

# Parallel Image Correlation: Case Study to Examine Trade-Offs in Algorithm-to-Machine Mappings \*

JAMES B. ARMSTRONG

*Advanced Product Development, Sarnoff Real Time Corporation, 301B College Road East, Princeton, NJ 08543-5202, USA*

jba@srtc.com

MUTHUCUMARU MAHESWARAN

MITCHELL D. THEYS

HOWARD JAY SIEGEL

MARK A. NICHOLS, AND KENNETH H. CASEY

*Parallel Processing Laboratory, School of Electrical and Computer Engineering, 1285 Electrical Engineering Building, Purdue University, West Lafayette, IN 47907-1285, USA*

maheswar@ecn.purdue.edu

theys@ecn.purdue.edu

hj@purdue.edu

**Editor:** Hamid R. Arabnia

**Abstract.** Performance of a parallel algorithm on a parallel machine depends not only on the time complexity of the algorithm, but also on how the underlying machine supports the fundamental operations used by the algorithm. This study analyzes various mappings of image correlation algorithms in SIMD, MIMD, and mixed-mode environments. Experiments were conducted on the Intel Paragon, MasPar MP-1, nCUBE 2, and PASM prototype. The machine features considered in this study include: modes of parallelism, communication/computation ratio, network topology and implementation, SIMD CU/PE overlap, and communication/computation overlap. Performance of an implementation can be enhanced by using algorithmic techniques that match the machine features. Some algorithmic techniques discussed here are additional communication versus redundant computation, data block transfers, and communication/computation overlap. The results presented are applicable to a large class of image processing tasks. Case studies, such as the one presented here, are a necessary step in developing software tools for mapping an application task onto a single parallel machine and for mapping the subtasks of an application task, or a set of independent application tasks, onto a heterogeneous suite of parallel machines.

**Keywords:** image correlation, Intel Paragon, MasPar MP-1, MIMD, mixed-mode, nCUBE 2, PASM prototype, scalability, SIMD.

## 1. Introduction

Performance of a parallel algorithm on a parallel machine depends not only on the time complexity of the algorithm, but also on how the underlying machine supports the fundamental operations used by the algorithm. This research is an application-driven study of the trade-offs that exist when a parallel implementation is designed

---

\* This work was supported by the National Aeronautical and Space Administration under grant number NGT-50961, by the Office of Naval Research under grant number N00014-90-J-1937, and by the DARPA/ITO Quorum Program under NPS subcontract number N62271-97-M-0900. Some of the equipment used was supported by the National Science Foundation under grant number CDA-9015696.

for a given task on a given target machine. The application considered, image correlation, is representative of a large class of window-based image processing techniques. In most low-level image processing tasks, the same set of instructions is applied to all of a two-dimensional array of picture elements (*pixels*) of an image, where each pixel is a grey-level value for that position in the image. Thus, most such image processing algorithms are data parallel in nature [33]. *Image correlation* (or image template matching) determines the “degree of similarity” between a *template* (i.e., a small image) and any area in the image with the same dimensions as the template.

Assume a distributed memory parallel machine with  $P$  PEs (a processing element consists of a processor and memory pair) connected via a logical mesh. The image is divided into  $P$  subimages that are distributed among the  $P$  PEs such that the PEs will need to interchange some of their pixels so the template can be fully matched at the “edges” of the subimages. In the *dynamic complete sums algorithm*, non-local pixels required by a PE are transferred only when they are first needed during the course of computation. As a variation to the dynamic complete sums algorithm, the *complete sums algorithm* performs all non-local pixel transfers before the start of any computation. In the *partial sums algorithm*, each PE performs all possible computations on its local data and then transfers its partial results to those PEs that require them. A mathematical study of the dynamic complete sums and partial sums algorithms is provided in [31]<sup>1</sup>. The operations performed in these algorithms for image correlation are representative of a wide variety of window-based image processing tasks, such as image smoothing, image convolution, and 2-D median filtering. Consequently, the analyses presented here can be extended to a large class of data-parallel algorithms. Section 2 describes the computations involved in the image correlation process. In Section 3, a summary of related work is presented. The three algorithms for image correlation are explained in Section 4.

