ELSEVIER

Information Processing Letters 66 (1998) 247–253

Information Processing Letters

Relaxed barrier synchronization for the BSP model of computation on message-passing architectures

Jin-Soo Kim [*], Soonhoi Ha, Chu Shik Jhon

Seoul National University, Department of Computer Engineering, Sinlim-dong, Kwanak-gu, Seoul 151, South Korea

Received 1 August 1997; received in revised form 2 February 1998
Communicated by S.E. Hambrusch

Keywords: Bulk Synchronous Parallel (BSP) model; Barrier synchronization; Message-passing architectures; Parallel processing

1. Introduction

The *Bulk Synchronous Parallel* (BSP) model of computation [9] was first proposed by Valiant as a bridging model between hardware and software for general-purpose parallel computation. The main objective of the model is to provide an abstract machine which allows the design of parallel programs that can be executed efficiently on a variety of architectures. A BSP abstract machine consists of a collection of N identical processors, each with local memory, connected by an interconnection network whose characteristics are modeled only by the barrier synchronization overhead (L), and the worst rate at which randomly-addressed data can continuously be delivered (g). 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.

In the BSP model, the computation is structured as a sequence of *supersteps*, each followed by a barrier synchronization. In each superstep, a processor performs operations on local data 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 [4]. However, the cost of the barrier synchronization on 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. A BSP programming model

Hill et al. have proposed BSPlib [5] as a standard library to integrate various approaches to BSP program-

[*] Corresponding author. Email: jinsoo@comp.snu.ac.kr.

PII: S0020-0190(98)00061-1

ming. One way of performing data communication in BSPlib is to use a Direct Remote Memory Access (DRMA) facility that provides operations to put/get data into/from the memory of a remote processor. In this section, we define a model for BSP programming which is based on BSPlib by the following minimal operations.[1]

- $SYNC_i$ performs a barrier synchronization in P_i;
- $PUT_i(j, r, v)$ puts data v in P_i into a region specified by r in P_j;
- $PUSHREG_i(r)$ registers a region r in P_i;
- $READ_i(r)$ reads a registered region r in P_i.

Assume that there are $P_0, \ldots, P_{N-1}$ processors. A *region* r defines contiguous memory locations, which can be represented by a tuple $r = \langle \alpha, \delta \rangle$, α and δ denote the start address of the region and its size, respectively. Because the same data structure is not necessarily stored at the same address in all processors, $PUSHREG_i(r)$ *registers* a region r so that it can be a target of data transfer between processors. Such registration maps an individual region r to a global index, denoted $\rho(r)$, that is the same in all processors. Once a region r is registered in P_i, it becomes a *registered region*.

For two regions $r = \langle \alpha, \delta \rangle$, $r' = \langle \alpha', \delta' \rangle$ in the same processor, r is a *subregion* of r', denoted $r \subset r'$, if and only if $\alpha' \leqslant \alpha$ and $\alpha + \delta \leqslant \alpha' + \delta'$. r and r' are *disjoint*, denoted $r \cap r' = \emptyset$, if and only if either $\alpha + \delta < \alpha'$ or $\alpha' + \delta' < \alpha$. Let R_i be a set of registered regions in P_i. A region $r_i = \langle \alpha_i, \delta_i \rangle$ in P_i can be mapped to a region r_j in P_j by a mapping function $\gamma_{i,j}(r_i)$, where $r_j = \gamma_{i,j}(r_i) = \langle \tilde{\alpha}_j + \alpha_i - \tilde{\alpha}_i, \delta_i \rangle$ for some $\tilde{r}_i = \langle \tilde{\alpha}_i, \tilde{\delta}_i \rangle \in R_i$ and $\tilde{r}_j = \langle \tilde{\alpha}_j, \tilde{\delta}_j \rangle \in R_j$ such that $r_i \subset \tilde{r}_i$, $\tilde{\delta}_i = \tilde{\delta}_j$ and $\rho(\tilde{r}_i) = \rho(\tilde{r}_j)$. The effect of $PUT_i(j, r, v)$ is to transfer data v into the contiguous memory locations, specified by a region $\gamma_{i,j}(r)$ in P_j.

