Efficient Barrier Synchronization Mechanism for the BSP Model on Message-Passing Architectures

Jin-Soo Kim Soonhoi Ha Chu Shik Jhon

Department of Computer Engineering
Seoul National University
Seoul 151-742, KOREA
{jinsoo, sha, csjhon}@comp.snu.ac.kr

Abstract

The Bulk Synchronous Parallel *(BSP) model of computation can be used to develop efficient and portable programs for a range of machines and applications. However, the cost of the barrier synchronization used in the BSP model is relatively expensive for message-passing architectures. In this paper, we relax the barrier synchronization constraint in the BSP model for the efficient implementation on message-passing architectures.*

In our relaxed barrier synchronization, *the synchronization occurs at the time of accessing non-local data only between the producer and the consumer processors, eliminating the exchange of global information. From the experimental evaluations on IBM SP2, we have observed that the relaxed barrier synchronization reduces the total synchronization time by 45.2% to 61.5% in FT, and 28.6% to 49.0% in LU with 32 processors.*

1. Introduction

The *Bulk Synchronous Parallel* (BSP) model of computation [8] was first proposed by Valiant as a bridging model between hardware and software for general-purpose parallel computation. Goudreau *et.al.* [3] have already shown that the BSP model can be used to develop efficient and portable programs for a range of machines and applications.

A BSP abstract machine consists of a collection of identical processors, each with local memory, connected by a communication network. The computation is structured as a sequence of *supersteps*, each followed by a barrier synchronization. In each superstep, a processor performs a number of local operations on data present in its local memory and sends messages to other processors. A message sent from one processor during a superstep is not visible to the destination processor until the subsequent superstep.

The barrier synchronization used in the BSP model can be implemented efficiently using locks, semaphores or cache coherence protocols for shared-memory architectures [5]. However, the cost of the barrier synchronization in message-passing architectures is relatively expensive because processors are synchronized by exchanging messages. The cost usually grows as the number of processors increases.

In this paper, we relax the barrier synchronization constraint in the BSP model for the efficient implementation on message-passing architectures. Direct implementation of the barrier synchronization does not allow any processor to proceed past the synchronization point until all processors reach that point. Instead, in our *relaxed barrier synchronization*, the synchronization occurs at the time of accessing non-local data only between the producer and the consumer processors, eliminating the exchange of global information.

2. The BSP Model on Message-Passing Architectures

Hill *et.al.* have proposed BSPlib [4] as a standard library to integrate various approaches to BSP programming. Figure 1 shows a code fragment which uses BSPlib routines. The program broadcasts the value of `v` stored in processor 0 to all the other processors.

`bsp_sync()` performs a barrier synchronization and identifies the end of a superstep. One way of performing data communication in BSPlib is to use a Direct Remote Memory Access (DRMA) facility that provides routines to put data into the local memory of a remote processor (`bsp_put()`), or to get data from a remote processor (`bsp_get()`). When `bsp_put()` in figure 1 is executed, the value of integer `v` is remotely stored into the memory of other processors at address `&x+0`. `bsp_pid()` and `bsp_nprocs()` return the index of the processor and the number of processors available in the system, respectively.

1063-7133/98 $10.00 © 1998 IEEE

```
int i, v, x;
...
bsp_push_reg(&x, sizeof(int));
bsp_sync();

if (bsp_pid() == 0)
    for (i = 0; i < bsp_nprocs(); i++)
        bsp_put(i, &v, &x, 0, sizeof(int));
bsp_sync();

printf("broadcasted value = %d\n", x);
```

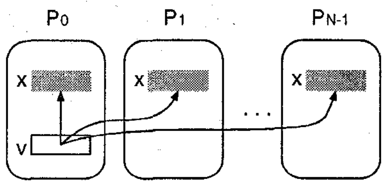

Figure 1. An example BSP program

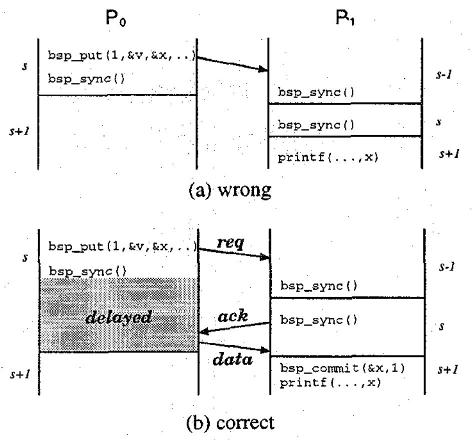

Figure 2. When the destination is too slow

Because the data structure x is not necessarily stored at the same address in all processors, bsp_push_reg() *registers* the address of x to the system so that it can be a target of data transfer in DRMA routines. Such registration maps the local address of the variable x to a global index that is the same in all processors.