Due to certain trade-offs between the SIMD and MIMD modes of parallelism, some sequences of instructions are performed better in one mode than in the other [26, 30]. For example, because of the single synchronized instruction stream in an SIMD program, the if-then-else clauses are serialized. This causes underutilization of PEs, because the PEs active for the “then” clause are disabled for the “else” clause and vice versa. However, due to the implicit synchronization in an SIMD program, less inter-PE communication overhead is incurred in SIMD execution. To take advantage of the benefits of both the SIMD and MIMD modes of parallelism, mixed-mode machines have been built. An SIMD/MIMD *mixed-mode* system can dynamically switch between the SIMD and MIMD modes of parallelism at instruction-level granularity with generally negligible overhead [30]. Examples of machines that have been built with mixed-mode capability are EXECUBE [15], MeshSP [11], OPSILA [6], PASM [27, 32], TRAC [18], and Triton/1 [22].

In this study, the three image correlation algorithms are implemented on four different parallel machines and the trade-offs are analyzed. The machines are: a commercial SIMD machine (MasPar MP-1 with 16K PEs), a commercial hypercube-based MIMD machine (nCUBE2 with 64 PEs), a commercial mesh-based MIMD machine (Intel Paragon with 140 PEs), and an experimental mixed-mode proto-

type (PASM with 16 PEs). Performance of a parallel algorithm on a parallel machine depends not only on the time complexity of the algorithm, but also on how the underlying machine supports the fundamental operations used by the algorithm. The machine features considered in this study include: modes of parallelism, communication/computation ratio, network topology and implementation, SIMD CU/PE overlap, and communication/computation overlap. Performance of an implementation can be enhanced by using algorithmic techniques that match the machine features. Some algorithmic techniques discussed here are additional communication versus redundant computation, data block transfers, and communication/computation overlap. Sections 5 through 7 present descriptions and analyses of the implementations on these machines.

Case studies, such as the one presented here, are necessary for understanding and quantifying the interaction of important application characteristics with important machine features. The knowledge gained from such studies is valuable in developing software tools for mapping an application onto a parallel machine or a set of independent application tasks onto a heterogeneous suite of parallel machines. Furthermore, studying the interaction of application characteristics with machine features provides valuable information that can be useful for developing methodologies for mixed-machine heterogeneous computing, where an application task is decomposed into subtasks and executed across a suite of different parallel machines [28].

## 2. Image Correlation

Image correlation involves determining the position at which a  $r$ -by- $c$  template  $t$  “best matches” an equal-sized portion  $y$  of an input image. That is, for an  $R$ -by- $C$  input image,  $I$  and a given image coordinate ( $row, col$ ) such that  $0 \leq row \leq (R-r)$  and  $0 \leq col \leq (C-c)$ ,  $y[k, l] = I[row + k, col + l]$  for all coordinates  $(k, l)$ , where  $0 \leq k < r$  and  $0 \leq l < c$ .

To measure the quality of fit of the template to the image data, a single number called the *coefficient of determination* [20, 31],  $\rho^2$ , is computed as follows, where the summations are performed over  $k$  and  $l$ , for  $0 \leq k < r$  and  $0 \leq l < c$  (i.e.,  $\sum \sum$  denotes  $\sum_{l=0}^{c-1} \sum_{k=0}^{r-1}$ ):

$$S_{tt} = \sum \sum t[k, l]^2 - \frac{1}{rc} (\sum \sum t[k, l])^2 \quad (1)$$

$$S_{ty} = \sum \sum t[k, l]y[k, l] - \frac{1}{rc} (\sum \sum t[k, l]) (\sum \sum y[k, l]) \quad (2)$$

$$S_{yy} = \sum \sum y[k, l]^2 - \frac{1}{rc} (\sum \sum y[k, l])^2 \quad (3)$$

$$\rho^2 = \frac{S_{ty}^2}{S_{tt}S_{yy}} \quad (4)$$

and is bounded such that  $\rho^2 \leq 1$ . When  $\rho^2 = 1$ , there exists a perfect match between the template  $t$  and the  $r$ -by- $c$  portion of the image  $y$ , whereas  $\rho^2 = 0$  implies that  $t$  and  $y$  are uncorrelated with respect to one another.

