12-1-1994

Data Parallel Algorithms

Howard Jay Siegel
Purdue University School of Electrical Engineering

Lee Wang
Purdue University School of Electrical Engineering

John John E. So
Purdue University School of Electrical Engineering

Muthucumaru Maheswaran
Purdue University School of Electrical Engineering

DATA PARALLEL ALGORITHMS

HOWARD JAY SIEGEL
LEE WANG
JOHN JOHN E. SO
MUTHUCUMARU MAHESWARAN

TR-EE *94-38*
DECEMBER 1994

Data Parallel Algorithms

to appear as a chapter in:

Handbook of Parallel and Distributed Computing

edited by Albert Y. Zomaya, McGraw-Hill, 1995

Howard Jay Siegel
Lee Wang
John John E. So
Muthucumaru Maheswaran

Parallel Processing Laboratory
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907-1285, USA
{hj, lwang, johnjohn, rnaheswar)@ecn.purdue.edu

December 1994

Purdue University
School of Electrical Engineering
Technical Report TR-EE 94-38

This research was supported by NRaD under contract number N68786-91-D-1799 and by a Fullbright Scholarship.

Data Parallel Algorithms

Howard Jay Siegel. Lee Wang. John John E. So. and Muthucumaru Maheswaran

Purdue University

Page

Abstract

Data parallelism is a model of parallel computing in which the same set of instructions is applied to all the elements in a data set. **A** sampling of data parallel algorithms is presented. The examples are certainly not exhaustive, but address many issues involved in designing data parallel algorithms. Case studies are used to illustrate some algorithm design techniques; and to highlight some implementation decisions that influence the overall performance of a parallel algorithm. It is shown that the characteristics of a particular parallel machine to be used need to be considered in transforming a given task into a parallel algorithm that executes effectively.

DATA PARALLEL ALGORITHMS

Howard Jay Siegel, Lee Wang, John John E. So, and Muthucumaru Maheswaran

Purdue University

1. OVERVIEW

As the size, hardware complexity, and programming diversity of parallel systems continue to evolve, the range of alternatives for implementing a task on these systems grows. Choosing a parallel algorithm and implementation becomes an important decision, and the choice has a significant impact on the execution time of the application.

Data parallelism is a model of parallel computing in which the same set of instructions is applied to all the elements in a data set [Mas91, Wil93]. A sampling of data parallel algorithms is presented. The examples are certainly not exhaustive, but address many issues involved in designing data parallel algorithms. Case studies are used to illustrate some algorithm design techniques and to highlight some implementation decisions that influence the overall performance of a parallel algorithm. It is shown that the characteristics of a particular parallel machine to be used need to be considered in transforming a given task into a parallel algorithm that executes effectively.

This report focuses on algorithm design techniques for mapping data parallel algorithms onto large scale (i.e., 2^6 to 2^{16} processors) distributed memory parallel machines. The ***SIMD*** (single instruction stream – multiple data stream) model of parallelism is used to demonstrate the techniques presented; however, the methods can be used with the ***MIMD*** (multiple instruction stream – multiple data stream), the ***SPMD*** (single program – multiple data stream, which is

This research was supported by NRaD under contract number N68786-91-D-1799 and by a Fulbright Scholarship.

a subclass of MIMD), and the ***mixed-mode*** (hybrid SIMD/MIMD) models of parallelism as well.

Data parallel programs typically exhibit a high degree of *uniformity* (operations to be performed are uniform across the data set) and are often well suited to the SIMD model because only a single instruction stream is necessary [BeS91, Jam87]. Implementing these applications on SIMD machines is often more cost effective than solving them using MIMD machines [HiS86]. Among the advantages of the SIMD mode of parallelism that can be exploited are implicit synchronization that allows more efficient inter-processor communication, the ability to overlap scalar operations on the control unit with the operations on the processing elements, and the need for only one program [BeS91, SiA92c]. Mixed-mode processing and the trade-offs between the SIMD and MIMD modes of parallelism are discussed further in [SiA95].

In Section 2, the SIMD machine model used in this report and other machine models are discussed. The choices for data distribution among the processing elements are explored in Section 3. The effect of the data distribution on execution time is demonstrated using an image smoothing algorithm. In addition, the impact of the interconnection network topology on the number of data transfers required to perform the computations is also studied. It is shown that the optimum data distribution is dependent on the architecture of the machine in use and the application to be implemented in parallel. Section 4 examines overlapping the operations of the control unit and the processing elements and uses an image correlation algorithm to illustrate how this overlap can be optimized.

Section 5 discusses parallel reduction operations. A sampling of matrix and vector operations is covered in Section 6. Parallel implementations of matrix transpose, matrix-by-matrix multiplication, and matrix-by-vector multiplication are presented. In Section 7, the effect on execution time of increasing the number of processors used (scalability of the algorithm) and the impact of partitioning the system for subtask parallelism are explored. It is shown that increasing the number of processors used does not always yield a decrease in execution tirne or an increase in system efficiency. In Section 8, the computation of multiple quadratic forms is used as an illustrative example on how scalability can be achieved using a suite of algorithms.

2. MACHINE MODEL

As stated previously, the data parallel algorithm studies will be based on the SIMD machine model [Fly66], and most of the results will also be applicable to the SPMD model on MIMD machines and to both models on mixed-mode systems. In the SIMD model, only a single sequence of instructions is present, but each instruction is executed simultaneously in an arbitrary set of processors, with each processor operating on its own data. Typically, an SIMD machine consists of N processors, N memory modules, a CU (control unit), and an interconnection network. Each processors is paired with a memory module to form a *PE* (processing element). The CU broadcasts instructions in sequence to the PEs and all enabled PEs execute the same instruction at the same time. Thus, there is a single instruction stream. Each enabled PE executes the instructions on the data in its own associated memory module, resulting in multiple data streams. The interconnection network allows communication among the PEs [Sie90]. Examples of SIMD systems that have been built include the Thinking Machines CM-2 [Hil85], DAP [Hun89], Illiac IV [BaB68], MasPar MP-1 [Bla90] and MP-2, MPP [Bat80], and STARAN [Bat74, Bat77].

Figure 1 shows the SIMD machine model used in this report. This model is referred to as a *PE–to–PE configuration* [Sie90] or *physically distributed memory organization.* There are $N = 2^n$ PEs and the PEs are numbered from 0 to N – 1. The PE's memory is only used to store data, not instructions.

The CU is made up of a processor, a memory, and an instruction broadcast queue. The CU fetches and decodes the program instructions from its own memory. Only one program exists, a portion of which is to be executed on the CU and the other portion of which is to be executed on the PEs. Typically, the CU executes the control flow instructions, e.g., loop indexing, and broadcasts the data processing instructions to the PEs. The disabled PEs must remain idle while the instruction that was broadcast from the CU is executed in enabled PEs.

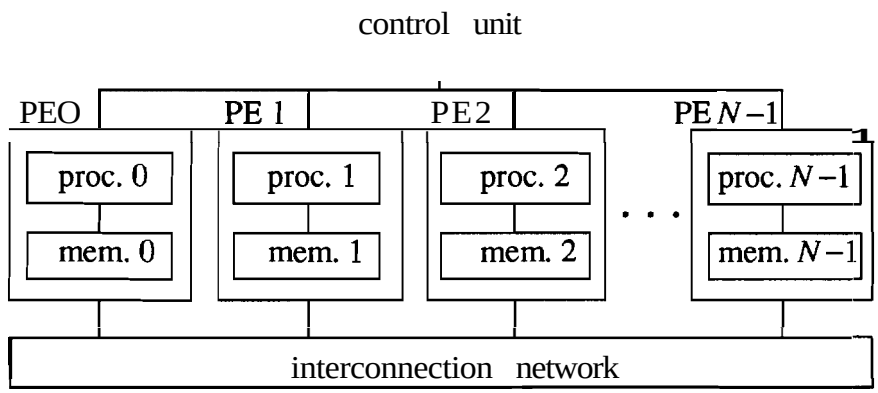

Figure 1: Distributed memory model of an SIMD machine.

As an example, consider the execution of the ***if–then–else*** construct, where the if-conditional involves data local to each PE. First, the enable status of the PEs prior to the execution of the if-then-else construct is saved. All disabled PEs will remain disabled during the execution of the if-then-else construct. An enabled PE is kept enabled for the "then" clause if the result of the if-conditional test is true for the PE. Otherwise, it is disabled for the "then" clause. Then the instructions of the "then" clause are broadcast from the CU and executed by the enabled PEs. Next, the PEs that were enabled for the "then" clause are disabled and the PEs enabled prior to the if-then-else construct, but disabled for the "then" clause, are enabled for the "else" clause. The CU broadcasts the instructions of the "else" clause and the enabled PEs execute these instructions. At the end of the if-then-else construct, the previously saved enable status is restored, i.e., each PE returns to the enable state it was in before the execution of the if-then-else construct.

The CU broadcasts a single stream of instructions from the instruction broadcast queue to the PEs in the computational engine. It will not issue the next instruction until all the enabled PEs have completed executing the current instruction, thereby implicitly synchronizing the PEs

at instruction-level granularity. Once the CU processor determines the instructions that should be placed in the queue, it may proceed with its own execution while the queued. instructions are broadcast to the PEs. This concurrent CU and PE execution is called *CU/PE* overlap [KiN91, SiS95], which is discussed in Section 4.

Recall that most methods discussed in this report can also be adopted for SPMD operation on existing MIMD machines (e.g., Thinking Machines CM-5 [HiT93], nCUBE 2 [HaM89], Intel Paragon, and Cray T3D) and mixed-mode systems (e.g., OPSILA [DuB88], PASM [SiS95], and Triton/1 [PhW93]). In contrast to the SIMD machine model, the MIMD machine model [Fly66] does not have a CU, but consists of N independent PEs. The PEs are connected to one another via an interconnection network, as in the SIMD case. The PE memory stores the program in addition to the data. Each PE executes its own instruction stream, and all PEs operate asynchronously with respect to one another. Any synchronization among PEs needed has to be explicitly specified in the program. The SPMD model is a subclass of the MIMD model [DaG88]. In the SPMD model, all PEs execute the same program asynchronously. Each of the PEs takes its own control path during the course of the execution. In a mixed-mode system [FiC91], the PEs can switch between the SIMD and the MIMD modes of parallelism at instruction-level granularity

There is a great variety of interconnection networks that can be used in parallel machines (e.g., see [NaS93, Sie90, VaR94]). Considering the impact of network selection on each algorithm described is beyond the scope of this report. Therefore, unless otherwise specified, a network that efficiently supports the inter-PE communications needed is assumed for each algorithm. Furthermore, unless otherwise specified, inter-PE communication will be measured in number of data words sent; other metrics, not considered here, include number of distinct network path establishments and size of the data items transferred. An example of the impact of the network selected is included in Section 3 to introduce the issues involved.

3. IMPACT OF DATA DISTRIBUTION

3.1. Introduction

One issue that must be addressed in designing data parallel algorithms is how data should be mapped across the PEs. If the mapping of the data across the PEs is optimized,,it is possible to gain significant performance improvements for a particular algorithm [BlN92].

One class of problems that exhibit data parallelism are window-based tasks. Examples of these tasks include image correlation [ArN91, SiS82], image smoothing [SiA92c, SiS81], and range image segmentation [GiW92]. This section demonstrates the impact of data distribution across the PEs on the computation time and the communication time for a given algorithm. Image smoothing is used as a representative of window-based algorithms to illustrate how varying the mapping across the PEs can affect the execution time.

3.2. The Serial Image Smoothing Algorithm

Smoothing is a procedure applied to an image to reduce noise. An M × M image A is stored in memory as a two-dimensional array (matrix) where each element, called a picture element, or pixel, is an integer whose value represents the gray-level intensity of the corresponding point in the discretized image. For each non-edge pixel (i, j) in A, a 3 × 3 window centered at (i, j) is used to generate the corresponding pixel in the M × M smoothed image A'. A serial algorithm to accomplish this is:

for i = 1 to M−2 do

for $j = 1$ to $M-2$ do

$$A'(i,j) = [A(i-1,j-1) + A(i,j-1) + A(i+1,j-1) + A(i-1,j) + A(i,j) +$$
$$A(i+1,j) + A(i-1,j+1) + A(i,j+1) + A(i+1,j+1)] / 9$$

In the case of an edge pixel, no calculation is performed and the pixel itself is taken to be the smoothed value. Because there are 4M − 4 edge pixels in the M × M image A, the serial time

complexity is the time to execute $M^2 - (4M - 4) = O(M^2)$ smoothing operations. For M = 4,096, this is 16,760,836 smoothing operations, (approximately M^2).

3.3. Parallel Implementation Using Square Subimages

In this report, it is assumed that when implementing an algorithm on a parallel machine, the goal is to minimize the execution time. Factors that, in general, help to do this include decreasing inter-PE communications and balancing the workload among the PEs so that as many PEs as possible are concurrently executing the algorithm.

Assume N PEs are logically arranged as a $\sqrt{N} \times \sqrt{N}$ grid. The M × M image A is mapped onto the PEs by superimposing the image onto the PE grid. Thus, eaclh PE stores an $(M/\sqrt{N}) \times (M/\sqrt{N})$ square subimage, as shown in Figure 2(a). Therefore, each PE performs up to M^2/N smoothing operations. All PEs smooth their subimages simultaneously, i.e., at most N smoothing operations can be performed concurrently at each step, one in each PE.

