A Comparison of Different Message-Passing Paradigms for the Parallelization of Two Irregular Applications

Seungjo Bae
Sanjay Ranka

CRPC-TR94505
July, 1994

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Submitted to the Journal of Supercomputing

A Comparison of Different Message-Passing Paradigms for the Parallelization of Two Irregular Applications

Seungjo Bae and Sanjay Ranka

4–116 Center for Science and Technology
School of Computer and Information Science
Syracuse University
Syracuse, NY 13244

July 28, 1994

Submitted to the Journal of Supercomputing

Abstract

We present experimental results for parallelizing two breadth-first search-based applications on the CM-5 by using two different message-passing paradigms, one based on *send/receive* and the other based on *active messages*. The parallelization of these applications requires fine-grained communication. Our results show that the *active messages*-based implementation gives significant improvement over the *send/receive*-based implementation. The improvements can largely be attributed to the lower latency of the *active messages* implementation.

1 Introduction

Two different message-passing paradigms are available on the CM-5 for point-to-point communication. One is the classical message-passing paradigm using ***send/receive*** functions, and the other uses the ***active messages*** developed by von Eicken et al. [11]. This paper describes the parallelization of Wolff cluster algorithm [8, 19] and Lee's maze routing algorithm [15, 12, 20, 1] using the above message-passing paradigms available on the CM-5. These applications require fine-grained communication (small messages) for efficient parallelization.

We have studied the tradeoffs of two optimizations to reduce the overhead of the communication cost for the above applications:

1. *Reducing the number of messages:* In the send/receive implementation, latency is very large compared to the actual cost of sending a few bytes of information. This can largely be attributed to the low-level messages needed for the handshaking required between source and destination processors. Hence a great deal of emphasis has to be placed on minimizing the number of messages sent, which implies a local coalescing of messages. Any communication requests during local computation are not processed immediately, but are saved in temporary communication buffers until local computation is finished. This typically entails extra copying costs. When message coalescing is not used, each processor immediately sends a message whenever communication is required. Potentially, this can reduce the idle time of the receiving processor.

2. *Overlapping communication with computation:* Another way to reduce the overhead of communication is to overlap computation with communication, which is done by using a nonblocking communication function to start communication. If useful computation can be performed until the message reply comes (if a reply is required), communication time can potentially be overlapped.

There are two implementations of the active messages paradigm available on the CM-5: CMMD Active Message Layer (CMAML), which is the protocol-less transport layer on which the CMMD functions are built [10], and Strata, which is the multi-layer communications library developed at MIT. Strata is compatible with CMMD as well as with CMAML and has been shown to have slightly lower communication overhead than the CMAML implementation [5, 6]. The active messages implementation on the CM-5 (CMAML and Strata) has been shown to have a significantly lower startup cost than send/receive message-passing functions, a reduction achieved largely by removing the handshaking required in previous send/receive-based protocol.

The above two message-passing paradigms and optimizations can potentially result in 12 different implementations (Figure 1). We discuss five of these schemes (Figures 2 and 3) in this paper and present extensive experimental results on the performance of the applications using these schemes. These applications require fine-grained communication for effective parallelization. Our experimental results suggest that the two overlapping schemes without message coalescing using Strata and CMAML active messages provide significant performance improvements over other

schemes. However, active messages-based implementations require careful attention to polling and synchronization.

The rest of the paper is organized as follows. In Section 2 we describe two applications—Wolff cluster algorithm and Lee's maze-routing algorithm. We discuss the parallelization of these two applications in Section 3. In Section 4 we present the performance of basic primitives using the different message-passing paradigms, and in Section 5 we describe five communication schemes. Section 6 presents the experimental results.

2 Applications

2.1 Wolff Cluster Algorithm

The Wolff cluster algorithm is a single-cluster Monte Carlo algorithm for the Ising model [8, 19]. The algorithm [8, 16, 19, 13] is given in Figure 4 and a simple example is presented in Figure 5 which assumes that the bond activation probability p is one (a site is always connected with its neighboring site if both sites have the same spin values).

One method used for growing a cluster in the Wolff cluster algorithm is the *ants-in-the-labyrinth* algorithm. This algorithm is similar to breadth-first search on an undirected graph [3, 8]. An initial site which acts as the first element of a cluster is selected at random from a lattice and an ant is placed on it. The ant propagates by placing a child on each of its four neighboring sites with bond activation probability p. Each ant in the first generation of ants checks each of four neighboring sites in turn and places a child of its own with probability p on any unoccupied site. The cluster continues to expand until no more ants are produced [8].

For the sequential ants algorithm we need queues that allow two operations, *enqueue* and *dequeue*. The pseudo code[1] is shown in Figure 6, [4]. The difference between this algorithm and breadth-first-search is that there is no global adjacency matrix (or list). When a site is removed from a queue, the bond is generated dynamically between the site and its neighbors.

The Wolff cluster algorithm is a variation on the Swendsen-Wang cluster algorithm [16]. The main difference is that in the former a single cluster is grown from a site selected at random and flipped with probability 1, while in the latter multiple clusters are grown and all clusters are formed and flipped with probability 1/2. Both algorithms use the same bond activation probabilities [13, 17, 18]. The sequential algorithm for the Wolff cluster algorithm is more efficient than the Swendsen-Wang algorithm [3], but it is more difficult to parallelize the Wolff cluster algorithm because it involves only a single cluster, while the Swendsen-Wang algorithm involves the entire lattice [3, 8].

Let the cluster size be M (the number of sites in a cluster).[2] The sequential ants algorithm takes time $\Theta(M)$. The size of a cluster depends on the bond activation probability p. The value of p depends on the value of β (Figure 4). A larger value of β implies larger p and M. The

[1] In the Wolff cluster algorithm a free-boundary lattice is used, hence the whole lattice is wrapped around in both directions.

[2] The exact value of M can be known only after one sweep is finished.

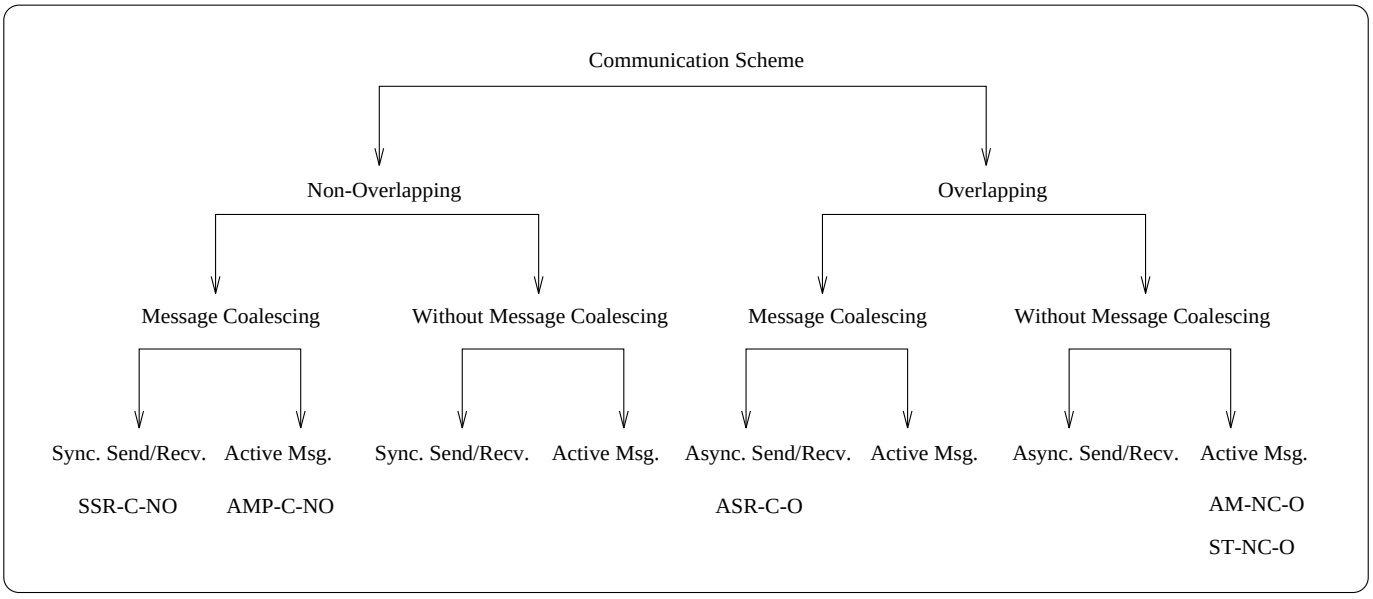


Figure 1: Twelve different communication schemes

Abbr.	Description
SSR	Synchronous Send/Receive
ASR	Asynchronous Send/Receive
AM	CMAML Active Message
AMP	CMAML Active Message–PUT
ST	Strata Active Message
C	Message Coalescing
NC	Without Message Coalescing
O	Overlapping
NO	Non-Overlapping

Figure 2: Abbreviations used in Figure 1

Abbreviation	Communication scheme		
	Message paradigm	Coalescing.	Overlapping
SSR-C-NO	Synchronous Send/Receive	Yes	No
AMP-C-NO	CMAML Active Message–PUT	Yes	No
ASR-C-O	Asynchronous Send/Receive	Yes	Yes
AM-NC-O	CMAML Active Message	No	Yes
ST-NC-O	Strata Active Message	No	Yes

Figure 3: Summary of different schemes for which performance was measured

1. Pick a site i_0 at random as the first site in a cluster.
2. Grow a cluster from a site i (initially, i_0) in the cluster by connecting bonds to nearest neighbor j with probability $p(i,j)$,
   where
   $p(i,j) = 1 - \exp[-\beta(1 + S_i S_j)]$,
   S_i, S_j = spin value,
   $\beta = \frac{J}{KT}$,
   J = interaction strength between two spins,
   K = Boltzmann's constant,
   T = temperature.
   If no site is expanded, halt.
3. Flip the spins in the cluster expanded in Step 2.
4. Go to Step 2.

Figure 4: Wolff's cluster algorithm

execution time of the parallel ants algorithm depends on the size as well as the shape of a cluster.

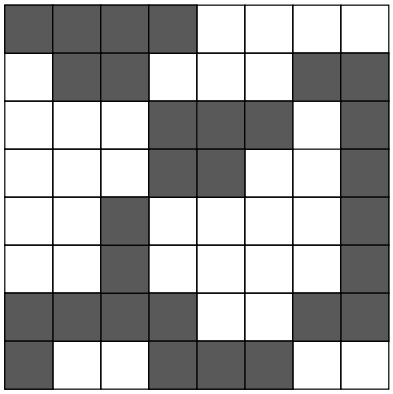
2.2 Lee's Maze-Routing Algorithm

Lee's maze-routing algorithm is a well-known routing algorithm in VLSI circuits [15, 12, 20]. In this paper we consider only the single layer case in which a surface can be represented by cells in a two-dimensional grid. The grid and the cell correspond to the lattice and the site, respectively, in the Wolff cluster algorithm. Unlike the Wolff cluster algorithm, the grid has four boundaries. The goal is to find a shortest path from a source cell to a destination cell. Some cells are blocked and therefore cannot be on the shortest path [20].

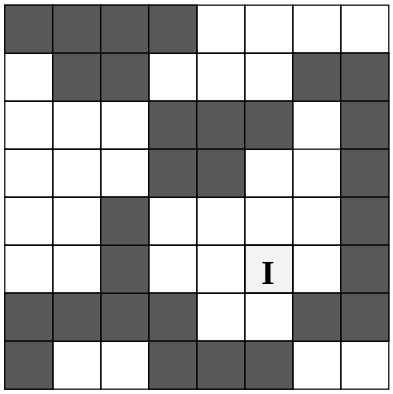
Lee's maze-routing algorithm consists of three phases (Figure 7) [20, 1]. The first phase, *wave-front expansion*, is similar to the ants-in-the-labyrinth algorithm and uses a breadth-first-search for expanding the wavefront. However, in Lee's maze-routing algorithm any two neighboring cells in a grid are connected unless one of the sites is blocked. During breadth-first-search every visited cell has a label which is used for the *path recovery phase*. This label represents the search direction (i.e., the direction of the parent cell) in the grid. A sweeping queue is maintained for keeping all the leaves of the local spanning tree. This queue is used in the sweeping phase.

In the *path recovery phase* the path from the destination cell to the source cell is traced and all the cells along the path are blocked. In the final *sweeping phase* all labels made in the wavefront expansion phase are cleared for the next routing phase. A simple example for these three phases is presented in Figure 8. The pseudo code of the sequential Lee's maze-routing algorithm is given in Figure 9.

