A Message Ordering Problem in Parallel Programs*

Bora Uçar and Cevdet Aykanat

Department of Computer Engineering, Bilkent University, 06800, Ankara, Turkey
{ubora,aykanat}@cs.bilkent.edu.tr

Abstract. We consider certain classes of parallel program segments in which the order of messages sent affects the completion time. We give characterization of the subject class of parallel programs and propose a solution to minimize the completion time. With a sample parallel program, we experimentally evaluate the effect of the solution on a PC cluster.

Keywords. fine grain computation, message latency, message ordering

Technical Report:	Department of Computer Engineering,
	Bilkent University, 06800 Ankara, Turkey.
	Available at http://www.cs.bilkent.edu.tr/publications/.
	Date: May 10, 2004.

* This work is partially supported by the Scientific and Technical Research Council of Turkey under grant 103E028

1 Introduction

We consider certain classes of parallel program segments with the following characteristics. First, there is a small-to-medium grain computation between two communication phases which are referred to as pre- and post-communication phases. Second, local computations cannot start before the pre-communication phase ends, and the post-communication phase cannot start before the computation ends. Third, the communication in both phases is irregular and sparse. That is, the communications are performed through point-to-point send and receive operations, where the sparsity refers to both the number and volume of the essages. These traits appear, for example, in the sparse-matrix vector multiply $y = Ax$, where matrix A is partitioned on the nonzero basis and also in the sparse-matrix-chain vector multiply $y = ABx$, where matrix A is partitioned along columns and matrix B is partitioned conformably along rows. In both examples, the x-vector entries are communicated just before the computation and the y-vector entries are communicated just after the computation.

There has been a vast amount of research in partitioning sparse matrices to effectively parallelize computations by achieving computational load balance and by minimizing the communication overhead [2, 7, 8]. As noted in [7], most of the existing methods consider minimization of the total message volume. Depending on the machine architecture and problem characteristics, communication overhead due to message latency may be a bottleneck as well [5]. Furthermore, the maximum message volume and latency handled by a single processor may also have crucial impact on the parallel performance. In our previous work [12], we addressed these four communication-cost metrics in 1D partitioning. Çatalyürek and Aykanat [3, 4] proposed hypergraph models for two-dimensional (2D) coarse-grain and fine-grain sparse matrix partitioning. In the coarse-grain model, a matrix is partitioned in a checkerboard like style. In the fine-grain model, a matrix is partitioned on the nonzero basis. In another work [10], we addressed the same four metrics in 2D fine-grain partitioning of sparse matrices. However, these are not sufficient to minimize the total completion time of the above kind of parallel programs. Since the phases do not overlap, the receiving time of a processor, and hence the issuing time of the corresponding send operation play an important role in the total completion time.

There may be different solutions to the above problem. One may consider balancing the number of messages per processor both in terms of sends and receives. This strategy would then has to partition the computations with the objectives of achieving computational load balance, minimizing total volume of messages, minimizing total number of messages, and also balancing the number of messages sent/received on the per processor basis. However, combining these objectives into a single function to be minimized would challenge the current state of the art. For this reason, we take these problems apart from each other and decompose the overall problem into stages, each of which involving a certain objective. We first use standard models to minimize the total volume of messages and maintain the computational load balance across processors using highly respected methods, such as graph and hypergraph partitioning methods. Then, we minimize the total number of messages and maintain a loose communication volume load balance, and in the meantime we address the minimization of the maximum number of messages sent by a single processor. After this stage, the communication pattern is determined. In this paper, we suggest to append one more stage in which the send operations of processors are ordered to address the minimization of the total completion time.

2 Message Ordering Problem and a Solution

We make the following assumptions. The computational load imbalance is negligible. All processors begin the pre-communication phase at the same time because of the possible global synchronization points and balanced computations that exist in the other parts of the parallel program. The parallel system has a high latency overhead so that the message transfer time is dominated by the start-up cost because of the sparsity of the message volumes. By the same reasoning, the receive operation is assumed to incur negligible cost to the receiving processor. For the sake of simplicity, the send operations are assumed to take unit time. Under these assumptions, once a send is initiated by a processor at time t_i, the sending processor can continue with some other operation at time t_{i+1}, and the receiving processor receives the message at time t_{i+1}. This assumption extends to concurrent messages destined for the same processor. The rationale behind these assumptions is that, the start-up costs for all messages destined for a certain processor truly overlap with each other.