Generally, a processor can freely read or write its registered regions like any other local data structures. However, we assume that the contents of the registered regions are changed only by *PUT* operations. We define an operation $READ_j(\tilde{r})$ to distinguish a processor's read accesses to the registered regions. Although there is no corresponding routine in BSPlib, in our model, $READ_j(\tilde{r})$ operation is conceptually added before the first instruction which accesses any data in $\tilde{r}$ for a superstep.

By executing $SYNC_i$ at a superstep s, P_i performs a barrier synchronization and proceeds to the next superstep $s + 1$. We use such notations as $SYNC_i^s$, $PUT_i^s(j, r, v)$, $PUSHREG_i^s(r)$, and $READ_i^s(r)$, to specify operations at a specific superstep s. We say two operations, *PUT* and *READ*, are *dependent* on $\tilde{r}_j$, denoted

$$PUT_i^s(j, r, v) \rightarrow READ_j^{s'}(\tilde{r}_j),$$

if and only if $READ_j^{s'}(\tilde{r}_j)$ is the first operation which satisfies $\gamma_{i,j}(r) \subset \tilde{r}_j$ and $s < s'$.[2]

Because the BSP model imposes some restrictions on the use of these operations, only the subset of BSP programs is meaningful. In this paper, we are interested in a class of BSP programs called *well-formed BSP programs* (*WFBP*).

Definition 1. A BSP program is a *well-formed BSP program (WFBP)*, if it satisfies the following conditions.

(1) For all $\tilde{r}, \tilde{r}' \in R_i$, $\tilde{r} \cap \tilde{r}' = \emptyset$ if $\tilde{r} \neq \tilde{r}'$.
(2) For all $PUT_i(j, r, v)$ in P_i, there exist $\tilde{r}_i \in R_i$ and $\tilde{r}_j \in R_j$ such that $r \subset \tilde{r}_i$ and $\rho(\tilde{r}_i) = \rho(\tilde{r}_j)$.
(3) For all $PUT_i^s(j, r, v)$ in P_i, there exists a $READ_j^{s'}(\tilde{r}_j)$ in P_j such that $PUT_i^s(j, r, v) \rightarrow READ_j^{s'}(\tilde{r}_j)$.
(4) If $PUT_i^{s_i}(j, r_i, v_i) \rightarrow READ_j^{s'}(\tilde{r}_j)$ and $PUT_k^{s_k}(j, r_k, v_k) \rightarrow READ_j^{s'}(\tilde{r}_j)$, then $\gamma_{i,j}(r_i) \cap \gamma_{k,j}(r_k) = \emptyset$, where $i \neq k$ or $s_i \neq s_k$.

Definition 1(2) shows that for a $PUT_i(j, r, v)$ operation, r should be a subregion of previously registered region $\tilde{r}_i$, which is unique by Definition 1(1). Moreover, the destination processors should have a registered region with the same global index as $\rho(\tilde{r}_i)$. Each *PUT* operation should have a matching *READ* operation at the later superstep, as specified in Definition 1(3). Note that for a *READ* operation, there can be more than one dependent *PUT* operations. However, as shown in Definition 1(4), the destination of those *PUT* operations should be disjoint in $\tilde{r}_j$, so as not to make some value lost or useless.

Now we define the *consistency* of a WFBP as follows.

[1] The first three correspond to `bsp_sync()`, `bsp_put()` and `bsp_push_reg()` routines of BSPlib, respectively.

[2] Note that s' need not be $s + 1$, the next superstep of s.

Definition 2. An execution of a WFBP is *consistent* if the value returned on *READ* operation is always the same value written by the dependent *PUT* operation. That is, between two dependent operations such that $PUT_i(j, r, v) \to READ_j(\tilde{r}_j)$, the value in a region $\gamma_{i,j}(r)$ in P_j should be v at the time of *READ* operation.

The main purpose of the barrier synchronization in the BSP model is to ensure the consistency by proceeding to the next superstep only when all the processors finish the current superstep. For a $PUT_i^s(j, r, v)$ operation, P_i does not start the superstep $s+1$ unless the value v is transferred to P_j. Therefore, it can be easily shown that $READ_j^{s'}(\tilde{r})$ operation in the subsequent superstep always accesses the correct value.

