Automatic Parallel Program Conversion from Shared-Memory to Message-Passing

Hsien–Hsin Lee Edward S. Davidson

Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, Michigan 48109
{linear,davidson}@eecs.umich.edu

ABSTRACT

It is an elaborate work to parallelize an application on message-passing machines since there is few good software or compilers supported for dealing with the interprocessor communication. Accordingly, it is inefficient for programmers to understand the communication behavior and explicitly specify them in the code by hand. In this paper, three schemes are presented for alleviating this time-consuming and error-prone task. Scheme A and B are two obvious methods to implement the communication code while the Scheme C is a systematic methodology for generating a functional and efficient message-passing code in an automatic manner.

1 Introduction

Message-passing and shared-memory are two predominant communication models supported in multiprocessor systems today. Shared memory machines provide programmers a shared address space supported by the machine's hardware, protocols and operating system. Thus shared data can be managed and moved among the processing nodes automatically when parallel programs are executed. On the other hand, message passing machines share common data through explicit message transactions which need to be specified by programmers in their high level application codes.

In terms of communication cost, ping-pong effect, cache coherence, false sharing and superfluity in transferred memory blocks often make shared-memory systems more inefficient than message-passing systems. Many people believe that better overall performance can be achieved by using message-passing communication substrates. However, in message-passing systems, programmers must understand when and where each data element is updated and needs to be sent to those nodes that own stale data copies in order to achieve a consistent execution of a program. This work, which must presently all be done by hand, is very elaborate and prone to error. Thus, porting an application from a shared-memory platform to a message-passing platform is a difficult, time-consuming, and costly task for programmers, especially, when an efficient message-passing program is required.

```
 DO PID =1, num_of_procs
  DO k =K_BEGIN(PID), K_END(PID)
   DO IL=1, NLAY
    DO i=1, SEG(PID,k)
     len=size*(J_END(PID,k,i)-J_BEG(PID,k,i)+1)
      call MP_BCAST(V(J_BEG(PID,k,i),k,IL,1), len,
&                    (PID-1), allgrp)
    END DO
   END DO
  END DO
 END DO
```

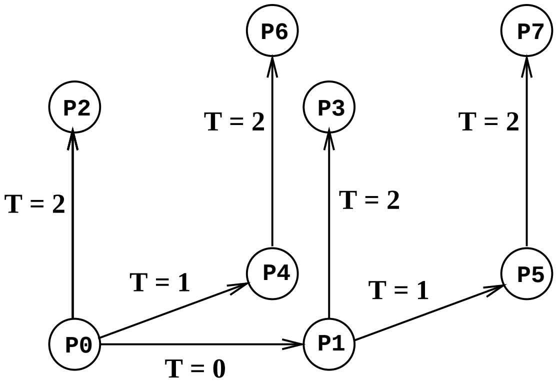


Figure 1: Broadcasting algorithm and recursive doubling algorithm

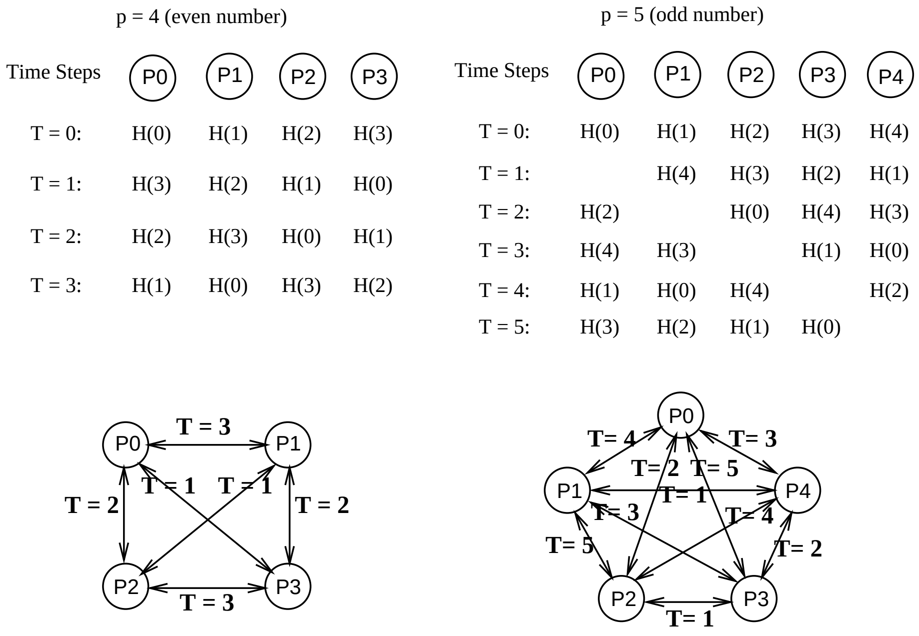
In this research, we discuss three approaches including a systematic methodology to parallelizing an application (Indian Ocean circulation simulator) on a pure message-passing machine (IBM SP2), based on a parallelized version of shared-memory code on a shared-memory machine (KSR2). Via an automated manner, the systematic method is proposed to simplify the task of specifying the routines for sending and receiving messages, where they need to be used and what data need to be transferred. Based on this approach, programmers can generate a functional and efficient message-passing code automatically from an analysis of parallel trace results which are collected by running the application on a shared-memory system.

The three approaches are presented in Section 2, 3 and 4, respectively. Some of our preliminary results on the IBM SP2 and a comparison with the KSR2 are reported in Section 5. We conclude and discuss our future work in Section 6.