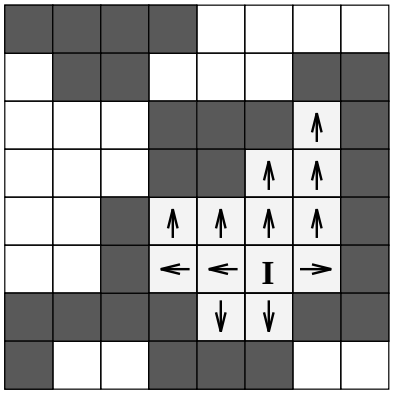
The time complexity of this algorithm depends on the distance (D) from a source cell to a destination cell and the blocking pattern of the cells. In the following analysis we assume the grid is an obstacle-free infinite grid. In the wavefront expansion phase, if the depth of a spanning tree during the breadth-first-traversal is d, then the size of the current wavefront is $1\,(d = 0)$



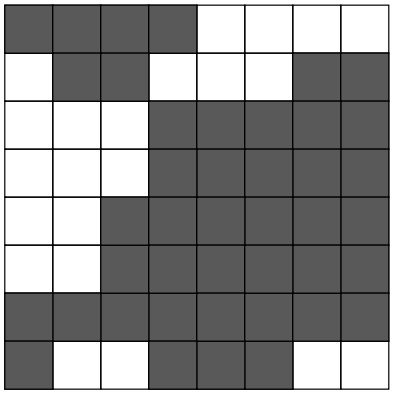
(a) Initial State



(b) After selection of an initial random site



(c) After cluster expansion



(d) Final state

Site with spin value = -1

Site with spin value = 1

I Initial random site of a cluster

Site in a cluster

Figure 5: Wolff cluster algorithm

```
1  procedure Ants-in-the-labyrinth
2      select a site S at random
3      Q ← empty-queue
4      mark[S] ← cluster
5      spin[S] ← −spin[S]
6      enqueue(S, Q)
7      while Q ≠ ∅ do
8          dequeue(P, Q)
9          for each neighbor C of P do
10             if (mark[C] ≠ cluster) and (spin[P] ≠ spin[C]) then
11                 r ← random(0..1)
12                 if r < p then    {p is a bond activation probability}
13                     mark[C] ← cluster
14                     spin[C] ← −spin[C]
15                     enqueue(C, Q)
```

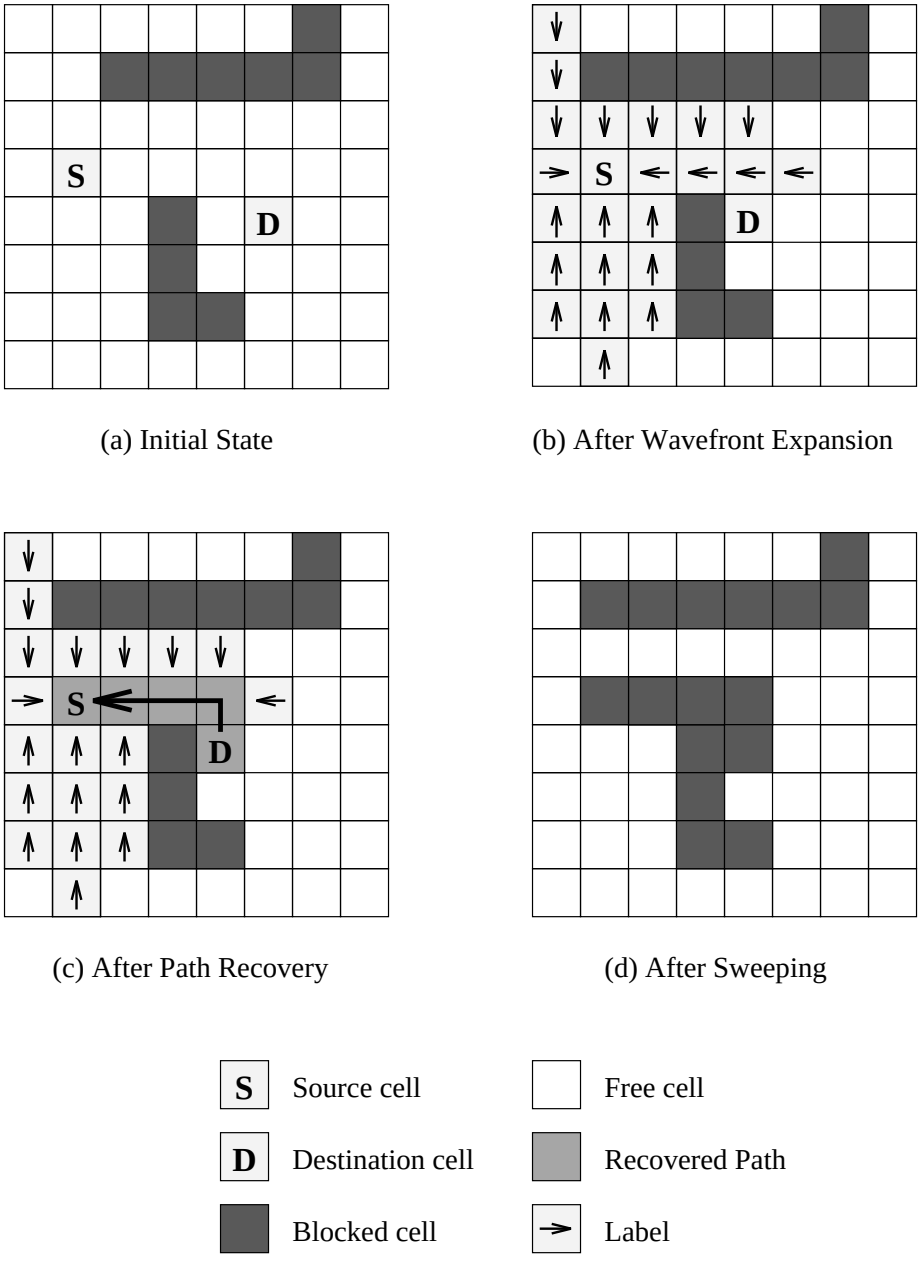
Figure 6: Pseudo code for the sequential *ants-in-the-labyrinth*

1. *Wavefront Expansion*
   Breadth-first-search starting from the source cell is performed. Directional labels such as north, south, east and west are assigned to every visited cell. This label is used in the path recovery phase for identifying the path. This expansion continues until the destination cell is visited.

2. *Path recovery*
   Path from the destination cell to the source cell is traced back by using the label of each cell on the path. Each cell on the path is blocked.

3. *Sweeping*
   Labels of cells visited during the wavefront expansion phase are cleared.

Figure 7: Lee's maze-routing algorithm



(a) Initial State

(b) After Wavefront Expansion

(c) After Path Recovery

(d) After Sweeping

Figure 8: Lee's maze-routing algorithm

```
1  procedure Maze-Routing(Source, Destination)
2      pathfound ← Wavefront-Expansion(Source, Destination)
3      if pathfound = true then
4          call Path-Recovery(Source, Destination)
5      call Sweeping()


6  procedure Wavefront-Expansion(S, D)
7      EQ ← empty-queue
8      enqueue(S, EQ)
9      while EQ ≠ ∅ do
10         dequeue(C, EQ)
11         expanded ← false
12         for each of four neighbors of C(say, N) do
13             if N = D then
14                 status[D] ← direction of N to C
15                 return true
16             else
17                 if status[N] = free then
18                     expanded ← true
19                     case direction of N from C of
20                         north : status[N] ← south
21                         south : status[N] ← north
22                         east  : status[N] ← west
23                         west  : status[N] ← east
24                     enqueue(N, EQ)
25         if expanded = false then
26             enqueue(C, SQ)
27     return false


28 procedure Path-Recovery(S, D)
29     C ← D
30     while C ≠ S do
31         label ← status[C]
32         status[C] ← block
33         case label of
34             north : C ← neighbor cell on the north of C
35             south : C ← neighbor cell on the south of C
36             east  : C ← neighbor cell on the east of C
37             west  : C ← neighbor cell on the west of C
38     status[S] ← block
```

Figure 9: Pseudo code for the sequential *Lee's maze-routing algorithm*

```
39 procedure Sweeping
40     for Q ← EQ, SQ do
41         while Q ≠ ∅ do
42             dequeue(C, Q)
43             finished ← false
44             repeat
45                 case status[C] of
46                     north : status[C] ← free;    C ← neighbor cell on the north of C
47                     south : status[C] ← free;    C ← neighbor cell on the south of C
48                     east  : status[C] ← free;    C ← neighbor cell on the east of C
49                     west  : status[C] ← free;    C ← neighbor cell on the west of C
50                 otherwise
51                     finished ← true
52             until finished = true
```

Figure 9: (*Cont.*) Pseudo code for the sequential *Lee's maze-routing algorithm*

or $4d\,(d > 0)$. Each cell on the current wavefront should check its four neighbors to determine if they are already visited. Since the depth is D after the expansion is finished, it takes time $\Theta(D(D+1)) = \Theta(D^2)$ [4]. In the path-recovery phase every cell on the path is visited, so this phase takes time $\Theta(D)$. The sweeping phase is almost the same as the expansion phase, but since we do not need to check any neighbors, this phase requires time $\Theta(D^2)$.

3 Parallel Algorithms

Both applications require parallelization of a breadth-first search. Effective parallelization requires maintaining good load balance while minimizing communication.

We used the cyclic column-wise striped partitioning (Figure 10) scheme for vertically partitioning the lattice in both algorithms [2, 14, 12]. Let the width of each partition be w. Assuming a lattice of size $N \times N$ and the number of processors equal to $nproc$, the partition width (w) must be in the range $1 \leq w \leq N/nproc$.[3] As w becomes smaller we expect better load balancing but higher communication overhead. The optimal partition width w is determined experimentally.

In the path-recovery phase of Lee's maze-routing algorithm we do not expect any parallelism. Only one processor is active at a time, and the remaining processors are idle. The performance in parallel would be worse than in the sequential case because of the communication overhead. Since the sweeping phase does not require any communication (i.e., only local computations are needed on each processor), our main concern here is to decrease communication time in the wavefront expansion phase.

The wavefront expansion phase of Lee's maze-routing algorithm is similar to the Wolff cluster algorithm. Thus in the following we provide details only of the parallel Wolff cluster algorithm (ants-in-the-labyrinth algorithm). The parallel ants-in-the-labyrinth algorithm consists of five

[3] For the sake of simplicity, we assume that N, $nproc$, and w are the power of two.

[Notation]

N : size of row and column of the global lattice
Nr : size of row of the local lattice
Nc : size of column of the local lattice
$nproc$: total number of processors
w : width of a partition
$pnum$: processor number where $0 \leq pnum < nproc$
(i,j) : coordinate of a site in the global lattice, where $0 \leq i,j < N$
(r,c) : coordinate of a site in the local lattice,
where $0 \leq r < Nr, 0 \leq c < Nc, Nr = N, Nc = \frac{N}{nproc}$

[Mapping Functions]

P : Global lattice $\rightarrow \{0,\ldots,nproc-1\}$
$P(i,j) = pnum$,
where $pnum = (j$ `div` $w)$ `mod` $nproc$
M : Global lattice $\rightarrow$ local lattice
$M(i,j) = (r,c)$,
where $r = i$,
$c = j$ `mod` $w + w \times (j$ `div` $(w \times nproc))$

Figure 10: Mapping function and notation used

steps (Figure 11). Each processor maintains a queue that stores the sites on the current wavefront in its local lattice. During local expansion of the current wavefront (Step 2 of Figure 11), each site on the current wavefront is removed from the queue and used as a parent site to expand the cluster. The bonds between the parent site and each neighboring site are generated with a probability p[4] unless the neighboring site is already an element of the cluster. If the parent site is on the boundary of a partition (i.e., on the left or right edge of a partition),[5] we may need communication with the left or right processor. This communication step (Step 3 of Figure 11) can be performed in three different ways—*non-overlapping scheme with message coalescing* (C-NO), *overlapping scheme with message coalescing* (C-O), and *overlapping scheme without message coalescing* (NC-O).

In Step 4, if any message is received from another processor, then each site in the message is checked to determine whether it can become a new element of the cluster. A synchronization step is required after completing Steps 2, 3, and 4 to ensure all the wavefronts have been completed (i.e., that the cluster stops growing) [20]. This can easily be done by using a global reduction function such as *logical AND*. The naive way is to synchronize all processors every time we finish Steps 2, 3, and 4, but this method increases the idle time of processors [20]. The cost of synchronization can be reduced by performing it every δ-steps (called the δ-synchronization scheme by Won and Sahni [20]). The value of δ is determined experimentally. The pseudo code using a δ-synchronization scheme is presented in Figure 12.[6] Steps 1 and 5 are common for

[4] Every processor uses a different seed for random numbers.

[5] If a local lattice consists of m partitions, there are $2m$ boundary edges.

[6] The effect of δ-synchronization is limited in the communication schemes using active messages. The details are

1. *Selection of an initial random site and initialization of the wavefront*
   Processor 0 selects an initial random site and broadcasts it to the other processors. If a processor has the initial site in its local lattice, it initializes the wavefront at the initial site.

2. *Local expansion of wavefront*
   If a processor has any site on the current wavefront of the cluster, it grows the cluster by expanding the site on the current wavefront within the local lattice. If there is any neighboring site of the site on the current wavefront, communication is required.

3. *Communication*
   Perform inter-communication if there is any communication request during the local expansion of wavefront step.

4. *Process communication messages*
   Process any received messages.

5. *Check empty wavefront*
   Every processor checks to see if the next local wavefront is empty. This step may require global reduction. If every processor has an empty local wavefront, halt. If not, repeat Steps 2, 3 and 4.

Figure 11: Parallel ants-in-the-labyrinth algorithm

all communication schemes. Steps 2, 3, and 4 (lines 20–22 of Figure 12) vary according to the communication scheme.

4 Message-Passing Paradigms on the CM-5

On the CM-5 there are two different message-passing paradigms for point-to-point communication. One is high-level CMMD message passing using send and receive functions which requires three-phase protocol, and the other is Active Messages developed by von Eicken et al. [11].

In the following subsections we describe the CM-5 data network and different message-passing paradigms in detail.

4.1 Data Network on the CM-5

On the CM-5 the Data Network consists of two independent but identical network interfaces that are called *left* and *right* interfaces. Each network interface has two memory-mapped FIFO buffers: one is outgoing FIFO to inject data to the network, and the other is incoming FIFO to extract data from the network. For sending data, a sending processor stores data to the outgoing FIFO. Once the data has been successfully transferred to the network interface, the data is sent to the receiver by the network interface, and a sending processor can continue incoming local computation. For receiving data, a receiving processor either polls data by checking the incoming FIFO or is notified by interrupt on arrival of data. [9].

