td></td>
<td>491</td>
</tr>
</tbody>
</table>

### 6.2 Global operation performance

Table 4 lists the timing of global operations. Cache refers to using line sending and buffer receiving with the format described in Table 2.

<table>
<thead>
<tr>
<th>Func.</th>
<th>Cells</th>
<th>Normal</th>
<th>LS</th>
<th>BR</th>
<th>LS+BR</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>dmax</td>
<td>64</td>
<td>1,970</td>
<td>1,840</td>
<td>975</td>
<td>597</td>
<td>136</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>2,630</td>
<td>2,480</td>
<td>1,320</td>
<td>809</td>
<td>198</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>2,970</td>
<td>2,770</td>
<td>1,480</td>
<td>913</td>
<td>224</td>
</tr>
<tr>
<td>dsum</td>
<td>64</td>
<td>1,990</td>
<td>1,850</td>
<td>983</td>
<td>586</td>
<td>119</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>2,650</td>
<td>2,470</td>
<td>1,320</td>
<td>792</td>
<td>163</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>3,000</td>
<td>2,800</td>
<td>1,500</td>
<td>911</td>
<td>190</td>
</tr>
</tbody>
</table>

The results of Normal, LS, BR, and LS+BR is similar to the results for the Pingpong benchmark. The best results were obtained by using a cache format message. Because the message size is one third that of ordinary messages, data transfer timing is reduced. The result of dsum() is faster than dmax() because dmax() needs more calculation, such as setting the index.

### 6.3 Results of larger applications

Table 5 lists the results of benchmarks with and without message handling functions. Figure 9 shows performance results normalized by normal mode. The benchmarks are briefly explained as follows:

<table>
<thead>
<tr>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
</tr>
<tr>
<td>256</td>
</tr>
<tr>
<td>512</td>
</tr>
</tbody>
</table>

Better performance may be achieved by using the native library calls, which utilize the RTC broadcast functions, as given here:

<table>
<thead>
<tr>
<th>Func.</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>dmax</td>
<td>92.1</td>
</tr>
<tr>
<td>dsum</td>
<td>74.8</td>
</tr>
</tbody>
</table>
Table 5: Performance results

<table>
<thead>
<tr>
<th>Application name</th>
<th>Cells (conf)</th>
<th>Problem size</th>
<th>Normal Timing (seconds)</th>
<th>LS Timing (seconds)</th>
<th>BR Timing (seconds)</th>
<th>LS+BR Timing (seconds)</th>
<th>Speedup ratio (Normal/LS+BR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINPACK</td>
<td>64 (8x8)</td>
<td>1000</td>
<td>8.73</td>
<td>7.95</td>
<td>6.65</td>
<td>4.83</td>
<td>1.81</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2000</td>
<td>35.8</td>
<td>33.0</td>
<td>31.4</td>
<td>26.2</td>
<td>1.37</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4000</td>
<td>197</td>
<td>187</td>
<td>186</td>
<td>170</td>
<td>1.16</td>
</tr>
<tr>
<td>SLALOM</td>
<td>256 (16x16)</td>
<td>1000</td>
<td>7.44</td>
<td>6.31</td>
<td>5.12</td>
<td>3.05</td>
<td>2.44</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2000</td>
<td>21.0</td>
<td>18.5</td>
<td>15.8</td>
<td>10.8</td>
<td>1.94</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4000</td>
<td>80.4</td>
<td>71.7</td>
<td>69.2</td>
<td>54.9</td>
<td>1.46</td>
</tr>
<tr>
<td>SCG</td>
<td>256 (16x16)</td>
<td>100</td>
<td>1.65</td>
<td>1.70</td>
<td>0.93</td>
<td>0.61</td>
<td>2.70</td>
</tr>
<tr>
<td></td>
<td></td>
<td>200</td>
<td>4.22</td>
<td>4.21</td>
<td>2.69</td>
<td>1.98</td>
<td>2.13</td>
</tr>
<tr>
<td></td>
<td></td>
<td>400</td>
<td>14.3</td>
<td>13.7</td>
<td>11.58</td>
<td>9.97</td>
<td>1.43</td>
</tr>
<tr>
<td></td>
<td>512 (16x32)</td>
<td>100</td>
<td>1.71</td>
<td>1.76</td>
<td>0.93</td>
<td>0.58</td>
<td>2.95</td>
</tr>
<tr>
<td></td>
<td></td>
<td>200</td>
<td>3.89</td>
<td>3.98</td>
<td>2.29</td>
<td>1.63</td>
<td>2.39</td>
</tr>
<tr>
<td></td>
<td></td>
<td>400</td>
<td>12.2</td>
<td>12.4</td>
<td>9.34</td>
<td>7.51</td>
<td>1.62</td>
</tr>
<tr>
<td>SLALOM</td>
<td>16</td>
<td>1067</td>
<td>64.9</td>
<td>65.4</td>
<td>61.9</td>
<td>59.8</td>
<td>1.09</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>1666</td>
<td>66.9</td>
<td>67.7</td>
<td>63.2</td>
<td>60.1</td>
<td>1.11</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>2205</td>
<td>68.3</td>
<td>70.2</td>
<td>64.7</td>
<td>60.2</td>
<td>1.13</td>
</tr>
</tbody>
</table>

