Vector Prefix and Reduction Computation on Coarse-Grained, Distributed-Memory Parallel Machines

Seungjo Bae
Parallel Programming Section
ETRI
Taejon, Korea
sbae@computer.etri.re.kr

Dongmin Kim
Dept. of CIS
Syracuse University
Syracuse, NY
kimd@top.cis.syr.edu

Sanjay Ranka
Dept. of CISE
University of Florida
Gainesville, FL
ranka@cise.ufl.edu

Abstract

Vector prefix and reduction are collective communication primitives in which all processors must cooperate. We present two parallel algorithms, the direct algorithm and the split algorithm, for vector prefix and reduction computation on coarse-grained, distributed-memory parallel machines. Our algorithms are relatively architecture independent and can be used effectively in many applications such as Pack/Unpack, Array Prefix/Reduction Functions, and Array Combining Scatter Functions, which are defined in Fortran 90 and in High Performance Fortran. Experimental results on the CM-5 are presented.

1. Introduction

On coarse-grained, distributed-memory parallel machines the prefix and reduction primitives are collective communication primitives, in which all processors have to cooperate with each other [8, 2, 9, 7]. We can expand prefix and reduction to element-wise vector prefix and reduction. Suppose there are P processors, and each processor P_i has a local vector of size M, $V_i(0 : M-1)$, where $0 \leq i < P$. In *exclusive vector prefix* with binary operator $\oplus$ and its *identity* $\mathcal{I}_\oplus$, each processor P_i has the resulting vector $F_i(0 : M-1)$ such that $F_i(j) = \mathcal{I}_\oplus \oplus V_0(j) \oplus \cdots \oplus V_{i-2}(j) \oplus V_{i-1}(j)$ for all j $(0 \leq j < M)$. In *vector reduction* with binary operator $\oplus$, the resulting vector $R(0 : M-1)$ is stored in all processors. Hence $R(j) = V_0(j) \oplus V_1(j) \oplus \cdots \oplus V_{P-2}(j) \oplus V_{P-1}(j)$ for all j $(0 \leq j < M)$. In the remainder of the paper we will assume that prefix is exclusive.

In this paper we present two algorithms for vector prefix and reduction: the *direct algorithm* and the *split algorithm*. The direct algorithms are extensions of hypercube algorithms for prefix and reduction, respectively [8, 9]. Under the assumption that P is a power of two, in the split algorithm P full binary trees (FBT) of depth $\lg P$ are embedded to P processors, and then P reduction or prefix is simultaneously performed on the P FBTs of depth $\lg P$ [2, 7] When $M = \Omega(\lg P)$, the split algorithm is cost optimal.

This paper is organized as follows. The direct algorithms for vector prefix and vector reduction are presented in Section 2. In Sections 3 and 4 the split algorithms for vector prefix and vector reduction are presented, respectively. A new extended primitive, vector prefix-reduction, is introduced in Section 5. Experimental results are presented in Section 6 and, finally, our conclusion is stated in Section 7.

2. Direct algorithm

The direct algorithms for vector prefix and reduction are extensions of hypercube algorithms for prefix and reduction, respectively. The hypercube algorithms on $\lg P$-dimensional hypercube consist of $\lg P$ steps [8, 9]. Readers are referred to [8, 9] for detailed algorithms.

In the direct algorithms for vector prefix and reduction we use the same algorithms as hypercube algorithms for prefix and reduction, except that the message size increases M times. The time taken by communication in the direct algorithms is bounded by $\mathcal{O}(\lg P(\tau + \mu M))$ where τ is the start-up cost and μ is the per-element transfer time. Also, the local computation takes time $\mathcal{O}(2M \lg P)$ in prefix and $\mathcal{O}(M \lg P)$ in reduction.[1] Note that in the direct algorithm for prefix we need to maintain the reduction value at each step.