Let *send-lists* $S_1(p)$ and $S_2(p)$ denote the set of messages, distinguished by the ranks of the receiving processors, to be sent by processor P_p in pre- and post-communication phases, respectively. For example, $\ell \in S_1(p)$ denotes the fact that processor P_ℓ will receive a message from P_p in the pre-communication phase. For $\ell \in S_1(p)$, we use $s_1(p,\ell)$ to denote the completion time of the message from P_p to P_ℓ, i.e., P_p issued the send at time $s_1(p,\ell)-1$, and P_ℓ received the message at time $s_1(p,\ell)$. We use $s_2(p,\ell)$ for the same purpose for the post-communication phase. Let W be the amount of computation performed by each processor. Let

$$r_1(p) = \max_{j: p \in S_1(j)} \{s_1(j,p)\} \tag{1}$$

denote the point in time at which processor P_p receives its latest message in the pre-communication phase. Then, P_p will enter the computation phase at time

$$c_1(p) = \max\{|S_1(p)|, r_1(p)\}, \tag{2}$$

i.e, after sending all of its messages and receiving all messages destined for it in the pre-communication phase. Let

$$r_2(p) = \max_{j: p \in S_2(j)} \{s_2(j,p)\} \tag{3}$$

denote the point in time at which processor P_p receives its latest message in post-communication phase. Then, processor P_p will reach completion at time

$$c_p = \max\{c_1(p) + W + |S_2(p)|, r_2(p)\}, \tag{4}$$

i.e., after completing its computational task as well as all send operations in the post-communication phase and after receiving all post-communication messages destined for it. Using the above notation, our objective is

$$minimize\{\max_p\{c_p\}\}, \tag{5}$$

i.e, to minimize the maximum completion time. The maximum completion time induced by a message ordering is called the bottleneck value, and the processor that defines it is called the bottleneck processor. Note that the objective function depends on the time points at which the messages are delivered.

In order to clarify the notations and assumptions, consider a six-processor system as shown in Fig. 1(a). In the figure, the processors are synchronized at time t_0. The computational load of each processor is of length five-units and shown as a gray rectangle. The send operation from processor P_k to P_ℓ is labeled with $s_{k\ell}$ on the right-hand side of the time-line for processor P_k. The corresponding receive operation is shown on the left-hand side of the time-line for processor P_ℓ. For example, processor P_1 issues a send to P_3 at time t_0 and completes the send at time t_1 which also denotes the delivery time to P_3. Also note that P_3 receives a message from P_5 at the same time. In the figure, $r_1(1) = c_1(1) = t_5$, $r_2(1) = t_{10}$ and $c_1 = t_{15}$. The bottleneck processor is P_1 with the bottleneck value $t_b = t_{15}$.