3. Relaxed barrier synchronization

For message-passing architectures, single-sided operation such as *PUT* can not be satisfied without the service of the destination processor, because there is no facility to access the remote memory directly. On the other hand, a processor does not know how many *PUT* operations it has to serve for other processors before proceeding to the next superstep.

To solve this problem, the current implementation of *SYNC* operation for message-passing architectures consists of two phases [8]. In the first phase, all processors exchange information about the number of *PUT* operations and their destination regions. 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 the 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 *PUT* operation, it should inform the 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.

Our approach is to relax the barrier synchronization constraint in the BSP model so that the exchange phase can be eliminated and the synchronization can occur at the time of reading registered regions only between the dependent processors. Processors are not globally synchronized any more, and each processor may execute a different superstep according to its relative speed and synchronization requirements.

We change the semantics of *READ* operation slightly and add an argument n such as $READ_j(\tilde{r}, n)$ to guarantee the consistency of a WFBP without resort to barrier synchronization. n is the number of dependent *PUT* operations for the *READ* operation, and it means that the registered region $\tilde{r}$ should be updated n times before being read. We associate a count value with each registered region $\tilde{r}$, denoted $\sigma(\tilde{r})$, which is initially set to zero. As the content of the registered region is updated via a message, $\sigma(\tilde{r})$ is increased by 1.

To prevent a registered region from being overwritten incorrectly, we use a handshaking mechanism using special control messages called *req* and *ack*. A message m sending from P_i to P_j is denoted by $m = \langle i, j, t \mid d_1, d_2, \ldots, d_n \rangle$, where $t \in T = \{req, ack, data\}$ denotes a message type and $d_1, d_2, \ldots, d_n$, the body of m. Before sending a *data* message, a processor issues a *req* message to the destination processor and receives an *ack* message. A *req* message holds the current superstep number of the sender, say s'. The destination processor accepts the request by sending an *ack* message back, if s' is less than its own superstep number. Otherwise, the acknowledgement is delayed until the destination processor reaches the superstep s'. Hence, the superstep number is used as a timestamp which represents the speed of each individual processor.

Fig. 1 outlines an implementation of the relaxed barrier synchronization on message-passing architectures. In Fig. 1, M_O and M_I are used to record the requests issued from the current superstep and the requests from the other processors that wait to be acknowledged, respectively. Incoming messages are handled either in *SYNC* or in *READ* operation. Note that a request from the same superstep is not accepted in *READ* operation.

Theorem 1. *For a given WFBP $\mathcal{P}$, let $\mathcal{P}'$ be a BSP program constructed by replacing $READ_j(\tilde{r})$ in $\mathcal{P}$ with $READ_j(\tilde{r}, n)$, where n is the number of dependent PUT operations for the READ operation.*

```
• $PUT_i^s(j, r, v)$
      Send $m_r = \langle i, j, req \mid reqid, s, \rho(\tilde{r}), \alpha - \tilde{\alpha}, \delta \rangle$ to $P_j$,
        where $r = \langle \alpha, \delta \rangle \subset \tilde{r} = \langle \tilde{\alpha}, \tilde{\delta} \rangle$, for some $\tilde{r} \in R_i$;
      $M_O \leftarrow M_O \cup \{(reqid, v)\}$;
      $reqid \leftarrow reqid + 1$;

• $SYNC_i^s$
      For each $m_i = \langle k, i, req \mid id, s', \rho', \Delta, \delta' \rangle \in M_I$
        such that $s = s'$,
      {
          $RECEIVE_i(k, id, \rho', \Delta, \delta')$;
          $M_I \leftarrow M_I - \{m_i\}$;
      }
      While $(M_O \neq \emptyset)$
          $HANDLE\_MESSAGE_i(s, 1)$;
      $s \leftarrow s + 1$;
      $reqid \leftarrow 0$;

• $READ_i^s(\tilde{r}, n)$
      While $(\sigma(\tilde{r}) < n)$
          $HANDLE\_MESSAGE_i(s, 0)$;
      $\sigma(\tilde{r}) \leftarrow 0$;

• $PUSHREG_i^s(r)$
      $R_i \leftarrow R_i \cup \{r\}$;

• $HANDLE\_MESSAGE_i(s, e)$
      For an incoming $m = \langle k, i, req \mid id, s', \rho', \Delta, \delta' \rangle$
          if $((s' < s) \vee (e = 1 \wedge s = s'))$
              $RECEIVE_i(k, id, \rho', \Delta, \delta')$;
          else
              $M_I \leftarrow M_I \cup \{m\}$;
      For an incoming $m = \langle k, i, ack \mid id \rangle$
      {
          send $m_d = \langle i, k, data \mid v \rangle$ to $P_k$,
            where $(id, v) \in M_O$;
          $M_O \leftarrow M_O - \{(id, v)\}$;
      }

• $RECEIVE_i(k, id, \rho', \Delta, \delta')$
      Send $m_a = \langle i, k, ack \mid id \rangle$ to $P_k$;
      Receive $m_d = \langle k, i, data \mid v \rangle$ from $P_k$;
      $r' \leftarrow v$, where $r' = \langle \tilde{\alpha} + \Delta, \delta' \rangle$ and $\rho(\tilde{r}) = \rho'$,
        for some $\tilde{r} = \langle \tilde{\alpha}, \tilde{\delta} \rangle \in R_i$;
      $\sigma(\tilde{r}) \leftarrow \sigma(\tilde{r}) + 1$;
```