To smooth the pixels at the edge of a subimage, pixels from spatially adjacent subimages must be transferred, as shown in Figure 2(b). PE i requires at most $M/\sqrt{N}$ pixels from each of the four PEs directly adjacent to it and one pixel from each of the four PEs diagonally adjacent to it. Thus, at most, $4M/\sqrt{N} + 4$ inter-PE data transfers are required per PE to smooth the entire image A. Just as the PEs can all smooth simultaneously, all N PEs can transfer data simultaneously (recall from Section 2 that an appropriate network is assumed to support this). The time complexity of the parallel algorithm when operating on an M × M image A with N PEs is the sum of M^2/N smoothing operations and $4M/\sqrt{N} + 4$ inter-PE data transfers (where up to N smoothing operations or up to N data transfers can occur concurrently). Assuming that the time to perform one transfer is equal to τ times the time to perform one smoothing operation, the speedup, S, of the parallel version over that of the serial version of the algorithm is defined as:

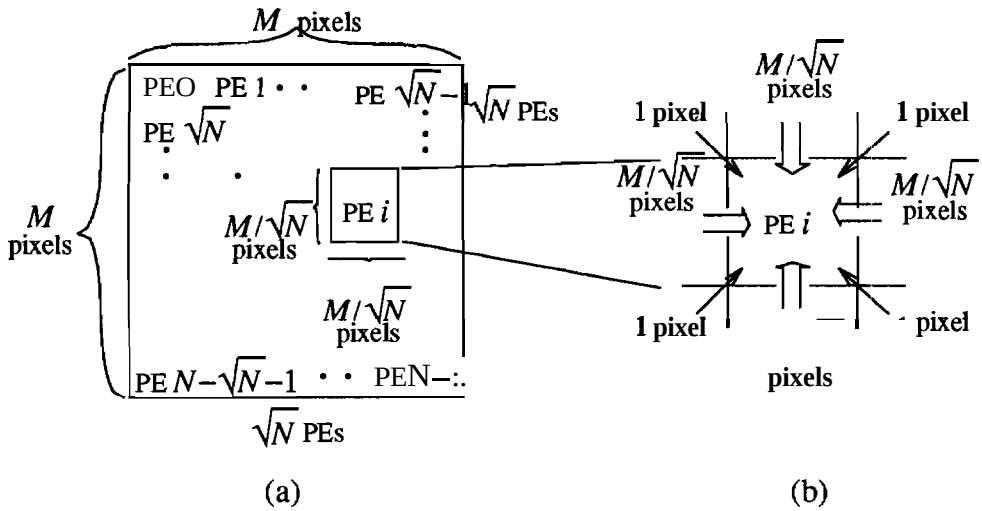

Figure 2: (a) Data allocation and (b) pixel transfers for image smoothing using square subimages.

$$S = \frac{\text{serial time complexity}}{\text{parallel time complexity}} = \frac{(M-2)^2}{\frac{M^2}{N} + \tau\left(\frac{4M}{\sqrt{N}} + 4\right)}$$

For example, if $M = 4{,}096$ and $N = 256$, the number of inter-PE data transfers required is 1,028 and 65,536 smoothing operations are performed. If $\tau = 1$, the speedup is $4094^2/66{,}564 = 252$. (The value of τ is machine architecture dependent and could be less than, equal to, or greater than 1.)

In the above speedup calculation, the complexities are computed by only counting the smoothing operations and the inter-PE data transfers. It is assumed that the uni-processor machine that executes the serial algorithm and each processor in the parallel machine are of equivalent computing power. Theoretically, the maximum possible speedup is N. This is achieved when the total workload is equally distributed among the PEs, no overhead (e.g. no

inter-PE data transfers) is incurred, and all PEs are always active during the execution. If the time to perform an inter-PE data transfer becomes much less than the time to perform a smoothing operation, which is the case, for example when using the Xnet in the MasPar MP-1 [Bla90], the speedup will be closer to N. The speedup will never equal N even if inter-PE communication is ignored because the PEs containing the edge pixels of the M × M image A will be disabled for some smoothing operations and are therefore underutilized for some steps of the algorithm. This demonstrates that inter-PE data transfers and *masking*, i.e., disabling some PEs for some operations, result in a less than perfect speedup [SiA92c].

Consider the number of inter-PE data transfers required per PE. Instead of using square subimages, suppose the M^2/N pixels are mapped to each PE such that the subimage in a PE has r pixels per row and c pixels per column, where $rc = M^2/N$. To minimize the number of inter-PE data transfers per PE, which is $2r + 2c + 4$, replace r with $M^2/(cN)$ and minimize the expression with respect to c:

$$\frac{d}{dc}(2r + 2c + 4) = \frac{d}{dc}\left[\frac{2M^2}{cN} + 2c + 4\right] = 0$$

This yields $c = M/\sqrt{N}$, subject to the constraint that c is an integer. It follows that $r = M/\sqrt{N}$. Thus, $r = c$, i.e., each PE should contain a square subimage [SiS82]. For example, if M = 4,096 and N = 256, each PE is assigned $M^2/N = (4{,}096)^2/256 = 65{,}536$ pixels. If the pixels are mapped to each PE as a 256×256 ($r = c = 256$) square subirnage, the number of inter-PE data transfers needed is $4 \times 256 + 4 = 1{,}028$. If instead the 65,536 pixels are mapped to each PE as a $64 \times 1{,}024$ ($r = 64$, $c = 1{,}024$ or $r = 1{,}024$, $c = 64$) rectangular subimage, the number of inter-PE data transfers needed is $2 \times 1{,}024 + 2 \times 64 + 4 = 2{,}176$.

3.4. Parallel Implementation Using the Horizontal (Row) Stripe Method

In the preceding subsection, it is shown that the way data is distributed across the PEs dictates the number of inter-PE data transfers required. An alternative method of mapping data among the PEs is based on distributing consecutive rows of data, as opposed to square subimages, to the PEs. This approach results in a decreased number of calculations performed per PE compared to using square subirnages.

Recall that in the image smoothing algorithm, the border (edge) pixels of image A do not require any smoothing operations. In general, for some window-based image processing tasks these border pixels may play an important role in the selection of the data distribution method. The ***horizontal stripe method*** allows window-based algorithms to take advantage of the fact that no calculations need to be performed on the column border pixels by distributing the column border pixels evenly among all the PEs, thereby decreasing the total number of :required calculations [GiW92]. The square subimage method does not take advantage of this fact because the border pixels are distributed unevenly among the PEs; some PEs will not get any border pixels and consequently must perform the maximum number of smoothing operations (M^2/N). These PEs dictate the time required to perform the necessary computations to finish the: task.

Using the same image smoothing algorithm discussed in the previous subsection, the M × M image A is divided into N rectangular (MIN) × M subimages, where M ≥ N, as shown in Figure 3. Each PE stores M/N rows by M columns of pixel data. The number of pixels assigned to each PE remains unchanged (still equal to M^2/N). However, PE i will now require a total of 2M pixels from its two neighboring PEs; M pixels will be required from PE $i-1$ and anotlher M pixels will be required from PE $i+1$. Recall that it is always assumed that all N PEs can transfer data simultaneously. The number of inter-PE transfers increases, for N > 4, relative to the square subimage method discussed in previous subsection (from $4M/\sqrt{N}+4$ to $2M$). However, for the stripe method, the border pixels located on the left side and the right side of the image are uniformly distributed among all the PEs. Because calculations on these column border pixels are not

performed, once the inter-PE data transfers are complete, each PE will perform a total of at most (M/N) × (M−2) smoothing operations. Thus, each PE avoids smoothing at least 2M/N image edge pixels when using the horizontal stripe method compared with using square subimages.

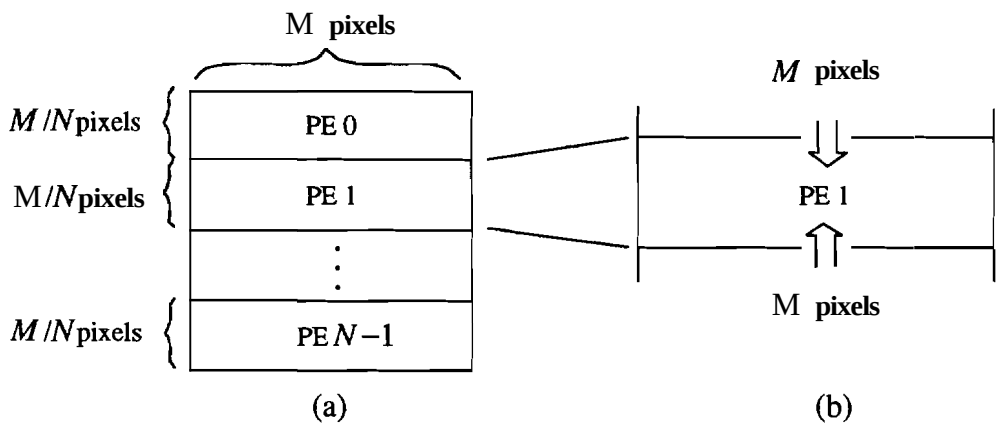

Figure 3: (a) Data allocation and (b) pixel transfers for image smoothing using the horizontal stripe method.

Comparing the two schemes, the square distribution scheme requires fewer inter-PE transfers but more smoothing operations. Although the horizontal stripe method requires more transfers, it has the potential of decreasing the execution time due to the reduced number of smoothing operations resulting from not processing the column border pixels. For image smoothing, the square subimage method, as shown in Subsection 3.3, may be best, but for other window-based image processing algorithms, e.g., range image segmentation, it may be outperformed by the horizontal strip method [GiW92].

The results derived so far in this subsection used a 3 × 3 window. The derivations can be generalized for a w × w window, with more complex operations being performed on the set of data values within the window. For a task with a w × w window, the square subimage method

performs M^2/N operations and the horizontal stripe method performs $(M^2/N)-2\lfloor w/2 \rfloor$(MIN) operations, where $\lfloor x \rfloor$ is the floor of x, i.e., the greatest integer less than or equal to x. In the case of the square subimage method, each PE must transfer $4\lfloor w/2 \rfloor \times (M/\sqrt{N}) + 4(\lfloor w/2 \rfloor)^2$ data elements. In the horizontal stripe method, $2\lfloor w/2 \rfloor$M data elements are transferred per PE. As the window size w increases, the number of operations required by the horizontal stripe method decreases. The time complexity of an operation increases as $O(w^2)$. The number of operations performed by the horizontal strip method is $\Delta T_{op} = 2\lfloor w/2 \rfloor$(MIN) less than that of the square subimage method. But the number of inter-PE transfers performed by the square subimage is $\Delta T_{transfer} = 2\lfloor w/2 \rfloor M - 4\lfloor w/2 \rfloor (M/\sqrt{N}) - 4(\lfloor w/2 \rfloor)^2$ less than that of the horizontal stripe method. If the computation time per operation is greater than or equal to $\lceil \Delta T_{transfer}/\Delta T_{so} \rceil \times$ the transfer time per data item, where $\lceil x \rceil$ is the ceiling of x, i.e., the smallest integer greater than or equal to x, then the horizontal stripe method is better. For example, if $w = 21$, $M = 4{,}096$, and $N = 256$, then $\Delta T_{op} = 320$ and $\Delta T_{transfer} = 71{,}280$. If the computation time per operation equals $\lceil 71{,}280/320 \rceil = 223 \times$ transfer time, then the horizontal stripe method is better. For a task such as image correlation, where each operation involves more than $w^2 = 441$ multiplications and additions, this could be the case (it is machine architecture dependent).

Thus, for window-based tasks, distribution of the image across PEs should not always be done by using square subimages. The subimage window size and the task to be performed have to be considered when deciding which data distribution scheme to use. In general, the horizontal stripe method may result in a decreased execution time whenever calculations on the column border pixels are not needed and inter-PE transfers are relatively fast compared to calculations needed for each pixel position.

3.5. Impact of a Specific Interconnection Network on Execution Time

Interconnection Networks Considered

The impact of a particular interconnection network on the time complexity of the parallel algorithm for a particular application is discussed next. The same data distribution across the PEs is used to compare the performance of an algorithm on two parallel machines with topologically distinct interconnection networks [SiA92b].

Consider two hypothetical parallel processing systems where the only difference between the two systems is the topological structure of their interconnection networks: one is a ***mesh*** with no wrap-around connections from one edge to another, and the other is a ring (Figure 4). All other system parameters are assumed identical, including the number of PEs, the computing and communication hardware, the operating system, the language, the communication protocols, and the link bandwidth. Both networks have N PEs labeled from 0 to $N-1$. In the mesh topology, PE i is directly connected to PEs $i-1$, $i+1$, $i+\sqrt{N}$, and $i-\sqrt{N}$. In the ring topology, PE i only has direct connections to PEs $i-1$ and $i+1$. For both networks, all PEs can send data simultaneously in the same direction.

To illustrate the impact of different interconnection network topologies while holding all other factors identical, the same SIMD image smoothing algorithm is executed on both systems. The two distribution methods described in Subsections 3.3 and 3.4 are used to compare the two architectures.

Mesh versus Ring Using Square Subimages

First consider the system with a mesh interconnection network. Figure 2 depicts how pixels from an $M \times M$ image are mapped onto a $\sqrt{N} \times \sqrt{N}$ mesh of PEs as square subimages. Pixels from N subimages of size $(M/\sqrt{N}) \times (M/\sqrt{N})$ are mapped onto PEs so that adjacent subimages are mapped to adjacent PEs in the mesh. As discussed earlier, each PE performs at most M^2/N smoothing operations and $4M/\sqrt{N} + 4$ inter-PE data transfers. Figure 5(a) illustrates the

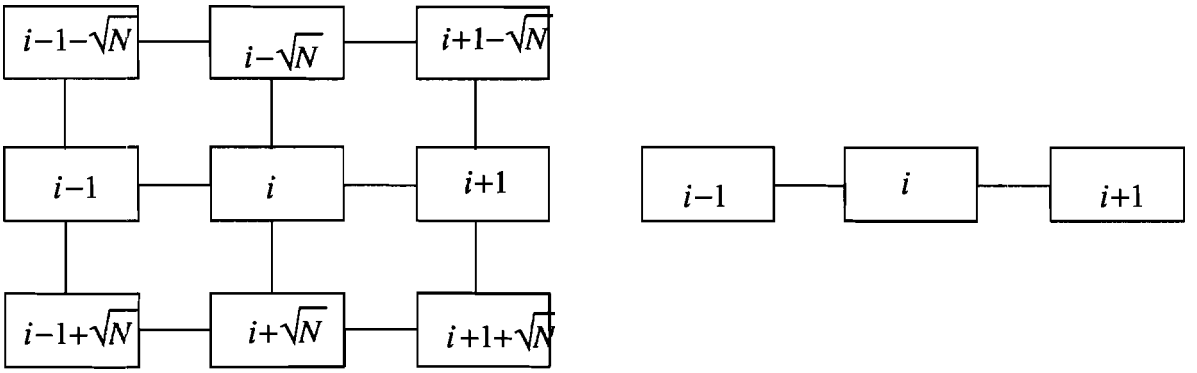

Figure 4: Connection patterns for PE i for (a) the mesh and (b) the ring topologies.

required data transfers for PE 6 for N = 16.

Next, consider a system with a ring interconnection network. It is assumed that the image is distributed among the PEs as square subimages in the same way as described above for the mesh. For the general problem of smoothing $M \times M$ images using N PEs, it is shown below that a total of $2M\sqrt{N} + 2M + 4$ inter-PE data transfers are required for the ring. Figure 5(b) is an example showing the required data transfers for PE 6 for N = 16. This example will be used in the following general description.