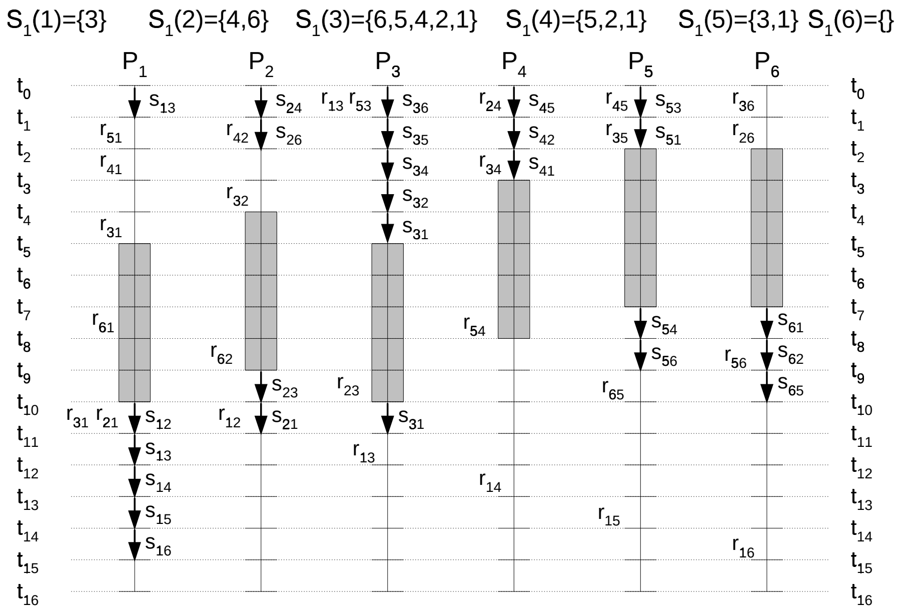
Reconsider the same system where the messages are sent according to the order as shown in Fig. 1(b). In this setting, P_1 is also a bottleneck processor with value $t_b = t_{11}$.

Note that if a processor P_p never stays idle then it will reach completion at time $|S_1(p)| + W + |S_2(p)|$. The optimum bottleneck value cannot be less than the maximum of these values. Therefore the order given in Fig. 1(b) is the best possible. Let processors P_q and P_r be the maximally loaded processors in the pre- and post-communication phases respectively, i.e., $|S_1(q)| \geq |S_1(p)|$ and $|S_2(r)| \geq |S_2(p)|$ for all p. Then, the bottleneck value cannot be larger than $|S_1(q)| + W + |S_2(r)|$. The setting in Fig. 1(a) attains this worst possible bottleneck value.

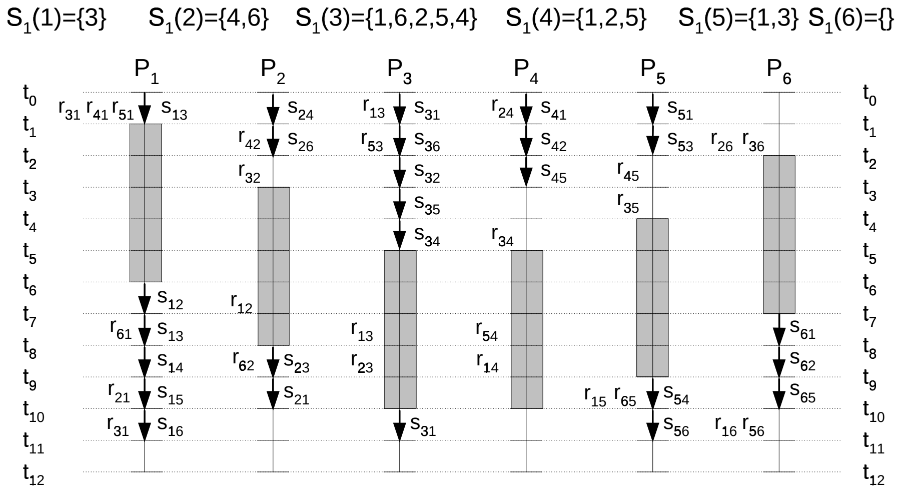
Observe that in a given message order, the bottleneck occurs at a processor with an outgoing message. Meaning that, for any bottleneck processor that receives a message at time t_b, there is a processor which finishes a send operation at the same time. Therefore, for a processor P_p to be a bottleneck processor we have to have