For message-passing architectures, single-sided communication routines such as bsp_put() and bsp_get() cannot be satisfied without the service of the destination processor, because there is no facility to access the remote memory directly[1]. On the other hand, a processor generally does not know how many requests it has to serve for other processors before proceeding to the next superstep.

To solve this problem, the current implementation of bsp_sync() for message-passing architectures consists of two phases [7]. In the first phase, all processors exchange information about the number, sizes, and destination addresses of messages. This exchange phase also serves as a barrier synchronization for the superstep. After the first phase, each processor knows how many messages to receive from other processors. Actual data communication is performed in the second phase.

The exchange phase generates a large number of messages in every superstep. Even when a processor does not have any outgoing message, it should inform other processors of the fact explicitly. Therefore, the overhead of the exchange phase is significant on message-passing architectures, especially when there is a large number of processors to synchronize and when the communication is slow compared to the computation. Although postponing communication until the end of local computation may increase the performance by making use of combining and reordering messages [7], congestion in the communication network is inevitable when large amounts of data are exchanged between processors.

[1]Cray T3D is an exception because it supports remote-memory accesses.

3. Relaxed Barrier Synchronization

In the BSP model, data accessed in a superstep should be one either held locally or came from other processors in the previous supersteps, which is defined as a *consistency rule*. The consistency rule should not be violated for the correct execution of any BSP program. Traditional barrier synchronization is a strict method to ensure the consistency rule by proceeding to the next superstep only when all the processors finish the current superstep. Instead, in our *relaxed barrier synchronization*, we eliminate the exchange phase of the barrier synchronization, but still make the execution of the BSP program produce consistent results.

Figure 2(a) and 3(a) illustrate two problematic cases that the bsp_put() request of P_0 arrives at P_1 for the code fragment in figure 1, when bsp_sync() does not perform global barrier synchronization. Now, processors may stay in different supersteps. For the correct execution, the bsp_put() issued in the superstep s in P_0 should make the value of v visible at the start of the superstep $s+1$ in P_1. Therefore, to be consistent in figure 2(a), P_0 should be delayed in bsp_sync() until the corresponding P_1 reaches the same superstep s.

The proposed scheme meets the consistency rule by a handshaking mechanism using special control messages called *req* and *ack*. Before sending a *data* message, P_0 issues a *req* message to the destination processor P_1, as shown in figure 2(b). A *req* message holds the current superstep number of the sender. P_1 compares the sender's superstep number s with its own superstep number $s-1$, and delays the acknowledgement since the request arrives too early. At the start of every bsp_sync(), processors check if they have any request that is formerly delayed and should be acknowledged in the current superstep. For such

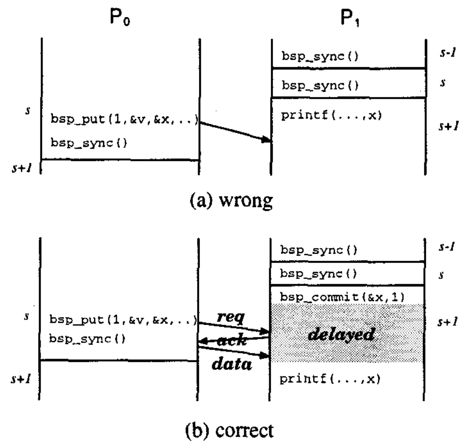

Figure 3. When the destination is too fast

requests, the processor sends an *ack* message and then receives the corresponding data. Hence, in our relaxed barrier synchronization, the superstep number is used as a timestamp which represents the speed of individual processors.

Unfortunately, this handshaking does not solve the problem shown in figure 3(a), where the request is delivered to the destination too late. P_1 should wait for P_0 until P_0 sends the needed data before actually using the value of x. However, P_1 does not know x should be received before starting the superstep $s + 1$, since no information is given. To remedy this problem, we introduce a primitive operation called `bsp_commit()`, which has the following syntax.