Data transfer between a sending processor and a network interface can be performed in two different ways [10]:

described in Section 5.

```
1   procedure Parallel-Wolff-Cluster
2       {Step 1: Selection of an initial random site followed by a broadcasting}
3       if pnum = 0 then
4           select an initial random site S, and broadcast it
5       else
6           receive information about the selected random site S from processor 0
7       {Step 1: Initialization of the wavefront}
8       EQ ← empty-queue
9       new-counter ← 0
10      if this processor has S in its local lattice then
11          spin[S] ← −spin[S]
12          mark[S] ← cluster
13          enqueue(S, EQ)
14          new-counter ← 1
15      all-empty-queue ← false
16      while all-empty-queue = false do
17          for i ← 1 to delta do
18              old-counter ← new-counter
19              new-counter ← 0
20              {Step 2, 3, 4: The procedure Expansion(old-counter, new-counter) }
21              {varies according to the communication scheme }
22              call Expansion(old-counter, new-counter);
23          {Step 5: Check if local wavefront is empty}
24          if EQ = ∅ then
25              all-empty-Queue ← true
26          Global-Reduction-AND(all-empty-queue)
```

Figure 12: Pseudo code for the *parallel Wolff cluster algorithm*

1. Blocking function: Until data is accepted by a network interface, a sending processor is blocked. The processor continues to check the status of the network interface.

2. Nonblocking function: If data is not accepted by a network interface, a sending processor continues its computation rather than getting blocked. The processor periodically checks the status of the network interface.

The network interface has no DMA and accepts only five-words-per-packet messages (one word for the message head and four words for data). If a message size is larger than four words, a processor should first packetize the message and then transfer data to the network interface packet by packet [6, 9]. Overlapping communication with computation is limited on the CM-5 due to the low network capacity (average ten packets) and network latency [6].

4.2 Send/Receive-Based Message Passing

CMMD supports two kinds of message-passing functions, *synchronous* message passing and *asynchronous* message passing. In the synchronous send/receive case, a blocking-send function and a blocking-receive function are used; both a sender and a receiver are synchronized using three-phase

protocol. In the asynchronous send/receive case, a nonblocking send function and a nonblocking receive function are used [10].[7]

Synchronous Send/Receive

To establish a communication link between a sender and a receiver when using the synchronous message-passing function, a two-way handshake is required as a preliminary step. First, a sender sends a *Request* signal to the receiver and waits for its reply. When the receiver receives the sender's request, it sends its *Reply* signal back to the sender. After the two-way handshake is finished, the sender and the receiver are synchronized implicitly. The sender then prepares its message and transmits it to the receiver [10]. The time taken by the two-way handshake is overhead on both a sender and a receiver, and this time should be added to the software overhead.

Suppose the send function is initiated on the sender at time x, and the receive function is initiated on the receiver at time y. Then one of two nodes should wait for its partner's response during $|x-y|$. The time interval $|x-y|$ can be regarded as idle time. Notice that the sender should be blocked from time x to the time that it transfers the last byte of message. The receiver is blocked from time y to the time that it receives the last byte of the message. Figure 13 shows the mechanism of synchronous send/receive message transmission. Another synchronous send/receive function, *simultaneous-send-receive*, is available in CMMD. This allows a node to send and receive a message simultaneously and is useful for communication patterns like circular shift.

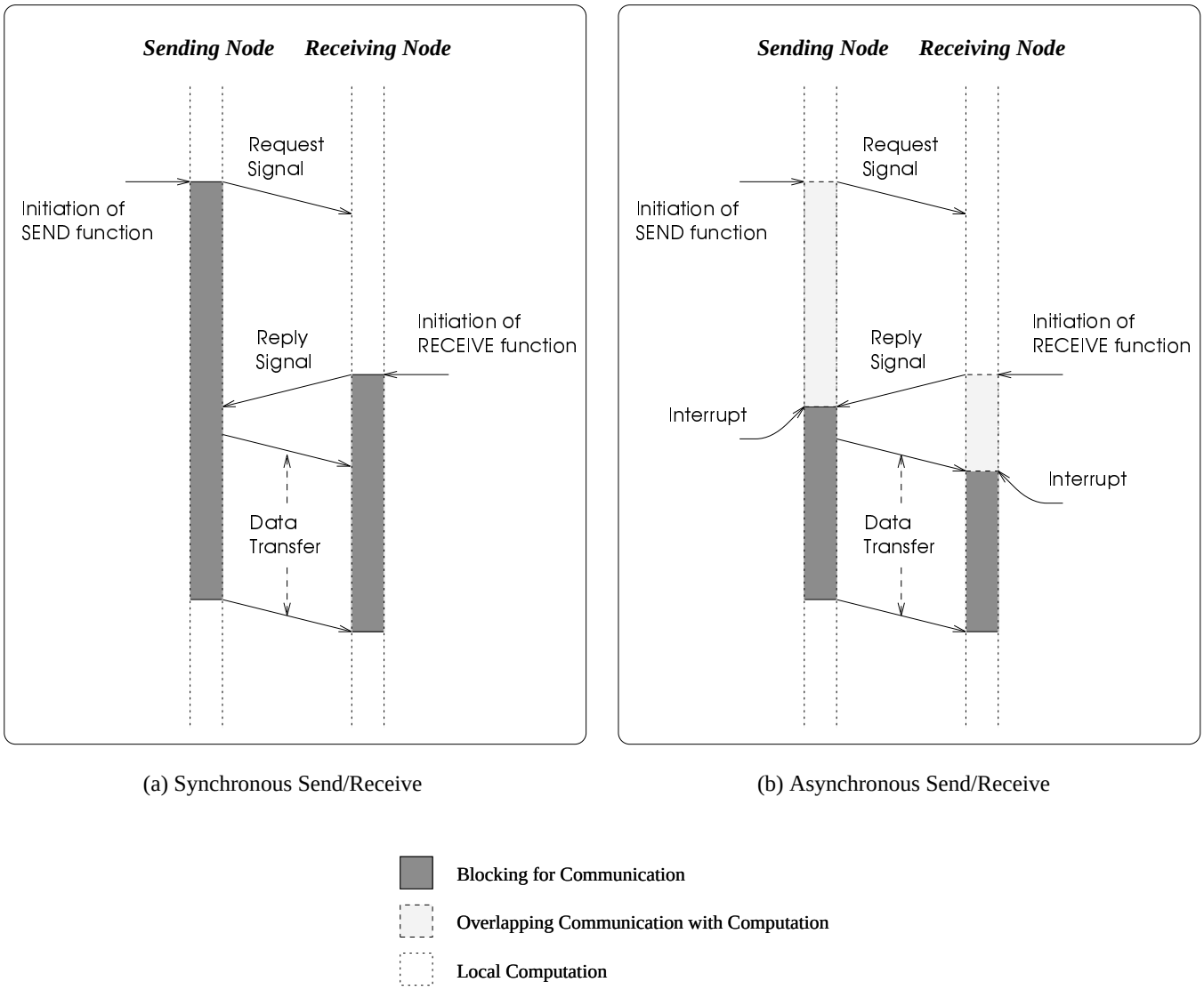
Asynchronous Send/Receive

Asynchronous message passing is usually interrupt-driven. A node signals its intention to send or receive a message, then continues other work until its partner signals that it is ready. At the point that both a sender and a receiver are ready to send and receive, the current process on the sender is interrupted, and the control is moved into the CMMD interrupt handler. A message is then transmitted to the receiver. After the last byte is transmitted, the control is returned into the main thread in the program executed on the sender. On the receiver the arrival of a message causes an interrupt. Once all messages arrive and are stored in local memory, the control is returned into the main thread [10].

Unlike synchronous message passing, a node is not blocked while it waits for its partner to be ready. Therefore, it is possible to do useful work during that period. One of the two nodes might do other work (i.e., ongoing local computation or initiation of another communication) from the time that the send or receive function is initiated to the time the message is actually transmitted or received on a processor (Figure 13).

When using the asynchronous send function, the sender and the receiver must maintain their buffers in their original states until the message has actually been sent or received. This can be rectified by using a *buffering-send* for which the CMMD copies the data into a temporary buffer

[7] Two terms, *blocking* and *nonblocking* used here are different from those mentioned in Section 4.1. In Section 4.1, these two terms are used to define different ways for transfering data from a processor to a network interface on a sender. On the other hand, here they are used to define different ways for cooperating a sender and a receiver.



(a) Synchronous Send/Receive

(b) Asynchronous Send/Receive

Figure 13: Communication mechanism of Send/Receive message-passing paradigm on the CM-5

so that the sender can modify its original buffer. However, efficiency is decreased because of the extra time required to copy the original buffer into a temporary buffer [10].

4.3 Active Messages

There are at least two different implementations of Active Messages available on the CM-5:

1. CM Active Message Layer (CMAML), a protocol-less transport layer on which the high-level CMMD functions are built [10], and

2. Strata, which is the multi-layer communications library developed at MIT and is compatible with CMMD as well as with CMAML [5, 6].

CMAML and Strata provide two kinds of message-transfer functions, Active Messages and Block Data Transfer (Active Messages-PUT), which is similar to *PUT* in *Split–C* developed in [11].

An active message on the CM-5 is a single communication packet (five words) that consists of the address of a handler function (first word) invoked on the destination processor and arguments (remaining four words) to the handler function. This requires programming using a SPMD programming model so that a sender can know the address of any code run in the receiver. By using a handler function associated with every active message we can integrate a communication message into computation [11]. Since there is no buffering on a receiving processor the handler function is invoked immediately when an active message is received by interrupt or polling [10].

The block data transfer (Active Messages-PUT) function transfers a continuous data block from one source processor to a single destination processor. Unlike CMMD send and receive functions, a source node and a destination node should agree as to the size of the data block. A handler function may be used in both the source processor and in the destination processor. The startup latency is much smaller than the send/receive functions [10].

Sending Active Messages

Unlike synchronous and asynchronous message-passing functions, two-way handshaking is not required when using active messages, since there is no explicit receive function on a receiver. For this reason the receiver does not identify the sender when receiving active messages. Active messages are one packet long,[8] and a sender can send active messages without two-way handshaking. However, active messages are blocking functions for a sender when used in the main thread of control. Each active message is associated with its handler function whose role is to integrate communication messages into the local computation by invoking itself on a receiver [11]. For a sender, a handler function can be regarded as a remote procedure call (with limited capabilities) executed on a receiver. An active message consists of the address of the handler function executed on a receiver and arguments to be passed to the remote function. On a receiver, receipt of an active message causes a corresponding handler function to be invoked immediately [10].

CMAML provides three types of active messages on either network interface [10]:

[8] A packet consists of five words, and one word is four bytes on the CM-5.

- *Request message*: A request message can be sent from the main thread of control using left interface to the data network. It is a blocking function and alternates trying to send an active message with receiving data until the active message is sent. For receiving data it polls both interfaces.

- *Reply message*: A reply message may be sent from a handler function (usually of request messages). It is a blocking function and alternates trying to send an active message with receiving data until the active message is sent. When trying to send and to receive it uses only the right interface.

- *Remote Procedure Call (RPC) message*: A remote procedure call (RPC) message can be sent either from the main thread of control or from handler functions using both interfaces to the data network. An RPC message sent from the main thread of control is a blocking function. Until the active message is sent, it alternates trying to send an active message with receiving data. When trying to send and receive data, it alternates sending attempts on both interfaces and receiving attempts on both interfaces. If an RPC message is sent from a handler function, it is a nonblocking function. However, it is guaranteed that an RPC message is transferred to the network interface before the control returns from the handler function to the main thread of control.

Request and reply messages are usually used for cases in which one node sends a *request message* for requesting data and then the other node sends back the requested data using a *reply message* within the handler function of the received *request message*. In cases when a request-reply communication pattern is not required, RPC messages can be used instead to give better performance since they use both interfaces [10].

Strata provides two more transfer functions in addition to the three CMAML active messages mentioned above. Each of these five functions specifies which interface of the data network is used when sending or polling active messages. Strata active messages have been explained in detail in [5].

Receiving Active Messages

On the receiving node any incoming active messages can be received using either automatic interrupt or explicit polling. In either case, active messages are fetched from the data network and corresponding handler functions are invoked immediately [10]. If the receiving node can predict when the message will arrive explicit polling is more suitable and efficient. Polling has much lower overhead than automatic interrupt has. When any incoming active messages are received via explicit polling, one should poll often to prevent network congestion [10, 6].

CMAML and Strata provide various polling functions, each of which uses the left, right, or both interfaces to the data network for receiving incoming active messages. Strata provides a wider range of polling functions [10, 5]. Most of CMAML and Strata active message functions alternate trying to send an active message with polling any incoming active message. Therefore, if a processor is sending active messages as well as receiving active messages (as in circular shift),

explicit polling is not always necessary. One of the exceptions is CMAML RPC, which does not poll at all unless it fails to send an active message [10, 7].

Block Data Transfer

Data can be transferred in blocks on CMAML using Active Messages-PUT. There is neither internal two-way handshaking nor an explicit receive function. The following steps are needed between a sender and a receiver before actual transfer of data [10]:

1. Two nodes should agree which receive port (say rport) will be used. The receive port is a data structure containing all information. Since there is no explicit receive function, the sender should send data to the rport agreed upon by both nodes.

2. Two nodes should agree on the length of the message. As each data byte arrives, the byte counter of the rport is decremented. When this counter becomes zero, the rport's handler function is executed.

3. The receiver should initialize the counter of the rport to the number of bytes it expects to receive in a transfer before the sender begins to transmit the message.