The serial image correlation algorithm can be viewed as sliding the template across the input image in row major order and computing  $\rho^2$  for each  $r$ -by- $c$  portion of the input image. The maximum  $\rho^2$  value and its location are the desired output. Therefore, once a  $\rho^2$  value is computed for a match position, it is compared to the current maximum  $\rho^2$  value and if it is greater, it replaces the maximum in value and location. If either  $S_{tt}$  or  $S_{yy}$  is zero,  $\rho^2 = \infty$ . This condition exists when either the template  $t$  or a portion of the input image  $y$  has the same value for every element. It is assumed that  $S_{tt}$  does not equal 0. For those match positions where  $S_{yy}$  is zero, the  $\rho^2$  value is not computed and thus the current maximum  $\rho^2$  value and its position are not altered. Typically, uniform areas (i.e., where  $S_{yy}$  is zero) correspond to background color or areas of no interest and therefore most applications ignore (bypass) such areas.

The execution time for the image correlation algorithm is dominated by the time to compute the  $\sum \sum t[i, j]y[i, j]$ ,  $\sum \sum y[i, j]$ , and  $\sum \sum y[i, j]^2$  values for all possible match positions. The  $\sum \sum t[i, j]$  and  $\sum \sum t[i, j]^2$  values involve only the template elements and are computed once. The execution time of the image correlation algorithm is independent of the actual pixel values in the image. Therefore, in the experiments reported here, the image is initialized by randomly generated pixel values and the template is initialized by copying the pixel values from a randomly chosen template position in the image.

### 3. Related Work

A square,  $C$ -by- $C$  input image matrix  $I$  and a square,  $c$ -by- $c$  template matrix  $t$  are considered here to compare the different algorithms presented in the reviewed papers. The serial image correlation algorithm has a complexity given by  $O(C^2c^2)$ . Because template matching is an important task that is carried out in many image processing applications, a great deal of effort has been spent on developing efficient parallel algorithms for this task. Many previous papers have examined different algorithms to perform template matching and implemented these algorithms using different interconnection networks (e.g., [2, 7, 8, 16, 17, 19, 23, 24]).

The work presented in our paper builds on the research described in the above papers and [31]. The major difference between the research presented here and the other papers is that the work presented here looks at issues involved in mapping image correlation algorithms onto different machines with different modes of parallelism, different types of interconnection networks, and how various implementations can exploit the particular machine features. For these examples of related work, the differences from the research here is described further in this section.

In [23], and [24] two algorithms for image correlation on MIMD hypercube multiprocessors are described. One algorithm assumes a fine-grain MIMD hypercube (i.e., the cost of an interprocessor communication is comparable to the cost of a basic arithmetic instruction) and the other assumes a medium-grain hypercube. The work presented in [23] and [24] is different from our work, because their goal was to find an efficient implementation on fine and coarse grained MIMD hypercubes.

Whereas, the goal of the study presented here is to examine the issues involved in mapping the image correlation algorithms onto a variety of machines.

In [7], two algorithms for SIMD hypercube computers is described, one for  $C^2c^2$  processors and another one for  $C^2$  processors. Because these algorithms require  $C^2$  or more number of processors, they are different from the algorithms considered in this research. Also, a similar difference exists between the algorithm described in [8] and our work. Three different interconnection networks are used in [8] and an algorithm with  $P = C^2$  is presented for each network that can solve the template matching problem in  $O(c^2)$  time. A generalized convolution algorithm for a mesh architecture is presented in [19]. This algorithm initially assumes  $P = C^2$ , and discusses  $P < C^2$  without showing any actual algorithms. The approach in [19] differs from the one presented here because here algorithms are given for  $P < C^2$ . In [16], the authors designed “simple elegant parallel algorithms” for template matching using an SIMD hypercube. The first algorithm uses  $P = C^2$  processors, the second algorithm uses  $P = C^2c^2$  processors, and the third algorithm uses  $P = O(C^2)$  processors. The approaches presented in [16] differ significantly from the one presented here because here  $P < C^2$ , and they do not embed a mesh in the hypercube. A mesh connected array processor arrangement is used in [17] to implement a 2-D convolution scheme where  $P = C^2$ . The paper examines using diamond, rectangular, and round templates. The approach presented in [17] differs from the one presented here because here  $P < C^2$ , and here we are concerned with only rectangular templates.

A parallel stereocorrelation algorithm is used to study a reconfigurable multi-ring network (RMRN) in [2]. Stereocorrelation is a statistical procedure that derives depth information from a pair of pictures of the same scene but from different positions. The computation in stereocorrelation is similar to that in the image correlation that is discussed in this paper. The work presented in [2] is different from that here because in [2] stereocorrelation is used as an example to study the properties of the RMRN network and here image correlation is used to study the trade-offs in mapping an algorithm onto different parallel machines.