- `bsp_commit (void *addr, int n)`

`addr` should be the address of a previously registered data structure. We associate a count variable with each registered data structure, which is initially set to 0. As the content of the data structure is updated via messages, the count variable is increased by 1. The `bsp_commit()` asserts that the count variable of `addr` should be `n` before proceeding to the next computation. The value of `n` can be larger than 1 when a processor gathers data from more than one processors into the same data structure.

`bsp_commit()` should be placed just before the statement which accesses non-local data for all supersteps. By inserting `bsp_commit(&x, 1)` before the `printf()` statement as shown in figure 3(b), it is explicitly specified that the value of x should be updated once. Therefore, P_0 and P_1 synchronize each other not on the basis of supersteps, but on the basis of data dependency.

Most scientific and engineering applications have regular communication patterns and the location and the message count of `bsp_commit()` routine can be determined easily. BSPlib somewhat simplifies this task, as the destination of `bsp_put()` is always the address of the registered data structure. Therefore, it is enough to insert `bsp_commit()`

```
• bsp_put(pid,src,dest,offset,size)
  {   add the request to request_table;
      send a req message to pid;                                    }

• bsp_sync()
  {   for any request in acknowledge_table which should be
        acknowledged in the current superstep
      {   send an ack message and receive data messages;
          count(data)++;
          remove the request from acknowledge_table;            }
      while there are pending requests in request_table
          handle_messages(IN_SYNC);
      sstep_no++;                                                   }

• bsp_commit(addr,n)
  {   while (count(addr) < n)
          handle_messages(IN_COMMIT);
      count(addr) ← 0;                                              }

• handle_messages(in)
  {   receive a message;
      if (request message)
          if ((req_sstep_no < sstep_no) ||
            ((in == IN_SYNC) && (req_sstep_no == sstep_no)))
          {   send an ack message and receive data messages;
              count(data)++;                                    }
          else record the request in acknowledge_table;
      else if (acknowledge message)
      {   send data messages;
          remove the corresponding request from request_table;  }   }
```

Figure 4. Implementation of the relaxed barrier synchronization

just before the statement which accesses the registered data structure in each superstep. Moreover, collective communication routines such as broadcast, fold, scan, gather, scatter and sort, can be modified to use `bsp_commit()` inside of their routines, which reduces the chance of manual insertion to the program text. For the benchmark programs FT and LU used in section 4, it was enough to insert `bsp_commit()` to just one and twelve locations, respectively, to make them work with the relaxed barrier synchronization.

For some pointer-based applications, the destination processor may not know the exact number of incoming messages in a certain superstep. In such cases, the relaxed barrier synchronization can emulate the traditional implementation by explicitly exchanging the information on the number of messages before actual data communication.

Figure 4 outlines an implementation of the relaxed barrier synchronization on message-passing architectures. Each processor maintains two tables called *request_table* and *acknowledge_table*. They are used to record the requests issued from the current superstep and the requests from other processors that is being delayed, respectively. **count()** is a macro which returns the count value of the given registered data structure. *sstep_no* and *req_sstep_no* represent the superstep number of its own, and that of the requester's, respectively.

Incoming messages are handled either in `bsp_sync()`

or in bsp_commit(). The decision whether the request should be accepted or not is also affected by the location where the message is handled. In bsp_sync(), the request can be accepted if the requester's superstep number is less than or equal to the current superstep number. However, bsp_commit() does not allow the request from the same superstep.

So far, we have paid attention to bsp_put() routine only. We believe that bsp_get() can be replaced with equivalent bsp_put() routine without much effort.

4. Experimental Results

We have implemented and verified the relaxed barrier synchronization by modifying the Oxford BSP toolset, version 0.72α[2]. We used the BSP version of NAS Parallel Benchmark (NPB) 2.1 [6] as benchmark programs. The NPB suite consists of two kernels called MG and FT, and three simulated computational fluid dynamics (CFD) codes called LU, SP and BT. The BSP version is converted from the original MPI version by Antoine Le Hyaric in Oxford University[3]. Among those five programs, FT and LU in class A are used for the experiments in this paper.

To evaluate the effect of the communication speed, we run the same program on IBM SP2 using two different communication subsystems, US (User Space) and IP (Internet Protocol). When IP subsystem is used, the *High Performance Switch (HPS)* of SP2 can be shared with other IP jobs, but the communication cost is more expensive than that of US.