The $M/\sqrt{N}$ pixels along the right vertical boundary of each subimage must be transferred from PE $i-1$ to PE i (e.g., 5 to 6). This requires $M/\sqrt{N}$ inter-PE data transfers. Likewise, transferring the $M/\sqrt{N}$ pixels from the left vertical subimage boundary from PE $i+l$ to PE i (e.g., 7 to 6) also requires $M/\sqrt{N}$ inter-PE data transfers. To transfer the $M/\sqrt{N}$ pixels along the upper horizontal subimage boundary from PE $i-\sqrt{N}$ to PE i (e.g., 2 to 6) requires $\sqrt{N} \times (M/\sqrt{N}) = M$ inter-PE data transfers because PE $i-\sqrt{N}$ and PE i are separated by $\sqrt{N}$

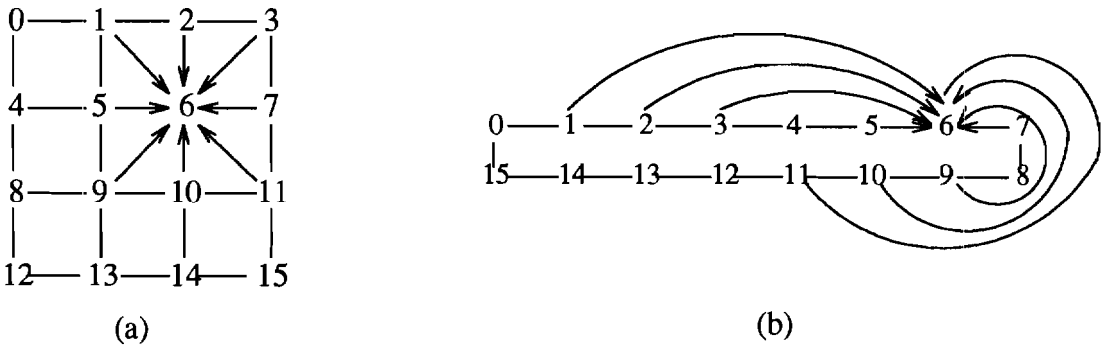

Figure 5: The required data transfers to PE 6 for (a) the mesh and (b) the ring topologies for $N = 16$.

links. Likewise, to transfer the $M/\sqrt{N}$ pixels along the lower horizontal subimage boundary from PE $i+\sqrt{N}$ to PE i (e.g., 10 to 6) requires M inter-PE data transfers. Next consider sending the corner pixels needed by PE i. Pixels from the diagonally adjacent PEs can be sent through intermediate PEs. The required pixels from PEs $i-\sqrt{N}+1$ (e.g., 3) and $i+\sqrt{N}+1$ (e.g., 11) have already been transferred to PE i+l (e.g., 7), so only two transfers are needed to move these pixels to PE i. The case for the pixels from the upper and lower left diagonal PEs is similar. Thus, the total number of inter-PE data transfers for the ring is $2M/\sqrt{N} + 2M + 4$. For N > 1, this is greater than the number of required inter-PE data transfers for the mesh, given by $4(M/\sqrt{N}) + 4$. For example, if M = 4,096 and N = 256, the number of inter-PE transfers for the mesh is 1,028 and for the ring it is 8,708.

Mesh versus Ring Using the Horizontal Stripe Method

Instead of using square subimages, consider the horizontal stripe method. A mapping of pixel values from the N rectangular $(M/N) \times M$ subimages onto the ring topology is shown in Figure 6. Pixel values along the horizontal boundaries of the rectangular subimages are transferred to the neighboring PEs, which requires a total of $2M$ inter-PE data transfers for the ring topology; M pixels are transferred from PE $i-1$ to PE i and M pixels are transferred from PE $i+1$ to PE i. Thus, the total number of inter-PE data transfers is reduced from $2M/\sqrt{N} + 2M + 4$ using square subimages to $2M$ using the horizontal stripe method for the ring network.

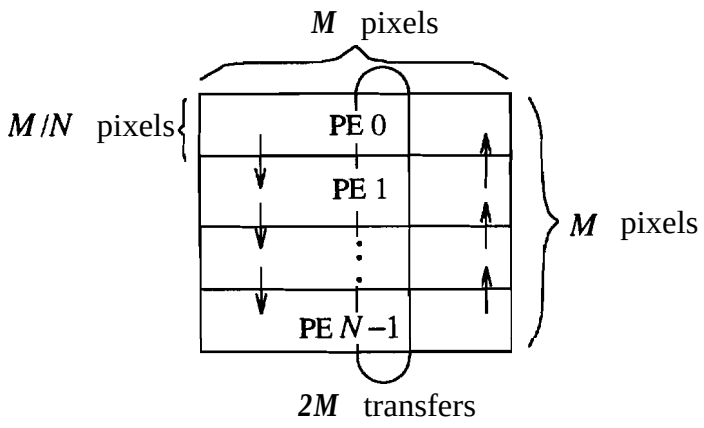

Figure 6: Mapping $(M/N) \times M$ subimages onto the PEs of the ring, showing the directions of transfers.

For the mesh network with no wrap-around connections between edges, if the horizontal stripe method is used, a total of $2M(\sqrt{N} + 1)$ inter-PE data transfers are required as demonstrated in Figure 7. For any PE i that is not on the right vertical edge of the N PE mesh, M transfers are needed to transfer pixels from PE i to PE $i+1$. For any PE i that is not on the left vertical edge of the N PE mesh, another M transfers are required to transfer pixels from PE i to

PE $i-1$. These horizontal transfers are shown in Figure 7(a). In addition, M pixels have to be transferred from PE $i \times \sqrt{N} - 1$ to PE $i \times \sqrt{N}$, $1 \le i \le \sqrt{N}-1$. This will require $\sqrt{N} \times$ M transfers because the source PE is $\sqrt{N}$ links away from the destination PE. Similarly, PE $i \times \sqrt{N}$ will send M pixels to PE i $\times \sqrt{N} - 1$, $1 \le i \le \sqrt{N}-1$. This will also require $\sqrt{N} \times$ M inter-PE data transfers. These transfers between PEs on the left and right vertical edges are depicted. in Figure 7(b). Thus, the number of inter-PE transfers for the mesh network when using the horizontal stripe method is $M + M + M\sqrt{N} + M\sqrt{N} = 2M(\sqrt{N} + 1)$. This is much greater than that required for the ring network when using the same method for data distribution. For example, if for M = 4,096 and N = 256, the number of inter-PE transfers for the ring is 8,192 and for the mesh it is 139,264. In this case, the ring outperforms the mesh.

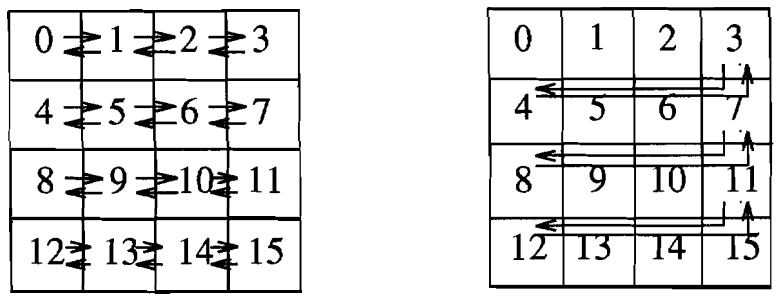

Figure 7: Mapping (MIN) × N subimages onto the PEs of the mesh with the directions of transfers shown; (a) 2M transfers, (b) $2(\sqrt{N}M)$ transfers.

Best Case Comparison

Based on the above analyses, in terms of the number of inter-PE data transfers, the square subimage data distribution is best for the mesh network, the horizontal stripe method is best for

the ring network, and the best mesh approach is better than the best ring approach. However, recall from Subsection **3.4,** the horizontal stripe method can also avoid processing the column border pixels, reducing the calculations performed by each **PE.** As was discussed in that subsection, consider a w × w window and a window-based image processing task more complex than smoothing. The comparison between the best mesh and best ring cases is the same as the comparison between the square subimage and horizontal stripe methods in that subsection. Thus, depending on the window size and the image processing task, the ring may be better than the mesh (as well as less expensive to implement).

Another aspect of the interaction of the algorithm, data distribution, and interconnection network is the number of distinct network settings (i.e., path changes). In certain types of network implementations, the number of network settings used can impact performance (even for a fixed total number of data items to be transferred). In many network implementations, due to software and/or hardware overhead, establishing the path for transmitting data is a significant portion of the transfer time, and is independent of the number of words to be transferred using that setting. Thus, it may be important to minimize the number of network settings used.

Consider image smoothing with **3 × 3** windows and the horizontal stripe rnethod of Figure **6. PE** i stores rows $i \times MIN$ to $(i+1) \times MIN - 1$ of the image. In general, it receives row $i \times MIN - 1$ from **PE** $i-1$, and row $(i+1) \times MIN$ from **PE** $i+1$. **PE** i smootlhes the pixels in rows $i \times M/N$ to $(i+1) \times MIN - 1$ of the image. Instead of using these two network settings, a single setting can be used by only sending rows $(i+1) \times MIN$ and $(i+1) \times M/N + 1$ from **PE** $i+1$ to **PE** i. In this case, **PE** i smoothes rows $i \times MIN + 1$ to $(i+1) \times MIN$ of the image. With this approach, the number of data items transferred is still $2M$ and the maximum number of pixels smoothed in any **PE** is still M^2/N. This can be generalized for w × w windows. Also, this approach can be applied to square subimages, reducing the number of network settings from four to two.

3.6. Summary

As has been shown, the way in which data are distributed among the PEs may influence both the number of inter-PE transfers needed and the number of computations to be performed. Conversely, the topology of the inter-PE network of the machine may influence which data distribution is best. Interestingly, even if the problem domain is limited to window-based image processing tasks, no one data distribution is best for all window sizes and image operations. Thus, it is important to carefully analyze all of these factors to find the best data distribution.

4. CU/PE OVERLAP

In this section, the impact of CU/PE overlap on execution time in SIMD mode is examined. The CU CPU initiates parallel computations by sending blocks of SIMD code to the CU instruction broadcast queue. Once in the queue, each SIMD instruction is broadcast to all the PEs while the CU CPU can be performing its own computations. This property is called CU/PE overlap [ArN91, BeS91, KiN91]. CUPE overlap can improve the overall performance of a program because CU execution and PE execution can occur concurrently. This aspect of data parallel algorithms is the only one discussed in this report that is not directly applicable to MIMD machines.

Image correlation [ArN91, SiS82] is used to examine the effects of CUPE overlap. Image *correlation* involves determining the position at which an image template best matches a portion of an input image. Let x denote an $r \times c$ template array, let A be an R × C input image array, and let y be an r × c portion of A. Also, let a given image coordinate be: (row, col) such that $0 \le row \le (R - r)$ and $0 \le col \le (C - c)$, and let $y(i,j) = A(row+i, col+j)$ for all coordinates (i, j) where $0 \le i < r$ and $0 \le j < c$. The *coefficient* of determination, D^2, is computed to determine the quality of fit of the template to the overlapped data of the input image. One part of this calculation is to compute $\sum\sum x(i, j)y(i, j)$ for each possible match position for $0 \le i < r$ and $0 \le j < c$.

CUPE overlap is an analytical quantity that can be maximized. Consider the above computation. Two possible approaches to compute this two-dimensional summation are given in C-like notation in Figures 8 and 9 (based on [ArN91]). Assume that the instruction broadcast queue is empty when the CU starts execution of either code segment and that the queue is large enough such that it never overflows during execution. For simplicity, it is assumed that the match position is (0,0) and the size of the template never exceeds that of the subimage. The image data are distributed among the PEs by segmenting the input image into rectangular subimages, each of which has the size R' × C' The template and subimage arrays are represented as one-dimensional

arrays temp[] and *subimage*[], respectively. The scalar variable *xysum* accumulates the desired sum.

The same task is performed by both code segments, but the workload is distributed differently between the CU and the PEs. The first approach, shown in Figure 8, overworks the PEs by performing all the array indexing calculations on the PEs; whereas the second approach, shown in Figure 9, overworks the CU by letting the CU perform all the array indexing calculations. The numbers along the left and right sides of each figure provide approximate statement execution times in microseconds for the CU and for the PEs, respectively, as :measured on the PASM prototype.

```
CU                                                                  PE

14    for ( i = 0; i < r; i++ ) { /* 4, 8, 6 */
17        send–short( &i, i );      /* send i (8 bits) from CU to PEs */      8
14        for ( j = 0; j < c; j++ ) {  /* 4, 8, 6 */
17            send–short( &j, j );     /* send j (8 bits) from CU to PEs */   8
 6            simdbegin                /* load PE instruction block into queue */
                  xysum += temp[c × i + j] × subimage[ C' × i + j];          90
              simdend
          }
      }
```

Figure 8: Code segment overworking the PEs.

The time complexities can be determined by examing the code segments. For the "for" statements,,the time to initialize the loop-control variable is *4*, the time to test the end-of-loop condition is 8, and the time to increment the loop-control variable is *6*. For the approach shown in Figure 8, the time complexities of the CU and the PEs are $T^a_{CU} = r \times (14 + 17) + (4 + 8) + r \times (c \times (14 + 17 + 6) + (4 + 8)) = 37rc + 43r + 12$ and

```
CU                                                                          PE
    tbase = temp[];              /* initialize template pointer */
    ibase = subimage[];          /* initialize subimage pointer */
14  for ( i = 0; i < r; i++ ) {  /* 4, 8, 6 */
32      tptr = tbase + c × i − 1;   /* increment template row pointer */
32      iptr = ibase + C' × i − 1; /* increment subimage row pointer */
14      for ( j = 0; j < c; j++ ) { /* 4, 8, 6 */
10          tptr += 1;           /* increment template column pointer */
10          iptr += 1;           /* increment subimage column pointer */
17          send_int( &tptr, tptr ); /* send template pointer (16 bits) from CU to PEs */  8
17          send_int( &iptr, iptr ); /* send subimage pointer (16 bits) from CU to PEs */  8
 6          simdbegin            /* load PE instruction block into queue */
                xysum += (*tptr) × (*iptr);                                  34
            simdend
        }
    }
```

Figure 9: Code segment overworking the CU.

$T^a_{PE} = r \times 8 + r \times c \times (90 + 8) = 98rc + 8r$, respectively. For $r \geq 1$ and $c \geq 1$, $T^a_{PE} > T^a_{CU}$. If $r = c = 7$, $T^a_{CU} = 2{,}126$ and $T^a_{PE} = 4{,}858$. For the approach shown in Figure 9, $T^b_{CU} = r \times (74 \times c + 90) + 12 = 74rc + 90r + 12$ and $T^b_{PE} = 50cr$, respectively. For $r \geq 1$ and $c \geq 1$, $T^b_{CU} > T^b_{PE}$. If $r = c = 7$, $T^b_{CU} = 4{,}268$ and $T^b_{PE} = 2{,}450$.