Fig. 1. An implementation of relaxed barrier synchronization.

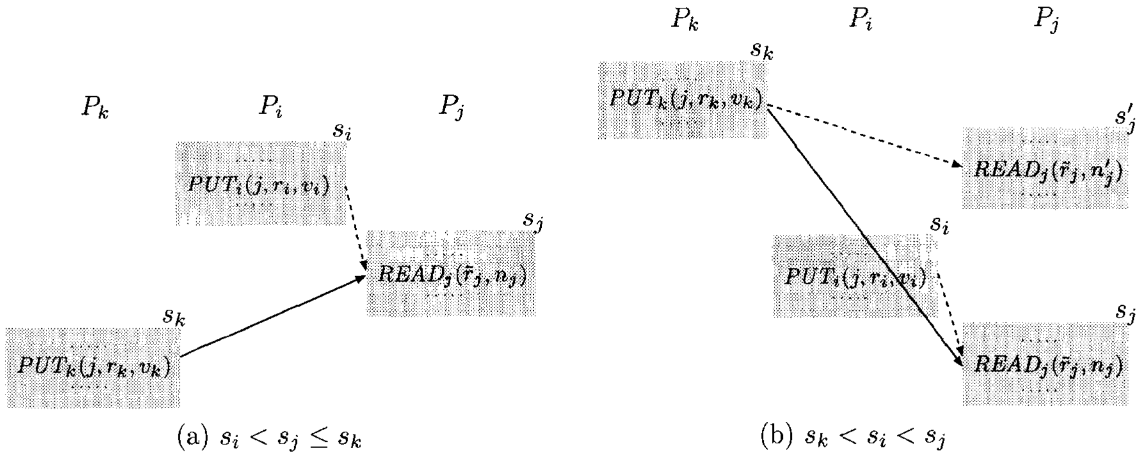

Fig. 2. Cases for accessing incorrect value in $READ_j^{s_j}(\tilde{r}_j, n_j)$.

Then, an execution of $\mathcal{P}'$ under the relaxed barrier synchronization in Fig. 1 is consistent.

Proof. Consider a pair of dependent operations such that $PUT_i^{s_i}(j, r_i, v_i) \rightarrow READ_j^{s_j}(\tilde{r}_j, n_j)$. To be inconsistent, $READ_j^{s_j}(\tilde{r}_j, n_j)$ should receive n_j messages, among which one is sent from another $PUT_k^{s_k}(j, r_k, v_k)$ operation, where $\gamma_{i,j}(r_i) \cap \gamma_{k,j}(r_k) \neq \emptyset$, $PUT_k^{s_k}(j, r_k, v_k) \not\rightarrow READ_j^{s_j}(\tilde{r}_j, n_j)$ and $\gamma_{k,j}(r_k) \subset \tilde{r}_j$. If s_k is between s_i and s_j, that is, $s_i \leqslant s_k < s_j$, $PUT_k^{s_k}(j, r_k, v_k)$ and $READ_j^{s_j}(\tilde{r}_j, n_j)$ will be dependent each other. Therefore, there are two cases, either $s_i < s_j \leqslant s_k$ or $s_k < s_i < s_j$, as shown in Fig. 2. (The dotted lines in Fig. 2 represent the dependency between *PUT* and *READ* operations.)