Figure 5 shows the average time spent for the synchronization in each processor (T_{sync}), where the left and the right column denote the performance of the original implementation (ORG) and that of the relaxed barrier synchronization (RBS), respectively. They are measured by the sum of the time spent in bsp_sync() for ORG, and in bsp_sync() and bsp_commit() for RBS. For ORG, we further divided the synchronization time into $T_{barrier}$ and T_{comm} to represent the time for the exchange phase and the communication phase, respectively. In reality, $T_{barrier}$ includes not only the duration of the exchange phase (T_{xch}), but also the waiting time caused by the variation in the completion times of the computation steps (T_{wait})[4]. In any case, the synchronization times were significantly reduced by using the relaxed barrier synchronization. For the system consisting of 32 processors, 61.5% (US) and 45.2% (IP) of the original synchronization time were reduced for FT, and 28.6% (US) and 49.0% (IP) for LU with the relaxed barrier synchronization.

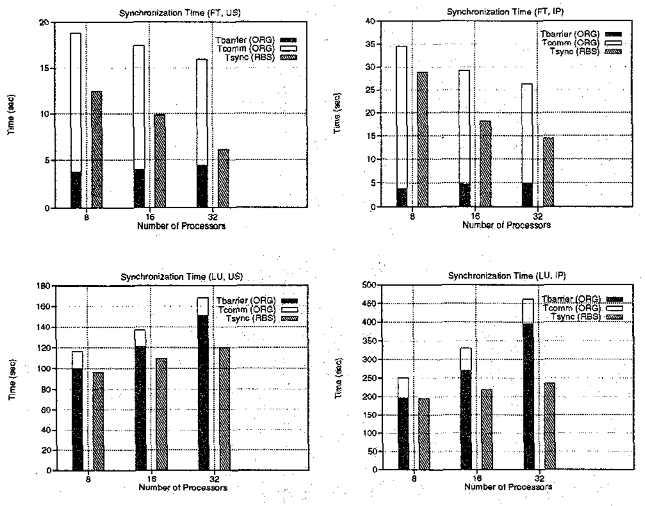

Figure 5. Changes in synchronization time

Figure 6 plots the relative speedup of the relaxed barrier synchronization with respect to the total execution time of the original implementation. In general, the computation time is reduced by almost half as the number of processors doubles. Therefore, as the number of processors increases, we can expect much more speedup because the synchronization time occupies the larger portion of the total execution time.

4.1 FT

Using the BSP cost model, we can estimate $T_{comm} = g \cdot \sum_{i=0}^{S-1} m_i = g \cdot M$, where m_i denotes the maximum size of messages generated for a superstep i, and S the total number of supersteps[5]. In FT, all-to-all exchange is performed to transpose an array, where each processor sends part of its data to every other processors. This results in the large value of M during a small number of supersteps.

In FT, the value of M for N processors is roughly decreased to $M/2$ for $2 \cdot N$ processors. g does not increase as fast as that rate, and this is the reason that T_{comm} decreases as N increases. On the other hand, $T_{barrier}$ slightly increases mainly due to the increase in T_{xch}.

Because the relaxed barrier synchronization eliminates the exchange phase, the upper bound of the difference in T_{sync} should be $T_{barrier}$. However, T_{sync} in RBS is always smaller than T_{comm} in ORG, even though it includes the waiting time. This means that the relaxed barrier synchronization not only eliminates T_{xch}, but also reduces the actual data transfer time, T_{comm}. The reason would be that the relaxed barrier synchronization reduces the possibility of con-

[2]It is freely available by anonymous ftp at ftp://ftp.comlab.ox.ac.uk/pub/Packages/BSP/.

[3]For details, refer to http://merry.comlab.ox.ac.uk/oucl/users/hyaric/doc/BSP/NASfromMPItoBSP/.

[4]Practically, T_{xch} and T_{wait} cannot be measured separately. Likewise, T_{sync} in RBS is the sum of T_{comm} and T_{wait}.

[5]Here, we do not normalize g with respect to processor speed.