3. Split algorithm for vector prefix

The split algorithm consists of two stages, *split stage* and *union stage*, each of which consists of $\lg P$ steps. First of

[1]We use constants in complexity for the purpose of comparison, although the constants are insignificant in terms of asymptotic complexity.

all, we make the following assumptions in presenting the split algorithm:

- A binary operator $\oplus$ is associative and has identity $\mathcal{I}_{\oplus}$.
- P and M are powers of two, thus let $d = \lg P$ and $d_v = \lg M$. Later the extensions to the case in which P or M are not powers of two will be described.
- $[b_{d-1}b_{d-2}\cdots b_1 b_0]$ is the binary representation of an index for a processor or vector element. For bit positioning we regard the lowest and highest bits as bit 0 and bit $(d-1)$, respectively.
- Define an *inverse-level* of a tree, s, such that $s = d - l$ where l is a level in a tree and $0 \leq s, l \leq d$. Note that we designate the level of a root as 0.

In the following sections we will briefly describe the PRAM algorithms for prefix and reduction and then explain how to embed P FBTs of depth $\lg P$ in P processors. Finally, we will present in detail the split and union stages of the split algorithm for vector prefix. Note that, for the sake of simplicity, the split algorithm will be presented under the assumption that $M = P$, and then later the extensions to the case in which $M \neq P$ will be described.

3.1. PRAM algorithm for prefix and reduction

The PRAM algorithm for exclusive prefix consists of two stages, *upward traverse* and *downward traverse*. In the upward traverse all P input data are assigned to all leaves in order. Then, at each level of an FBT (from $d-1$ to 0), the value of node A, $R(A)$, is decided as follows:

$$R(A) \leftarrow R(left[A]) \oplus R(right[A]), \qquad (1)$$

where $left[A]$ and $right[A]$ are node A's left and right children [2, 7]. Therefore, the root (at level 0) will have the final resulting value for reduction. Note that the PRAM algorithm for reduction requires only one stage, *upward traverse*, and the result of the upward traverse in prefix is the same as that in reduction.

In the downward traverse the root is first initialized to $\mathcal{I}_{\oplus}$. Then, at each level of an FBT (from 0 to $d-1$), each node A with the current value $F(A)$ contributes the values of its two children as follows [2, 7, 9]:

$$\begin{aligned} F(left[A]) &\leftarrow F(A) \\ F(right[A]) &\leftarrow F(A) \oplus R(left[A]). \end{aligned} \qquad (2)$$

Readers are referred to [2, 7] for details of PRAM algorithms.

3.2. Embedding P full binary trees

Let T_i be an FBT of depth d. Then each FBT T_i is in charge of prefix on vector element i where $0 \leq i < P$. Each of P FBTs is embedded into P processors according to the following embedding rules:

1. Each processor P_i is mapped to the root of FBT $T_{\mathsf{rev}(i)}$.
2. In each FBT processor $P_i = P_{[b_{d-1}\cdots b_s b_{s-1} b_{s-2}\cdots b_0]}$ to be mapped to a node at inverse-level s $(0 < s \leq d)$ has a left and right child, and its children are mapped to $P_{[b_{d-1}\cdots b_s 0 b_{s-2}\cdots b_0]}$ and $P_{[b_{d-1}\cdots b_s 1 b_{s-2}\cdots b_0]}$, respectively.

Here the function $\mathsf{rev}()$ is a bit-reversal function such that the binary representation of an input value with a given width of bits is bit-reversed [4]. For example, if a bit-width is 4, $\mathsf{rev}(7) = 14$. In the split stage, d is the width of bits in $\mathsf{rev}()$.

After embedding P FBTs on P processors, it should be noted that at each inverse-level s (i.e., level $d - s$) each pair of two processors, (P_k, P_l), such that k xor $l = 2^s$, have the same two parents, P_k and P_l, where xor is a bitwise exclusive-or operation. Now we need to identify the FBTs to which a processor belongs at each level.

Theorem 1 *At level l (inverse level $s = d - l$) processor $P_i = P_{[b_{d-1}\cdots b_s b_{s-1} b_{s-2}\cdots b_0]}$ belongs to 2^l FBTs such that*

$$T_{[b_0\cdots b_{s-2} b_{s-1}\underbrace{*\cdots*}_{l}]},$$

where $0 \leq l \leq d$, $0 \leq i < P$, and each of lower l bits, $*$, is either 0 or 1.