## 4. Parallel Algorithm Mappings

### 4.1. Common Portion of the Parallel Algorithm Mappings

It is assumed that the  $P$  PEs are logically arranged as a  $\sqrt{P}$ -by- $\sqrt{P}$  array of PEs (the physical interconnection of the PEs may not correspond to this). PE  $M$ , for  $0 \leq M < P$ , is located at position  $(m, n)$  for  $m, n < \sqrt{P}$  and  $M = m\sqrt{P} + n$ . The input image is dimensioned  $R$ -by- $C$ , where  $R, C \gg \sqrt{P}$  and both  $R$  and  $C$  are multiples of  $\sqrt{P}$ . The input image is partitioned into  $P$  subimages (as shown in Figure 1) and each PE’s subimage is initially dimensioned as an  $R_S$ -by- $C_S$  matrix, where  $R_S = R/\sqrt{P}$  and  $C_S = C/\sqrt{P}$ . To accommodate the pixel data that is transferred into a PE from other PEs, each PE’s initial subimage is extended by  $r - 1$  rows and  $c - 1$  columns, for a total size of  $(R_S + r - 1)$ -by- $(C_S + c - 1)$  (where the initial subimage is in rows 0 to  $R_S - 1$  and columns 0 to  $C_S - 1$ ). The

template is dimensioned  $r$ -by- $c$ , where  $r \leq R_S$  and  $c \leq C_S$ . Because the template's dimensions are typically small, a copy of the entire template is stored on each PE. The template and the subimages are stored in row major order and accessed by pointers as if they were one-dimensional arrays.

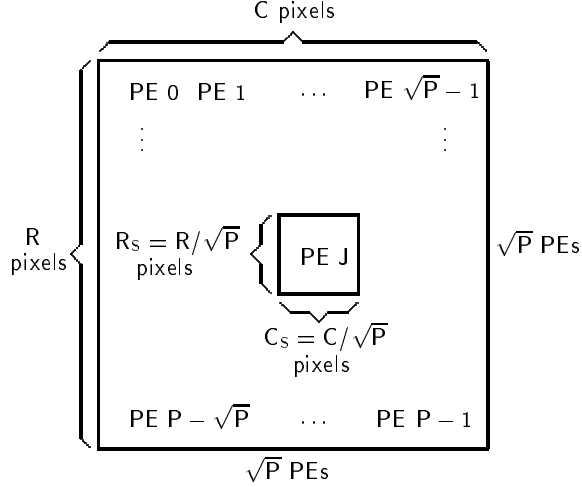


Figure 1. Partitioning an  $R$ -by- $C$  input image into  $P$   $R_S$ -by- $C_S$  PE subimages.

For these parallel algorithm mappings, each  $r$ -by- $c$  portion of a PE's input subimage for which  $\rho^2$  is computed is referred to as a *match position* and represented symbolically as `match_position[i, j]`, where  $(i, j)$  is the PE-subimage coordinate that overlaps the pixel in the upper left corner of the template.

The algorithm can be decomposed into three distinct phases. *Phase I* consists of the parallel computation of  $S_{tt}$ . In *Phase II*, all PEs can simultaneously compute  $S_{ty}$ ,  $S_{yy}$ , and  $\rho^2$  values for each `match_position[i, j]`, where  $0 \leq i < R_S$  and  $0 \leq j < C_S$  (i.e., wherever the upper corner of the template overlaps a position within a PE's subimage). To reduce extraneous loops, computations for the  $S_{ty}$ ,  $S_{yy}$ , and  $\rho^2$  terms are interleaved, so that for each new match position all three terms are calculated before moving to the next match position. After each of the PEs determines which of its  $R_S C_S$  local  $\rho^2$  values is its local maximum value, *Phase III* utilizes a recursive doubling operation to combine the  $P$  local maximum values of  $\rho^2$  into one global maximum value.