All three steps can be accomplished by an extra communication between a sender and a receiver. However, depending upon the application requirements, all three steps are not always necessary. In many regular problems the receiver knows the message size. For such cases Step 2 is not necessary. If the communication pattern is repetitive, Step 1 is required only once. Since each node knows the senders, they can be assigned fixed rport numbers. Step 3 is needed for every transfer. Active messages-PUT is a blocking function when used in the main thread of control, hence the sender is blocked until the last byte of data is transferred to the network interface.

Strata provides two active messages-PUT functions. These functions use both interfaces when trying to send data. When trying to receive data, one polls both interfaces, but the other polls only the right interface [5].

Block data is transferred in a series of single-packet active messages. Since packet ordering is not guaranteed on the CM-5 network [11], each message header contains the offset of data and an rport number. On arrival of each active message, a special system handler function is invoked on a receiving processor. This handler function stores data (four words) in the corresponding rport according to the offset of data. After all messages are received, user-defined handler function is invoked, and this handler function transfers data from the rport to the local communication buffer [5, 10].

Active messages and active messages-PUT provide the application programmer a great deal of flexibility. Since they have much lower startup cost than the send/receive message paradigm, they can be used in both a synchronous and an asynchronous manner.

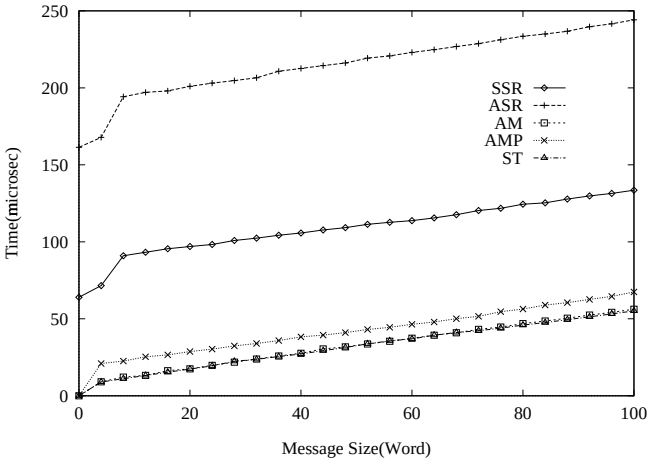


Figure 14: Communication time for one-to-one communication

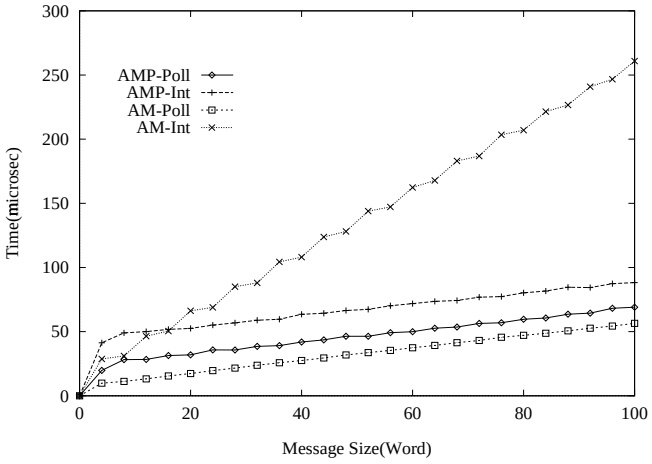


Figure 15: Effect of using interrupt vs. polling in CMAML RPC active messages and CMAML active messages-PUT

4.4 Performance Evaluation

Here we investigate two message-passing paradigms in detail and present experimental results for one-to-one communication, as well as two-directional circular shift. The latter is the communication pattern required for the parallel Wolff cluster algorithm and Lee's maze-routing algorithm.

We measured the communication time for sending data, (i.e., one-to-one communication) between two directly connected nodes (the number of links between two nodes is two on the CM-5). At least 1000 iterations were performed and half of the time for a round-trip message was measured. The two nodes were synchronized before measuring the time. In asynchronous message passing, *buffering-send* was used. In active messages, CMAML RPC active message, Strata RPC active message, and CMAML active messages-PUT (Block Data transfer) were used. Active messages were received by polling functions that use both left and right interfaces.

For a message of size S words, we sent $\left\lceil \frac{S}{4} \right\rceil$ RPC active messages. Figure 14 shows the average communication time (microsecond) for one-to-one communication for different message paradigms. From these results we can see that Strata and CMAML active messages give the lowest communication time. The difference of communication time between Strata and CMAML is very small. Note that the communication cost for all message paradigms shows the same slope, hence the difference in communication time is due to the difference of the software overhead. When the send/receive message paradigm is used for small sized messages, the software overhead dominates the actual data transfer time.

We also investigated the effect of using interrupt vs. polling in CMAML RPC active messages and CMAML active messages-PUT. We measured the communication time for CMAML RPC active messages and active messages-PUT via both automatic interrupt and explicit polling using the same polling function as used in the previous experiment. Figure 15 shows the performance results for different message sizes on the CM-5. Active messages using explicit polling gave the

best results. The application programmer should carefully consider this interrupt overhead when sending large numbers of active messages as compared to sending a large message using active messages-PUT. Sending active messages or active messages-PUT via automatic interrupt is more convenient to use, but is less efficient.

We also measured the communication time for a two-directional circular shift of distance one—that is, each processor communicates in a ring pattern in both directions (to the left and to the right). In synchronous send/receive cases the *simultaneous-send-and-receive* function has been used instead of the blocking-send and blocking-receive functions. In all other cases the same communication functions as those used in the first experiment have been used.

In one-to-one communication a sender continues to send multiple active messages, while a receiver receives all active messages by polling repetitively.

In two-directional circular shift each node needs not only to send active messages but also to receive active messages. Strata active messages always poll the interfaces to the data network when sending active messages, thus only limited polling is required to ensure all messages are received after all active messages are sent.

CMAML RPC active messages do not poll at all unless they fail to send an active message. An explicit polling may be necessary when multiple active messages are sent. It is important to poll often to prevent network congestion. The naive method is to poll explicitly for every active message sent. This method might give a poor performance due to the polling overhead. Suppose we have M active messages to be sent and received. Let $\mathcal{F}$ define the number of active messages sent, after which polling is performed explicitly. We investigated various polling methods using CMAML RPC active messages with various $\mathcal{F}$ values ($\mathcal{F}$ = 1, 5, 10 and M) in two-directional circular shift. Figure 16 shows the results for various polling methods. The polling method of $\mathcal{F} = 1$ gave the worst performance, and that of $\mathcal{F} = 5$ gave the best results. For the following experiments, the value $\mathcal{F}$ was fixed to 5.

Communication time for two-directional circular shift using active messages is given in Figure 17. Unlike one-to-one communication, the communication time for CMAML active messages is higher than that of active messages-PUT for message sizes greater than about 60 words. Strata-based active messages give a better performance than that of CMAML active messages-PUT when the message size is less than 100 words, but the difference of communication time between Strata-based active messages and CMAML active messages-PUT decreases as the size increases.

Figure 18 shows the communication time for two-directional circular shift using send/receive message-passing paradigms and the active messages paradigm. As in one-to-one communication, active messages-based implementation outperforms send/receive message-passing based primitives. Asynchronous send/receive has a worse performance than synchronous send/receive. However, if the idle time is efficiently overlapped with useful computation, asynchronous communication might give a better performance.

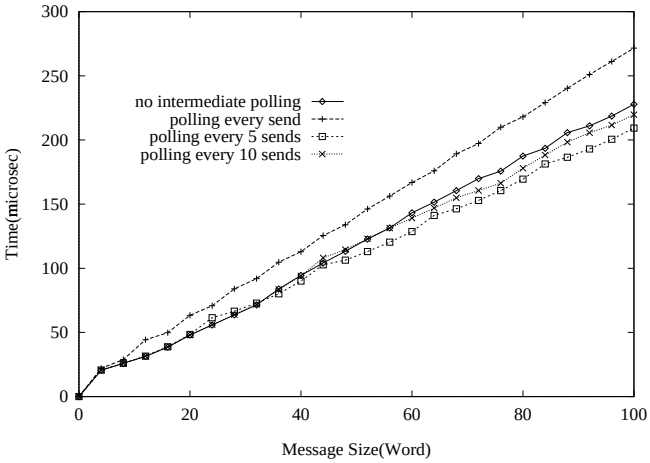


Figure 16: Comparison of different polling strategies of CMAML active messages for two-directional circular shift (the size on the x-axis is the message size to be sent in each direction)

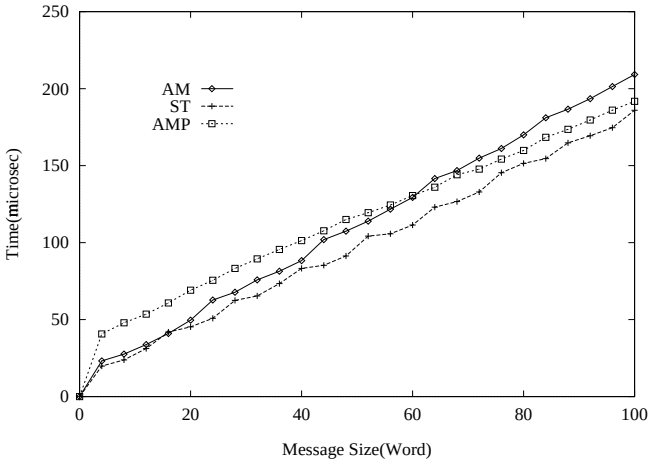


Figure 17: Communication time for two-directional circular shift using active messages (the size on the x-axis is the message size to be sent in each direction)

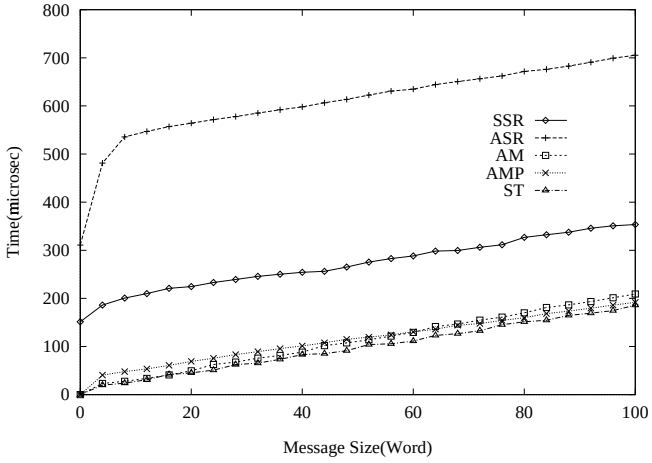


Figure 18: Communication time for two-directional circular shift using send/receive message-passing paradigm and active message paradigm (the size on the x-axis is the message size to be sent in each direction)

5 Communication Schemes

In the Lee's maze routing algorithm, a processor needs to communicate the coordinate of a remote site to a neighboring processor whenever it expands a site on the boundary. In the case of Wolff cluster algorithm, the spin value of the local parent site is also needed. The size of communication is at most three words (two words in Lee's maze-routing algorithm) per site. Each processor expands sites on the current local wavefront at each cycle. The total number of sites requiring communication is relatively small. If the partition width is w, each processor has $2 \times N/(w \times nproc)$ boundary edges. This makes not performing message coalescing a viable option only for the active messages-based implementation.

As mentioned earlier, each processor performs the following three processing steps at each cycle (Steps 2, 3, and 4 in Figure 11):

- *Local Expansion*: local computation requiring only local data.
- *Communication*: preparing and sending message to update remote site in neighboring processors.
- *Remote Expansion* (*Processing Communication Message*): Computation on local data using the messages received from neighboring processors.

Figure 19 shows the structure of the three processing steps (Local Expansion (LE), Communication (IC), and Remote Expansion (RE), respectively) that need to be performed in each processor, and the data flow between these steps at each cycle for the different communication schemes.

In the non-overlapping-with-message-coalescing scheme, three processing steps are performed sequentially on every processor. In the IC step each processor cooperates with its two neighboring processors to exchange coalesced messages. At the end of cycle k, a new wavefront of distance k is completely formed.

IC and RE steps can be delayed by one cycle in order to overlap local computation with communication [20]. At the beginning of the cycle k, each processor has two inputs: partial local wavefront of distance $k-1$ expanded at cycle $k-1$ (which is the input for the LE step), and candidates for remote wavefront of distance $k-1$ (which is the input for the IC step). In the LE step a partial local wavefront of distance k is generated from the partial local wavefront of distance $k-1$, and candidates for a remote wavefront of distance k are formed. In the IC step each processor exchanges the candidates for a remote wavefront of distance $k-1$ with its neighbors. Since there is no dependency between the LE and IC steps of a given cycle, these two steps can potentially be overlapped using asynchronous messages. In the RE step, the candidates for a local wavefront of distance $k-1$ received from their neighbors are checked to see if they can be part of a local wavefront of distance $k-1$. Any candidates that are part of a local wavefront (of distance $k-1$) are also expanded, thus the partial local wavefront of distance $k-1$ and k, as well as the candidates for a remote wavefront of distance k, are completed. Let $l(k)$ be a partial local wavefront of distance k made in the LE step, and $r(k-1)$, $r(k)$ be a partial wavefront of distance

$k-1$ and k generated in the RE step, respectively, (i.e., $r(k)$ is expanded from $r(k-1)$). Consider a local site x such that $x \in l(k) \bigcap r(k-1)$. It is possible that a site at distance $k-1$ was marked with a distance k because the LE step is performed before the RE step.[9] Removing this discrepancy requires that we record the distance of each expanded site; any site x such that $x \in l(k) \bigcap r(k-1)$ may require a correction. However, extra memory and computation are required to resolve the conflict.

Overlapping communication without a message coalescing scheme can be executed easily by using active messages. An active message is sent as soon as communication is requested. The handler function corresponding to each active message performs an RE step on a receiving node.