Proof by induction on l:
[Induction base] $P_i = P_{[b_{d-1}b_{d-2}\cdots b_1 b_0]}$ belongs to $T_{\mathsf{rev}(i)} = T_{[b_0 b_1\cdots b_{d-2} b_{d-1}]}$ at level 0 by embedding rule 1.
[Induction hypothesis] Suppose at level l (inverse-level $s = d - l$) $P_{[b_{d-1}\cdots b_s b_{s-1} b_{s-2}\cdots b_0]}$ belongs to 2^l FBTs such that $T_{[b_0\cdots b_{s-2} b_{s-1}\underbrace{*\cdots*}_{l}]}$.
[Induction step] At level $l+1$ (inverse-level $s - 1 = d - (l+1)$) $P_{[b_{d-1}\cdots b_s b_{s-1} b_{s-2}\cdots b_0]}$ has two parents such as $P_{[b_{d-1}\cdots b_s 0 b_{s-2}\cdots b_0]}$ and $P_{[b_{d-1}\cdots b_s 1 b_{s-2}\cdots b_0]}$ according to embedding rule 2. Moreover, by induction hypothesis, two parents belong to $T_{[b_0\cdots b_{s-2} 0\underbrace{*\cdots*}_{l}]}$ and $T_{[b_0\cdots b_{s-2} 1\underbrace{*\cdots*}_{l}]}$ at level l, respectively. Therefore, $P_{[b_{d-1}\cdots b_s b_{s-1} b_{s-2}\cdots b_0]}$ belongs to 2^{l+1} FBTs such that $T_{[b_0\cdots b_{s-2}\underbrace{*\cdots*}_{l+1}]}$.

3.3. Split stage

In the split stage we traverse each FBT upward. At this time the operation in Equation (1) is performed on each FBT, which is needed to take care of each element of a resulting vector, since $M = P$. According to Theorem 1, at level l $(0 \le l \le d)$ each processor has to take care of 2^l elements of a vector.

As a preliminary step all elements of an input vector are copied to resulting vector R. Now the split stage consists of d steps, each of which corresponds to each pair of inverse-levels $(s, s+1)$ on FBTs where $0 \le s < d$. At the beginning of step s each processor $P_i = P_{[b_{d-1}\cdots b_s b_{s-1}\cdots b_0]}$ is initially in charge of the resulting subvector of size 2^l, $R_i^s([b_0 \cdots b_{s-1} \underbrace{* \cdots *}_{l=d-s}])$. Note that in the presentation of the split algorithm we will use the notations, R_i^s (for the split stage) and F_i^s (for the union stage), to describe the resulting subvectors on processor P_i, which corresponds to inverse-level s on FBTs.

If $b_s = 0$, P_i keeps $R_i^s([b_0 \cdots b_{s-1} 0 * \cdots *])$ (the first half), and sends $R_i^s([b_0 \cdots b_{s-1} 1 * \cdots *])$ (the second half) to its partner $P_{j=(i \text{ xor } 2^s)}$. On the other hand P_j keeps the second half of the current subvector and sends the first half to its partner P_i. If $b_s = 1$, P_i and P_j will take the place of P_j and P_i, respectively.

At each level l $(0 < l \le d)$ of FBTs in the union stage, the basic operation given in Equation (2) is used. Hence, after communicating with the partner in the split stage, each processor P_i must save the subvector received from its left child. More precisely, if $b_s = 0$, P_i should save the current keeping subvector: $Left_i^s(0 : 2^{l-1} - 1) \Leftarrow R_i^s([b_0 \cdots b_{s-1} 0 * \cdots *])$. Otherwise, P_i saves the message of size 2^{l-1} to be received from its partner: $Left_i^s(0 : 2^{l-1} - 1) \Leftarrow R_j^s([b_0 \cdots b_{s-1} 0 * \cdots *])$. Here notation $Left_i^s$ is used to describe the vector to be saved at step s on processor P_i, and $\Leftarrow$ is an element-wise vector assignment.

After saving $Left_i^s$, each processor combines the keeping subvector with the receiving subvector according to the given binary associative operator. Now, P_i is ready to take care of the new resulting subvector at step $s+1$, $R_i^{s+1}([b_0 \cdots b_s \underbrace{* \cdots *}_{l-1}])$, the size of which is reduced by half. After the final step $(s = d-1)$, processor P_i to be mapped to the root of FBT $T_{\text{rev}(i)}$ has the final resulting value $R(\text{rev}(i)) = R_i^d(\text{rev}(i))$.