2 Scheme A: Broadcast all updated data to all other processors

Since there is typically no information present in a domain decomposition file that identifies shared vs. local data, it is quite difficult to determine the appropriate data transactions between processors for implementing a message-passing code. In this section, a very straightforward method is proposed for implementing message-passing code. First of all, we analyze and identify the shared data arrays in the application. Due to the unknown details regarding exactly how these arrays are shared between processors, we take a simple but wasteful approach in Scheme A. Whenever a shared array is updated by a particular processor, the values will be sent to every other processor by inserting message-passing calls at the original synchronization barriers of the shared-memory code. This approach ensures the functional correctness of the message-passing code by assuming that each processor has the most up-to-date copy of each shared array. Nevertheless, this programming method will introduce a lot of communication overhead due to redundant messages.

Two communication schemes described in the following were used in our current implementation of this approach.



p = 4 (even number)

Time Steps	P0	P1	P2	P3
T = 0:	H(0)	H(1)	H(2)	H(3)
T = 1:	H(3)	H(2)	H(1)	H(0)
T = 2:	H(2)	H(3)	H(0)	H(1)
T = 3:	H(1)	H(0)	H(3)	H(2)

p = 5 (odd number)

Time Steps	P0	P1	P2	P3	P4
T = 0:	H(0)	H(1)	H(2)	H(3)	H(4)
T = 1:		H(4)	H(3)	H(2)	H(1)
T = 2:	H(2)		H(0)	H(4)	H(3)
T = 3:	H(4)	H(3)		H(1)	H(0)
T = 4:	H(1)	H(0)	H(4)		H(2)
T = 5:	H(3)	H(2)	H(1)	H(0)	

Figure 2: Examples of pairwise message exchange

2.1 Broadcasting

In this scheme, many-to-many communications are performed by the way that every processor broadcasts to every other processor its updates on decomposed shared arrays. Since each processor knows the arrays' partition index domains of all others, thus the receiver end will put their received data into the correct arrays' index range after the communication finishes. The algorithm of our implementation is shown in figure 1.

From the experiments suggested in [1], the collective routine, *MP_BCAST*, provided in the MPL message-passing library of the IBM SP2 for broadcasting is used. This routine employs a recursive doubling (binary tree)algorithm. The graph in figure 1 illustrates the data communication paths and the corresponding time steps of the recursive doubling algorithm, which takes $O(\lg p)$ time to broadcast a message from a particular processor $P0$ to the other $(p-1)$ processors. The total time complexity of this scheme for each of p processors to broadcast to all others is $O(p \lg p)$.

2.2 Pairwise Message Exchange

In this section, pairwise message exchange, an efficient way to implement many-to-many communications is demonstrated. For communication with an even number of processors, for each of several steps the processors can be divided into disjoint pairs and each pair can exchange their updated data with one another. The number of steps we need in this case for p processors will be $\frac{C_2^p}{(p/2)}$, that is, $p-1$ steps. For communication with an odd number of processors, at each step one processor will be left out while the other processors exchange their data. Hence, after p steps, each processor will have received the updated data from all others. The time complexity of pairwise message exchange for p processors is thus $O(p)$, which is a lower order of complexity

```
pair = ((num_of_procs .eq.  even) ?  num_of_procs :  num_of_procs - 1)
/* Packing the sender's message */
length_out = accumulate(PID, array1, array2, ..)
send_buffer = pack_data(array1, array2, ..)
/* Starting pair the processors and exchange their messages */
do i = 0, pair - 1
  if (i .ne.  PID .or.  pair .eq.  num_of_procs) then
    if (PID .eq.  pair) then
      buddy_PID = i
    else if (PID .eq.  i) then
      buddy_PID = pair
    else
      buddy_PID = mod(pair+2*i-PID, pair)
    end if
    /* Recognize the number of bytes to be received from its buddy proc */
    length_in = accumulate(buddy_PID, array1, array2, ..)
    /* Pairwise message exchange */
    call MP_BSENDRECV(send_buffer, length_out, (buddy_PID-1),
       mesg_type, recv_buffer, length_in, (buddy_PID-1), NBYTES)
    /* Unpacking the receiver's messages */
    unpack(buddy_PID, recv_buffer, array1, array2, ..)
  end if
end do
```

Figure 3: Algorithm for pairwise message exchange

than the broadcasting algorithm proposed in the previous section. Regarding exchanging of data between two processors, the IBM Message Passing Library (MPL) [3] provides a library routine called *MP_BSENDRECV* which effectively accomplishes a pairwise exchange.

Two examples are illustrated in figure 2. At the beginning, each processor p owns a locally updated partial array $H(p)$. Data will be exchanged in each step based on the above description. In 3 steps for $p = 4$ (a case with an even number of processors) and 5 for $p = 5$ (a case with an odd number of processors), each processor will have exchanged all its locally updated data with all the others. The detailed algorithm is shown in figure 3.