The following subsections describe the programming details of the various communication schemes.

Non-overlapping Scheme with Message Coalescing

In this scheme communication is performed after all processors finish their local wavefront expansion, which requires saving all communication requests during the local wavefront expansion step in temporary buffers. Each processor begins to process the communication buffer as soon as it finishes its receiving operation [20]. The pseudo code of *Expansion* is given in Figure 20. The variable *old* represents the number of sites on the current wavefront, and the variable *new* records the number of sites expanded from the current wavefront. The routine *Local–Expansion* is almost the same as in the sequential algorithm, the only difference being that a check needs to be made for any communication requests. Since we need only two-directional communications, the communication requests are saved in the left or right communication buffers. The communication step (line 10 of Figure 20) can be completed using synchronous send/receive or active messages-PUT.

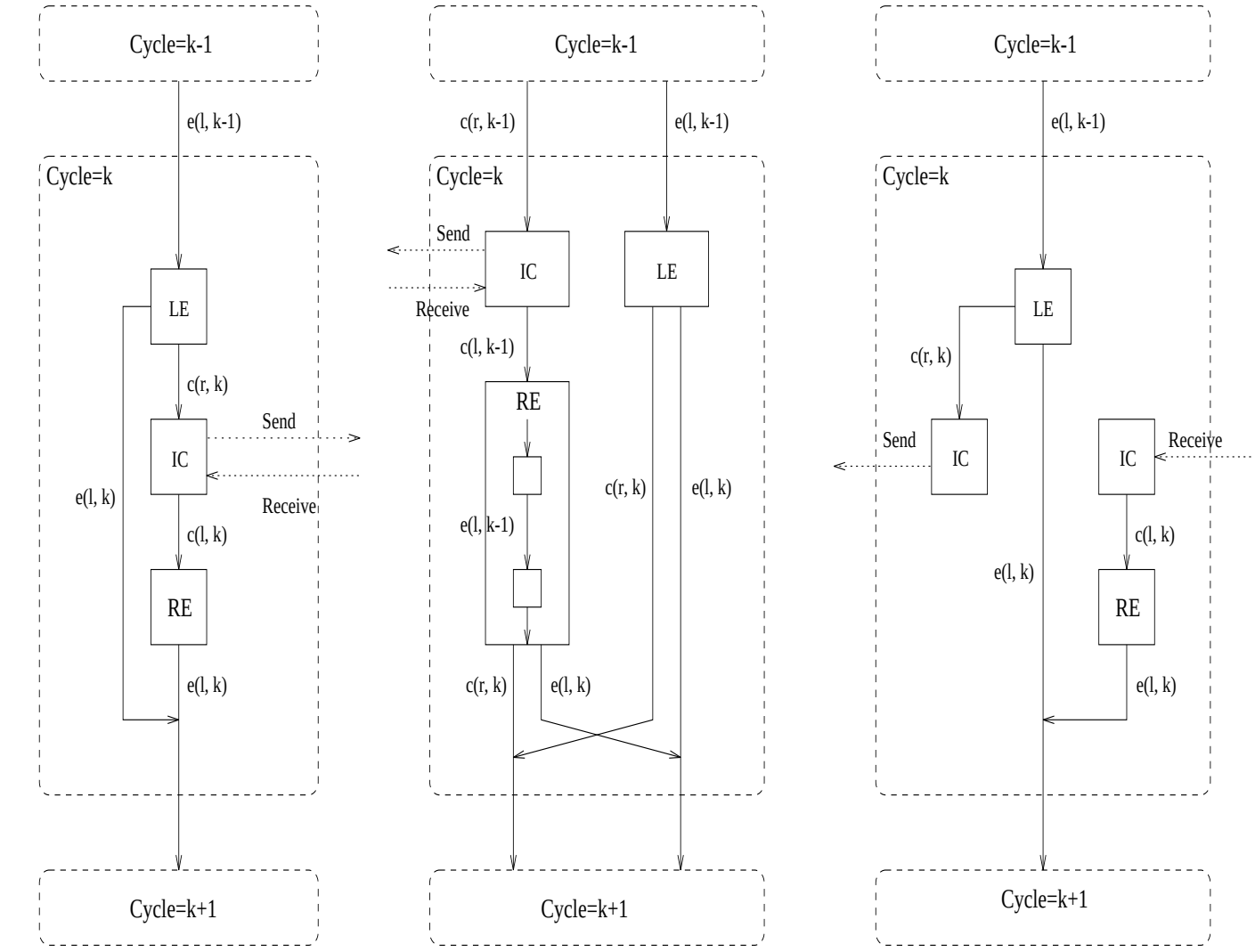
Since the wavefront is irregular in size and shape, the receiver needs to know the size of the communication buffer for local processing (Step 4 of Figure 11). For the send/receive implementation, the receiver specifies an oversized buffer (Figure 21). Each processor is locally synchronized with its right and left processors at each cycle.

Performing *Communication* using active messages-PUT (Figure 22) requires the following steps:

1. Since the communication pattern is regular (two-directional circular shift), Step 1 is required in the initial phase only.

2. In order for a sender and a receiver to agree on the size of the communication buffer, each processor sends the size of the communication buffer to its left and right neighboring nodes as the preliminary step (lines 2–7 of Figure 22). This is done by sending another active message, which sets the rport size counter on the receiver. All nodes must be synchronized before sending this preliminary active message (line 3 of Figure 22), which prevents any node from changing the rport information before a receiver finishes processing its communication

[9] This is only required for Lee's maze-routing algorithm.



LE = Local Expansion	e(l, k) = expanded element of distance k in local domain
IC = Communication	c(r, k) = candidate of distance k in remote domain
RE = Remote Expansion	c(l, k) = candadate of distance k in local domain

Figure 19: Structure of three processing steps, Local Expansion, Communication and Remote Expansion for different communication schemes

```
1  procedure Expansion(old, new)
2      {Step 2: Local Wavefront Expansion}
3      clear buffer SRB, SLB, RRB, RLB
4      {SR(L)B: Send-to-the-Right(Left) Buffer}
5      {RR(L)B: Receive-from-the-Right(Left) Buffer}
6      for i ← 1 to old do
7          dequeue(parent, EQ)
8          call Local–Expansion(parent, new)
9      {Step 3 and 4: Communicate with coalesced message and process received comunication buffer}
10     call Communication(new)


11 procedure Local–Expansion(P, new)
12     for each neighbor C of P do
13         if C is a local site then
14             if (mark[C] ≠ cluster) and (spin[P] ≠ spin[C]) then
15                 r ← random(0..1)
16                 if r < p then    {p is a bond activation probability}
17                     mark[C] ← cluster
18                     spin[C] ← −spin[C]
19                     enqueue(C, EQ)
20                     new ← new + 1
21         else
22             {Message coalescing}
23             put C and spin[P] into SRB(or SLB) according to their destination
```

Figure 20: Pseudo code of *Expansion* for the non-overlapping scheme with message coalescing (C-NO) in the parallel Wolff cluster algorithm

buffer. Another barrier (line 8 of Figure 22) is required to ensure that a sender begins to send data only after the rport counter is set on a receiver.

Thus two barriers and two extra active messages (to the left and to the right) are required at each cycle. Although an efficient barrier mechanism using a control network is provided on the CM-5, it can potentially increase the idle time of the processors. When using active messages-PUT in the above fashion, the interrupt overhead can be avoided by using the explicit polling function.

A sender and a receiver can be locally synchronized in the same sense as the synchronous send/receive message-passing paradigm. These schemes require extra communication and make the algorithm more complicated than that using barriers.

Overlapping Scheme with Message Coalescing

In the overlapping scheme with message coalescing we overlap the computation (Step 2 of Figure 11) and the communication (Step 3 of Figure 11) by using asynchronous communication

```
1  procedure Communication(new)
2      sync-send-and-receive(left processor, SLB, right processor, RRB)
3      sync-send-and-receive(right processor, SRB, left processor, RLB)
4      call Process-Received-Buffer(new, RRB)
5      call Process-Received-Buffer(new, RLB)
```

Figure 21: Pseudo code of *Communication* for the non-overlapping scheme with message coalescing and using synchronous Send/Receive (SSR-C-NO) in the parallel Wolff cluster algorithm

```
1  procedure Communication(new)
2      sizeSL ← size of SLB; sizeSR ← size of SRB
3      barrier();
4      active-message(left processor, sizeSL, sizeRR)
5      active-message(right processor, sizeSR, sizeRL)
6      poll-while(sizeRR)
7      poll-while(sizeRL)
8      barrier();
9      if sizeSL > 0 then
10         active-message-PUT(left processor, SLB)
11     if sizeSR > 0 then
12         active-message-PUT(right processor, SRB)
13     if sizeRR > 0 then
14         poll-while(RRB)
15         call Process-Received-Buffer(new, RRB)
16     if sizeRL > 0 then
17         poll-while(RLB)
18         call Process-Received-Buffer(new, RLB)
```

Figure 22: Pseudo code of *Communication* for the non-overlapping scheme with message coalescing and using Active Message–PUT (AMP-C-NO) in the parallel Wolff cluster algorithm

```
1  procedure Process-Received-Buffer(new, Buf)
2      while Buf ≠ ∅ do
3          get a site C and its remote parent's spin value Pspin from Buf
4          if (mark[C] ≠ cluster) and (spin[C] ≠ Pspin) then
5              r ← random(0..1)
6              if r < p then
7                  mark[C] ← cluster
8                  spin[C] ← −spin[C]
9                  enqueue(C, EQ)
10                 new ← new + 1
```

Figure 23: Pseudo code of *Process-Received-Buffer* for the non-overlapping scheme with message coalescing (C-NO) in the parallel Wolff cluster algorithm

functions. The pseudo code is given in Figure 24.[10] Each processor first declares its readiness for sending and receiving data and then immediately starts its local expansion step, thus the communication step is overlapped with local computation (lines 3–8 of Figure 24). Note that we use *buffering-send* because we need to modify a communication buffer immediately during the local expansion step, regardless of the fact that the communication buffer is transferred.[11] Each processor is synchronized locally with only its right and left processors. All processors are synchronized globally only every δ cycles.

Overlapping Scheme Without Message Coalescing

The total amount of communication required by Wolff cluster algorithm and Lee's maze-routing algorithm at each iteration is relatively small. During the local wavefront expansion phase updating of nonlocal sites is achieved by sending an active message containing the coordinate for a remote site and the spin value of a local parent site (lines 22–27 of Figure 25).

The variable $sizeSR$ (or $sizeSL$) keep track of the total number of active messages sent at each cycle. A periodic polling method is used for CMAML implementation. This polling is done whenever expansion of a site is completed. No explicit polling is required when using Strata active message function, since Strata send-functions always poll whenever sending an active message. When an active message is received by polling, the handler function is invoked on the receiver and each site in the message is checked to see if this site can become a new element of a cluster by using the handler function (lines 28–36 of Figure 25). The variable $countR$ or $countL$ keep track of the total number of active messages received at each cycle.

After finishing local expansion, each processor, using another extra active message, sends to its neighbors the total number of active messages that it has sent. This ensures that all messages have arrived before proceeding to the next cycle (by comparing the value of $sizeRR$ with $countR$ or $sizeRL$ with $countL$). Since packet-ordering is not guaranteed on the CM-5 [11], each node should record the number of active messages to be received to ensure that there is no pending message before it proceeds to the next cycle.

Each processor should set the variable $countR$ or $countL$ to zero at each cycle before receiving any message from its two neighbors to correctly keep track of the active messages received. This requires synchronization between every processor and its two neighboring processors at each cycle.

The naive method is to synchronize all processors after they finish the current expansion and set the values $countR$ and $countL$ to zero. This global synchronization scheme (Figure 26) is simple and requires only one extra message (for $sizeSR$ or $sizeSL$) in each direction. But using barrier at each cycle can potentially increase the idle time of all processors. If work-loads for all processors are balanced, this global synchronization scheme will be perform well.

A local synchronization scheme can be used alternatively (Figure 27). In this local synchroniza-

[10] The routine *Local–Expansion* is the same as in Figure 20.