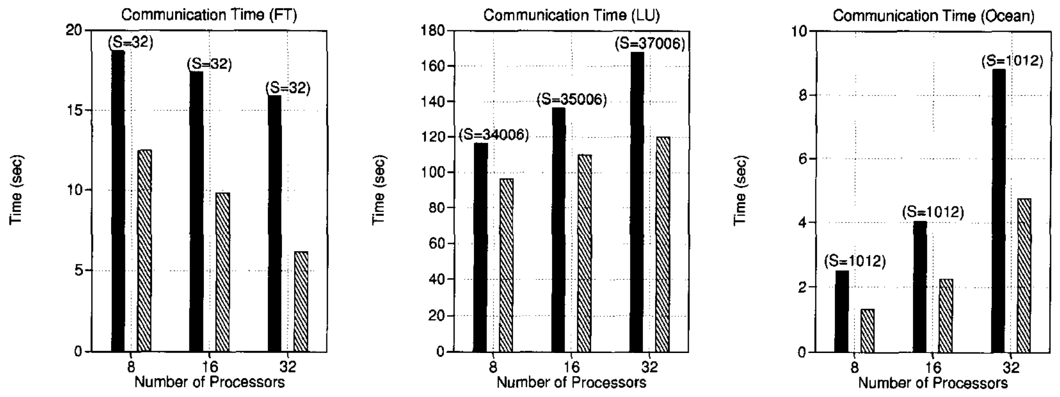

Fig. 3. The reduction in the communication time.

Case (1): $s_i < s_j \leqslant s_k$. In this case, although the *req* message of $PUT_k^{s_k}(j, r_k, v_k)$ may arrive at P_j before the superstep s_j, the *ack* message is delayed until the $SYNC_j$ operation at the superstep s_k, as shown in Fig. 1. Therefore, it is impossible for $READ_j^{s_j}(\tilde{r}_j, n_j)$ operation to read v_k.

Case (2): $s_k < s_i < s_j$. Without loss of generality, we assume that $READ_j^{s_j}(\tilde{r}_j, n_j)$ is the first inconsistent *READ* operation in P_j. For $PUT_k^{s_k}(j, r_k, v_k)$, there exists a $READ_j^{s'_j}(\tilde{r}_j, n'_j)$ such that

$$PUT_k^{s_k}(j, r_k, v_k) \to READ_j^{s'_j}(\tilde{r}_j, n'_j).$$

s'_j should be less than s_j, because otherwise $PUT_k^{s_k}(j, r_k, v_k)$ and $READ_j^{s_j}(\tilde{r}_j, n_j)$ will be dependent each other. Then, $READ_j^{s'_j}(\tilde{r}'_j, n'_j)$ is also inconsistent because it fails to receive the value v_k from its dependent $PUT_k^{s_k}(j, r_k, v_k)$ operation. This contradicts the assumption that $READ_j^{s_j}(\tilde{r}_j, n_j)$ is the first inconsistent *READ* operation in P_j.

From the cases (1) and (2), any *READ* operation reads the correct value under the relaxed barrier synchronization. □

4. Experimental evaluation

4.1. Results

We have implemented and verified the relaxed barrier synchronization on IBM SP2 by modifying the Oxford BSP toolset, version 0.72α.[3] Three benchmark programs, FT, LU and Ocean, were used in our experiments. FT and LU are parts of NAS Parallel Benchmark (NPB) 2.1. The original MPI version was converted to the BSP version by Antoine Le Hyaric in Oxford University.[4] Ocean is one of the barrier-intensive applications in SPLASH benchmark and was used for the BSP model in [3].