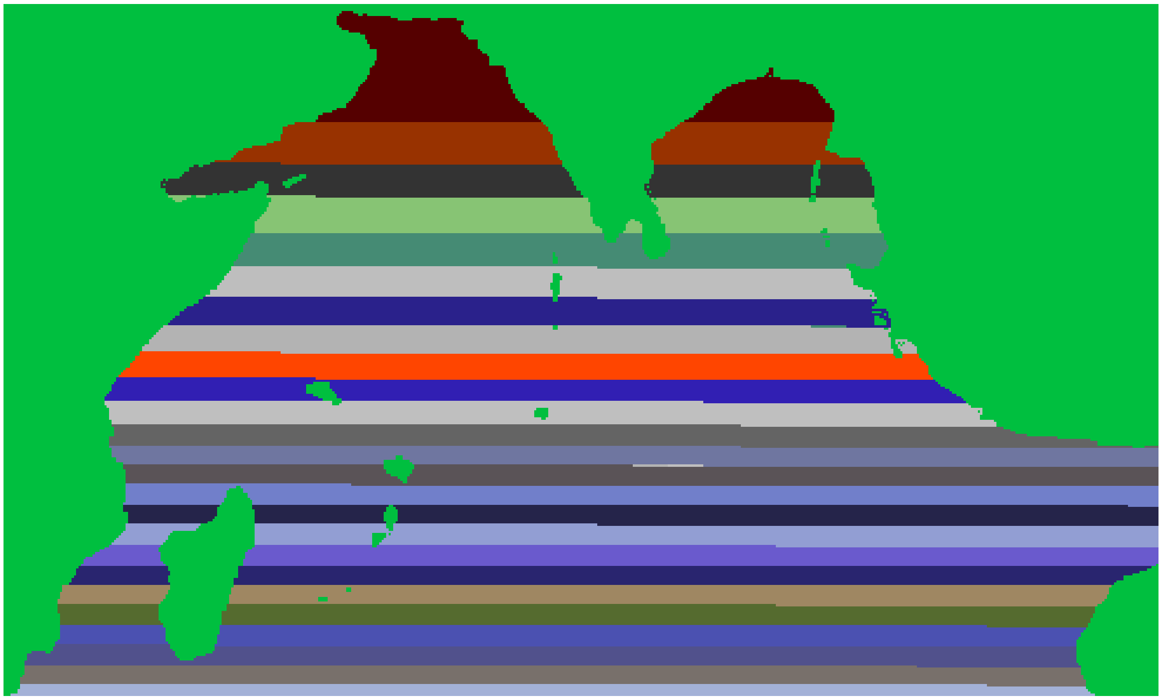


Figure 4: Indian ocean partition for 25 processors

3 Scheme B: Send updated arrays only to neighboring processors when possible

Without doubt too much redundant information is communicated among the processors in Scheme A for almost any application, i.e. unless every processor performs updates and all processors need all the updated data. In many parallel applications, a processor shares only those data elements lying on the boundaries of its sub-domain and to share these only with the neighbors that share the relevant boundary. For example, as shown in figure 4, one data decomposition of an Indian Ocean circulation code uses a horizontal partitioning. Each processor has one neighbor processor sharing each of two boundaries. Therefore, each processor can pass the locally updated shared arrays only to the two neighbor processors instead of passing them to all of the other processors. However, before using this method, it must be guaranteed that each array to be communicated only with neighbor processors are needed only by those neighbors that share a boundary with the transmitting processor; otherwise, the array is broadcast to all of the other processors. To make sure that this condition is satisfied, in our current implementation, we identified the array-sharing status using the analysis of Scheme C which is discussed in the next section.

To determine the neighbors of a particular processor, after the domain decomposition file is read in, we build the neighbors list for each processor by checking the ownership (a processor) of the elements adjacent to each data element that is allocated to a processor as shown in figure 5. A processor's updated shared data will merely be multi-casted to the neighbors identified in its list so as to eliminate some of the redundant communication.

4 Scheme C: Send each updated array element only to processors that use it

In Scheme C, a systematic and automatic methodology is presented to assist programmers in writing an efficient message-passing code based on an existing shared-memory code.

Distributed shared-memory (DSM) machines are built on top of an underlying message-passing substrate [4]. In message-passing multicomputers, the shared data are sent or received by accessing this network (substrate) directly. The messages received from the network are put into buffers which will be accessed later by the processor. Shared-memory multiprocessors, on the other hand, provide an enhanced interface between the processors and the network which provides globally shared addresses of the desired data and maintains system-wide data coherence. This enhanced interface provides a single system-wide virtual memory address space to programmers. Based on this concept, we developed a methodology that will analyze a shared-memory code to assist us in writing a functional and efficient code that will run on message-passing machines. This methodology can simplify the error-prone task of specifying the explicit message-passing routines and can eliminate a large amount of the messages communicated on the network when using Scheme A or Scheme B.

4.1 Overview of the Methodology

The block diagram of this methodology is shown in figure 6. This diagram can be divided into five major phases. The output of each phase can be produced by a specific automatic tool that we have developed.

For writing a message-passing program, the most critical and laborious work is to identify the interprocessor data movements and translate these movements into explicit messages. Our basic notion in this approach is to obtain this information directly from the execution of the shared-memory program. In the first phase of this scheme, the shared-memory code is instrumented for generating a parallel trace by inserting a call to trace routines at points in the call where memory accesses or synchronizations occur. During the execution of the instrumented program, a trace is generated for each participating processor. In our experiments, a parallel tracing tool, *K-Tracer* [2], was developed and used on our host shared-memory machine, the KSR2. In the second phase, these parallel traces are split into several smaller parallel traces based on several disjoint ranges of the address space. This step provides more manageable files and improves the performance in later phases. The split parallel traces are independent of one another, thus they can be processed concurrently in the follow-up phases.