Because the CU broadcasts the instructions to the PEs, the time before the PEs receive their first instructions and the time for the PEs to execute the final instructions broadcast to them have to be considered. Let T_{final} denote the time for the PEs to execute the last instructions broadcast to them minus the time to perform CU computation after the final SIMD block; has been broadcast, i.e., T_{final} is the time some PE continues to execute after the CU is done. If the difference is less than zero, set $T_{final} = 0$. For the first approach, the CU increments and then checks the index value of i and j after it broadcasts the final SIMD block, and thus

$T^a_{final} = (8 + 90) - (6 + 6 + 8 + 6 + 8) = 64$. Let $T_{startup}$ be the PE idle time during the first iteration. T^b_{final} and $T^b_{startup}$ can be calculated similarly. For the first approach, $T^a_{startup} = (4 + 8 + 17 + 4 + 8 + 17) - 8 = 50$. For $r = c = 7$, the total execution time of the first approach is

$$\max(T^a_{CU} + T^a_{final},\ T^a_{startup} + T^a_{PE}) = \max(2,126 + 64,\ 50 + 4,858) = 4,908$$

and for the second approach, the total execution time is

$$\max(T^b_{CU} + T^b_{final},\ T^b_{startup} + T^b_{PE}) = \max(4,268 + 8,\ 134 + 2,450) = 4,276$$

By balancing the workload between the CU and the PEs, CU/PE overlap can be maximized, i.e., $|T_{CU} + T_{final} - T_{PE} - T_{startup}|$ is minimized, and thus the execution time can be reduced. A third approach, shown in Figure 10, attempts to minimize $|T_{CU} + T_{final} - T_{PE} - T_{startup}|$ by migrating two indexing operations from the CU to the PEs. The total execution time for the third approach is

$$\max(T^c_{CU} + T^c_{final},\ T^c_{startup} + T^c_{PE}) = \max(1,860 + 26,\ 112 + 2,758) = 2,870$$

Thus, by balancing the operations performed on the CU and on the PEs, it is possible to achieve better performance.

In summary, this section has shown that on SIMD machines that have CU/PE overlap capability, CU/PE overlap can be maximized to reduce the algorithm execution time, especially when loop-intensive computations are involved. Thus, to achieve the best performance, it is important to examine the code and to balance CUPE overlap whenever possible.

CU		PE
	tbase = temp[]; /* initialize template pointer */	
	ibase = subimage[]; /* initialize subimage pointer */	
14	for (i = 0; i < r; i++) (/* 4, 8, 6 */	
32	tptr = tbase + c × i − 1; /* increment template row pointer */	
32	iptr = ibase + C′ × i − 1; /* increment subimage row pointer */	
17	send_int(&tptr, tptr); /* send template pointer (16 bits) from CU to PEs */	8
17	send_int(&iptr, iptr); /* send subimage pointer (16 bits) from CU to PEs */	8
14	for (j = 0; j < c; j++) (/* 4, 8, 6 */	
6	simdbegin /* load PE instruction block into queue */	
	tptr += 1; /* increment template column pointer */	10
	iptr += 1; /* increment subimage column pointer */	10
	xysum += (*tptr) × (*iptr);	34
	simdend	
	}	
	}	

Figure 10: Code segment optimizing CU/PE overlap.

5. PARALLEL REDUCTION OPERATIONS

5.1. Introduction

Many problems require that all the elements of a data set be combined in some :fashion, e.g., the sum or product of all elements of an array. These operations are known as reduction operations, i.e., a single value is computed as the result of an operation on the data set. Parallel reduction operations can only be applied to associative operations, e.g., sum, product, min, and max. It is important to study parallel algorithms for reduction operations because data elements are generally distributed across the PEs on parallel machines. As examples, recursive doubling and parallel prefix are discussed in detail in this section.

5.2. A Single Reduction Operation on a Single Set of Data

Recursive doubling is used to demonstrate the concept of a single reduction operation on a single set of data in this subsection. The recursive doubling procedure, sometimes also called tree summing, is an algorithm that combines a set of operands distributed across PEs [Sto80]. Consider the example of finding the sum of M numbers. Sequentially, this requires one load and $M-1$ additions, or approximately M additions. However, if these M numbers are distributed across N = M PEs, the parallel summing procedure requires $\log_2 N$ transfer-add steps, where a transfer–add is composed of the transfer of a partial sum to a PE and the addition of that partial sum to the PE's local sum. Let t_{add} be the time required to execute an addition and $t_{transfer-add}$ be the time to execute a transfer-add. Then the speedup of this algorithm is $(M \times t_{add}) / (\log_2 N \times t_{transfer-add}) = O(N/\log_2 N)$, assuming t_{add} and $t_{transfer-add}$ are of the same order.

This is demonstrated for M = N = 8 in Figure 11. Assume that the goal is to calculate $\sum_{i=0}^{7} A(i)$ and A (i) is initially stored in PE i. First, each odd numbered PE i transfers its data item to PE $i-1$ (at time t_0). Each PE receiving a data item adds it to the data item it is storing (at

time t_1). PE 2 sends its partial sum (A (2) + A (3)) to PE0, and PE 6 sends its partial sum (A (6) + A (7)) to PE 4 (at time t_2). Each of PEs 0 and 4 adds its received partial sum to its stored partial sum (at time t_3). Then, PE 4 sends its partial sum to PE 0 (at time t_4). PE 0 then computes the complete sum (at time t_5). All of the inter-PE transfers can be adjusted so that the complete sum could be computed by any one of the PEs.

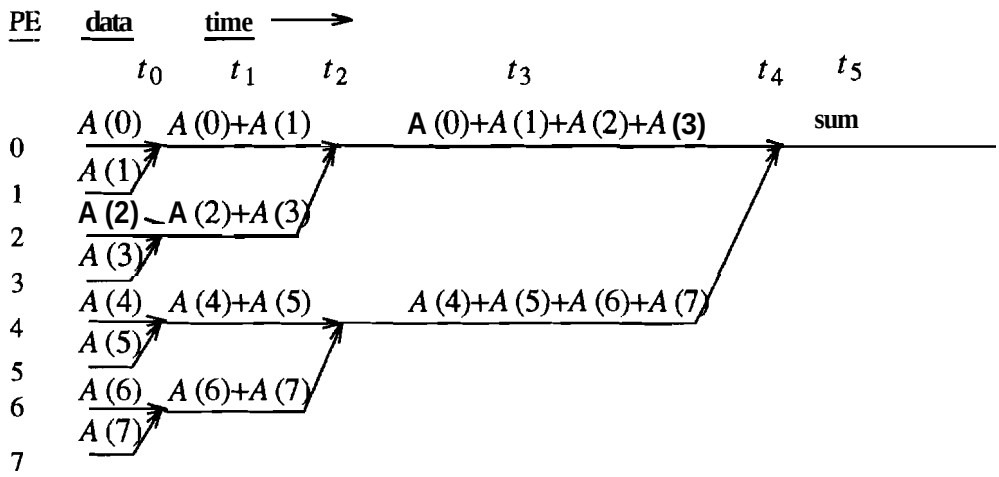

Figure 11: Recursive doubling using result-to-one-PE technique for $M = N = 8$, where sum $= \sum_{i=0}^{7} A[i]$.

The recursive doubling technique can also be applied to a set of operands whose size is greater than the number of PEs. Let M be the number of operands and let $N = 2^n$ be the number of PEs, addressed from 0 to N − 1, where PE P's address in binary is $p_{n-1}...p_1p_0$. Let each PE store $\lfloor M/N \rfloor$ of the operands and let M mod N PEs receive one additional operand from the remaining M mod N operands. First, each PE sequentially sums its local data, requiring at most $\lceil M/N \rceil$ operations. Once the local sums are obtained, then $\log_2 N$ transfer-adds are performed. In general, in the j-th inter-PE transfer, where j proceeds from 0 to $\log_2 N - 1$, PE $P = p_{n-1}...p_{j+1}10...0$ sends its partial sum to PE $P' = p_{n-1}...p_{j+1}00...0$. PE P' combines the

received partial sum and its previously computed local partial sum to form a new local partial sum. After $\log_2 N$ transfer-adds, PE 0 will contain the complete sum. The speedup is $(M \times t_{add})/(\lceil M/N \rceil \times t_{add} + \log_2 N \times t_{transfer-add})$. The greater the difference between M and N, M > N, the closer the speedup is to N. If M < N, then this technique can still be applied. In this situation, only M PEs would be used.

The above demonstrates the ***result-to-one-PE*** technique, where the global result will be available in a single PE. An alternative is the ***result-to-every-PE*** technique, in which each PE will have the global result. This is shown in Figure 12 for M = N = 8. Let a ***transfer-op*** be composed of a transfer of an operand to a PE and an operation combining this operand and a local operand of this PE. As in the previous technique, once the local results have been obtained, $\log_2 N$ transfer-ops are made to find the global result. In transfer-op j, where j proceeds from 0 to $\log_2 N - 1$, PE $P = p_{n-1}...p_1 p_0$ exchanges partial results with PE $P' = p_{n-1}...\overline{p_j}...p_1 p_0$. After $\log_2 N$ transfer-ops, each PE will have the global result in its local memory.

Consider the trade-offs between the result-to-one-PE and result-to-every-PE techniques. In SIMD mode, the result-to-one-PE approach has the overhead of disabling PEs in transfer-op steps other than the first step. Thus, even if the result is not needed in all PEs, the result-to-every-PE method may be faster because the masking overhead for disabling PEs is avoided. For SIMD or MIMD mode, when the result is only needed in one PE, using the result-to-every-PE method may cause unnecessary delays in the interconnection network. These trade-offs would have to be balanced for a particular SIMD architecture.

5.3. Multiple Reduction Operations on a Single Set of Data

In the previous subsection, performing one reduction operation on a single data set was discussed. In this subsection, performing multiple reduction operations on a single set of data is reviewed. The min/max problem, i.e., finding the minimum and the maximum values from a set of data simultaneously, is analyzed as an illustrative example.

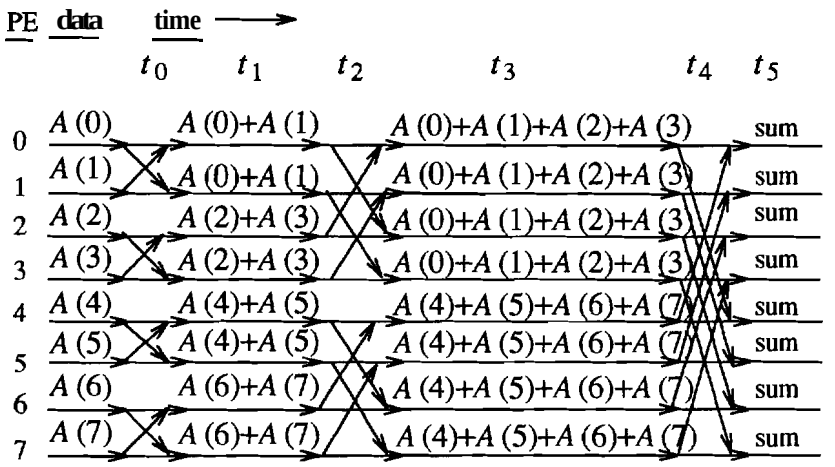

Figure 12: Recursive doubling using result-to-every-PE technique.

A straightforward method of solving the min/max problem in parallel is to use the recursive doubling procedure twice: first to find the maximum value and then to find the rninimum value. A better technique is to divide the PEs into two groups, such that one group finds the maximum value while the other group simultaneously finds the minimum value. Figure 13 demonstrates the process of finding the rninimum and the maximum values of a data set of $M = 16$ elements on $N = 8$ PEs.

For simplicity, assume that the data set has M elements, where M is an integer multiple of N. Each PE is assigned MIN elements. In the first step, each PE finds the local minimum value and the local maximum value. Then in the next step, PEs are divided into two groups: the upper half and the lower half. A PE in the upper half has its high-order address bit $p_{n-1} = 0$ and a PE in the lower half has its high-order address bit $p_{n-1} = 1$.

In the third step, each PE $P = p_{n-1} \ldots p_1 p_0$ exchanges local values with PE $P' = \bar{p}_{n-1} \ldots p_1 p_0$. A PE in the lower half transfers its local minimum value, while a PE in the upper half transfers its local maximum value. Each PE in the lower half computes a new local

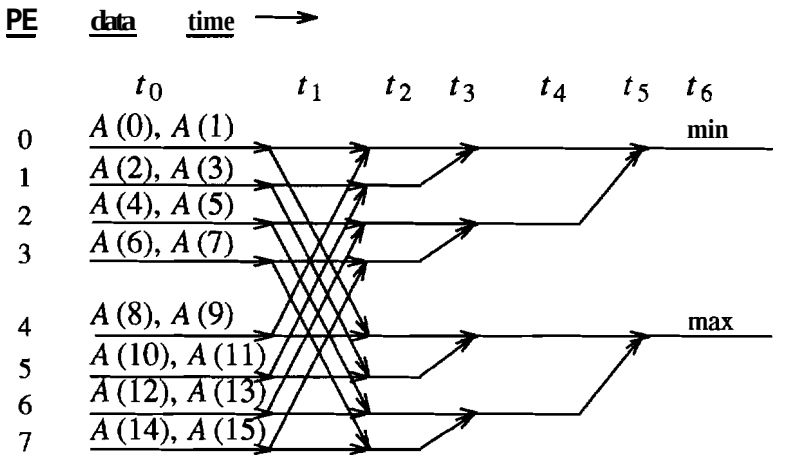

Figure 13: Finding the minimum and maximum values concurrently for $M = 16$ and N = 8 by adapting the result-to-one PE technique.

maximum value, by choosing the larger value from its original local maximum value and the value it received. Each PE in the upper half computes a new local minimum value by choosing the smaller value from its original minimum value and the value it received. Then, the lower half of the PEs do transfer-max operations and the upper half of the PEs do transfer-min operations independently but concurrently (see Figure 13). As a result of this step, the global maximum value is known by one or all PEs in the lower half (depending on which technique is selected: result-to-one-PE or result-to-all-PEs) and the global minimum value is known by one or all PEs in the upper half. If needed, the maximum and minimum values can be exchanged between the two groups (in a single inter-PE transfer) so that each group can have both the nnaximum and the minimum values. The total number of transfer-compares is $1 + \log_2(N/2)$, where the first term comes from the initial exchange of local values, and the second term comes from the concurrent recursive doubling processes.

In general, the technique can accommodate K reduction operations on a single set of data, where K ≥ 2. The number of transfer-ops is given by $(K-1)+\log_2(N/K)$, assuming that N and K are powers of two and the number of operands is an integer multiple of N. Again, the first term and the second term come from the initial exchange of local values and from the concurrent recursive doubling processes, respectively.

5.4. A Single Reduction Operation on Multiple Sets of Data

Recursive doubling can also be applied to one reduction operation on multiple sets of data. An example is global histogramming [SiA92c, SiS8 l], also known as vector *summation*, which will be studied next.