3.4. Union stage

In the second (union) stage we traverse each FBT downward. First of all, as a preliminary step on each processor P_i, we need to initialize in the following way: $F_i^d(\text{rev}(i)) \leftarrow \mathcal{I}_\oplus$. The union stage consists of $d = \lg P$ steps, each of which corresponds to each pair of levels $(l-1, l)$ on FBTs where $0 < l \le d$. The union stage can be performed by reversing the sequence of operations in the split stage. It follows that each step of the union stage l $(0 < l \le d)$ corresponds to step s in the split stage such that $s = d - l$. Note that step l starts from 1, since it corresponds to step $d-1$ of the split stage.

At each step l $(0 < l \le d)$ in the union stage, each node $P_i = P_{[b_{d-1}\cdots b_s b_{s-1}\cdots b_0]}$ needs to perform operations defined in Equation (2). If $b_s = 0$, P_i first performs $F_i^{s+1}([b_0 \cdots b_{s-1} 0 * \cdots *]) \oplus Left_i^s(0 : 2^{l-1} - 1)$, and then sends the result to its right child, where $s = d - l$. Otherwise, only after P_i sends $F_i^{s+1}([b_0 \cdots b_{s-1} 1 * \cdots *])$ to its left child, does it perform $F_i^s([b_0 \cdots b_{s-1} 1 * \cdots *]) \Leftarrow F_i^{s+1}([b_0 \cdots b_{s-1} 1 * \cdots *]) \oplus Left_i^s(0 : 2^{l-1} - 1)$.

Two subvectors, the kept and received vectors, are then concatenated, which results in a new subvector of size $2^l = 2^{d-s}$. The size of a new resulting subvector is twice as large as those two initial subvectors. At the end of the union stage each processor P_i $(0 \le i < P)$ has a complete resulting vector $F(0 : M-1) = F_i^0(0 : M-1)$. Readers are referred to [1] for details of the algorithm.

3.5. Extensions

[Case I: $M > P$] Each FBT should be responsible for reduction on M/P vector elements. This can be done by evenly partitioning an input vector into continuous P subvectors, and then regarding each subvector of size $M/P = 2^{d_v - d}$ as one unit.

[Case II: $M < P$] $T_{[b_0 b_1 \cdots b_{d_v-2} b_{d_v-1} b_{d_v} \cdots b_{d-2} b_{d-1}]}$, is in charge of reduction on one vector element such as $[b_0 b_1 \cdots b_{d_v-2} b_{d_v-1}]$. That is, only the d_v most significant bits in the index of a FBT is used to identify the assigned vector element. It should be noted that during the first $(d - d_v)$ steps $(1 \le l \le d - d_v)$ in the union stage each processor does not need to communicate at all with its partner, and it needs to update the resulting value locally only if bit s of the processor number is one.

[Case III: P or M is not a power of two] If P is not a power of two, $P_v = 2^{\lceil \lg P \rceil} - P$ virtual processors need to be used, so $P + P_v$ FBTs of depth $\lceil \lg P \rceil$ can be embedded to $P + P_v$ processors. Since $P_v < P$, no actual processor needs to take care of more than one virtual processor. Therefore, the time taken by the split algorithm with P not being a power of two will be the same as that taken by the split algorithm with P being a power of two in terms of asymptotic complexity. Note that the input vector on each virtual processor must be initialized to identity values. If M is not a power of two, we need only to add some dummy vector elements.

3.6. Analysis

First of all let us assume that $M > P$. Then, at the beginning of step s $(0 \leq s < \lg P)$ in the split stage each processor has a subvector of size $2^{d-s}M/P = M/2^s$. One half of the subvector is sent to the partner, and then the other half, which will be kept, is updated by using the received message. Also, each node P_i must save a subvector of size $M/2^{s+1}$ received from its left child. Therefore, in the split stage the total size of messages or subvectors to be updated (or saved) at each processor is $\sum_{s=0}^{\lg P-1} \frac{M}{2^{s+1}} = M\left(1 - \frac{1}{P}\right) = \mathcal{O}\left(M\right)$, thus the cost for the split stage is bounded by $\mathcal{O}(\tau \lg P + \mu M + 2M)$.

In the union stage each processor needs to exchange a message with its partner and to perform the operation in Equation (2), thus the cost for the union stage is bounded by $\mathcal{O}(\tau \lg P + \mu M + M)$.