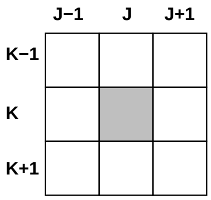


Figure 5: Check eight neighbors for an array with dimension (J,K)

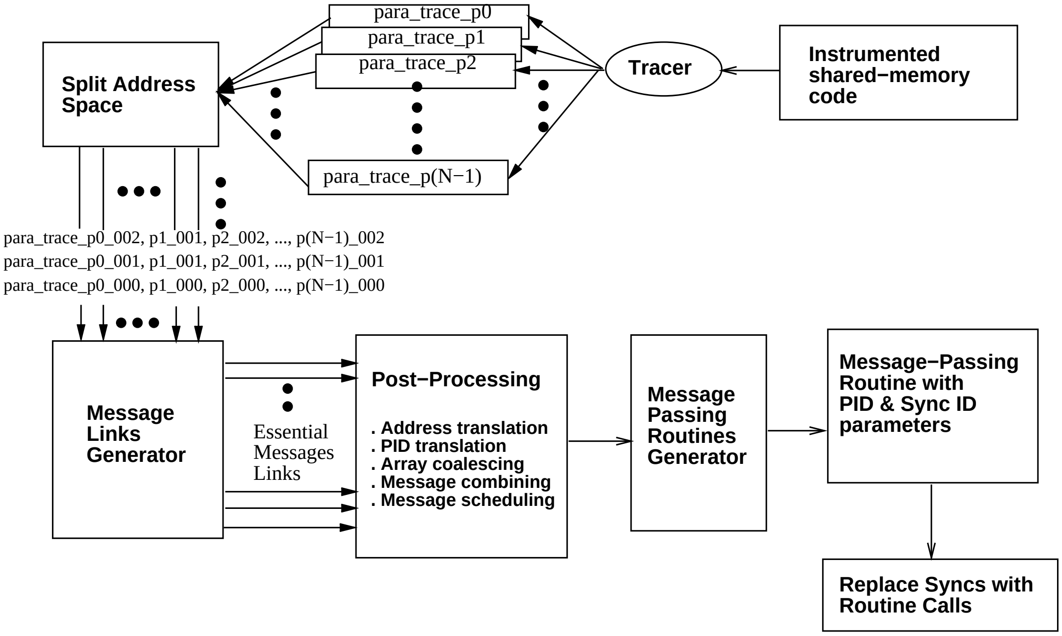

para_trace_p0
para_trace_p1
para_trace_p2
para_trace_p(N−1)
Tracer
Instrumented shared−memory code
Split Address Space
para_trace_p0_002, p1_001, p2_002, ..., p(N−1)_002
para_trace_p0_001, p1_001, p2_001, ..., p(N−1)_001
para_trace_p0_000, p1_000, p2_000, ..., p(N−1)_000
Message Links Generator
Essential Messages Links
Post−Processing
. Address translation
. PID translation
. Array coalescing
. Message combining
. Message scheduling
Message Passing Routines Generator
Message−Passing Routine with PID & Sync ID parameters
Replace Syncs with Routine Calls

Figure 6: Overview of the automated scheme

Through the phases of *Message Links Generator*, *Post Processing* and *Message Passing Routines Generator*, the essential messages are organized in one message-passing routine for our final message-passing code. Then synchronizations in the parallel code are replaced with calls to this routine. The detailed mechanisms of the method and tools are described in the following sections.

4.2 Generating Message Links

A tool called *Message Links Generator* was developed for the purpose of generating the essential message links. This tool reads in each of the split parallel traces and generate essential message links in the format shown in figure 7. Each message link consists of a memory address and a message tag. A message tag is composed of four fields, source processor ID (SP), destination processor ID

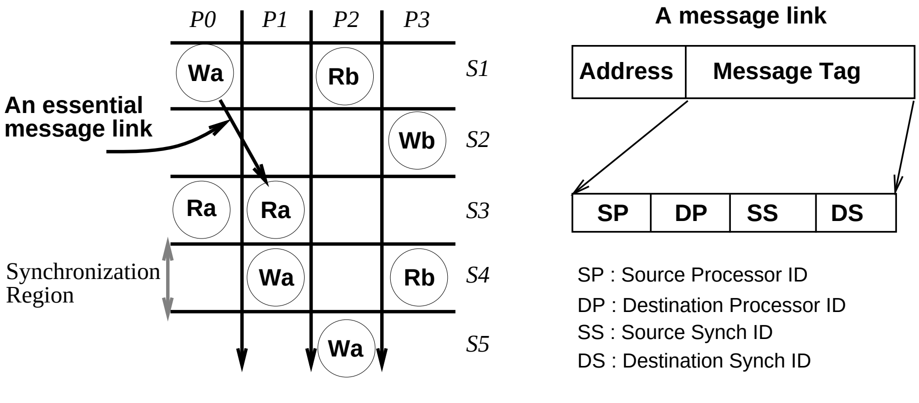


Figure 7: Essential Message and Message Link Format

(DP), source synchronization region ID (SS) and destination synchronization region ID (DS), where ID stands for identification number. Processor IDs indicate which processor the message is from and where the message should be sent. Synchronization region IDs provide us the sequential timing constraints that are used in message scheduling.