$$c'_p = c_1(p) + W + |S_2(p)| \tag{6}$$

as a bottleneck value. Hence, our objective reduces to


S1(1)={3} S1(2)={4,6} S1(3)={6,5,4,2,1} S1(4)={5,2,1} S1(5)={3,1} S1(6)={}
P1 P2 P3 P4 P5 P6
t0 … t16

(a) A sample message order which produces worst completion time


S1(1)={3} S1(2)={4,6} S1(3)={1,6,2,5,4} S1(4)={1,2,5} S1(5)={1,3} S1(6)={}
P1 P2 P3 P4 P5 P6
t0 … t12

(b) A sample message order which produces best completion time

Fig. 1. Worst and best orderings of the messages

$$minimize\{\max_p\{c'_p\}\}. \tag{7}$$

Also observe that the bottleneck processor and value remains as is, for any ordering of the post-communication messages. Therefore, our problem reduces to ordering the messages in the pre-communication phase. From these observations we reach the very intuitive idea of having the maximally loaded processor in the post-communication phase to be in the first position in each send-list. This will make the processor with maximum $|S_2(p)|$ to enter the computation phase as soon as possible. Extending this to the remaining processors we develop the following algorithm. For each processor P_q define $key(q) = |S_2(q)|$. Then each processor P_p sorts its send-list $S_1(p)$ in descending order according to the key-values of the receiving processors. These sorted send-lists determine the message ordering in the pre-communication phase, where the ordering in the post-communication phase is arbitrary.

Theorem 1. *The above algorithm obtains the optimal solution that minimizes the maximum completion time.*

Proof. We take an optimal solution and then by using "switching arguments" we modify it to have each send-list sorted in descending order of key-values. The details of the proof can be found in [11].

Consider an optimal solution. Let processor P_k be the bottleneck processor finishing its sends at time t_b. For each send-list in the pre-communication phase, we perform the following operations.

For any P_ℓ with $key_k \leq key_\ell$ where P_k and P_ℓ are in the same send-list $S_1(p)$, if $s_1(p,\ell) \leq s_1(p,k)$ then we are done, if not swap $s_1(p,\ell)$ and $s_1(p,k)$. Let $t_s = s_1(p,\ell)$ before the swap operation. Then, we have $t_s + W + key_\ell \leq t_b$ before the swap. After the swap we will have $t_s + W + key_k$ and $t_h + W + key_\ell$ for some $t_h < t_s$, for the processors P_k and P_ℓ. These two values are less than t_b, i.e., we did not disturb the optimality of the solution.

For any P_j with $key_j \leq key_k$ where P_j and P_k are in the same send-list $S_1(q)$, if $s_1(q,k) \leq s_1(q,j)$ then we are done, if not swap $s_1(q,k)$ and $s_1(q,j)$. Let $t_s = s_1(q,k)$ before the swap operation. Then, we have $t_s + W + key_k \leq t_b$. After the swap operation we will have $t_s + W + key_j$ and $t_h + W + key_k$ for some $t_h < t_s$ for processors P_j and P_k, respectively. Clearly, these two values are less than or equal to t_b, i.e., we again have an optimal solution.

For any P_u and P_v that are different from P_k with $key_u \leq key_v$ in a send-list $S_1(r)$, if $s_1(r,v) \leq s_1(r,u)$ then we are done, if not swap $s_1(r,u)$ and $s_1(r,v)$. Let $t_s = s_1(r,v)$ before the swap operation. Then, we have $t_s + W + key_v \leq t_b$. After the swap operation we will have $t_s + W + key_u$ and $t_h + W + key_v$ for some $t_h < t_s$, for P_u and P_v respectively. These two values are clearly less than or equal to t_b. Therefore, for each optimal solution we have an equivalent solution in which all send-lists in pre-communication phase are sorted according to the key values of the processors. Since the sorted order is unique with respect to the key values, the above algorithm is correct.

3 Experiments

In order to see whether the findings in this work help in practice we have implemented a simple parallel program which is shown in Fig 2. In this figure, each processor first posts its non-blocking receives and then sends its messages in the order as they appear in the send-lists. In order to simplify the effects of the message volume on the message transfer time, we set the same volume for each message. We have used LAM [1] implementation of MPI and `mpirun` command without `-lamd` option. The parallel program were run on a Beowulf class [9] PC cluster with 24 nodes. Each node has a 400MHz Pentium-II processor and 128MB memory. The interconnection network is comprised of a 3COM SuperStack II 3900 managed switch connected to Intel Ethernet Pro 100 Fast Ethernet network interface cards at each node. The system runs Linux kernel 2.4.14 and Debian GNU/Linux 3.0 distribution.