Similarly, the total cost for the split algorithm for vector prefix with $M < P$ is bounded by $\mathcal{O}\left(\tau \lg PM + \mu\left(2M + \lg \frac{P}{M}\right)\right) + \mathcal{O}\left(3(M + \lg \frac{P}{M})\right)$.

Comparison with direct algorithm The direct algorithm consists of $\lg P$ steps, while the split algorithm consists of $2 \lg P$ steps. Thus, the total number of communications in the split algorithm is at most two times greater than that in the direct algorithm. However, if P is relatively large, the time taken by local computation in the split algorithm will be less than that in the direct algorithm. Moreover, data transfer time, the term associated with μ, also will be less than that in the direct algorithm. Actually, if $M \gg \lg P$, the total cost for startup would be dominated by the data transfer time, even though the startup time, τ, is relatively larger than the per-element transfer time, μ, on parallel machines. Therefore, as P and M increase, the split algorithm will outperform the direct algorithm.

4. Split algorithm for vector reduction

Basically, the split algorithm for vector reduction can be performed in a manner similar to that presented in the split algorithm for vector prefix. At this time the operations given in Equation (2) should be changed to $R(left[A]) \leftarrow R(A)$ and $R(right[A]) \leftarrow R(A)$. Also, the final resulting value stored in the root after the split stage will be the initial value of the root in the union stage.

Since we do not deal with a vector, $Left$, the total cost for the split algorithm for vector reduction is bounded by, if $M \geq P$, $\mathcal{O}\left(2\left(\tau \lg P + \mu M\right)\right) + \mathcal{O}\left(M\right)$ and otherwise $\mathcal{O}\left(\tau \lg PM + \mu\left(2M + \lg \frac{P}{M}\right)\right) + \mathcal{O}\left(M + \lg \frac{P}{M}\right)$.

5. Vector prefix-reduction

In many applications such as PACK/UNPACK, Array Prefix and Reduction, and Array Combining Scatter Functions, which are defined in Fortran 90 and in High Performance Fortran [3, 6], we need to perform not only vector reduction, but also vector prefix with the same input vector [1]. Thus, in this section we define a new primitive, *vector prefix-reduction*, which is nothing but a combination of vector prefix and vector reduction.

Direct algorithm In the direct algorithm for vector prefix we maintain the reduction value at each step. Thus the direct algorithm for vector prefix can be used directly for vector prefix-reduction without any modification, and the cost will be the same as that for the direct algorithm for vector prefix.

Split algorithm The split stage for vector prefix can be used for vector prefix-reduction without any modification. Also, note that the basic structure of the second (union) stage in vector prefix is the same as that in vector reduction. Therefore, at each step in the union stage of the split algorithm for vector prefix-reduction, each processor needs only to send and receive two messages instead of one. Thus we have extra cost only in communication, not in local computation. Compared with the time taken by the split algorithm for vector prefix, the extra cost entailed in combining two primitives will be $\mathcal{O}(\tau \lg P + \mu M)$ if $M \geq P$, and otherwise will be $\mathcal{O}(\tau \lg M + \mu M)$.

6. Experimental results

All six algorithms presented in this paper were programmed in C on the CM-5. To evaluate the performance, the experiments were conducted for various M and P such that $M = 2^i$ and $P = 2^j$ where $2 \leq i \leq 12$ and $2 \leq j \leq 7$. Associative binary operator $+$ was used in the experiments, where the type of each vector element was integer. Also, active messages were used in one-to-one communication [10, 5]. We measured total execution time for each algorithm except the direct algorithm for vector prefix-reduction, since it is the same as the direct algorithm for vector prefix. Notice that only partial results are presented in Figure 1 due to space limitation. Full results are available in [1].

Effect of M Each plot in Figure 1 shows the total execution time (in milliseconds) for each algorithm with a fixed P. Once P is equal to or greater than 16, the slopes of all split algorithms are less than those of all direct algorithms.