An essential message link is defined as a data transaction path which is required between two processors each time that a true interprocessor data dependency exists between them. Anti-dependency and output-dependency between processors, which cause problems and have to be considered previously during program parallelization, need not be considered here. Because inter-processor dependencies only exist between distinct synchronization regions, if one processor writes a data element, then that element can be used in the same synchronization region only by the same processor that wrote the element. It can be used by other processors only in subsequent synchronization regions until some processor writes that element again. If the data is read by other processors in subsequent synchronization regions, then to maintain data consistency a message link has to be built from the writing processor to those processors that read the data. Such a message link is called an *essential* message link. Let a data read and a data write be represented as $R(a, P, S)$ and $W(a, P, S)$, respectively, where a stands for the name of the data element, P is the processor ID and S is the synchronization region ID. A message link is created for each WR pair that share the same a where the R occurs after the W and before the next W, where the order is determined by the S field. A message link is denoted as $Mesg(a, SP, DP, SS, DS)$, where SP and SS are P and S from W and DP and DS are from R.

For example, in figure 7, variables a and b are read(R) and written(W) by four processors in five distinct synchronization regions. The only interprocessor communication takes place when there is a true interprocessor dependency, i.e. only between $W(a, P0, S1)$ and $R(a, P1, S3)$. A message, $Mesg(a, P0, P1, S1, S3)$, needs to be generated for this dependency. It is not necessary to have message links built between $W(a, P0, S1)$ and $W(a, P2, S5)$, or between $R(b, P2, S1)$ and $W(b, P3, S2)$ although there exists output-dependency and anti-dependency in these two cases, respectively.

4.3 Post Processing

After all the essential message links are built by the previous phase, a few post-processing techniques are applied to these messages. Basically, the following procedures are considered in our current implementation.

- Address translation
- Processor ID translation
- Array coalescing
- Message combining
- Message scheduling

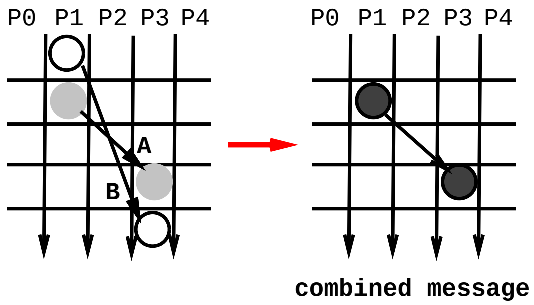


Figure 8: Example of combining messages with different synchronization ranges

4.3.1 Address and Processor ID Translation

Since the message links are generated from the parallel traces, the memory addresses are numerically resolved virtual addresses. For writing the message-passing routines, these numerical addresses have to be translated into array element names. *Address translation* translates these memory addresses from numerical format into array names with the corresponding indices. *Processor ID translation* translates the physical processor ID into a symbolic processor ID which will be used as a symbolic source or destination in the message-passing program. The address and processor ID mapping information are collected by $K - Tracer$ during the run of the instrumented shared-memory code.

4.3.2 Array Coalescing and Messages Combining

Variables with contiguous memory addresses that are communicated from the same source to the same destination and have the same synchronization region IDs can be combined into larger messages, this technique is called *array coalescing*. Due to the large overhead incurred by the initialization effect for each message, a larger combined message is preferable to several smaller messages.

A more aggressive optimization scheme is to combine non-contiguous addresses into larger messages. The message combining approach is combine messages with the same SP and DP whose synchronization ranges (from SS up to, but not including DS) overlap. For example, assume that SP(k), DP(k), SS(k) and DS(k) represent variable k's source PID, destination PID, source synch ID and destination synch ID, respectively. Then message A and message B can be combined if *i)* DP(A)=DP(B) and SP(A)=SP(B) and *ii)* DS(A) $>$ SS(B) and DS(B) $>$ SS(A) are both satisfied. An example is shown in figure 8.

Another effective way to reduce the number of messages is re-routing the message path through intermediate processors. Figure 9 gives an example of a sample data distribution and the results after our tools are applied on it. Figure 9(a) is the data distribution obtained from the parallel traces. Each circle accounts for a data reference, either a read or write. Each vertical band is the trace for a particular processor while each horizontal band is for a synchronization region. Through our *Message Links Generator*, the essential message links for each data element are generated as