Let an M × M input image be mapped onto N PEs such that each PE holds M^2/N pixels, as in the image smoothing example discussed in Subsection 3.3. Global histogramming involves computing B bins, where each bin has two attributes associated with it: (1) the range of pixel values it represents and (2) the number of pixels in the entire image that have values within that range. For this algorithm, it is assumed that N is an integer multiple of B. Each possible pixel value belongs to exactly one bin range. Each PE first computes a local B-bin histogram for the M^2/N pixels in its memory. Let $A(x,y)$ be the gray-level value of the subimage pixel in row x and column y, and let bin(i), where $0 \le i <$ B, be initialized to 0. If each PE contains an $(M/\sqrt{N}) \times (M/\sqrt{N})$ subimage, then an algorithm to compute the local B-bin histograms is:

```
for x = 0 to (M/√N) − 1
  for y = 0 to (M/√N) − 1
      bin(A(x,y)) = bin(A(x,y)) + 1
```

(For the serial algorithm, set N = 1.) The PEs then combine their local histograms to obtain the global histogram. This is a process of a single combining operation on multiple bins. One straightforward approach to compute the global histogram is to combine one bin at a time using recursive doubling, requiring B × $\log_2 N$ transfer-add steps.

Consider an overlapped recursive doubling procedure for combining local histograms, where all the bins are summed in an interleaved fashion. Figure 14 shows this process for N = 16 and B = 4. In the figure, (0, ..., 3) denotes the values of bins 0, ..., 3 accumulated in the PE. The N PEs are logically divided into NIB blocks of B PEs each. In the first $b = \log_2 B$ stages, each of the NIB blocks simultaneously combine their histograms. As a result, each PE in a block holds a different bin computed by summing the values of the corresponding local bins of the PEs in the block. This is done by dividing each block of PEs in half such that the PEs with lower addresses form one group and the PEs with the higher addresses form the other group. Each group accumulates the sums for half of the bins and sends the bins it is not accumulating to the other group. For example, in stage 0, PE 0 accumulates bins 0 and 1 from PE 0 and PE 2. Simultaneously, PE 1 accumulates bins 0 and 1 from PE 1 and PE 3. The next phase involves dividing each group of B 12 PEs into two groups of B 14 PEs using the same rule for partitioning PEs as in the previous phase. These two new groups only exchange those bins for which they had accumulated sums in the previous phase. This subdividing process continues until each group has only one PE. Once the subdividing process terminates, each PE will contain only one bin, and this bin will have the accumulated value for the PEs in the block that originally included this PE. Continuing the example above, PE 0 accumulates bin 0 from PE 0 and PE 1, while PE 1 accumulates bin 1 from PE 0 and PE 1.

In the next n – b stages, where $n = \log_2 N$, the partial histograms of all the blocks are combined by performing B simultaneous recursive-doubling operations. Each of these B recursive-doubling operations involves the NIB PEs that store the bins of same index. For the example in Figure 14, PEs 0, 4, 8, and 12 combine the bin 0 data, while PEs 1, 5, 9, and 13 combine the bin 1 data, etc. As a result, the histogram for the entire image is distributed over B PEs, where bin i is located in PE i, for $0 \le i < B$.

For the first b stages of the algorithm, $B/2^{j+1}$ transfer-adds are used at stage j, where $0 \le j < b$, for a total of B –1. At each stage j, where $b \le j < n$, one transfer-add occurs. Thus the final n – b stages require $\log_2(N/B) = n - b$ transfer-adds. The total number of transfer-adds

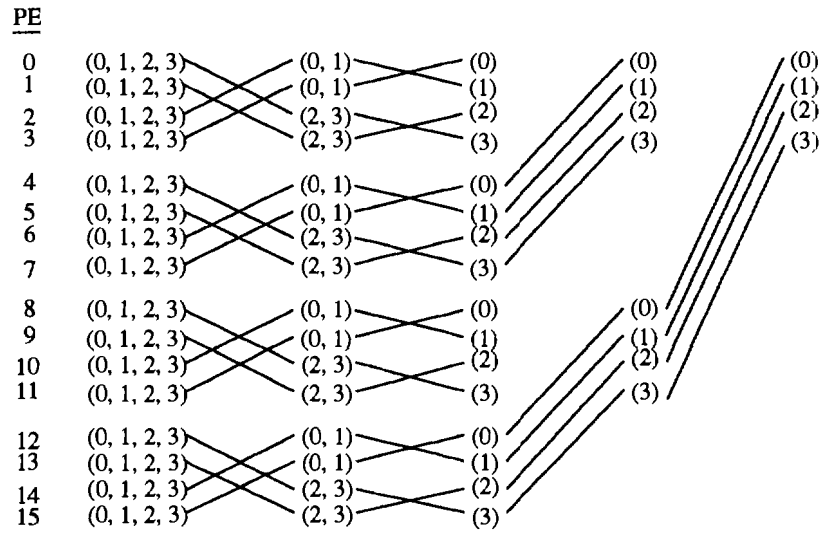

Figure 14: Global histogramming for B = 4 and N = 16.

needed to merge the local histograms using the overlapped recursive-doubling scheme is then $B - 1 + \log_2(N/B)$. For realistic values of N and B (e.g., N = 1,024 and B = 256), this is approximately $\log_2 N$ times faster then the straightforward approach of performing B recursive doubling in sequence.

5.5. A Generalized Form of Reduction Operations

A generalization of the techniques discussed in Subsections 5.2 to 5.4 is to apply multiple reduction operations on multiple sets of data. ***A distinct operation*** is defined as an associative operation on one set of data. Both one associative operation on multiple sets of data and multiple associative operations on one set of data are considered multiple distinct operations. For example, if min and max operations are applied on one set of data, they are considered two distinct operations. In the global histogramming example, the combining operation is applied on four bins, thus there are four distinct operations. It is obvious that the number of distinct operations is

always greater than or equal to the number of associative operations. Assume that each data set is of the same size M. Let N be the number of PEs used and let K be the number of distinct operations. For simplicity, it is assumed that M, N, and K are all powers of two, where $M \geq N$ and $K \leq N$, and each PE has M/N elements of each data set.

The generalized reduction operation process can be divided into three phases: (1) computing local partial results, (2) grouping PEs for different distinct operations, and (3) combining partial results. In the first phase, each PE computes K local partial results, one for each distinct operation on its local data. There are no inter-PE transfers in this phase, and the number of concurrent local operations is $K \times (M/N)$. In the second phase, the PEs are logically divided into N/K groups. A single PE accumulates the partial results with respect to one distinct operation for the block during this phase. As shown in the example for global histogramming, $K - 1$ transfer-op steps are needed to accomplish this phase. The last phase is to perform K recursive doubling procedures concurrently, one for each distinct operation. This is the same as the last step shown in the min/max and the global histogramming examples and needs $\log_2(N/K)$ transfer-ops. Thus, the total number of transfer-ops is $K - 1 + \log_2(N/K)$.

When $K > N$, i.e., when there are more distinct operations than the number of PEs, interleaved recursive doubling procedures can still be used to find the results. This approach is referred to as p–recursive doubling [WaN94]. In this case, $\rho = K$ and the final results of K distinct operations can be computed in $\log_2 N$ steps, where $\lceil K/2^{j+1} \rceil$ transfer-op operations occur at step j, for $0 \leq j < \log_2 N$. In the p-recursive doubling procedure, K is not required to be a power of two. Figure 15 illustrates how $N = 4$ PEs sum the elements for $K = 6$ sets of data, assuming each set has N elements. This requires five transfer-adds, three of which are for the first step and two for the second step.

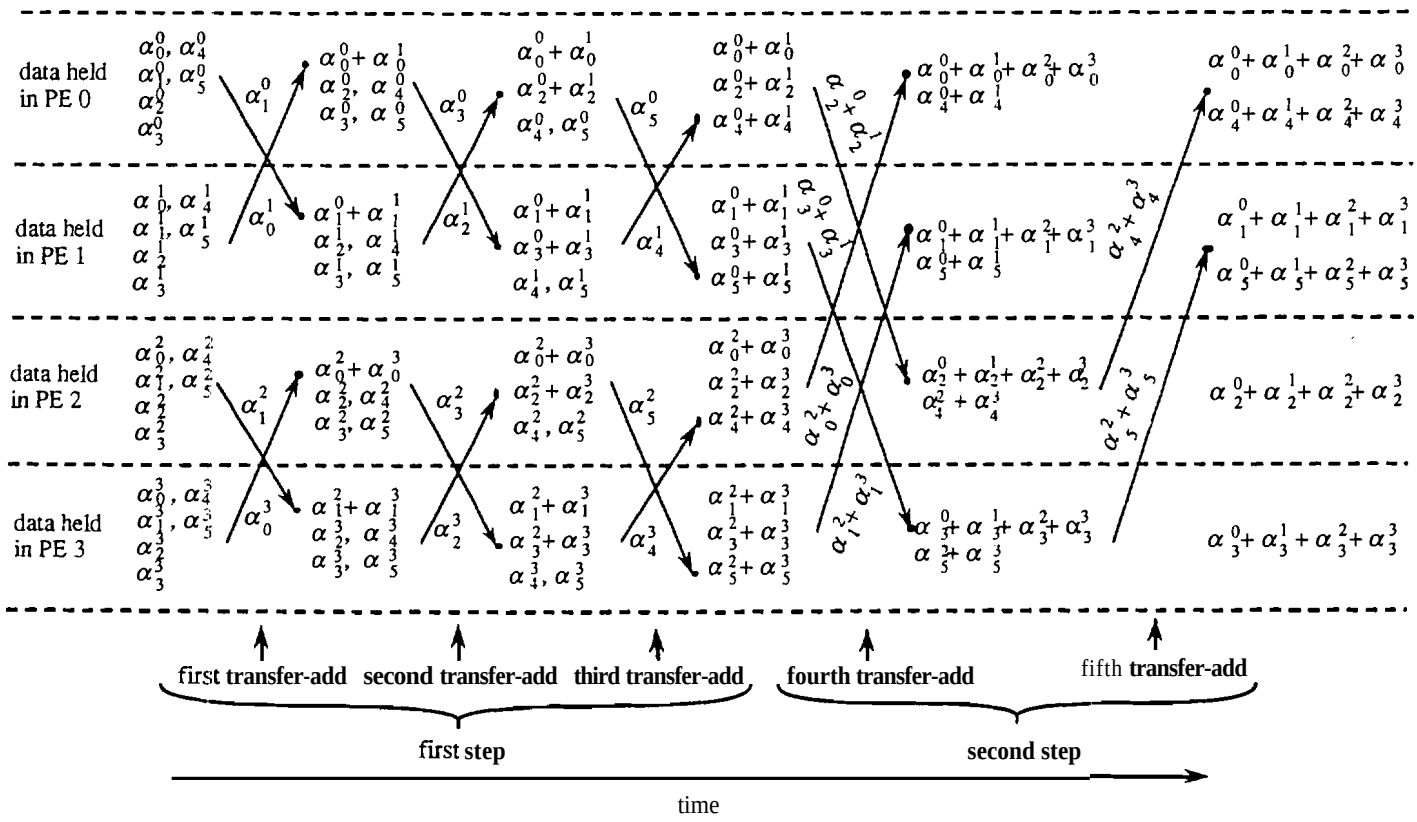

Figure 15: p-recursive doubling for $K = 6$ and $N = 4$, to compute $\sum_{b=0}^{5} \alpha_b^a$, for $0 \leq a < 6$.

5.6. Parallel Prefix

A parallel technique similar to recursive doubling is ***parallel prefix***. For A(i), $0 \le i < N$, the parallel prefix method can compute $B(j) = \sum_{i=0}^{j} A(i)$, $0 \le j < N$, in $\log_2 N$ transfer-add steps [Sto80].

It is assumed that PE i, $0 \le i < N$, has A(i) resident, and it will have B(*i*) as the result. Figure 16 demonstrates this procedure for the case N = 8. Initially, every PE executes B(i) = A(i). In transfer-add step j, where j proceeds from 0 to $\log_2 N - 1$, PE P sends its partial result B(P) to PE $P' = P + 2^j$ only if $P' < N$. PE P' computes a new partial result B(P') from its own partial result and the received value. After $\log_2 N$ transfer-op steps, PE i will have B(i).

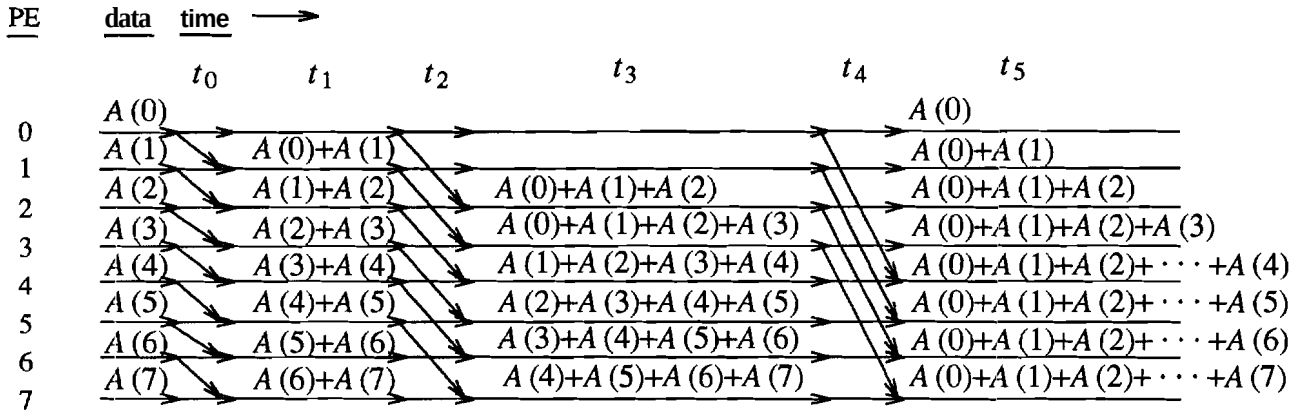

Figure 16: Parallel prefix for N = M = 8.

5.7. Summary

Various parallel reduction operation techniques for associative operations were discussed in this section. The choice of technique will depend primarily on the application task to be performed.

Because computers use finite precision arithmetic, applying certain associative operations (such as floating point addition) on the same set of data but in different sequence could give different results. However, this problem exists for serial as well as parallel machines.

6. MATRIX AND VECTOR OPERATIONS

6.1. Introduction

This section examines parallel matrix operations by considering three examples: matrix transposition, matrix-by-matrix multiplication, and matrix-vector multiplication. In each case, only one of many possible approaches is presented.

6.2. Matrix Transposition

Given an M × M matrix A, the transpose of A, A^T, is $A^T(i, j) = A(j, i)$, where $0 \le i, j < M$. The parallel matrix transposition technique described here is part of a two-dimensional FFT algorithm presented in [JaM86]. For simplicity, assume M = N, the number of the PEs used. The data is distributed across the PEs such that PE i contains row i of A, $0 \le i < N$. The goal is to have row i of A^T (i.e., column i of A) stored in PE i, $0 \le i < N$.