[11] As described earlier, *buffering-send* entails additional overhead for copying a message into a temporary buffer unless the message can be sent immediately. Alternatively, we could use multiple communication buffers instead of *buffering-send*.

```
1  procedure Expansion(old, new)
2      {Step 3: Communicate coalesced message}
3      call SendData(new)
4      clear buffter SRB and SLB
5      {Step 2: Local Wavefront Expansion with cycle > 0}
6      for i ← 1 to old do
7          dequeue(parent, EQ)
8          call Local-Expansion(parent, new)
9      {Step 4: Process of comunication buffers}
10     call Process-Buffer(new)


11 procedure SendData (new)
12     async-send(left processor, SLB)
13     async-send(right processor, SRB)
14     async-receive(right processor, RRB)
15     async-receive(left processor, RLB)


16 procedure Process-Buffer(new)
17     if sizeRR > 0 then
18         check if there is any pending message
19         call Process-Received-Buffer(new, RRB)
20     if sizeRL > 0 then
21         check if there is any pending message
22         call Process-Received-Buffer(new, RLB)


23 procedure Process-Received-Buffer(new, Buf)
24     while Buf ≠ ∅ do
25         get a site C and its remote parent's spin value Pspin from Buf
26         if (mark[C] ≠ cluster) and (spin[C] ≠ Pspin) then
27             r ← random(0..1)
28             if if r < p then
29                 mark[C] ← cluster
30                 spin[C] ← −spin[C]
31                 call Local-Expansion(C, new)
```

Figure 24: Pseudo code of *Expansion* for the overlapping scheme with message coalescing and using asynchronous Send/Receive(ASR-C-O) in the parallel Wolff cluster algorithm

```
1  procedure Expansion(old, new)
2      {Initialization step}
3      cycle ← cycle + 1
4      sizeSL ← 0; sizeSR ← 0
5      {Steps 2, 3 and 4: Wavefront Expansion with cycle > 0}
6      for i ← 1 to old do
7          dequeue(parent, EQ)
8          call Local-Expansion(parent, new)
9          poll(any pending active messages)
10     {Step 4: Check the end of cycle}
11     call Check-End-Of-Expansion()


12 procedure Local-Expansion(P, new)
13     for each neighbor C of P do
14         if C is a local site then
15             if (mark[C] ≠ cluster) and (spin[P] ≠ spin[C]) then
16                 r ← random(0..1)
17                 if r < p then    {p is a bond activation probability}
18                     mark[C] ← cluster
19                     spin[C] ← −spin[C]
20                     enqueue(C, EQ)
21                     new ← new + 1
22         else
23             {No Message coalescing communication: }
24             {    Handler function Handler is invoked on the receiver}
25             {    C and spin[P] are arguments passed to Handler }
26             active-message(right(or left) processor, C, spin[P], Handler)
27             sizeSR (or sizeSL) ← sizeSR (or sizeSL) + 1


28 procedure Handler(C, Pspin)
29     countR (or CountL) ← countR (or CountL) + 1
30     if (mark[C] ≠ cluster) and (spin[C] ≠ Pspin) then
31         r ← random(0..1)
32         if r < p then
33             mark[C] ← cluster
34             spin[C] ← −spin[C]
35             enqueue(C, EQ)
36             new ← new + 1
```

Figure 25: Pseudo code of *Expansion* for the overlapping scheme without message coalescing (NC-O) in the parallel Wolff cluster algorithm

```
1  procedure Check-End-Of-Expansion()
2      active-message(left processor, sizeSL, sizeRR)
3      active-message(right processor, sizeSR, sizeRL)
4      poll-while(sizeRR and sizeRL)
5      while (countR < sizeRR or countL < sizeRL) do
6          poll(any pending active messages)
```

* In this global synchronization scheme, lines 3–4 of Figure 25 should be replaced as follows:

```
    sizeSL ← 0; sizeSR ← 0
    countL ← 0; countR ← 0
    barrier()
```

Figure 26: Pseudo code of the global synchronization version of *Check-End-Of-Expansion* for the overlapping scheme without message coalescing (NC-O) in the parallel Wolff cluster algorithm

```
1  procedure Check-End-Of-Expansion()
2      endL ← false; endR ← false
3      active-message(left processor, sizeSL, sizeRR, cycle, cycleR)
4      active-message(right processor, sizeSR, sizeRL, cycle, cycleL)
5      while (endL = false) or (endR = false) do
6          poll(any pending active messages)
7          if (endL = false) and (cycleL = cycle) and (sizeRL = countL) then
8              countL ← 0
9              endL ← true
10             active-message(left processor, cycle, syncL)
11         if (endR = false) and (cycleR = cycle) and (sizeRR = countR) then
12             countR ← 0
13             endR ← true
14             active-message(right processor, cycle, syncR)
15     while (syncL ≠ cycle) or (syncR ≠ cycle) do
16         poll(any pending active messages)
```

Figure 27: Pseudo code of the local synchronization version of *Check-End-Of-Expansion* for the overlapping scheme without message coalescing (NC-O) in the parallel Wolff cluster algorithm

tion scheme, a processor may proceed to the next cycle $k+1$ without waiting for its neighboring processors to finish their current cycle k. After each processor receives all messages from its neighbor, it signals its neighbor to be allowed to proceed to the next cycle by sending another active message ($syncL$ or $syncR$). Each processor can proceed to the next cycle after it receives this signal from its neighbors (line 15–16 of Figure 27). Thus the local synchronization scheme requires two extra active messages (one is for $sizeSL$ or $sizeSR$ and the other is for $syncL$ or $syncR$) in each direction.

However, in Lee's maze-routing algorithm, the expansion of distance $k+1$ can start only after the expansion of distance k ends in order to find the shortest path. For this reason buffering the message is necessary on the receiver and at least two buffers in each direction are required. The remote expansion step can be overlapped with neither local expansion nor communication step. The handler function deposits the corresponding message in the buffer.

We compared the performance of this local synchronization scheme with that of the global synchronization scheme for both algorithm. In the Wolff cluster algorithm the local synchronization scheme gave better performance, but the global synchronization scheme worked better in the Lee's maze-routing algorithm.

Data to be sent to a processor's neighbors consists of two words (row and column coordinate of a remote site) in Lee's maze-routing algorithm, thus an active message can include the information of two remote sites instead of just one, which can improve the performance. In the Wolff cluster algorithm, data to be sent consists of three words (row and column coordinates of a remote site, and spin value of a local parent site). Since the spin value is either 1 or -1, and coordinate values are not negative, we can combine the spin value and column (or row) coordinate. Thus each active message can contain information for two remote sites, as in Lee's maze-routing algorithm.

6 Experimental Results

The parallel Wolff cluster algorithm and the parallel Lee's maze-routing algorithm in this paper have been programmed in C (hostless program). The number of processors for each of the experiments were 16 and 32, respectively. Separate sequential programs were written in C and were run on one node of the CM-5. For global communication (Steps 1 and 5 of Figure 11), CMMD synchronization and global reduction functions have been used. Strata barrier function and combine function have been used for Strata implementation. Experiments were conducted for three different lattices or grid (N=256, 512, 1024). Various partition widths, w, were used in our experiments. This was done to study the sensitivity of the performance to the partition width as well as to find the width that gave the best performance. When using δ synchronization the value of δ was chosen to be $0.1 \times (N/2)$ for the Wolff cluster algorithm and $0.2M$[12] for Lee's maze-routing algorithm [20].

For the Wolff cluster algorithm we fixed the inverse temperature β to the critical inverse

[12]If the coordinates of the source cell and the destination cell are (x_s, y_s) and (x_d, y_d), respectively, $M = |x_s - x_d| + |y_s - y_d|$ [20].

temperature β_c[13] because it generates clusters that are highly irregular in size and shape. The average execution time for one sweep was measured over 100 sweeps. A thermalization step involving 100 steps was used before taking these measurements.

As discussed earlier, the time complexity of the Wolff cluster algorithm depends completely on the size of a cluster which depends on the random numbers generated. Since each processor uses a different random seed, the bond activation probability p for a pair of sites can be different for different partition widths. In the overlapping scheme without message coalescing, the bond activation probability p for a pair of sites can be different even for the same partition width, since the packet-ordering is not guaranteed on the CM-5 [11]. Hence the scale time T_s for one sweep was used for performance comparisons:

$$T_s = \frac{T}{S} \times N^2,$$

where T=average execution time for one sweep and S = average cluster size. The results for the Wolff cluster algorithm are given in Figure 29.[14]

For Lee's maze-routing algorithm two different pre-blocked grids were used, one with 25% blocked cells and the other with 50% blocked cells. We measured the average execution time of the *wavefront expansion phase* of Figure 7 for ten instances. For each instance, source and destination cells were chosen randomly but were required to have a Manhattan distance, M, such as $0.4N \leq M \leq 0.6N$. The results are presented in Figure 30.[15]

Performance

The performance (speedup and efficiency) of five communication schemes using the best partition widths are given in Figures 31 through 34. For both applications the overlapping schemes without message coalescing using Strata and CMAML active messages gave the best and the second-best performance, respectively. This was true regardless of lattice size and number of processors for both applications and number of pre-blocked cells in Lee's maze-routing algorithm. The Strata-based implementation was marginally better than the corresponding CMAML implementation.

At the beginning of any expansion phase only the processor with the initial site is busy doing useful work. Effective parallelization of these applications require fast diffusion of the frontal expansion wave. Clearly, avoiding message coalescing as well as having a lower partition width (which is directly dependent on the startup latency cost) enhances the diffusion process and achieves better performance. However, it is interesting to compare the two paradigms for the case when message coalescing is used. This removes the diffusion effect (and the corresponding performance gain) which can be attributed to message coalescing. The experimental results show that the active messages-based implementation for the non-overlapping scheme with message coalescing outperformed the send/receive-based implementation even though the active message

[13] The critical inverse temperature β_c of the phase transition for the two-dimensional spin Model is $\frac{1}{2}\ln(\sqrt{2}+1) = 0.4406868\ldots$ [3].

[14] In the overlapping scheme without message coalescing, the local synchronization scheme has been used.

[15] In the overlapping scheme without message coalescing, the global synchronization scheme has been used.

based implementation required several global synchronizations to ensure program correctness. It also required exchanging message sizes. These results suggest that the handshaking costs along with the buffer management costs for general send/receive implementation are prohibitive enough that active messages-based implementations, which may require a great amount of synchronization and handshaking, can still provide a better performance.

The performance of the overlapping scheme with message coalescing and using asynchronous send/receive is not as good as the non-overlapping scheme with message coalescing and using synchronous send/receive. The extra overhead of asynchronous communication is not offset by the overlapped computation.[16]

For Lee's maze routing algorithm, better efficiency is achieved for the 25% pre-blocked grid than in the 50% pre-blocked grid. This is expected, as lower blocking implies better diffusion and better parallelization.

Effect of Partition Width

For both applications the performance sensitivity to the partition width and the best partition width has a different behavior for different communication schemes. For larger lattices the performance is more sensitive to the the choice of the partition width.

Amount of Communication Generated

We measured the average number of CMAML active messages at each cycle and the average number of cycles for the overlapping scheme without message coalescing with the best partition width w. Figures 35 and 36 show the average number of active messages sent by each node at each cycle and the total number of cycles in both algorithms. The maximum among all the processors is chosen as each node sends different number of active messages. Since the best partition width, w, for other schemes is not smaller and the amount of communication decreases as partition width increases, the values in Figures 35 and 36 represent an upper bound on communication.

The average number of active messages (sent to the left or right neighbor processor) at each cycle using the best partition width is less than 6 in the Wolff cluster algorithm and is less than 4 in the Lee's maze-routing algorithm (Figures 35 and 36). In the Wolff cluster algorithm two extra messages are required in each direction for the local synchrnization scheme, and in the Lee's maze-routing algorithm just one extra message is required in each direction for the global synchrnization scheme. Thus the average amount of communication at each cycle, excluding the extra messages for coordination, does not exceed 16 words (or 8 sites) and 12 words (or 6 sites) in each direction in both algorithms, respectively.

[16] As described earlier, the network interface has no DMA and accepts only a five-words-per-packet-long message (one word for the message head and four words for data). If a message is larger than four words, a processor should first packetize the message and then transfer the data to the network interface packet by packet. Overlapping communication (actual transmission without the idle time resulting from handshaking) with computation is limited on the CM-5 due to the low network capacity and network latency [6].

Scheme	Size of code	Ease of programming	Performance
SSR-C-NO	2	1	3
AMP-C-NO	3	4	2
ASR-C-O	4	2	4
AM-NC-O	1	3	1

Smaller values imply a smaller sized code, relative ease of programming, and better performance, respectively.

Figure 28: Relative comparison of four communication schemes

Local versus Global Synchronization

In the overlapping scheme without message coalescing, we compared the performance of the local synchronization scheme with that of the global synchronization scheme in both algorithms (Figures 37 and 38). In the Wolff cluster algorithm the local synchronization scheme gave better performances, but the global synchronization scheme worked better in the Lee's maze-routing algorithm. The local synchronization scheme in the Lee's maze-routing algorithm requires not only to send two extra messages, but also to manage multiple buffers on a receiver. These overheads are not offset by the potential decrease in idle time.

7 Conclusions

We have presented experimental results for several communication schemes for the parallel Wolff cluster algorithm and the parallel Lee's maze-routing algorithm on the CM-5. For both applications the *overlapping scheme without message coalescing using Strata and CMAML active messages* gave the best and the second-best performance, respectively. The performance difference between these two schemes was marginal. The active messages-based implementations outperformed send/receive message-passing due to their smaller startup cost as compared to send/receive. The time improvements required for using active messages was a factor of two to three over those required for using the send/receive paradigm. The *non-overlapping scheme with message coalescing and using active messages-PUT* gave a better performance than any scheme using the send/receive message paradigm.

Figure 28 shows the comparison of the five communication schemes presented in this paper for three features: size of code, ease of programming, and performance.[17] The AM-NC-O scheme has the smallest program because no extra work is required for message coalescing and the three processing steps (Local Expansion, Communication, and Remote Expansion) can be combined. The ASR-C-O scheme has the largest program. Extra work is required in the Remote Expansion step to overlap the Local Expansion step with the Communication step. The AMP-C-NO scheme using active messages-PUT requires extra coordination between a sender and a receiver, hence

[17]We have omitted the overlapping scheme without message coalescing that uses Strata active messages, since this scheme is the same as that using CMAML active messages except for the functions used.

the larger size than the SSR-C-NO scheme.

We found active messages-based programming to be much more difficult than programming using send/receive.[18] Programming for the AMP-C-NO scheme was the most difficult due to the explicit coordination between a sender and a receiver.

[18]This is also due to the fact that we have had much more experience programming with the send/receive paradigm, and this work was one of the first applications we wrote using active messages.