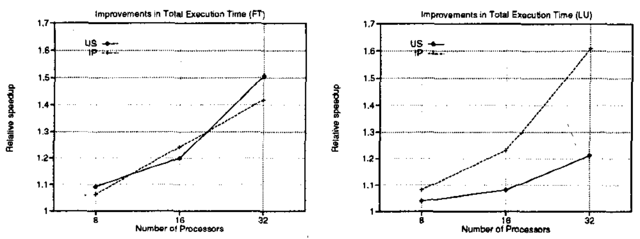

Figure 6. The relative speedup

gestion in the communication network by distributing the communication according to the relative speed of individual processors.

4.2 LU

The communication characteristics of LU is that small messages are generated and these messages are sent to east and south neighbors in a pipelined style. LU organizes a 2D grid of processors and each processor P_{ij} belongs to the rank $(i + j)$. The first processor P_{00} should finish its own local computations before sending the results to the processors of rank 1. In a second stage, P_{00} computes another block of data while P_{01} and P_{10} compute their first block and send it to rank 2 processors.

Unlike FT, T_{sync} increases as the number of processors increases and $T_{barrier}$ occupies most of T_{sync} in ORG. This is because that the number of supersteps and the pipeline depth increase as the number of processors increases. The increase in the number of supersteps results in the increase in T_{xch}. Also, the longer latency to fill the pipeline is added to T_{wait} as the pipeline depth increases.

The relaxed barrier synchronization improves the performance of LU by eliminating T_{xch}, which is substantial due to the large number of supersteps. In addition, it reduces T_{wait} as the synchronization occurs only between the processors which have data dependencies. For example, in the original implementation, P_{01} should be idle until all the processors finish the computation of the current stage. Instead, in the relaxed barrier synchronization, P_{01} can begin its own computation as soon as P_{00} finishes its computation.

5. Concluding Remarks

In this paper, we have introduced relaxed barrier synchronization for the efficient implementation of the BSP model on message-passing architectures. The proposed relaxed barrier synchronization preserves the consistency of a BSP program without global barrier synchronization.

Our approach is similar to [2] and [1], where they also employ a message counting scheme to trigger the beginning of new supersteps. However, a simple message counting scheme has a potential to produce inconsistent results if it just counts the number of incoming messages without controlling them based on their superstep numbers. Consider a situation where P_0 expects a message from P_1 at the superstep s and another from P_2 at the superstep $s + 2$. If the message of P_2 arrives at P_0 before the message of P_1, the simple message counting scheme will fail to preserve the consistency. This issue has not been addressed in the previous works. Another important distinction is that the previous works still synchronize on the basis of supersteps, while the relaxed barrier synchronization does on the basis of each data structure, which can maximize the overlap of communication and computation. In addition, their works are not presented in the framework of the proposed standard, BSPlib.

The benefits of the relaxed barrier synchronization can be summarized as follows. First, it eliminates the time required for the exchange phase, which can be substantial if a program consists of a large number of supersteps. Second, it reduces the waiting time because each processor can start its own computation as soon as data become available. Third, it reduces the network congestion by distributing the communication according to the relative speed of individual processors.

We are currently evaluating the relaxed barrier synchronization on other platforms including shared-memory architectures and network of workstations.

References

[1] R. D. Alpert and J. F. Philbin. cBSP: Zero-Cost Synchronization in a Modified BSP Model. Technical report, NEC Research Institute, 1997.

[2] A. Fahmy and A. Heddaya. Communicable Memory and Lazy Barriers for Bulk Synchronous Parallelism in BSPk. Technical Report BU-CS-96-012, Boston University, Sep. 1996.

[3] M. Goudreau, K. Lang, S. Rao, T. Suel, and T. Tsantilas. Towards Efficient and Portability: Programming with the BSP Model. In *Proc. 8th ACM Symp. on Parallel Algorithms and Architectures*, pages 1–12, 1996.

[4] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. Bisseling. BSPlib: The BSP Programming Library. Available at `http://www.bsp-worldwide.org/`, May 1997.

[5] J. M. D. Hill and D. B. Skillicorn. Practical Barrier Synchronization. Technical Report PRG-TR-16-96, Oxford University Computing Laboratory, 1996.

[6] W. Saphir, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.1 Results. Technical Report NAS-96-010, NASA Ames Research Center, Aug. 1996.

[7] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. Technical Report PRG-TR-15-96, Oxford University Computing Laboratory, Nov. 1996.

[8] L. G. Valiant. A Bridging Model for Parallel Computing. *Commun. ACM*, 33(8):103–111, Aug. 1990.