Let B be the base address of the location that holds the row of A in a PE and let B^T be the base address of the memory segment that is allocated to hold a row of A^T in a PE. The following code for PE i is executed by all PEs simultaneously to generate A^T.

```
(S1)  for j = 1 to M − 1 do
(S2)        fetch the value at location B + (i + j) mod M        /* A(i,i+j) */
(S3)        transfer from PE i to PE (i + j) mod M
(S4)        store the received value at location B^T + (i − j + M) mod M   /* A^T(i+j,i) */
(S5)  copy the value at location B + i to B^T + i      /* move A(i,i) to A^T(i,i) */
```

In statement S2, each PE fetches the value to be sent from its rnemory location B + (i + j) mod M. In S4, each PE stores the received value at location B^T + (i − j + M) mod M. When S1 through S4 are done, only the diagonal elements remain to be processed, which is done in S5. This is simply a matter of copying $A(i,i)$ to $A^T(i,i)$, which does not require any inter-PE transfers. Figure 17 shows the transposition process for $N = M = 4$

for $j = 1$. The total number of matrix elements that must be transferred is $M(M-1)$, because the M elements in the major diagonal are not transferred. At most, M elements can be moved in parallel in one transfer step. Thus, this algorithm uses $M-1$ transfers, which is the minimum possible given these data distributions. This technique can be directly extended for $M > N$, and each PE storing a square submatrix of A.

PE #				
0	(0,0)	(0,1)	(0,2)	(0,3)
1	(1,0)	(1,1)	(1,2)	(1,3)
2	(2,0)	(2,1)	(2,2)	(2,3)
3	(3,0)	(3,1)	(3,2)	(3,3)

(a)

PE #				
0	(0,0)	(1,0)	(2,0)	(3,0)
1	(0,1)	(1,1)	(2,1)	(3,1)
2	(0,2)	(1,2)	(2,2)	(3,2)
3	(0,3)	(1,3)	(2,3)	(3,3)

(b)

Figure 17: Matrix transpose for N = $M = 4$ for j = 1. The data transferred are underlined. (a) Matrix A and (b) matrix A^T across PEs.

6.3. Matrix-by-Matrix Multiplication

Given an $M \times M$ matrix A and an M × M matrix B, the product of A and B is an $M \times M$ matrix C = A × B whose elements are given by

$$C(q,r) = \sum_{h=0}^{M-1} A(q,h) \times B(h,r), \quad 0 \le q,r \le M-1$$

Using the straightforward approach, the serial execution time is $t_s = M^3 t_{op}$, where t_{op} is the time required for one addition and one multiplication.

Assume that N PEs are arranged in a logical $\sqrt{N} \times \sqrt{N}$ grid and the PEs are labeled from PE(0,0) to PE($\sqrt{N}-1$, $\sqrt{N}-1$). Consider two M × M matrices A and B partitioned into N submatrices A^{ij} and N submatrices B^{ij}, respectively, $0 \le i, j < \sqrt{N}$. Each of the submatrices is of size $M/\sqrt{N} \times M/\sqrt{N}$. Matrices A and B are superimposed onto the PE grid such that PE(i,j) has the submatrices A^{ij} and B^{ij}. PE(i,j) also allocates space for the submatrix C^{ij} of the result matrix C. To calculate C^{ij}, PE(i,j) needs the submatrices A^{ik} and B^{kj} for $0 \le k < \sqrt{N}$. That is, PE(i,j) needs data from PE(i,k) and PE(k,j), for $0 \le k < \sqrt{N}$. To acquire the required rubmatrices, all-to-all row and column broadcasts are performed by the PEs. Once the broadcasts are complete, PE(i,j) will have submatrices A^{ik} and submatrices B^{kj}, for $0 \le k < \sqrt{N}$. An algorithm that performs parallel matrix multiplication is shown below [KuG94]. The following code for PE(i,j) is executed by all PEs simultaneously.

```
for k = 1 to √N − 1 do
    send A^ij to PE(i, (j+k) mod √N)        /* all-to-all row broadcast */
for k = 1 to √N − 1 do
    send B^ij to PE((i+k) mod √N, j)        /* all-to-all column broadcast */
initialize submatrix C^ij to all zeros
for k = 0 to √N − 1 do
    C^ij = C^ij + A^ik × B^kj               /* local multiplication of submatrices */
```

Consider the first "for" loop of the above algorithm. Each "send" takes $(M^2/N) \times t_{transfer}$ time units, where $t_{transfer}$ is the inter-PE transfer time for a matrix element. Hence, the time taken by the "for" loop is $(\sqrt{N}-1)(M^2/N)t_{transfer}$. The total time for the first two "for" loops is $2(\sqrt{N}-1)(M^2/N)t_{transfer}$. The final "for" loop performs the computations necessary to calculate the submatrix C^{ij}. in time $\sqrt{N}(M/\sqrt{N})^3 t_{op} = (M^3/N)t_{op}$. The speedup achieved with N PEs is:

$$S = \frac{M^3 t_{op}}{(M^3/N)t_{op} + 2(\sqrt{N} - 1)(M^2/N)t_{transfer}}$$

More efficient, but more complex, algorithms for matrix-by-matrix multiplication can be found in the literature (e.g., [KuG94]).

6.4. Matrix-by-Vector Multiplication

Matrix-by-vector multiplication is a special case of matrix-by-matrix multiplication. Given an M × M matrix A and an M × 1 vector U, the product of A and U is an M × 1 vector V = A × U whose elements are given by

$$V(i) = \sum_{j=0}^{M-1} A(i,j) \times U(j), \quad 0 \le i < M$$

The straightforward serial implementation takes $O(M^2)$ time.

A parallel algorithm for matrix-by-vector multiplication is given below [KuG94]. For simplicity, assume M = N, where N is the number of PEs used. The elements of matrix A are distributed among the PEs such that each PE i, $0 \le i <$ N, is assigned row i of A. The vector U is stored in each PE. V(i) is computed on PE i by multiplying the row i of A that is stored in PE i with the vector U, i.e., PE i computes $V(i) = \sum_{j=0}^{M-1} A(i,j) \times U(j)$, $0 \le i <$ M. This will require M multiplications and M − 1 additions per PE. Assuming that the time to perform the multiplication is of the same order as the time to perform an addition, the resulting parallel time complexity is O(M). An O(M) speedup is attained by this parallel algorithm. This technique can be adapted for M > N.

7. MAPPING ALGORITHMS ONTO PARTIONABLE MACHINES

7.1. Introduction

Partitionable parallel machines are parallel processing systems that can be divided into independent or communicating subsystems, each having the same characteristics as the original machine [LiM87, SiS81]. The ability to form multiple independent subsystems to execute multiple tasks in parallel provides such partitionable parallel machines with the potential of achieving better performance. Most MIMD machines can be partitioned, either under system or user control. A ***multiple-SIMD*** machine includes multiple CUs so that it can be partitioned into independent SIMD subsystems [Nut77]. The CM-2 hardware design could support this [TuR88]. PASM is a prototype partitionable system where each submachine can operate using ***mixed mode*** parallelism [SiS95]. In this section, two aspects of mapping algorithms onto partitionable parallel machines are considered: the impact of increasing the number of PEs used, and the impact of subtask parallelism.

The time required to move data between the local PE memories and the system secondary memory (or external I/O devices) varies greatly among different configurations of parallel machines. Thus, for the analyses here, the simplifying assumption is made that there is not a significant difference in this memory transfer time due to varying the number of PEs used for a task. These analyses can be extended to include this memory transfer time if it is significant.

In this section, it is instructive to consider overhead and its impact on parallel efficiency. ***Overhead*** operations are those that are needed for parallel execution, but not for serial execution (e.g., inter-PE data transfers). The ***parallel efficiency***, E, of a parallel implementation measures the amount of overhead that is incurred:

$$E = \frac{\text{speedup}}{\#\ \text{PEs}} = \frac{\text{serial time}}{(\#\ \text{PEs}) \times \text{parallel time}}$$

That is, the efficiency is the speedup divided by the number of the PEs used to achieve this

speedup. Obviously, efficiency could decrease if the increase in the speedup grows more slowly than the increase in the number of the PEs used, i.e., improved efficiency and increased speedup are not mutually implied. The efficiency measure will be used in the next two subsections.

7.2. Impact of Increasing the Number of PEs

Increasing the number of PEs may or may not decrease the execution time. Two examples, image smoothing and recursive doubling, are analyzed to demonstrate the impact of increasing the number of PEs on performance [KrM88, SiA92c].

Recall from Subsection 3.3 that smoothing an M × M image with N PEs using square subimages requires that each PE perform M^2/N smoothing operations and $4M/\sqrt{N} + 4$ inter-PE data transfers. The execution time decreases as N increases ($N \leq M^2$) because both the number of smoothing operations and the number of inter-PE transfers are inversely proportional to N.

Let t_{so} be the time to perform a smoothing operation and $t_{transfer}$ the time to perform an inter-PE data transfer. The parallel efficiency for the smoothing algorithm can be approximated as $(M-2)^2 t_{so}/(M^2 t_{so} + (4M\sqrt{N} + 4N)t_{transfer})$. As N increases, the efficiency decreases. Thus, increasing N reduces the total execution time, but causes the efficiency to decrease. This is because as N increases, the PEs spend more time executing overhead operations (i.e., inter-PE transfers) relative to required computations (i.e., smoothing operations). For example, when $N = M^2$, the PEs may spend more time transferring data than smoothing. One must determine the metric that is important in a given situation: execution time or parallel efficiency.

The impact of increasing N has a different effect on the recursive doubling algorithm. Assume that MIN numbers are stored in each PE, where M is the number of operands and N is the number of PEs used. Both M and N are assumed to be powers of two, $M \geq N$, and $t_{transfer-add}$ and t_{add} are as defined in Subsection 5.2. The total execution time is $(M/N)t_{add} + (\log_2 N)t_{transfer-add}$.

Assume that a transfer-add takes τ times as long as an addition, i.e., $t_{transfer-add} = \tau \times t_{add}$. In this case, as N increases the number of local additions decreases but the number of transfer-

adds increases. Thus, as N increases, the execution time first decreases then increases. The minimum execution time is a function of τ. The efficiency of the parallel recursive doubling is $M/(M+\tau N\log_2 N)$. Thus, the parallel efficiency always decreases as N increases.

One way to find the number of PEs that will minimize the execution time is to determine the partial derivative of the execution time formula with respect to N. For the recursive doubling example, this yields $\partial T_{total}/\partial N = (-M/N^2 + \tau/(N\ln 2)) \times t_{add}$. The derivative is negative for $N < (M/\tau)\ln 2$ and is positive for $N > (M/\tau)\ln 2$. Thus, as N increases from 1 to $(M/\tau)\ln 2$, the execution time decreases, and as N increases beyond $(M/\tau)\ln 2$, the execution time increases. Let N_1 be the largest value such that $N_1 \leq (M/\tau)\ln 2$ and let N_2 be the smallest value such that $N_2 > (M/\tau)\ln 2$, where N_1 and N_2 are positive integers and are powers of two. Either N_1 or N_2, whichever gives a smaller T_{total}, is chosen as the number of the PEs to be used.

For example, if $\tau = 10$ and $M = 2^{14}$, then the execution time is $(2^{14}/N + 10\log_2 N) \times t_{add}$ time units. Figure 18 shows the execution times as N increases. Using the above procedure, 2^{10} PEs are chosen to achieve the minimum execution time. This result impacts the concept of maximizing machine utilization. Typically, one tries to make use of all the PEs available, with the goal of maximizing the number of concurrent operations to minimize the execution time. This study demonstrates that maximizing utilization (i.e., using the largest N possible) does not always mean minimizing execution time. It may be faster to partition the machine and use a subset of the PEs available.

Thus, whether increasing N reduces overall execution time is algorithm dependent. Furthermore, increased parallel efficiency may not imply decreased execution time, and vice versa. Similarly, increased utilization of PEs may not imply decreased execution time. and vice versa.

N	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}
time units	148	144	122	116	118	124	132	141

Figure 18: Execution time versus number of PEs (N) for $M = 2^{14}$ and $\tau = 10$.

7.3. Impact of Subtask Parallelism

The effect of partitioning a parallel task into smaller, concurrent subtasks can have an impact on performance. Consider the task of smoothing four images such that the total time to smooth all four is minimized. One way to do this is to smooth the four images one at a time, using all N PEs to smooth each image. Another way is to partition the task such that all four images are smoothed concurrently, each using N/4 PEs, as shown in Figure 19.

The time required to smooth the four $M \times M$ images in sequence is $4((M^2/N)t_{so} + (4(M/\sqrt{N}) + 4))t_{transfers})$, four times that to smooth a single image. For M = 512 and N = 1,024, this is $1{,}024 \times t_{so} + 272 \times t_{transfer}$. The total time required for N PEs to smooth the four images concurrently, each on N/4 PEs, is $(M^2/(N/4))t_{so} + (4(M/\sqrt{N/4}) + 4)t_{transfer}$ (because all four images are smoothed concurrently, it is the same as the time to smooth one image on $N/4$ PEs). For M = 512 and N = 1,024, this is $1{,}024 \times t_{so} + 132 \times t_{transfer}$. Thus, partitioning the system and exploiting subtask parallelism decreases the execution time by reducing the number of inter-PE transfers.

The number of inter-PE transfers required in the partitioning approach is $4 \times (M/\sqrt{N/4}) + 4$, whereas in the other approach, $4 \times (4(M/\sqrt{N}) + 4)$ inter-PE transfers are needed. This reduction in inter-PE transfers gives the partitioning approach a smaller execution time. For example, if M = 512 and N = 1,024, there are 132 versus 272 inter-PE transfers. Assuming that

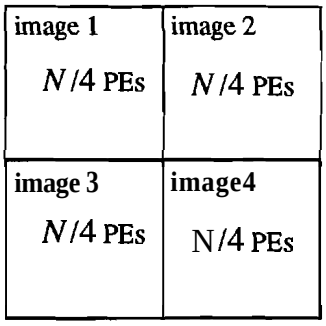

Figure 19: Smoothing four images concurrently on N PEs.

$t_{so} = t_{transfer} = 1$, the parallel efficiency of smoothing four 512×512 images in sequence, each using 1,024 PEs, is 78%, while the efficiency of smoothing all four images simultaneously on a system partitioned into four submachines of 256 PEs each is 88%. The efficiency of smoothing images concurrently is improved over that of smoothing images in sequence because the larger subimage size (32×32 versus 16×16) reduces the portion of the total execution time spent on doing inter-PE data transfers for $t_{so} = t_{transfer} = 1$ ($132/(32^2 + 132) = 11\%$ versus $68/(16^2 + 68) = 21\%$). The same number of smoothing operations are performed in both schemes. Therefore, in this case, partitioning improves both execution time and efficiency.