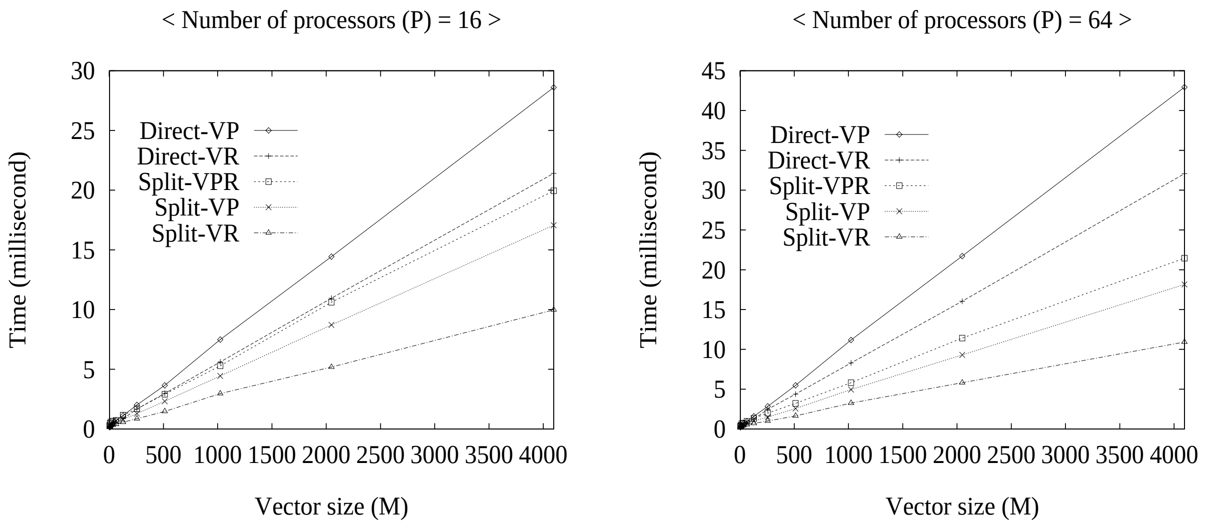

Figure 1. Performance evaluation of direct and split algorithms for vector prefix (VP), reduction (VR), and prefix-reduction (VPR) when $P = 16, 64$

Therefore, M is relatively large, all split algorithms outperform the direct algorithms. Moreover, as M increases, the split algorithms will have a substantially better performance as compared to the direct algorithms.

Effect of P As stated before, the performance of the direct algorithm is largely affected not only by P, but also by M, while that of the split algorithm mainly depends on M. In the direct algorithm we can see that a higher P shows a higher rate of increase in execution time. Thus, given a vector of size M, the time taken by the direct algorithm increases significantly with P. As M increases, the cost of the direct algorithm is more substantially affected by the increase of P. On the other hand, in the split algorithm, once P is relatively large, the slope does not increase significantly with P.

7. Conclusions

In this paper we have presented two algorithms for vector prefix and reduction: the *direct algorithm* and the *split algorithm*. The performance of the direct algorithm is greatly affected not only by P, but also by M, while that of the split algorithm mainly depends on M. Thus when P and M are relatively large, the split algorithm will give the better performance. If $M = \Omega(\lg P)$, the split algorithm is cost optimal and outperforms the direct algorithm by a factor of $\lg P$ in terms of asymptotic complexity. When P and M are relatively small, the direct algorithm will be a feasible option.

In addition, we have explained how to perform vector prefix and vector reduction simultaneously when two primitives use the same input vector.

Acknowledgments

The work of Sanjay Ranka was supported by AFMC and ARPA under contracts F19628-94-C-0057 and WM-82738-K-19 and a subcontract from Syracuse University. The content of the information does not necessarily reflect the position or the policy of the United States government and no official endorsement should be inferred.

References

[1] S. Bae. *Runtime Support for Unstructured Data Accesses on Coarse-Grained Distributed-Memory Parallel Machines*. PhD thesis, Syracuse University, August 1997.

[2] G. E. Belloch. Prefix Sums and Their Applications. Technical report, School of Computer Science, Carnegie Mellon University, November 1990.

[3] W. S. Brainerd, C. H. Goldberg, and J. C. Adams. *Programmer's Guide to FORTRAN 90*. McGraw Hill, 1990.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[5] T. M. Corporation. *CMMD version 3.0 Reference manual*, May 1993.

[6] High Performance Fortran Forum. *High Performance Fortran Language Specification: Version 1.1*, November 1994.

[7] J. JáJá. *An introduction to Parallel Algorithms*. Addison-Wesley Publishing Company, 1992.

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

[9] M. J. Quinn. *Parallel Computing: Theory and Practice*. McGraw Hill, 1994.

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