For match positions where the portion of the input image is not fully contained within a single PE's  $R_S$ -by- $C_S$  subimage, information will need to be transferred among PEs. By adhering to the restriction that  $r \leq R_S$  and  $c \leq C_S$ , the transferred data comes from immediately adjacent PEs. To complete all of the  $R_S C_S$  match position computations, all PEs require information from three neighboring PEs: PE  $J$  would communicate with PE  $J + 1$ , PE  $J + \sqrt{P}$ , and PE  $J + \sqrt{P} + 1$ . The PEs at the opposite "edges" of the logical mesh (e.g., PE  $\sqrt{P} - 1$  and PE  $\sqrt{P}$ ) do not need to communicate.

Phase I consists of the parallel computation of a value for the term  $S_{tt}$  using Equation (1). Because the equation requires template values and no input image values,  $S_{tt}$  is computed only once. If the template's dimensions are such that  $rc \geq P$ , then all  $P$  PEs can participate in the parallel computation; otherwise, only  $rc < P$  PEs can participate. To start the parallel computation, each of the  $P$  PEs is assigned  $\lfloor rc/P \rfloor$  template elements and  $(rc) \bmod P$  of those PEs are also assigned one of the remaining template elements. Next, each PE computes  $t^2$  for each of the template elements it holds, followed by each PE computing the local sum of its  $t$  values and the local sum of its  $t^2$  values. Finally, recursive doubling operations are used to obtain the global sums  $\sum \sum t[i, j]^2$  and  $\sum \sum t[i, j]$  from which each PE can compute  $S_{tt}$  via Equation (1).

Equation (3) is used to calculate  $S_{yy}$ , in parallel on all PEs, for each of the  $R_S C_S$  distinct match positions. A brief description of how  $S_{yy}$  is computed is necessary to understand the trade-offs discussed later. The terms  $\sum \sum y[l, k]$  and  $\sum \sum y[l, k]^2$ , referred to symbolically as  $\mathbf{ysum}[i, j]$  and  $\mathbf{ysumsq}[i, j]$ , respectively, for  $\mathbf{match\_position}[i, j]$ , are computed for all match positions within each PE's  $R_S$ -by- $C_S$  subimage based on the serial algorithm presented in [31]. To assist in the computation of  $\mathbf{ysum}[i, j]$  and  $\mathbf{ysumsq}[i, j]$ , two single array data structures,  $\mathbf{colsum}[\mathbf{k}]$  and  $\mathbf{colsumsq}[\mathbf{k}]$ , for  $0 \leq k \leq C_S$ , are used as intermediate storage. If  $\mathbf{subimage}[i, j]$  represents the pixel at coordinate  $(i, j)$  of a PE's subimage,  $\mathbf{colsum}[\mathbf{k}]$  is computed for the first row in each subimage via:

$$\mathbf{colsum}[\mathbf{k}] = \sum_{i=0}^{r-1} \mathbf{subimage}[i, \mathbf{k}] \text{ and } \mathbf{colsumsq}[\mathbf{k}] = \sum_{i=0}^{r-1} (\mathbf{subimage}[i, \mathbf{k}])^2 \quad (5)$$

When progressing from  $\mathbf{match\_position}[i-1, C_S-1]$  to  $\mathbf{match\_position}[i, 0]$  where  $1 \leq i < R_S - (r-1)$ , the  $\mathbf{colsum}[\mathbf{k}]$  and  $\mathbf{colsumsq}[\mathbf{k}]$  arrays are updated for  $0 \leq j < C_S$  via:

$$\mathbf{colsum}[j] = \mathbf{colsum}[j] - \mathbf{subimage}[i-1, j] + \mathbf{subimage}[i+r-1, j] \text{ and} \quad (6)$$

$$\mathbf{colsumsq}[j] = \mathbf{colsumsq}[j] - \mathbf{subimage}[i-1, j]^2 + \mathbf{subimage}[i+r-1, j]^2 \quad (7)$$

For  $\mathbf{match\_position}[i, j]$ , where  $R_S - (r-1) \leq i < R_S$ , the above equations can be used with additional data transfers. For  $\mathbf{match\_position}[i, 0]$ , where  $0 \leq i < R_S$ :

$$\mathbf{ysum}[i, 0] = \sum_{k=0}^{c-1} \mathbf{colsum}[\mathbf{k}] \text{ and } \mathbf{ysumsq}[i, j] = \sum_{k=0}^{c-1} \mathbf{colsumsq}[\mathbf{k}] \quad (8)$$