7.4. Summary

In this section, two aspects of mapping algorithms onto partitionable machines were discussed. Partitioning can be used to select a subset of the PEs for the task when it is faster then using all of them. Partitioning can also be used to improve performance by executing multiple subtasks simultaneously [NaM93].

8. ACHIEVING SCALABILITY USING A SET OF ALGORITHMS

A parallel algorithm is scalable if it is capable of delivering an increase in performance proportional to an increase of the number of processors utilized [Wil93]. However, one algorithmic approach to a task may not always be able to give the best performance for various input-data parameters and system parameters. Scalability can be better achieved by selecting one algorithm or some algorithm combination from a suite of algorithms to perform the task effectively as these parameters vary.

The set approach is to have a set of algorithms from which the most appropriate algorithm or combination of algorithms is selected based on the ratio between the data size and the target machine size. Parallel algorithms for computing multiple quadratic *forms* (MQFs) are discussed in this section as a case study in the design of scalable algorithms [WaN94]. Implementations of the MQF problem for various data-size/machine-size ratios were evaluated in great detail in [WaN94]. The goal of this section is not to discuss the details of the MQF study, but rather to use the results to demonstrate the importance of the set approach to achieve scalability.

Let a steering vector (s–vector) be an r × 1 vector of complex numbers and v be the total number of s-vectors. Define M to be an $r \times$ r matrix of complex numbers and M(i, j) to be the element of M in row i and column j, where $0 \le i, j < r$. The q-th s-vector is denoted by s_q, where $0 \le q < v$. Element m of the s-vector q is denoted by $s_q(m)$, where $0 \le m < r$. Let H denote the Hermitian transpose, i.e., the complex conjugate transpose of the s-vector. The MQF calculation can be defined as $w_q = s_q^H M s_q$, where $0 \le q < v$.

One parallel implementation used to solve the MQF problem is the *uncoupled* method. In this approach, no inter-PE communication is needed. By distributing v s-vectors evenly among N PEs, $\lceil v/N \rceil$ quadratic forms will be calculated in each of v mod N PEs, and $\lfloor v/N \rfloor$ on each remaining PE. Thus, the time to compute the MQFs is the same for $kN < v \le (k+1)N$, for $k \ge 1$.

Another approach used to solve the MQF problem is the coupled method, which does use inter-PE communications. Let PE(i,j) denote the PE in row i and column j in an a × b logical PE

grid, where $0 \le i < a$, $0 \le j < b$, and $a \times b = N$. The s-vectors are loaded into the PE memories such that an (r/a)-element subvector of the Hermitian of each s-vector, s_q^H, and an (r/b)-element subvector of at most $\lceil v/a \rceil$ s-vectors, s_q, are stored in each PE memory. Each PE also holds an $(r/a) \times (r/b)$ portion of M. After local computations are done, summations are performed, first within columns of PEs and later within rows of PEs, using the p-recursive doubling technique described in Subsection 5.5.

An implementation of these two approaches on the nCUBE 2 MIMD machine is shown in Figure 20, where r = 16, N = 16, 32, and 64, with b fixed at 16, and a varying such that $N = a \times b$. The execution times as a function of the number of steering vectors is shown for the uncoupled and coupled cases. As can be observed, the faster approach for a given N and r depends on v.

The uncoupled and coupled approaches can be combined by using the uncoupled approach to process $\lfloor v/N \rfloor \times N$ vectors and using the coupled approach to compute the remaining ones. For the nCUBE 2 implementation example of Figure 20, if v = 80 and $N = 64$, then 64 vectors could be processed by the uncoupled method (0.0049 seconds) and 16 vectors by the coupled method (0.0032 seconds), for a total time of 0.0081 seconds. To process 80 vectors by the uncoupled method by itself takes 0.0097 seconds, and the coupled method by itself takes 0.01 seconds.

Another variation is to exploit subtask partitioning, as was presented in Subsection 7.3. For the nCUBE 2 implementation example of Figure 20, processing 80 vectors using 64 PEs on the nCUBE 2 with the coupled approach requires 0.01 seconds, but by partitioning the machine into four submachines of 16 PEs, each processing 20 vectors, requires only 0.0077 seconds. Whether partitioning is advantageous depends on the values of v, r, and N.

In summary, choosing an optimal algorithm for the MQF problem is dependent on the characteristics of the input-data and the machine. By having a set of algorithms that solve the problem efficiently for various input-data parameters and system parameters, an algorithm (or

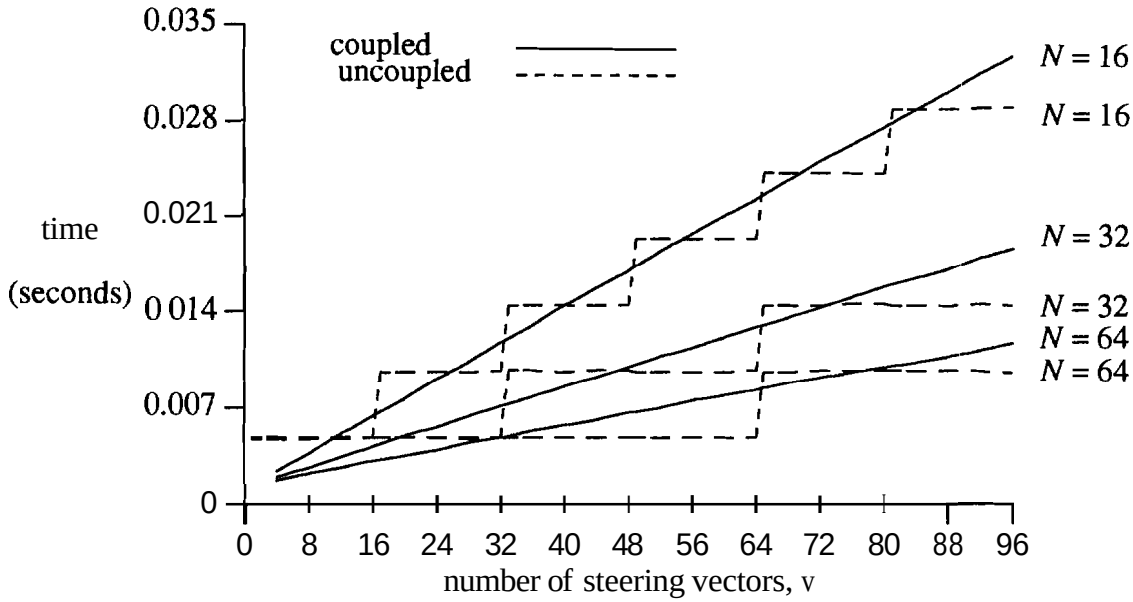

Figure 20: Execution time of the uncoupled and coupled data parallel methods for the nCUBE 2 for $r = 16$ and $N = 16, 32$ **and** 64.

combined algorithm) selection methodology can be implemented. .

9. CONCLUSIONS AND FUTURE DIRECTIONS

In designing algorithms for large-scale parallel machines, many issues must be addressed to devise an effective implementation. The importance of these issues cannot be overemphasized due to their direct impact on the performance of these algorithms. This report has surveyed some aspects of some of these issues. For more information, readers are encouraged to see the papers listed in the references and books on parallel algorithms, such as [Akl89, BeT89, Cha92, GoO93, JaG87, KrS92, KuG94, Kum91, LaD90, Mod88, Pit93]. Some potential future research problems in the design of data parallel algorithms are discussed below. However, the discussion is by no means exhaustive.

In Section 3, the impact of data distribution on the execution time was examined. It was shown that the best data distribution for one network may not be the best for another network. Future research is necessary to establish a methodology for determining an optimal data distribution for a given problem-machine combination. The factors that must be considered to design such a methodology for guiding data distribution decisions include (a) the problem parameters, (b) effectively utilizing the PEs, (c) the interconnection network topology, and (d) the inter-PE communication latency. The resulting methodology for data distribution should minimize the total overall execution time.

A machine-dependent issue that is not addressed in this report is the communication latency (i.e., the time needed to move a data item needed by one PE but located in another). Because this impacts the overall communication cost, to reduce the execution time, it is necessary to develop design techniques that reduce or hide the communication latency. The communication latency is dependent on factors such as (a) the software and hardware overhead for formatting the data to be transferred and establishing the path in the network, (b) the distance from the source PE to destination PE in number of links, (c) the amount of communication and computation overlap possible, and (d) the message size. One way to hide the latency is to initiate the data transfer before the data item is actually needed by the destination PE. Incorporating this

approach into one data parallel programming style, or having a compiler do it automatically, is important.

None of the case studies used in this report involved real-time I/O. One of the challenging areas where data parallel algorithms could be effectively applied is real-time processing, e.g., real-time image processing, where real-time I/O operations are integrated into the algorithm. The overall goal is to minimize or to hide the I/O latency. The issues that must be considered include (a) quantifying the I/O characteristics of the application, (b) characterizing the target machine, and (c) scheduling the I/O operations.

Algorithm scalability is another key issue that must be addressed in designing an algorithm for massively parallel machines. Ideally, an algorithm should be designed so that it can be scaled to future generation machines as they become available. Section 8 illustrated how to achieve scalability using a set of algorithms. One algorithm or combination of algorithms is selected based on the ratio between the data size and the target machine size to solve a given problem. Given the ratio, the algorithm(s) selected should achieve the best performance. Data parallel algorithm designers should consider this multiple algorithm approach to scalability. Research is needed to develop a methodology to automatically select an algorithm or combination of algorithms from a set of programs given information about the problem-size/machine-size ratio and other relevant information about the problem and the machine.

The examples of parallel matrix operations in Section 6 cover dense matrices. A related area that is receiving wide attention at present is operations on very large, very sparse matrices [KuG94]. Sparse matrix representation schemes impact the efficiency of the solution methods. The best serial algorithms cannot be easily parallelized. Hence, it is necessary to develop new algorithms and new representation schemes for sparse matrices that would effectively make use of massively parallel machines. For example, much research is needed in parallel sparse matrix factorization. Current understanding of this area is based on empirical studies. Further research is necessary to theoretically analyze the parallelism and scalability of this operation.

Exploiting concurrent execution of multiple subtasks can be important in reducing the execution time as demonstrated in Section 7. Given a partitionable machine and an application, a systematic approach is necessary to determine whether employing subtask parallelism will decrease the execution time [NaM93]. If subtask parallelism is to be employed, then it is necessary to determine the submachine sizes and the mapping of the subtasks onto the submachines that will result in the minimum execution time [ChD89]. The issues that need to be considered in the analysis include (a) identifying the individual subtasks, (b) determining the characteristics (computational requirements and modes of parallelism) of each subtask, (c) mapping the subtasks onto the submachines, and (d) determining the optimal sizes for the subnnachines. In the case of heterogeneous computing, where there are different parallel machines connected by high speed links, the analysis becomes even more complex (see [SiA95]).

An application that exhibits data parallelism can be implemented either on an SIMD machine or on an MIMD machine using SPMD mode. It is necessary to devise a systematic approach to aid the programmer in selecting the best type of machine architecture for the application under consideration. In making the choice, the trade-offs between SIMD and MIMD architectures discussed in [SiA95] must be considered. Once an architecture is chosen, the algorithm for the given application should be optimized with respect to the target architecture.

A data parallel algorithm can also be mapped onto a hybrid SIMD/MIMD mixed-mode machine (a survey of mixed-mode machines is in [SiA95]). When this is done, the programmer can specify which portion of the program should employ SIMD mode and which SPMD mode. In mapping an application onto a mixed-mode machine, two important issues that must be considered are: (a) where to switch modes within the program, and (b) how to identify the best execution mode (i.e., SIMD or SPMD) for each portion of the program. The use of both SIMD and SPMD modes to execute a single program becomes more complex when a suite of heterogeneous parallel machines is considered instead of a mixed-mode machine. In this case, mode switching implies switching between machines, which involves additional software and hardware overheads (see [SiA95]).

The previous paragraph discusses using both SIMD and SPMD modes within a single data parallel program. Mixed-mode machines and suites of heterogeneous parallel computers interconnected by high-speed links also offer the possibility of combining data parallelism and control parallelism (where each PE can follow a different program). The issues raised in the last paragraph can be extended to three choices: data parallelism using SIMD execution, data parallelism using SPMD execution, and control parallelism using MIMD execution. Designing such hybrid data/control parallel algorithms is another future direction for research.

An issue related to the data parallel algorithm design techniques is the development of an effective parallel programming environment [AdC94, Pan91]. Such an environrnent is crucial in supporting the implementation of data parallel algorithms on massively parallell machines. The components of this environment include (a) user-friendly interfaces in both textual and graphical forms, (b) portable parallel programming languages, (c) compilers, (d) libraries that contains optimized machine-dependent data parallel algorithms, and (e) tools (e.g., parallel program debuggers, performance analyzers). An effective parallel programming environment will reduce program development time, increase programmer productivity, ensure program portability, and improve performance. Increased programmer productivity is achieved by making use of the scalable libraries and tools. The portability is ensured by supporting the same parallel language and library functions across multiple hardware platforms. The improved performance can be attained by fine tuning the program using the tools and libraries provided.

Researchers have been working for many years to develop fully automatic techniques for parallelizing sequential algorithms [AdC94]. An alternative approach utilizes semi-automatic techniques for parallelization. The Data Parallel Meta Language (DPML) [FrM94] is one such effort. The idea is to develop a meta language to specify the communication patterns, data distribution strategies, and coordination of subtasks. The meta language provides; a model of the problem-machine combination to the compiler, so that the compiler can better parallelize the serial code. The input to the parallelizing compiler is the serial program augmented with the specification of the machine architecture and the problem characteristics in the meta language.

This meta language approach is also promising for heterogeneous computing. Further research is necessary to identify the information needed in a meta language and to design a meta language for heterogeneous computing environments.

This report has provided an overview of data parallel algorithm design techniques. The case studies used to illustrate the techniques and to highlight the impact of implementation decisions were based on the message-passing distributed-memory SIMD machine model. The analysis can be extended to other machine models as well, such as shared address space distributed-memory machines and various types of MIMD machines. However, additional analysis must be done to map the algorithms to those models efficiently, because of the different architectural properties of machines (e.g., multiple instruction streams). The remapping would not be necessary if the techniques discussed were based on an abstract machine model that subsumes all existing models. However, such a model does not presently exist. A future research problem is to develop one universal model or a small number of fundamental models that abstract the salient features of parallel machines and that will support machine-independent parallel programming. In addition, the model must provide realistic information on the relative costs of computation, communication, and synchronization [KoN93]. Therein lies the difficulty, i.e., developing a general machine-independent model that is "'precise enough' about performance without being 'too explicit' about the implementation details" of any machine [SiA92a]. The algorithm design techniques based on such a standard parallel machine model would be applicable to all existing parallel machine models. Thus, the design techniques for parallel algorithms could be unified. The advantages of this unified approach would include algorithm portability, algorithm scalability, run-time migration of tasks across parallel machines, and reduction in algorithm development time.