We extracted the communication patterns of some row-column-parallel sparse matrix-vector multiply operations on 24 processors. Table 1 lists minimum and maximum number of send operations per processor under columns *min* and *max*. Total number of messages is given under the column *tot*.

Table 1. Communication patterns and parallel running times on 24 processors.

Data	Communication pattern			Mssg order	Completion time				
					unit	milliseconds			
					Max	Message length (bytes)			
	min	max	tot		$\{c'_p\}$	8	64	512	1024
1-PRE	5	21	290	best	38	4.3	4.4	5.5	7.2
1-POST	6	22	358	worst	42	4.8	5.0	6.2	7.8
2-PRE	3	23	313	best	39	4.9	5.0	6.0	7.3
2-POST	11	22	370	worst	45	5.3	5.4	6.7	7.8
3-PRE	10	23	490	best	45	6.3	6.4	7.8	9.7
3-POST	15	23	504	worst	46	6.6	6.6	8.2	10.1
4-PRE	6	22	312	best	41	4.5	4.6	5.9	7.3
4-POST	10	20	356	worst	42	5.3	5.6	6.8	8.2
5-PRE	5	23	228	best	36	4.0	4.1	4.9	5.9
5-POST	7	13	228	worst	36	4.4	4.6	5.6	6.6
6-PRE	1	23	212	best	35	4.1	4.1	5.1	6.0
6-POST	4	17	236	worst	40	4.5	4.6	5.8	6.7
7-PRE	3	20	226	best	29	3.7	3.7	4.5	5.3
7-POST	7	17	253	worst	37	3.9	3.9	5.0	5.9
8-PRE	2	23	267	best	43	4.7	4.7	6.1	7.6
8-POST	4	22	278	worst	45	5.7	5.9	7.0	8.1
9-PRE	3	16	167	best	35	3.7	4.0	4.8	5.6
9-POST	4	20	273	worst	36	4.3	4.3	5.3	6.0
10-PRE	2	23	300	best	46	4.7	4.7	6.3	8.0
10-POST	10	23	316	worst	46	5.6	5.7	7.1	8.3
					W (Computation time):	0.00	0.01	0.06	0.11

```
 MPI_Barrier(MPI_COMM_WORLD);
 startTime = MPI_Wtime();
 for(iter = 0; iter < MAXITER; iter++){
    communication(preSendList, preSendCount,
                  preRecvList, preRecvCount,
                  sendBuf, recvBuf, iter);
    computation(sendBuf, recvBuf);
    communication(postSendList, postSendCount,
                  postRecvList, postRecvCount,
                  sendBuf, recvBuf, iter + 1 );
    MPI_Barrier(MPI_COMM_WORLD);
 }
 totTime = 1000.0*MPI_Wtime() - 1000.0*startTime;
```

(a) Parallel program segment

```
void computation(MSSGTYPE *sendBuf, MSSGTYPE *recvBuf)
{
 int i;
 for(i = 0; i < numProcs; i++){
    int j, indi = mssgSizes * i;
    for(j = 0; j < mssgSizes; j++){
       int indij = indi + j;
       sendBuf[indij]=(sendBuf[indij]+recvBuf[indij])/(MSSGTYPE)2;
    }
 }
 return;
}
```

(b) Local computation performed at each processor

```
void communication(int *sendList, int sendCount,
                   int *recvList, int recvCount,
                   MSSGTYPE *sendBuf, MSSGTYPE *recvBuf, int tag)
{
 int i;
 MPI_Request reqs[recvCount]; MPI_Status stats[recvCount];
 for(i = 0 ; i < recvCount; i++){
    int p = recvList[i], ind = p*mssgSizes;
    MPI_Irecv(&recvBuf[ind], mssgSizes, bMPITYPESTR, p,
               tag, MPI_COMM_WORLD,&reqs[i]);
 }
 for(i = 0; i < sendCount; i++){
    int p = sendList[i], ind = myId * mssgSizes;
    MPI_Send(&sendBuf[ind], mssgSizes,bMPITYPESTR,p, tag,MPI_COMM_WORLD);
 }
 if(recvCount > 0) MPI_Waitall(recvCount, reqs, stats);
 return;
}
```