For  $\mathbf{match\_position}[i, j]$ , where  $0 \leq i < R_S$ ,  $1 \leq j < C_S - (c-1)$ :

$$\mathbf{ysum}[i, 0] = \mathbf{ysum}[i, j-1] - \mathbf{colsum}[j-1] + \mathbf{colsum}[j+c-1] \text{ and} \quad (9)$$

$$\mathbf{ysumsq}[i, j] = \mathbf{ysumsq}[i, j] - \mathbf{colsumsq}[j-1] + \mathbf{colsumsq}[j+c-1] \quad (10)$$

For  $\mathbf{match\_position}[i, j]$ , where  $0 \leq i < R_S$  and  $C_S - (c-1) \leq j < C_S$ , data needs to be transferred. The data that is transferred differs between the dynamic

complete and partial sums algorithms as discussed in the next subsection. Once each PE has calculated  $\mathbf{y}_{\text{sum}}[\mathbf{i}, \mathbf{j}]$  and  $\mathbf{y}_{\text{sumsq}}[\mathbf{i}, \mathbf{j}]$ , each PE can compute  $S_{yy}$  for  $\text{match\_position}[\mathbf{i}, \mathbf{j}]$  via Equation (3).

Equation (2) is used to calculate  $S_{ty}$  in parallel on all PEs, for each of the  $R_S C_S$  distinct match positions. The three terms  $\sum \sum t[l, k]$ ,  $\sum \sum y[l, k]$ , and  $\sum \sum t[l, k]y[l, k]$  are computed for all match positions within each PE's  $R_S$ -by- $C_S$  subimage. Of these three terms,  $\sum \sum t[l, k]$  and  $\sum \sum y[l, k]$  are computed as discussed above. The remaining term,  $\sum \sum t[l, k]y[l, k]$ , is computed directly.

Because the value of  $S_{tt}$  does not change with the match position, the term  $\rho' = S_{ty}^2/S_{yy}$  is computed in parallel on all PEs, for each of the  $R_S C_S$  distinct match positions. Therefore, because  $S_{tt}$  is known to be non-negative, the match position that yields the maximum  $\rho'$  value would yield the maximum  $\rho^2$  value (Equation (4)).

Additionally, for each match position, two data-dependent conditionals are performed. One conditional determines whether  $S_{yy}$  is zero and the other is used to determine if the new  $\rho'$  value exceeds the current  $\rho'$  maximum. Once each PE has found its local  $\rho'$  maximum and its corresponding subimage location, a recursive doubling operation is used to find the global maximum and its corresponding input image location. To determine the location of the global maximum  $\rho'$ , a *position indicator* is used to encode the global coordinates into a single number to reduce inter-PE communications. For instance, if PE  $J$ 's local maximum  $\rho'$  value occurred at  $\text{match\_position}[\mathbf{i}, \mathbf{j}]$ , let  $i_{\text{global}} = (\lfloor J/\sqrt{P} \rfloor R_S + i)$  and  $j_{\text{global}} = (J \bmod \sqrt{P})C_S + j$ . Then the position indicator ( $pos$ ) is:  $pos = \sqrt{P}C_S i_{\text{global}} + j_{\text{global}}$ . Each PE passes the position indicator along with its maximum  $\rho'$  value to the recursive doubling routine. The routine's output is the ordered pair  $(\rho', pos)$ , where  $\rho'$  is the global maximum and  $pos$  is its corresponding location. Finally, the coefficient of determination,  $\rho^2 = \rho'/S_{tt}$ , and the coordinate  $(u, v)$  are computed, where  $u = \lfloor pos/(\sqrt{P}C_S) \rfloor$  and  $v = pos \bmod \sqrt{P}C_S$ .

## 4.2. Unique Portions of the Mappings

### 4.2.1. Dynamic Complete Sums versus Complete Sums

The only difference between the dynamic complete sums and complete sums algorithms is when data transfers occur in Phase II. In both algorithms, the information transferred among PEs are pixels. In the complete sums algorithm, all non-local pixels are transferred before the template traverses the input image. The dynamic complete sums algorithm transfers the non-local pixels during the template traversal only when they are first needed. Because the complete sums approach isolates the non-local pixel transfers from the match position computation, additional nested loops are used for cycling through the transfers and thus contributes additional loop overhead. However, by interleaving the transfers with computation in the dynamic complete sums approach, more overhead is associated with addressing the appropriate pixel to transfer and the location in which to place the received pixel. The overhead associated with both algorithms is shown in Section 5 to be offsetting. The isolation

of the inter-PE transfers in complete sums makes the use of message blocks and computation/communication overlap more viable.