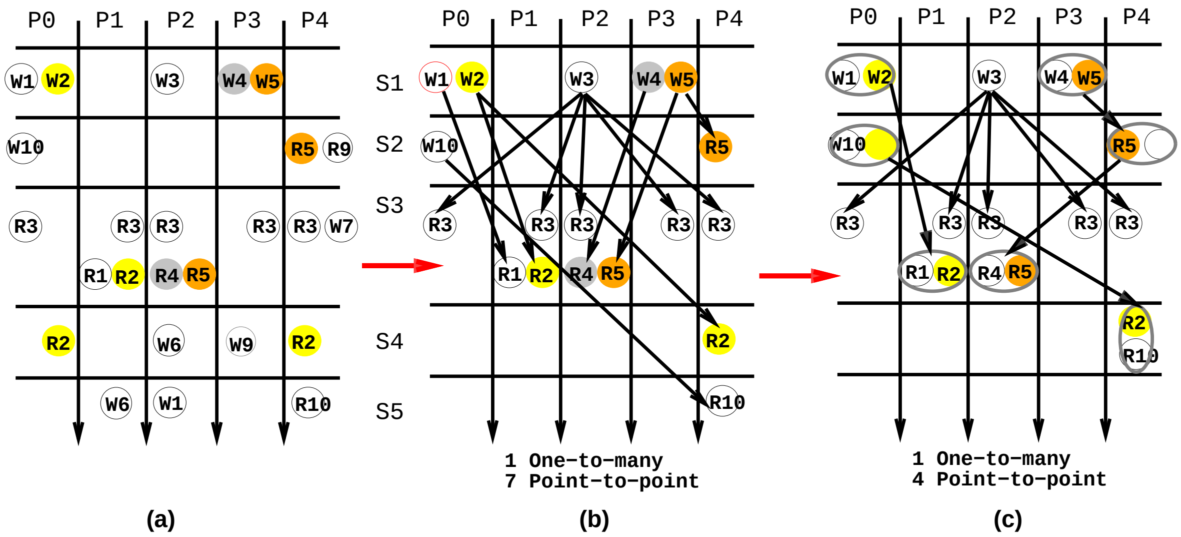


Figure 9: Example of the Scheme C

shown in figure 9(b) and some of the data elements which do not cause any communication are removed. In this example, the communication links are now composed of a single one-to-many message and seven point-to-point messages. After the optimization of the post-processing phase as shown in figure 9(c), some of the combinable messages are bound together, which results in a reduced number of point-to-point messages. Some of the messages, such as $Mesg(10, P0, P4, S2, S5)$, must be received earlier; and some, such as $Mesg(2, P0, P4, S1, S4)$, must be transmitted later because they are combined with another message, whose synchronization range overlaps, but is not identical. The synchronization ranges of these combined messages are reduced appropriately.

There is an interesting case which we call *intermediate message routing* in the message optimization. Notice that $Mesg(4, P3, P2, S1, S3)$ and $Mesg(5, P3, P2, S1, S3)$ are combined and routed through $P4$ to $P2$ so as to eliminate one message initialization overhead even though *Data 4* is not needed by $P4$. However, if the size of *Data 4* is big enough, we may not get any benefit from this scheme because the transfer time for *Data 4* could override the message initialization time and *Data 4* may also pollute $P4$'s memory.

4.3.3 Message scheduling for point-to-point communication

In section 2.2, we described an efficient communication scheme, pairwise message exchange, provided by some message-passing machines. Before scheduling the generated point-to-point communication messages, pairs of messages that are suitable for pairwise exchange will be grouped together so as to mask a communication start-up overhead.

In this step, we demonstrate an algorithm to schedule the messages. The purpose of scheduling the messages is to reduce the wait time for each processor. A non-proper schedule for the point-to-point message-passing code can result in unnecessary wait time in some processors. For example, in figure 10(a), a message is defined as $M(s, i, j)$, where s, i and j stand for the message weight, source processor ID and destination processor ID, respectively. The message weight can be calculated from the formulas of communication latency modeled in [1]. Nine messages are scheduled on five processors in the order shown in figure 10(a). The corresponding schedule and the communication latency are shown in figure 10(b), the total latency for passing these nine messages is 5700. Based on our algorithm described below, the selected order of the scheduled messages and the resulting communication latency are shown in 10(c) and (d). The total latency of the scheduled messages is reduced to 3300.

The scheduling algorithm is explained as follows. Define the processor weight function W_{p_i} as the time at which processor i finishes the last communication currently assigned to it. The message weight, W_{m_k}, of a particular unassigned message m_k in point-to-point communication is defined as the larger processor weight of the two processors involved in this message, i.e. $W_{m_k} = max(W_{p_i}, W_{p_j})$, where p_i and p_j are the processors involved in the message $M(s,i,j)$, i.e. the earliest time that this message can start if it is selected for scheduling next. The scheduling algorithm has the following steps. The algorithm is an earliest-start-time-first greedy algorithm with a shortest-message-first tie-breaker.

Step 1: Reset the values of processor weight function, W_{p_i}, and message weight function, W_{m_k}, to zero for each processor p_i and message m_k. Put all the messages to be scheduled into the message candidates list.

Step 2: Select a random message $M(s,i,j)$ from the message candidates list as the first one in the schedule.

Step 3: Update the weights W_{p_i} and W_{p_j} by adding the modeled communication latency, s, of the scheduled message to the weights of W_{p_i} and W_{p_j} of processors i and j, then select the larger value of W_{p_i} and W_{p_j} as the new weight for both p_i and p_j.

Step 4: Update the message weights W_{m_k} for each unscheduled message.