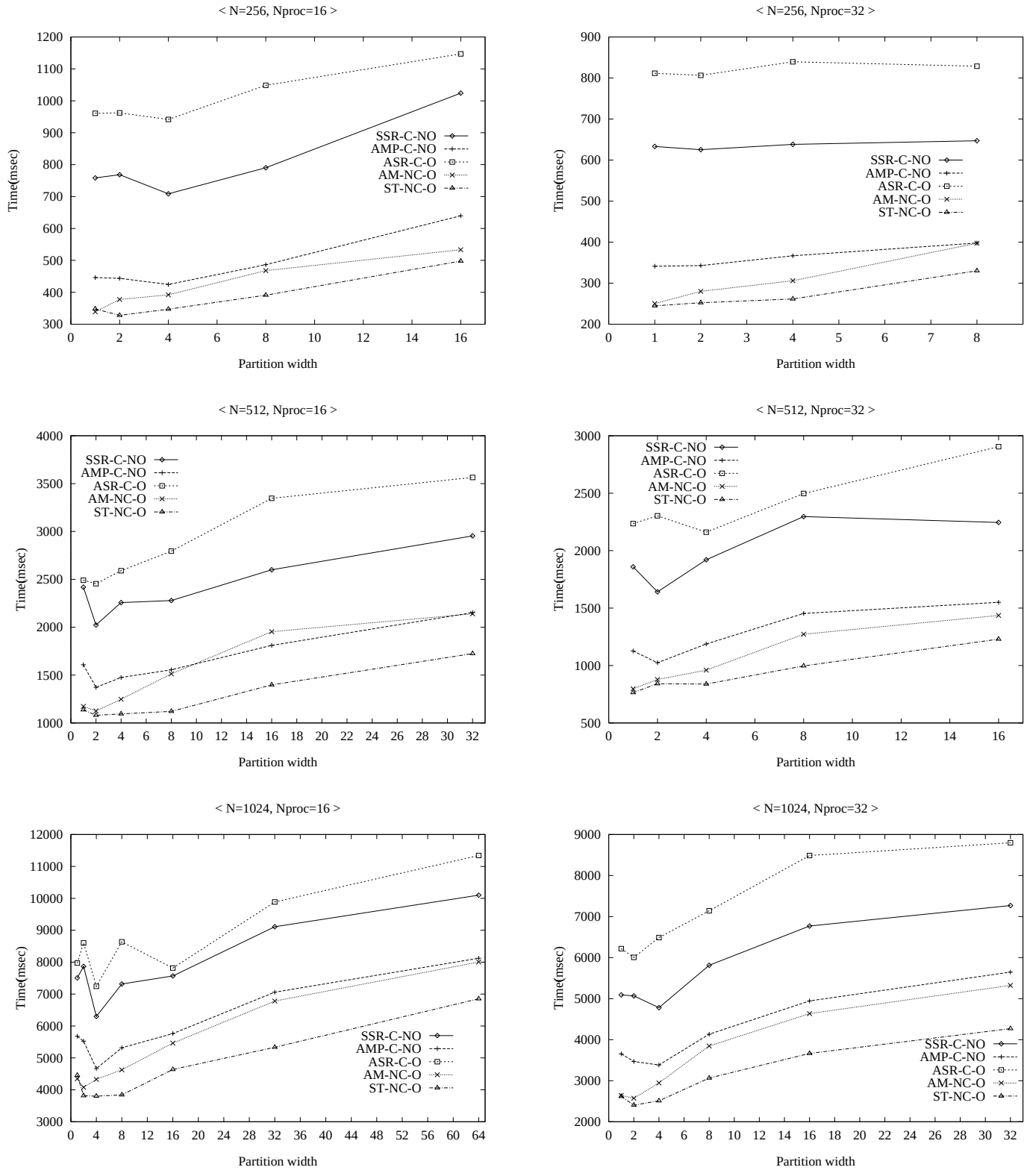


Figure 29: Comparison of scale time (msec) for one sweep with five communication schemes in the parallel Wolff cluster algorithm

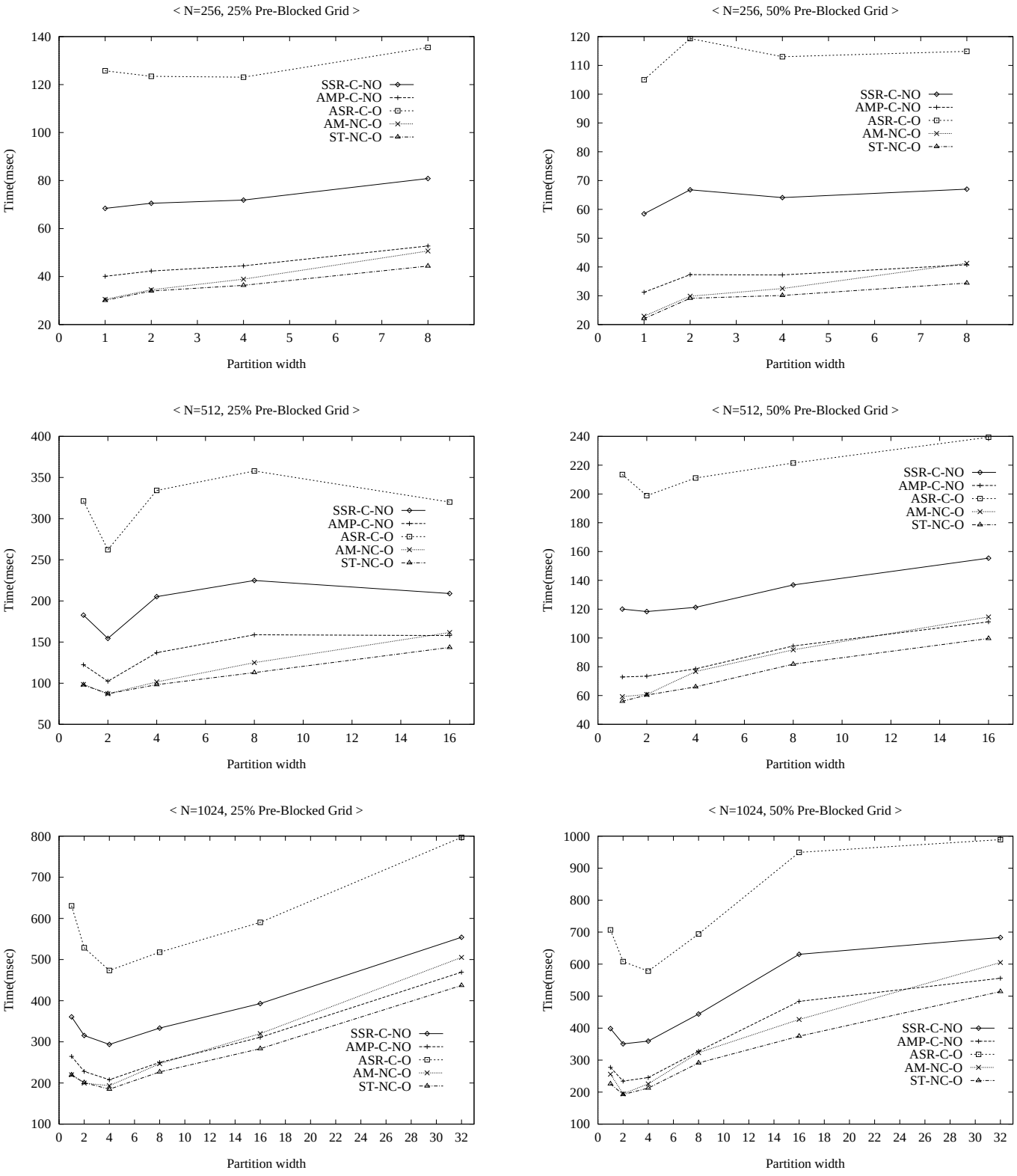


Figure 30: Comparison of execution time (msec) for one iteration with five communication schemes in the parallel Lee's maze-routing algorithm ($Nproc = 32$)

Lattice size(N^2)	256^2	512^2	1024^2
Sequential time	2124.664	8620.074	35314.967

$Nproc = 16$

Lattice size(N^2)	256^2		512^2		1024^2	
	Speed-up	Efficiency	Speed-up	Efficiency	Speed-up	Efficiency
SSR-C-NO	2.988	0.187	4.261	0.266	5.607	0.350
AMP-C-NO	5.004	0.313	6.281	0.393	7.568	0.473
ASR-C-O	2.256	0.141	3.512	0.219	4.875	0.305
AM-NC-O	6.270	0.392	7.655	0.478	8.659	0.541
ST-NC-O	6.481	0.405	7.978	0.499	9.290	0.581

$Nproc = 32$

Lattice size(N^2)	256^2		512^2		1024^2	
	Speed-up	Efficiency	Speed-up	Efficiency	Speed-up	Efficiency
SSR-C-NO	3.398	0.106	5.252	0.164	7.388	0.231
AMP-C-NO	6.226	0.195	8.414	0.263	10.438	0.326
ASR-C-O	2.635	0.082	3.990	0.125	5.880	0.184
AM-NC-O	8.494	0.265	10.806	0.338	13.721	0.429
ST-NC-O	8.680	0.271	11.282	0.353	14.680	0.459

Figure 31: Scale time (msec) for one sweep in the sequential Wolff cluster algorithm and evaluation of the parallel algorithm

25% Pre-Blocked Grid

Grid size(N^2)	256^2	512^2	1024^2
Sequential time	178.607	753.025	2532.771

Grid size(N^2)	256^2		512^2		1024^2	
	Speed-up	Efficiency	Speed-up	Efficiency	Speed-up	Efficiency
SSR-C-NO	2.610	0.082	4.875	0.152	8.624	0.270
AMP-C-NO	4.452	0.139	7.358	0.230	12.188	0.381
ASR-C-O	1.377	0.043	2.872	0.090	5.350	0.167
AM-NC-O	5.861	0.183	8.641	0.270	13.093	0.409
ST-NC-O	5.940	0.186	8.639	0.270	13.699	0.428

50% Pre-Blocked Grid

Grid size(N^2)	256^2	512^2	1024^2
Sequential time	131.068	470.749	2018.324

Grid size(N^2)	256^2		512^2		1024^2	
	Speed-up	Efficiency	Speed-up	Efficiency	Speed-up	Efficiency
SSR-C-NO	2.242	0.070	3.978	0.124	5.754	0.180
AMP-C-NO	4.198	0.131	6.457	0.202	8.611	0.269
ASR-C-O	1.248	0.039	2.368	0.074	3.491	0.109
AM-NC-O	5.720	0.179	7.945	0.248	10.371	0.324
ST-NC-O	5.948	0.186	8.403	0.263	10.480	0.327

Figure 32: Execution time (msec) for one sweep in the sequential Lee's maze-routing algorithm and evaluation of the parallel algorithm

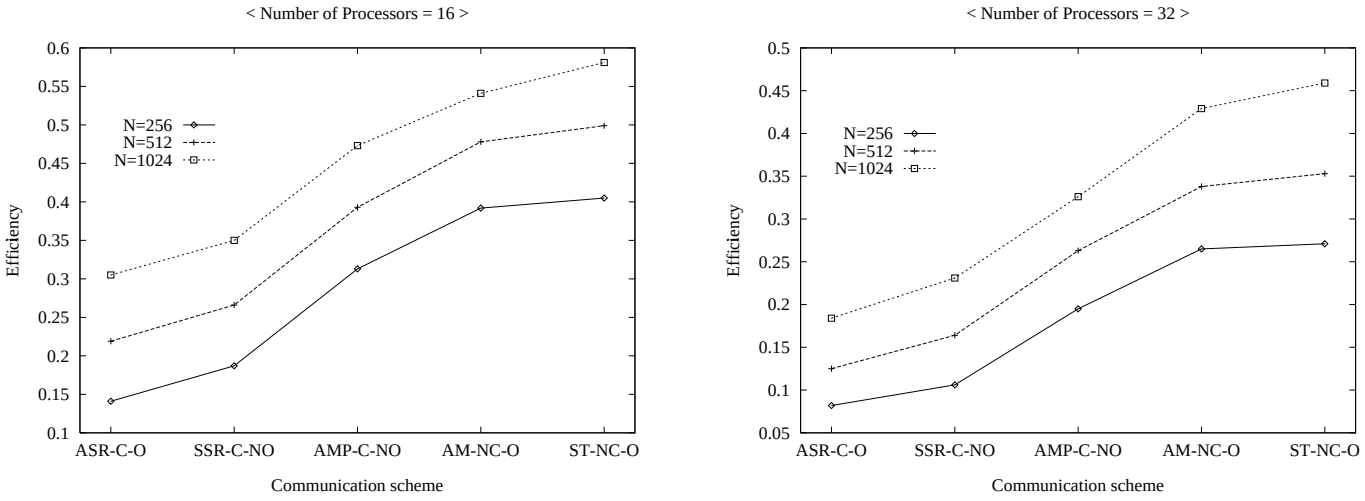


Figure 33: Efficiency of parallel Wolff cluster algorithm with five communication schemes