The technique used by the dynamic complete sums algorithm for interleaving these transfers with the match position computation is discussed further in this subsection. Furthermore, in this subsection it is explained why in the dynamic complete sums algorithm it is sufficient for PE  $J$  to receive data from PE  $J + 1$  and PE  $J + \sqrt{P}$  (see Figure 2(a)).

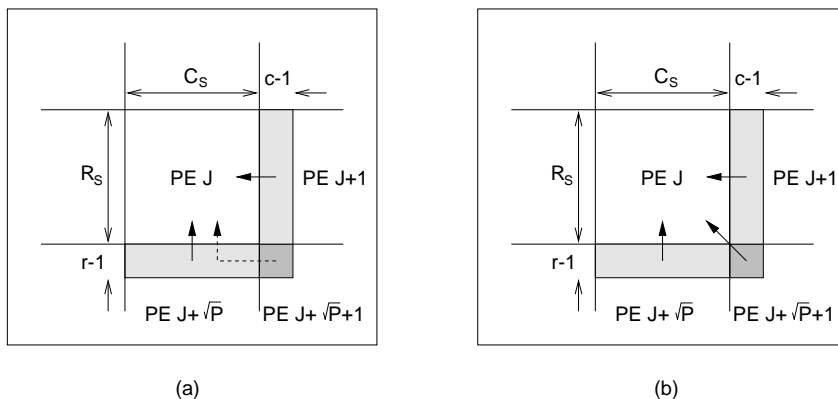


Figure 2. The PEs from which PE  $J$  receives data in the (a) dynamic complete sums and (b) partial sums algorithms.

First, consider match positions within PE  $J$  whose associated computations require the transfer of pixels from PE  $J + 1$ , i.e., each `match_position`[ $i$ ,  $j$ ] where  $0 \leq i < R_S - r$  and  $C_S - (c - 1) \leq j < C_S$ . Over all such positions the associated computations require all of PE  $J + 1$ 's pixels from columns 0 to  $c - 2$  of rows 0 to  $R_S - 1$  to be transferred to PE  $J$ . For each `match_position`[ $i$ ,  $j$ ] where  $i = 0$  and  $C_S - (c - 1) \leq j < C_S$  (i.e., those match positions in row 0), it is necessary for  $r$  pixels to be transferred for the associated computations. However, for each `match_position`[ $i$ ,  $j$ ] where  $1 \leq i < R_S - r$  and  $C_S - (c - 1) \leq j < C_S$  (i.e., those match positions not in row 0), only one new pixel needs to be transferred from PE  $J + 1$  because all of the other pixels are transferred and stored during previous match position computations. The example illustrated in Figure 3 uses a five-by-four template and a `match_position`[ $i$ ,  $j$ ] for  $r - 1 \leq i < R_S - r$  and  $C_S - (c - 1) \leq j < C_S$ . The pixels marked by a symbol other than a dot are those initially not resident in PE  $J$ . However, for a particular `match_position`[ $i$ ,  $j$ ], only the single pixel indicated by the + needs to be transferred. More precisely, at a `match_position`[ $i$ ,  $j$ ], the pixel at coordinate  $(e, f)$  within PE  $J + 1$  is transferred to PE  $J$  where  $e = i + (r - 1)$  and  $f = j - C_S + (c - 1)$ . The pixels marked by x's were previously transferred for `match_position`[ $u$ ,  $v$ ], where  $i - (r - 1) \leq u < i$  and  $C_S - (c - 1) \leq v \leq j$ . Those pixels marked by o's were previously transferred for `match_position`[ $w$ ,  $z$ ], where  $w = i$  and  $C_S - (c - 1) \leq z \leq j$ . By bringing in

one pixel at a time and only when first needed, no extraneous loops are generated, which minimizes loop overhead.