(c) Implementation of pre- and post-communication phases

Fig. 2. A simple parallel program

For each test case, we run the parallel program of Fig. 2 with small message lengths of 8, 64, 512, and 1024-bytes to justify the practicality of the assumptions made in this work. Both best and worst orderings are experimented. The best message orderings are generated according to the algorithm proposed in § 2. The worst message orderings are obtained by sorting the send-lists according to the key-values of the receiving processors in increasing order. In all cases, we used the same message ordering in the post-communication phase. In Table 1, we give the running times in milliseconds. We give the best among 20 runs (see [6] for choosing best in order to obtain reproducible results). In the table, we also give $\max_p\{c'_p\}$ for worst and best orderings with $W = 0$. In all cases, the best ordering always gives better completion time than the worst ordering. In theory, however, we did not expect improvements for the 5th and 10th cases, in which the two orderings give the same bottleneck value. This unexpected outcome may be resulting from the internals of the process that handles the communication requests. We are going to investigate this issue.

4 Conclusion

In this work, we addressed the problem of minimizing maximum completion time of a certain class of parallel program segments in which there exists a small-to-medium grain computation between two communication phases. We showed that the order in which the messages are sent affects the completion time and showed how to order the messages optimally in theory. Experimental results on a PC cluster verified the existence of the specified problem and the validity of the proposed solution. As a future work, we are trying to set-up experiments to observe the findings of this work in the parallel sparse matrix-vector multiplies. A generalization of the given problem addresses parallel programs that have multiple computation phases interleaved with communications. This problem is in our research plans.

References

1. G. Burns, R. Daoud, and J. Vaigl. LAM: an open cluster environment for MPI. In John W. Ross, editor, *Proceedings of Supercomputing Symposium '94*, pages 379–386. University of Toronto, 1994.
2. Ü. V. Çatalyürek and C. Aykanat. Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999.
3. Ü. V. Çatalyürek and C. Aykanat. A fine-grain hypergraph model for 2d decomposition of sparse matrices. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS), 8th International Workshop on Solving Irregularly Structured Problems in Parallel (Irregular 2001)*, April 2001.
4. Ü. V. Çatalyürek and C. Aykanat. A hypergraph-partitioning approach for coarse-grain decomposition. In *Proceedings of Scientific Computing 2001 (SC2001)*, pages 10–16, Denver, Colorado, November 2001.

5. J. J. Dongarra and T. H. Dunigan. Message-passing performance of various computers. *Concurrency—Practice and Experience*, 9(10):915–926, 1997.
6. W. Gropp and E. Lusk. Reproducible measurements of mpi performance characteristics. Technical Report ANL/MCS-P755-0699, Argonne National Laboratory, June 1999.
7. B. Hendrickson and T. G. Kolda. Graph partitioning models for parallel computing. *Parallel Computing*, 26:1519–1534, 2000.
8. B. Hendrickson and T. G. Kolda. Partitioning rectangular and structurally unsymmetric sparse matrices for parallel processing. *SIAM J. Sci. Comput.*, 21(6):2048–2072, 2000.
9. T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranaweke, and C. V. Packer. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, 1995.
10. B. Uçar and C. Aykanat. Minimizing communication cost in fine-grain partitioning of sparse matrices. In A. Yazıcı and C. Şener, editors, *in Proc. ISCISXVIII-18th Int. Symp. on Computer and Information Sciences*, Antalya, Turkey, Nov. 2003.
11. B. Uçar and C. Aykanat. Determining the order of messages in parallel programs. Technical Report BU-CE-0404, Department of Computer Engineering, Bilkent University, 2004.
12. B. Uçar and C. Aykanat. Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies. *SIAM J. Sci. Comput.*, Accepted, 2004.