In summary, there is a great deal of activity in the field of data parallel algorithms, as evidenced in the sampling of relevant references listed at the end of this report. The goal of this report was to provide an introduction to some of the issues germane to constructing effective data parallel programs. This section has discussed a variety of areas for future research directly

or indirectly related to data parallel algorithms. In general, the future directions for research relating to data parallel algorithms should lead to the mapping of applications onto parallel machines in ways that most effectively exploit the computing power these machines provide. In many cases, this may enable the performance of application tasks that would be infeasible without the computing power of parallel machines. Finally, when contrasting the data parallel approach to the control parallel approach, many researchers feel that the data parallel paradigm must be employed, at least within subtasks, to take advantage of massively parallel machines with a thousand or more processors.

ACKNOWLEDGMENTS

The authors thank K. H. Casey, R. Gupta, J. M. Siegel, M. Tan, M. Theys, and A. Y. Zomaya for their comments.

REFERENCES

[AdC94] V. Adre, A. Carle, E. Granston, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, J. Mellor-Crurnmey, S. Warren, and C. Tseng, "Requirements for Data-Parallel Programming Environments," ***IEEE Parallel and Distributed Technology Systems and Applications,*** Vol.2, No. 3, Fall 1994, pp. 48-57.

[Akl89] S. G. Akl, ***The Design and Analysis of Parallel Algorithms,*** Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[ArN91] J. B. Armstrong, M. A. Nichols, H. J. Siegel, and L. H. Jamieson, "Examining the Effects of CU/PE Overlap and Synchronization Overhead When 'Using the Complete Sums Approach to Image Correlation," ***Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing,*** Dec. 1991, pp. 224 - 232.

[BaB68] G. H. Barnes, R. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes, "The Illiac IV Computer," ***IEEE Trans. Computers,*** Vol. C-17, No. 8, Aug. 1968, pp. 746-757.

[Bat74] K. E. Batcher, "STARAN Parallel Processor System Hardware," ***AFIPS 1974 National Computer Conf.,*** May 1974, pp. 405-410.

[Bat77] K. E. Batcher, "STARAN Series E," ***1977 Int'l Conf. on Parallel Processing,*** Aug. 1977, pp. 140-143.

[Bat80] K. E. Batcher, "Design of a Massively Parallel Processor," ***IEEE Trans. Computers,*** Vol. C-29, No. 9, Sep. 1980, pp. 836-844.

[BeS91] T. B. Berg and H. J. Siegel, "Instruction Execution Trade-offs for SIMD vs. MIMD vs. Mixed-Mode Parallelism," ***Fifth Int'l Parallel Processing Symposium,*** May 1991, pp. 301 - 308.

[BeT89] D. P. Bertsekas and J. N. Tsitsiklis, ***Parallel and Distributed Computation,*** Prentice-Hall, New Jersey, 1989.

[Bla90] T. Blank, "The MasPar MP-1 Architecture," ***IEEE Compcon,*** Feb. 1990, pp. 20-24.

[BlN92] T. Blank and J. R. Nickolls, "A Grimm Collection of MIMD Fairy Tales," ***Proceedings of the Fourth Symposium on Frontiers of Massively Parallel Computation,*** Oct. 1992, pp. 448-457.

[Cha92] P. Chaudhuri, ***Parallel Algorithms: Design and Analysis,*** Prentice Hall, New York, New York, 1992.

[ChD89] C. H. Chu, E. J. Delp, L. H. Jamieson, H. J. Siegel, F. J. Weil, and A. B. Whinston, "A Model for an Intelligent Operating System for Executing Image Understanding Tasks on a Reconfigurable Parallel Architecture," ***J. Parallel and Distributed Computing,*** Vol. 21, No. 1, Jun. 1989, pp. 598-622.

[DaG88] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister, "A Single-Program-Multiple-Data Computational Model for EPEX/FORTRAN," ***Parallel Computing,*** Vol 7, No. 1, Apr. 1988, pp. 11-24.

[DuB88] P. Duclos, F. Boeri, M. Auguin, and G. Giraudon, "Image Processing on a SIMD/SPMD Architecture: OPSILA," ***Ninth Int'l Conf. Pattern Recognition,*** Nov. 1988, pp. 430-433.

[FiC91] S. A. Fineberg, T. L. Casavant, and H. J. Siegel, "Experimental Analysis of a Mixed-Mode Parallel Architecture Using Bitonic Sequence Sorting," ***J. Parallel and Distributed Computing,*** Vol. 11, No. 3, Mar. 1991, pp. 239-251.

[Fly66] M. J. Flynn, "Very High-Speed Computing Systems," ***Proceedings of the IEEE,*** Vol. 54, No. 12, Dec. 1966, pp. 1901-1909.

[FrM94] R. S. Francis, I. D. Mathieson, P. G. Whiting, M. R. Dix, H. L. Davis, and L. D. Rotstayn, "A Data Parallel Scientific Modelling Language," ***J. Parallel and Distributed Computing,*** Vol. 21, No. 1, Apr. 1994, pp. 46-60.

[GiW92] N. Giolmas, D. W. Watson, D. M. Chelberg, and H. J. Siegel, "A Parallel Approach to Hybrid Range Image Segmentation," ***Proceedings of the Sixth Int'l Parallel Processing Symposium,*** Mar. 1992, pp. 334-342.

[GoO93] G. Golub and J. M. Ortega, ***Scientific Computing An Introduction with Parallel Computing,*** Academic Press, Inc., California, 1993.

[HaM89] J. P. Hayes and T. Mudge, "Hypercube Supercomputers," ***Proceedings of the IEEE,*** Vol 77, No. 12, pp. 1829-1841.

[Hil85] W. D. Hillis, ***The Connection Machine,*** MIT Press, Cambridge, Massachusetts, 1985.

[HiS86] W. D. Hillis and G. L,, Jr. Steele, "Data Parallel Algorithms," ***Communications of the ACM,*** Vol. 29, No. 12, Dec. 1986, pp. 1170-1183.

[HiT93] W. D. Hillis and L. W. Tucker, "The CM-5 Connection Machine: A Scalable Supercomputer," ***Communications of the ACM,*** Vol. 16, No. 11, Nov. 1993, pp. 31-40

[Hun89] D. J. Hunt, "AMT DAP - A Processor Array in a Workstation Environment," ***Computer Systems Science and Engineering,*** Vol. 4, No. 2, Apr. 1989, pp. 107-114.

[JaG87] L. H. Jamieson, D. Gannon, and R. J. Douglass, eds., ***The Characteristics of Parallel Algorithms,*** MIT Press, Cambridge, Massachusetts, 1987.

[JaM86] L. H. Jamieson, P. H. Mueller, Jr., and H. J. Siegel, "FFT Algorithms for SIMD Parallel Processing Systems," ***J. Parallel and Distributed Computing,*** Vol. 3, No. 1, Mar. 1986, pp. 48-71.

[Jam87] L. H. Jamieson, "Characterizing Parallel Algorithms," in ***The Characteristics of Parallel Algorithms,*** L. H. Jamieson, D. G. Gannon, and R. J. Donglass, eds., The MIT Press, Cambridge, Massachusetts, 1987.

[KiN91] S. D. Kim, M. A. Nichols, and H. J. Siegel, "Modeling Overlapped Operation Between the Control Unit and Processing Elements in an SIMD Machine," ***J. Parallel and Distributed Computing,*** Vol. 12, No. 4, Aug. 1991, pp. 329-342.

[KoN93] J. S. Kowalik and K. W. Neves, "Software for Parallel Computing: Key Issues and Research Directions," in ***Software for Parallel Computation,*** J. S. Kowalik and L. Grandinetti, eds., Springer-Verlag, Germany, 1993, pp. 3-33.

[KrM88] R. Krishnamurti and E. Ma, "The Processor Partitioning Problem in Special-Purpose Partitionable Systems," 1988 ***Int'l Conf. on Parallel Processing,*** Vol. I, Aug. 1988, pp. 434-443.

[KrS92] L. Kronsjo and D. Shumsheruddin, ***Advances in Parallel Algorithms,*** Halsted Press, New York, New York, 1992.

[KuG94] V. Kumar, A. Grama, A. Gupta, and G. Karypis, ***Introduction to Parallel Computing: Design and Analysis,*** The Benjamin/Cummings Publishing Company, Redwood City, California, 1994.

[Kum91] V. K. P. Kumar, ed., ***Parallel Architectures and Algorithms for Image Understanding,*** Academic Press, Boston, Massachusetts, 1991.

[LaD90] S. Lakshmivarahan and S. K. Dhall, ***Analysis and Design of Parallel Algorithms : Arithmetic and Matrix Problems***, McGraw-Hill, New York, New York, 1990.

[LiM87] G. J. Lipovski and M. Malek, ***Parallel Computing,*** John Wiley & Sons, New York, New York, 1987.

[Mas91] ***Data-Parallel Programming Guide,*** MasPar Computer Corporation, Sunnyvale, California, 1991.

[Mod88] J. J. Modi, ***Parallel Algorithms and Matrix Computation,*** Oxford University Press, New York, New York, 1988.

[NaM93] W. G. Nation, A. A. Maciejewski, and H. J. Siegel, "A Methodology for Exploiting Concurrency Among Independent Tasks in Partitionable Parallel Processing Systems," ***J. Parallel and Distributed Computing,*** Special Issue on Performance of Supercomputers, Vol. 19, No. 3, Nov. 1993, pp. 271-278.

[NaS93] W. G. Nation, G. Saghi, and H. J. Siegel, "Properties of Interconnection Networks for Large-Scale Parallel Processing Systems," ***Int'l Summer Institute on Parallel Architectures, Languages, and Algorithms,*** July 1993, pp. 51-82

[Nut77] G. J. Nutt, "Multiprocessor Implementation of a Parallel Processor," ***Proceedings of the Fourth Annual Symposium on Computer Architecture,*** 1977, pp. 147-152.

[Pan91] C. M. Pancake, "Software Support for Parallel Computing: Where Are We Headed?," ***Communications of the ACM,*** Vol. 34, No. 11, Nov. 1991, pp. 53-64.

[PhW93] M. Philippsen, T. Warschko, W. Tichy, and C. Herter, "Project Triton: Towards Improved Programmability of Parallel Machines," ***26th Hawaii Int'l Conf. System Sciences,*** Jan. 1993, pp. 192-201.

[Pit93] I. Pitas, ***Parallel Algorithms For Digital Image Processing, Computer Vision, and Neural Networks,*** John Wiley & Sons, New York, New York, 1993.

[SiA92a] H. J. Siegel, S. Abraham, W. L. Bain, K. E. Batcher, T. L. Casavant, D. DeGroot, J. B. Dennis, D. C. Douglas, T. Y. Feng, J. R. Goodman, A. Huang, H. F. Jordan, J. R. Jump, Y. N. Patt, A. J. Smith, J. E. Smith, L. Snyder, H. S. Stone, R. Tuck, and B. W. Wah, "Report of the Purdue Workshop on Grand Challenges in Computer Architecture for the Support of High Performance Computing," ***J. Parallel and Distributed Computing,*** Vol. 16, No. 3, Nov. 1992, pp. 199-211.

[SiA92b] H. J. Siegel, J. K. Antonio, and K. Liszka, "Metrics for Metrics: Why is it Difficult to Compare Interconnection Networks or How Would You Compare an Alligator to an Armadillo?," ***Proceedings of the New Frontiers: A Workshop on Future Directions of Massively Parallel Processing,*** Oct. 1992, pp. 97-106.

[SiA92c] H. J. Siegel, J. B. Armstrong, and D. W. Watson, "Mapping Computer-Vision-Related Tasks Onto Reconfigurable Parallel Processing Systems," ***Computer,*** Vol. 25, No. 2, Feb. 1992, pp. 54-63.

[SiA95] H. J. Siegel, J. K. Antonio, R. C. Metzger, M. Tan, and Y. A. Li, "Heterogeneous Computing," to appear in ***Handbook of Parallel and Distributed Computing,*** A. Y. Zomaya, ed., McGraw-Hill, 1995.

[Sie90] H. J. Siegel, ***Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies, Second Edition,*** McGraw-Hill, New York, New York, 1990.

[SiS81] H. J. Siegel, L. J. Siegel, F. C. Kernrnerer, P. T. Mueller, Jr., H. E. Smalley, Jr., and S. D. Smith, "PASM: A Partitionable SIMD/MIMD System for hnage Processing and Pattern Recognition," ***IEEE Trans. Computers,*** Vol. C-30, No. 12, Dec. 1981, pp. 934-947.

[SiS82] L. J. Siegel, H. J. Siegel, and A. E. Feather, "Parallel Processing Approaches to Image Correlation," ***IEEE Trans. Computers,*** Vol. C-31, Mar. 1982, pp. 208-218.

[SiS95] H. J. Siegel, T. Schwederski, W. G. Nation, J. B. Armstrong, L. Wang, J. T. Kuehn, R. Gupta, M. D. Allemang, D. G. Meyer, and D. W. Watson, "The Design and Prototyping of the PASM Reconfigurable Parallel Processing System," in ***Parallel Computing: Paradigms and Applications,*** A. Y. Zomaya, ed., Chapman and Hall, London, United Kingdom, 1995 (in press).

[Sto80] H. S. Stone, "Parallel Computers," in ***Introduction to Computer Architecture***, H. S. Stone, ed., Science Research Associates, Chicago, Illinois, 1980.

[TuR88] L. W. Tucker and G. G. Robertson, "Architecture and Applications of the Connection Machine," ***Computer,*** Vol. 21, No. 8, Aug. 1988, pp. 26-38.

[VaR94] A. Varma and C. S. Ragavendra, eds., ***Interconnection Networks for Multiprocessors and Multicomputers: Theory and Practice,*** *IEEE* Computer Society Press, Los Alamitos, California, 1994.

[WaN94] M-C Wang, W. G. Nation, J. B. Armstrong, H. J. Siegel, S. D. Kim, M. A. Nichols, and M. Gherrity, "Multiple Quadratic Forms: A Case Study in the Design of Data-Parallel Algorithms," **J. *Parallel and Distributed Computing,*** Vol 21, No. 1, Apr. 1994, pp. 124-139.

[Wil93] G. V. Wilson, "A Glossary of Parallel Computing Terminology," IEEE ***Parallel and Distributed Technology,*** Vol 1, No. 1, Feb. 1993, pp. 52-67.