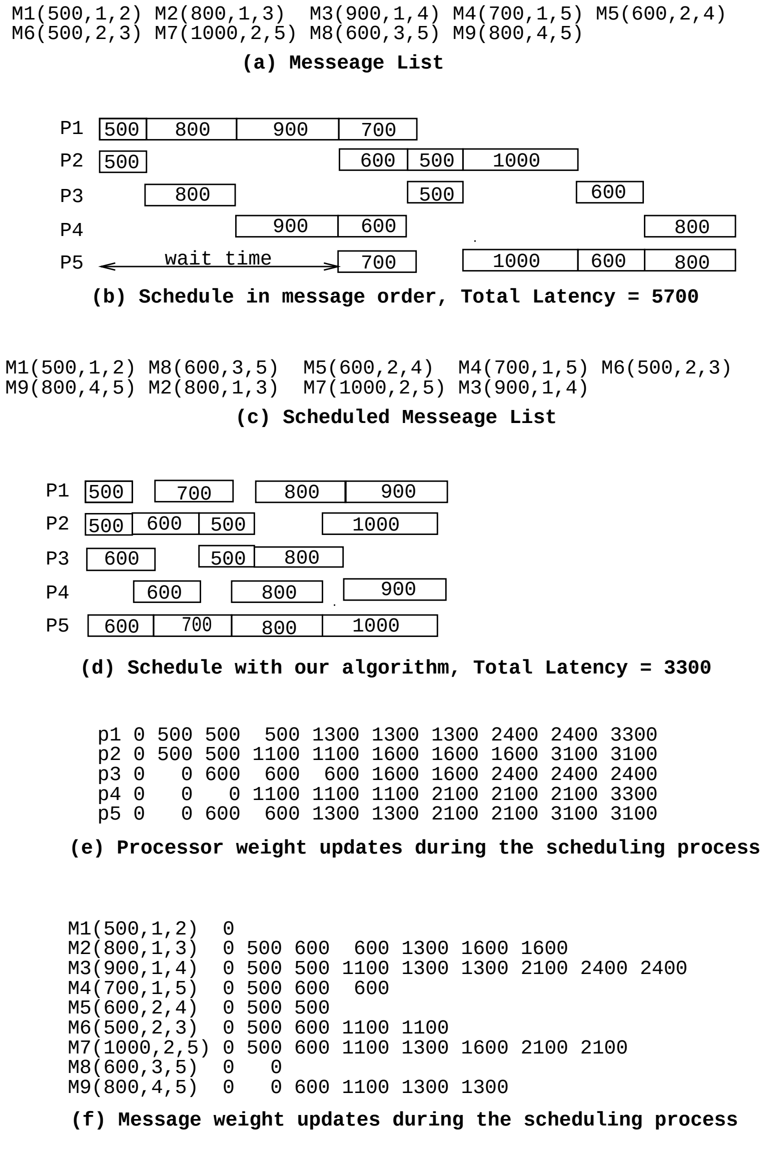
Step 5: Select the message with the minimum message weight as the message to be scheduled next. If there are several candidates, then select the one with the shortest message modeled communication.

Step 6: Repeat *Step 3* to *6* until the message candidate list is empty.

Figures 10(d) and 10(e) show the variation of processor weights and message weights during the scheduling process. The resulting final order of the schedule is given in figure 10(c).

4.4 Generating message-passing routines

The final phase of Scheme C is to generate the message-passing routines to be used in the message-passing code. Another automatic tool, *Message_Passing File Generator* reads in the generated formatted message links and generated a *Fortran* subroutine which will be inserted later in each original synchronization barrier of the shared-memory code to perform interprocessor communication. After these routines are generated by our automatic tool, we simply change each original synchronization barrier in shared-memory code into the corresponding message-passing subroutine call, as shown in figure 11. There are potential deadlock problems if the send/receive calls are not



```
M1(500,1,2) M2(800,1,3)  M3(900,1,4) M4(700,1,5) M5(600,2,4)
M6(500,2,3) M7(1000,2,5) M8(600,3,5) M9(800,4,5)
```

(a) Messeage List

(b) Schedule in message order, Total Latency = 5700

```
M1(500,1,2) M8(600,3,5)  M5(600,2,4)  M4(700,1,5) M6(500,2,3)
M9(800,4,5) M2(800,1,3)  M7(1000,2,5) M3(900,1,4)
```

(c) Scheduled Messeage List

(d) Schedule with our algorithm, Total Latency = 3300

```
p1 0 500 500  500 1300 1300 1300 2400 2400 3300
p2 0 500 500 1100 1100 1600 1600 1600 3100 3100
p3 0   0 600  600  600 1600 1600 2400 2400 2400
p4 0   0   0 1100 1100 1100 2100 2100 2100 3300
p5 0   0 600  600 1300 1300 2100 2100 3100 3100
```

(e) Processor weight updates during the scheduling process

```
M1(500,1,2)  0
M2(800,1,3)  0 500 600  600 1300 1600 1600
M3(900,1,4)  0 500 500 1100 1300 1300 2100 2400 2400
M4(700,1,5)  0 500 600  600
M5(600,2,4)  0 500 500
M6(500,2,3)  0 500 600 1100 1100
M7(1000,2,5) 0 500 600 1100 1300 1600 2100 2100
M8(600,3,5)  0   0
M9(800,4,5)  0   0 600 1100 1300 1300
```

(f) Message weight updates during the scheduling process

Figure 10: Example of message scheduling

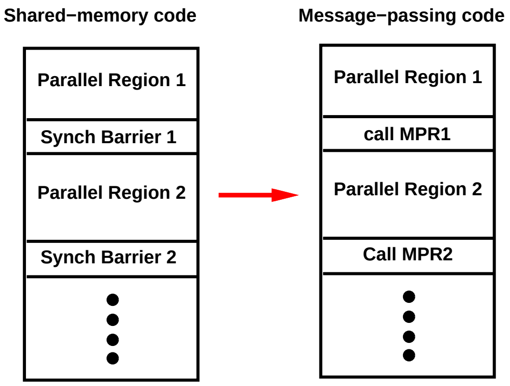


Figure 11: Conversion from shared-memory code to message-passing code

properly arranged in these routines. For example, suppose that $P1$ and $P2$ each have messages to send to the other. If the generated code has both receiving calls written ahead of the sending calls, then $P1$ and $P2$ will wait forever and never receive their message. To avoid deadlock problems, pairwise communication will be helpful here. (Another implementation that can be effective would use a non-blocking send for all sending processors first. Then after all processors send, each processor can start to receive by using a blocking receive. However, this scheme is unable to be implemented thus far, since our target machine, IBM SP2, has not yet actually supported the non-blocking communication functions.)