For the relaxed barrier synchronization, we have added a special primitive in BSPlib which corresponds to the *READ* operation. As we mentioned before, we insert this primitive just before the first statement which accesses a registered region in a superstep.

Fig. 3 shows the average time spent for the communication in each processor 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 *SYNC* for ORG, and in *SYNC* and *READ* for RBS. Note that they include not only the synchronization time but also the data transfer time. Numbers in parentheses represent the number of supersteps executed. In any case, the communication times were significantly reduced by using the relaxed barrier synchronization. For the system consisting of 32 processors, 61.5% of the original communication

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

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

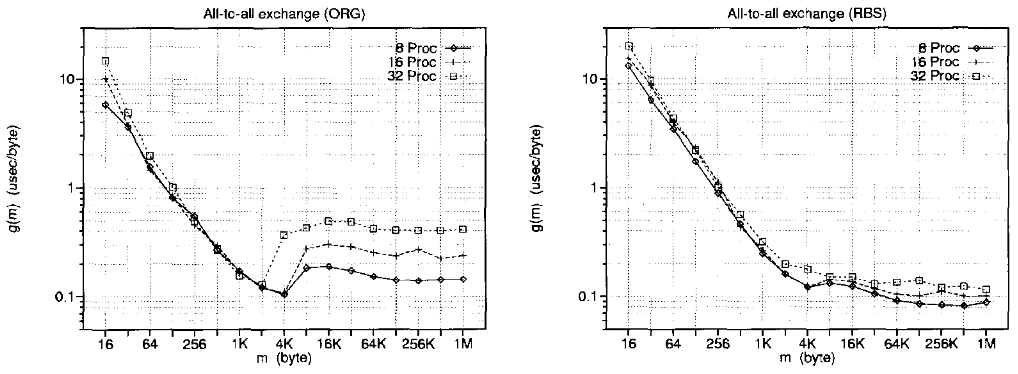

Fig. 4. BSP parameter g.

time was reduced for FT, 28.6% for LU, and 45.9% for Ocean with the relaxed barrier synchronization.

4.2. Impact on the BSP cost model

For a given BSP program consisting of S supersteps, the execution time is given by

$$\sum_{i=0}^{S-1} w_i + g \cdot \sum_{i=0}^{S-1} m_i + L \cdot S = W + g \cdot M + L \cdot S,$$

where w_i is the largest amount of work performed, and m_i the maximum size of messages generated for a superstep i.[5] Traditional implementation allows us to calculate the cost of a series of supersteps simply by summing the cost of each separate superstep. In the relaxed barrier synchronization, however, the communication occurs only between dependent processors and this will affect the cost model in several ways.

The most obvious effect is the elimination of the barrier synchronization cost L, because the relaxed barrier synchronization does not perform any global barrier synchronization. This cost, $L \cdot S$, can be substantial if there is a large number of processors or if S is large as in the case of LU. Normally, programmers attempt to compromise (1) the amount of local computation (W), (2) the size of messages (M), and (3) the number of supersteps (S), to minimize the execution time. However, under the relaxed barrier synchronization, S does not incur any overhead and programmers need not consider the number of supersteps.

[5] In this paper, we do not normalize g and L with respect to the processor speed.

Our scheme also affects the data transfer time. Fig. 4 plots $g(m)$, a unit cost to transmit a byte, as a function of message size m. It is measured using a microbenchmark in which each processor sends a single message of size m to all other processors. In both cases, $g(m)$ shows a saturated value $g(\infty)$ for large messages as expected in the BSP model. $g(\infty)$ is a good measure of the lower-bound rate of communication and it can be regarded as the BSP parameter g, because the BSP model defines g under conditions of continuous and bulk traffics. From Fig. 4, it is noted that the relaxed barrier synchronization slightly lowers the value of g. This means that our implementation is more efficient for the exchange of large messages, possibly due to the reduction in congestion delays. Actually, the improvement of FT in Fig. 3 mostly comes from the decrease in the data transfer time, because S is small compared with other applications.