Figure 9: Normalized performance results

LU decomposition: LINPACK

This program was originally written by Prof. Brent to reflect the problem area of solving dense systems of equations using block LU decomposition. Problem size \( n \) refers to a matrix of \( n \times n \) elements. Note that in this measurement, broadcast functions of RTC are not used. Thus, the results reported here are slower than those in [16].

Scaled conjugate gradient method: SCG

SCG solves the two-dimensional finite difference method using the scaled conjugate gradient method in which a matrix is scaled by diagonal elements. Problem size \( n \) refers to a \( n \times n \) mesh to be partitioned. The matrix to be solved is \( n^2 \times n^2 \) in size.

Slalom benchmark: SLALOM

SLALOM[17] is a computer benchmark which solves an optical radiosity in the interior of a box. The problem size is set to the number of patches that LS+BR mode can calculate in one minute.

These results reflect the results of the communication performance. LS+BR shows better performance overall the items. Receiving overhead, such as interrupts, is also dominant for large applications. When receiving overhead is reduced, reducing the sending overhead is effective. The problems we pointed out in Section 2.2, including line sending using the CPU, effects of the trap, and complicated ring buffer operation, become negligible.

7 Conclusion

We have introduced a scheme to perform interprocessor communication in a distributed memory parallel computer by using line sending and buffer receiving. We evaluated it by running several benchmarks, and proved that these mechanisms to support low-latency communication can speed up applications, even if these functions consume CPU time.

Reducing the overhead of sending messages speeds up the communication performance of a parallel machine which already has efficient receiving mechanisms. It is important to utilize cache memory for message passing functions. Line sending can initiate message transfer...
directly and immediately, and can be started in same way whether data is cached or not.

For buffer receiving, considerable overhead may be introduced in message searching, because in the current implementation, messages are searched from the oldest to the newest. If message arrival is scheduled well, messages that arrive first are read out first, and overhead is kept very small. However, if messages are not read out soon and remain in the ring buffer, the search time increases.

If many messages arrive at the same time, the ring buffer overflows. To prevent this, we should consider message scheduling. However, buffer receiving tolerates some unordered messages.

When a message is wrapped around in the ring buffer, a remainder of the message is appended to the bottom of the ring buffer. This is not a good approach because it wastes memory and requires copying. Invalidation of areas in the ring buffer that have been read are processed by software. These do not affect the present results.

Current implementations of line sending are blocked by network congestion, which is reported by the occurrence of traps. This does not cause performance problems, as the cost of a trap is negligible.

Our results show that line sending and buffer receiving can speed up many kinds of applications.

Acknowledgments

We thank Dr. Ishii, Mr. Shiraisi, and Mr. Uchida for their helpful suggestions, Prof. Richard Brent at Australian National University for use of the LU decomposition program, and Mr. Kenichi Hayashi for use of the SCG method program. We also thank the referees for their helpful comments and suggestions.

REFERENCES