5 Preliminary Results and Current Status

At this point, we have ported the Indian Ocean code on the IBM POWER2 node, and developed three fully functional versions of message-passing programs based on the two different broadcast communications patterns (one to many and pairwise message exchange) described in Scheme A and Scheme B. A fourth code based on Scheme C is functional except for some arrays in some synchronization regions which have irregular communication that varies among time steps. These communications could be corrected by refining Scheme C or by using Scheme A or B only for these and Scheme C for all others. The message byte count for Scheme C in Table 1 is an accurate count of what a refined Scheme C would produce. Preliminary results collected from these programs are as follows.

Table 1 is the actual number of data bytes communicated on the network by the Scheme A and C codes based on a 25-processor run for three time steps of the Indian Ocean code. As this table shows, there are many redundant data bytes communicated on the network when the message-passing code was implemented by Scheme A. In most of the processors, only about 1% of the data bytes transferred are regarded as essential (i.e. communicated in the Scheme C code). Many messages are removed or shortened by Scheme C to achieve a great performance improvement.

Table 2 compares the uniprocessor performance of the KSR2 and the IBM SP2 (POWER2 architecture). Note that IBM SP2 has a much faster single processor than the KSR2.

PID	Send(A)	Send(C)	Recv(A)	Recv(C)	Reduced Send(%)	Reduced Recv(%)
P0	98336640	102432	29909988	3552628	99.90%	88.12%
P1	29763360	299892	32767208	147696	98.99%	99.95%
P2	29910240	308460	32761088	157956	98.97%	99.52%
P3	29744352	301992	32768000	149328	98.98%	99.54%
P4	29759904	303096	32767352	150912	98.98%	99.54%
P5	30065760	303120	32754608	150552	98.99%	99.54%
P6	29706336	310416	32769584	155232	98.96%	99.53%
P7	29783232	277020	32766380	127008	99.07%	99.61%
P8	29942208	304008	32759756	152808	98.98%	99.53%
P9	29792736	255480	32765984	105060	99.14%	99.68%
P10	29810880	309444	32765228	155580	98.96%	99.53%
P11	30224736	313092	32747984	159756	98.96%	99.51%
P12	29746080	315468	32767928	161688	98.94%	99.51%
P13	29783232	294888	32766380	149208	99.01%	99.54%
P14	29988000	289776	32757848	137796	99.03%	99.58%
P15	29713248	299364	32769296	148464	98.99%	99.55%
P16	29790144	272580	32766092	120564	99.08%	99.63%
P17	29988000	269916	32757848	119076	99.10%	99.64%
P18	29740032	300072	32768180	147048	98.99%	99.55%
P19	29771136	247356	32766844	97368	99.17%	99.70%
P20	30044160	245256	32755508	96792	99.18%	99.70%
P21	29875680	234204	32762528	86808	99.22%	99.74%
P22	30034656	233676	32755904	87948	99.22%	99.73%
P23	29954304	217236	32759252	72372	99.27%	99.78%
P24	30907296	168024	32719544	34752	99.46%	99.89%
Total	816176352	6624000	816176352	6624000	99.19%	99.19%

Table 1: Comparison of the number of messages (in bytes) between Scheme A and C

Single Node (in seconds)	10 Days	30 Days	100 Days
KSR2	391.33	931.34	2369.24
IBM SP2	70.16	157.23	277.6

Table 2: Uniprocessor run on the KSR2 and IBM SP2

IBM SP2 (in seconds)	30 Days	120 Days	360 Days
P=1	117.36	359.63	1005.77
P=16 (Scheme A: broadcast)	122	377.9	1063
P=16 (Scheme A: pairwise)	78.66	178.4	444
P=16 (Scheme B)	76.35	139.7	320.5

Table 3: Preliminary results on the IBM SP2

	P = 1	P = 16
KSR2	1	3.685
IBM SP2 (Scheme A: broadcast)	1	1.29
IBM SP2 (Scheme A: pairwise)	1	1.99
IBM SP2 (Scheme B)	1	2.06

Table 4: Speedup of on the KSR2 and IBM SP2

Table 3 shows parallel performance for various schemes. Table 4 shows speedup vs. P for the KSR2 and for the Scheme A (with one-to-many broadcast) for the SP2, which is the least efficient scheme. Speedups for the other schemes are yet to be evaluated.

6 Conclusion and Future Work

In this research, we showed that tracing tools can be effectively applied in program conversion from shared-memory to message-passing machines in addition to their basic functions on debugging and analyzing the behavior of programs. Scheme C has shown that it can be used as an effective parallel program performance tuning tool on message-passing programs with stable communication patterns and can identify and remove the redundant messages, and exploit opportunities for message combining.

For portions of programs with irregular communication, Scheme B or if necessary Scheme A can be used. They are more robust but generate more traffic, some of which is generally redundant. More work needs to be done to improve the robustness and efficiency of these schemes, but these preliminary results are very encouraging.

References

[1] Eric L. Boyd, Gheith A. Abandah, Hsien-Hsin Lee, and Edward S. Davidson. Modeling computation and communication performance of parallel scientific applications: A case study of the IBM SP2. Technical report, CSE-TR-236-95, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, 1994.

[2] Shih-Hao Hung and Edward S. Davidson. Design of trace-driven simulation tools on the KSR1. Directed Study Report, University of Michigan, 1994.

[3] IBM Corporation, Kingston, NY. *IBM AIX Parallel Environment Parallel Programming Subroutine Reference*, 1994.

[4] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating message-passing and shared-memory: Early experience. In *Proceedings of the Fouth Symposium on Principles and Practices of Parallel Programming*, pages 54–63, 1993.