In Fig. 4, the product $g(m) \cdot m$ remains roughly constant for small messages, which means that the startup cost dominates for short message transfers. In this case, although $g(m)$ of RBS is larger than that of ORG mainly due to the overhead of handshaking, the elimination of L can offset the overhead.

On the other hand, W includes the waiting time caused by the variation of the completion times of the computation steps. Because each processor starts its own computation as soon as data become available, the relaxed barrier synchronization can reduce W to some extent. However, this reduction of the waiting time is hard to characterize and varies from application to application.

5. Related work

Our approach is similar to a message counting scheme used in Split-C [6]. Split-C also has a single-sided *store* operation which stores a value into a global location, and a barrier operation called `all_store_sync()` which ensures all the store operations are complete. Because `all_store_sync()` incurs communication overhead, and prevents processors from working ahead on their computation until all other processors are ready, Split-C provides another operation called `store_sync(x)`, which waits only until `x` bytes have been stored locally.

Recently, a similar message counting scheme has been adopted for the BSP model to trigger the beginning of new supersteps [2,1]. However, a simple message counting scheme has a potential to produce inconsistent results if it just counts the number of incoming messages without considering their superstep numbers. Consider a situation where P_0 expects a message from P_1 at superstep s, and another from P_2 at 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 satisfy the consistency. This issue has not been addressed in the previous works for the BSP model. In [6], it is mentioned for Split-C that the higher level program protocol must avoid potential confusion because `store_sync()` does not indicate which data has been deposited.

MPI-2 also has a mechanism for providing regions of memory on which a processor waits until a one-sided communication accesses that memory [7]. To solve the aforementioned problem, MPI-2 uses the notion of *epoch*. A target memory region can be accessed only within an *exposure epoch* and there is a one-to-one matching between *access epochs* of origin processors and *exposure epochs* of destination processors. Because an exposure epoch is started and completed by synchronization calls executed in the target processor, the destination should be explicitly involved in the communication.

Another important distinction from [2,1] is that they still synchronize on the basis of supersteps, while the relaxed barrier synchronization does on the basis of each data structure. Let us assume that a processor needs two messages for a superstep. In the relaxed barrier synchronization, the processor can start the corresponding computation as soon as it receives one of those messages unless they belong to the same registered region. In this way, the relaxed barrier synchronization maximizes the overlap of communication and computation. However, the processor should wait until both of messages arrive in the previous works. In addition, their works are not presented in the framework of the proposed standard, BSPlib.

6. Concluding remarks

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

Although BSPlib defines another DRMA operation *GET*, we have paid attention to the *PUT* operation only. We believe that any *GET* operation can be replaced with equivalent *PUT* operation without much effort.

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

References

[1] R.D. Alpert, J.F. Philbin, cBSP: Zero-cost synchronization in a modified BSP model, Tech. Rept., NEC Research Institute, 1997.

[2] A. Fahmy, A. Heddaya, Communicable memory and lazy barriers for bulk synchronous parallelism in BSPk, Tech. Rept. BU-CS-96-012, Boston University, 1996.

[3] M. Goudreau et al., Towards efficient and portability: Programming with the BSP model, in: Proc. 8th ACM Symp. on Parallel Algorithms and Architectures, 1996, pp. 1–12.

[4] J.M.D. Hill, D.B. Skillicorn, Practical barrier synchronization, Tech. Rept. PRG-TR-16-96, Oxford University Computing Laboratory, 1996.

[5] J.M.D. Hill et al., BSPlib: The BSP programming library. Available at http://www.bsp-worldwide.org/, 1997.

[6] A. Krishnamurthy, D.E. Culler, A. Dusseau, S.C. Goldstein, S. Lumetta, T. von Eicken, K. Yelick, Parallel Programming in Split-C, in: Proc. of Supercomputing, 1993, pp. 262–273.

[7] Message Passing Interface Forum, MPI-2: Extensions to the message-passing interface, 1997. Available at http://www.mpi-forum.org/.

[8] D.B. Skillicorn, J.M.D. Hill, W.F. McColl, Questions and answers about BSP, Tech. Rept. PRG-TR-15-96, Oxford University Computing Laboratory, 1996.

[9] L.G. Valiant, A bridging model for parallel computing, Comm. ACM 33 (1990) 103–111.