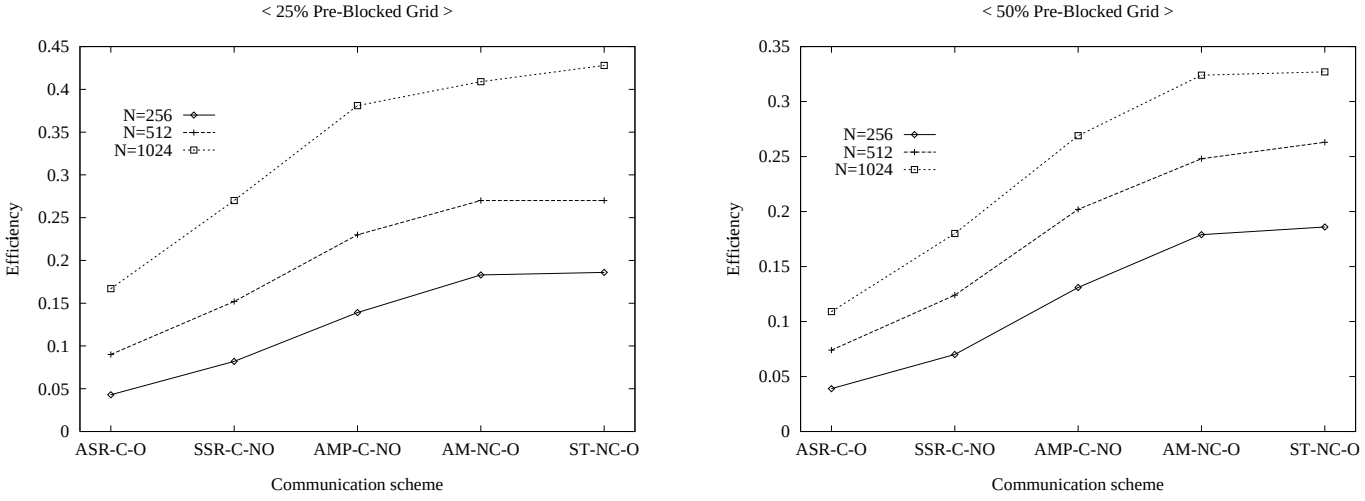


Figure 34: Efficiency of parallel Lee's maze routing algorithm with five communication schemes

Average number of active messages at each cycle

Lattice size(N^2)	256^2		512^2		1024^2	
	to left	to right	to left	to right	to left	to right
$Nproc = 16$	4.099	4.099	3.748	3.751	5.032	5.035
$Nproc = 32$	3.295	3.295	3.710	3.710	3.765	3.766

Total number of cycles

Lattice size(N^2)	256^2	512^2	1024^2
$Nproc = 16$	307.20	573.25	1326.00
$Nproc = 32$	347.40	581.75	1286.22

Figure 35: Average number of active messages sent by each node at each cycle and average number of cycles in the Wolff cluster algorithm using overlapping scheme without message coalescing

Average number of active messages at each cycle

Grid size(N^2)	256^2		512^2		1024^2	
	to left	to right	to left	to right	to left	to right
25% *Pre-Blocked Grid*	3.117	3.117	3.330	3.331	3.041	3.042
50% *Pre-Blocked Grid*	2.421	2.421	3.836	3.836	3.637	3.636

Total number of cycles

Lattice size(N^2)	256^2	512^2	1024^2
25% *Pre-Blocked Grid*	193.7	331.5	537.8
50% *Pre-Blocked Grid*	184.7	311.5	706.7

Figure 36: Average number of active messages sent by each node at each cycle and average number of cycles in the Lee's maze-routing algorithm using overlapping scheme without message coalescing

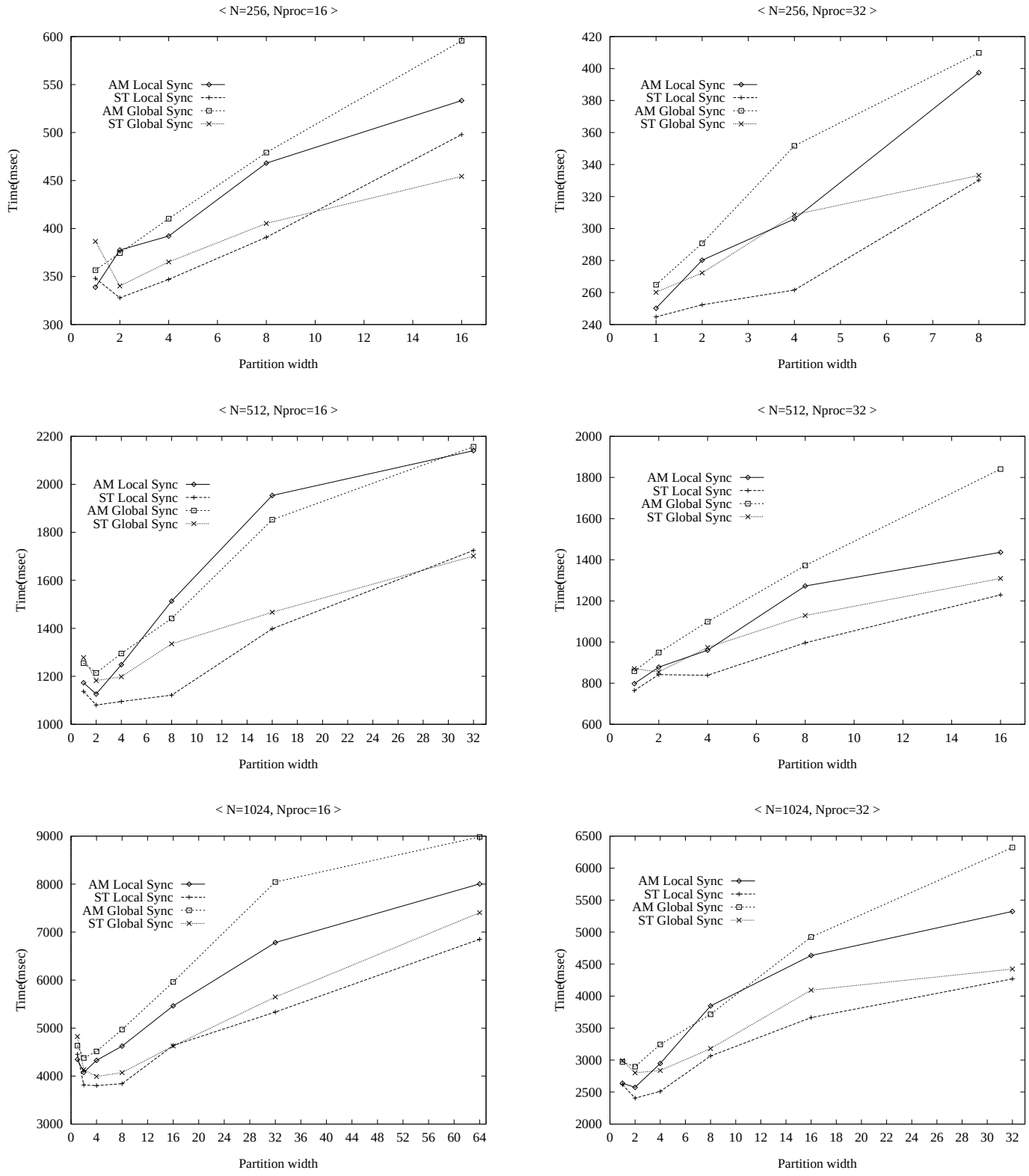


Figure 37: Comparison of scale time (msec) for one sweep with local synchronization scheme and global synchronization scheme in the overlapping scheme without message coalescing in the parallel Wolff cluster algorithm

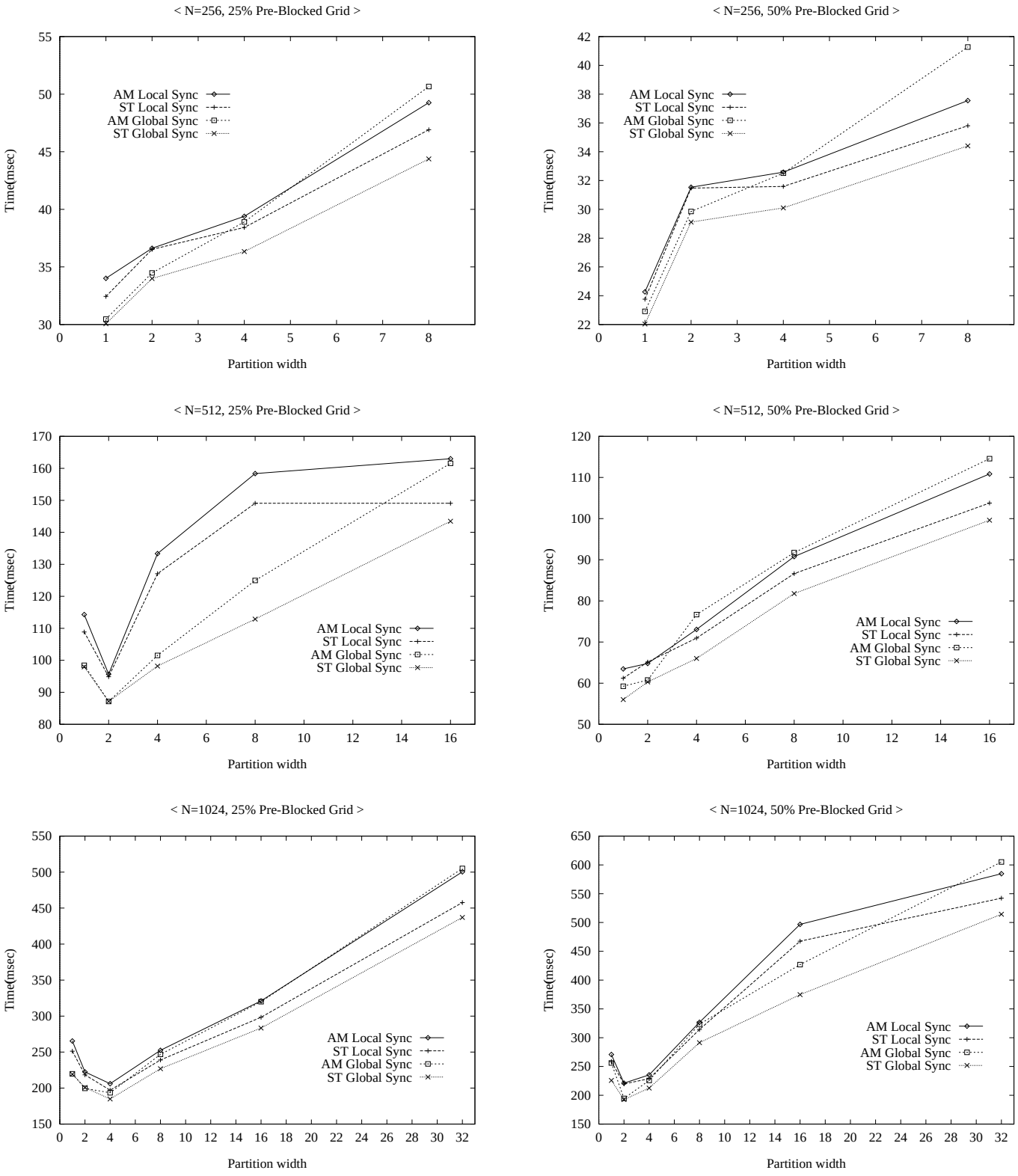


Figure 38: Comparison of execution time (msec) for one iteration with local synchronization scheme and global synchronization scheme in the overlapping scheme without message coalescing in the parallel Lee's maze-routing algorithm ($Nproc = 32$)

References

[1] C. Aykanat and T. Kurc. Efficient parallel maze routing algorithm on a hypercube multicomputer. In *Proc. 1991 Intl. Conf. on Parallel Processing, Vol 3*, pages 224–227, 1991.

[2] Seungjo Bae, Sung-Hoon Ko, and Paul Coddington. Parallel Wolff Cluster Algorithm. Technical Report SCCS-619, Northeast Parallel Architectures Center, Syracuse University, March 1994.

[3] Clive F. Baillie and Paul D. Coddington. Cluster Identification Algorithms for Spin Models—Sequential and Parallel. Technical Report C^3P-855, Caltech, June 1990.

[4] G. Brassard and P. Bratley. *Algorithmics: Theory and Practice*. Prentice Hall, 1987.

[5] E. A. Brewer and R. D. Blumofe. Strata: A Multi-Layer Communication Library. Technical Report (to appear), MIT Laboratory for Computer Science, February 1994.

[6] E. A. Brewer and B. C. Kuszmaul. How to get good performance from the CM-5 data network. In *International Parallel Processing Symposium*, 1994.

[7] F. T. Chong, S. D. Sharma, E. A. Brewer, and J. Saltz. Multiprocessor Runtime Support for Fine-Grained, Irregular DAGs. Submitted for publication, 1994.

[8] Paul D. Coddington. Lecture note. Lecture notes for the course CPS713 at Syracuse University, Spring 1993.

[9] Thinking Machines Corporation. *CM-5 Technical Summary*, November 1993.

[10] Thinking Machines Corporation. *CMMD version 3.0 Reference manual*, May 1993.

[11] T. Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a mechanism for integrated communication and computation. In *Proc. of ISCA 1992*, pages 256–266, 1992.

[12] Y. Fang, I. Yen, and R. Dubash. Improving the performance of Lee's maze routing algorithm on parallel computers via semi-dynamic mapping strategies. Technical Report CPS-93-35, Michigan State University, December 1993.

[13] D. W. Heermann and A. N. Burki. *Parallel Algorithms in Computational Science*. Springer-Verlag, 1990.

[14] V. Kurmar, A. Grama, G. Karypis, and A. Gupta. *Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings Publishing Company, Inc., 1994.

[15] C. Y. Lee. An algorithm for path connections and its applications. *IRE Trans, Electronic Computers*, EC–10:346–365, September 1961.

[16] R. H. Swendsen and J. S. Wang. Nonuniversal critical dynamics in Monte Carlo simulation. *Phys. Rev. Lett.*, 58(2):86, 1987.

[17] P. Tamayo, R. C. Brower, and W. Klein. Single-Cluster Monte Carlo dynamics for the Ising model. *J. Statis. Phys.*, 58(5):1083, 1990.

[18] J. S. Wang and R. H. Swendsen. Cluster Monte Carlo algorithm. *Physica A*, 167:565, 1990.

[19] U. Wolff. Lattice field theory as a percolation process. *Phys. Rev. Lett.*, 62(15):361, 1988.

[20] Y. Won and S. Sahni. Maze routing on a hypercube multiprocessors computers. In *Proceedings of Intrl. Conf. on Parallel Processing, St. Charles*, pages 630–637, August 1987.