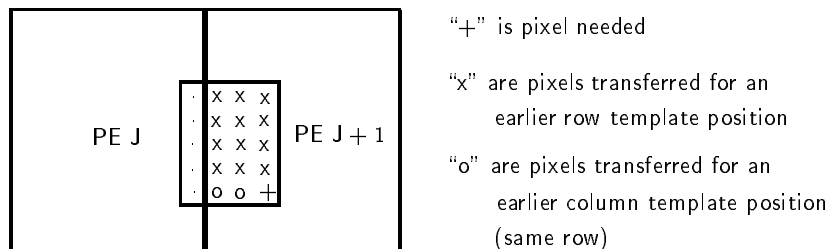


Figure 3. Pixel transfers from PE  $J + 1$  to PE  $J$ .

Next, consider match positions within PE  $J$  whose associated computations require the transfer of pixels from PE  $J + \sqrt{P}$ , i.e., each `match_position`[ $i$ ,  $j$ ] where  $R_S - (r - 1) \leq i < R_S$  and  $0 \leq j < C_S - c$ . Over all such positions, the associated computations require all of the pixels from rows 0 to  $r - 2$  of columns 0 to  $C_S - 1$  to be transferred to PE  $J$  from PE  $J + \sqrt{P}$ . For `match_position`[ $i$ ,  $j$ ] where  $R_S - (r - 1) \leq i < R_S$  and  $C_S - c \leq j < C_S$ , some pixels are required from PE  $J + \sqrt{P} + 1$ . Specifically, the pixels for rows 0 to  $r - 2$  of columns 0 to  $c - 2$  are needed. However, at this point in the algorithm, these pixels have already been transferred to PE  $J + \sqrt{P}$  and are in rows 0 to  $r - 2$  of columns  $C_S$  to  $C_S + c - 1$  of the subimage in PE  $J + \sqrt{P}$ . Thus, explicit communication with PE  $J + \sqrt{P} + 1$  is not necessary. For each `match_position`[ $i$ ,  $j$ ], where  $R_S - (r - 1) \leq i < R_S$  and  $j = 0$  (i.e., those match positions including column 0), it is necessary for  $c$  pixels to be transferred for the associated computations. For each `match_position`[ $i$ ,  $j$ ] where  $R_S - (r - 1) \leq i < R_S$  and  $1 \leq j < C_S$  (i.e., those match positions not including column 0), only one new pixel needs to be transferred from PE  $J + \sqrt{P}$  because all of the other pixels are transferred and stored during previous match position computations. Similar to Figure 3, the example shown in Figure 4 uses a five-by-four template and a `match_position`[ $i$ ,  $j$ ] for  $R_S - (r - 1) \leq i < R_S$  and  $1 \leq j < C_S$ . At a `match_position`[ $i$ ,  $j$ ], the pixel at coordinate  $(e, f)$  within PE  $J + \sqrt{P}$  is transferred to PE  $J$  where  $e = i - R_S + (r - 1)$  and  $f = j + (c - 1)$ .

**4.2.2. Dynamic Complete Sums versus Partial Sums** The partial sums algorithm stands in contrast to the dynamic complete sums and complete sums algorithms by the number of transfers, the information transferred, and the amount of computation done in Phase II. Figure 2(b) shows the PEs from which PE  $J$  receives data in the partial sums algorithm. In general, the partial sums algorithm does less additions and multiplications but more transfers than the other two algorithms.

At each match position, the sums  $\sum \sum t[i, j]y[i, j]$ ,  $\sum \sum y[i, j]$ , and  $\sum \sum y[i, j]^2$  are computed and used to compute  $S_{ty}$  and  $R_{ty}$ . In the dynamic complete sums algorithm, pixels were transferred to compute these sums for match positions whose

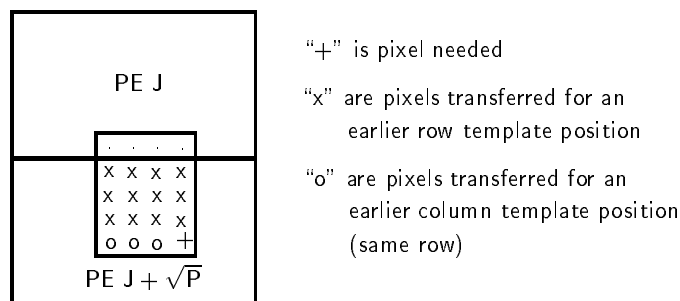


Figure 4. Pixel transfers from PE  $J + \sqrt{P}$  to PE  $J$ .

computations required pixels not in the local PE memory. However, in the partial sums algorithm, each PE computes as much of these summations as possible with pixels it contains in its local subimage. For those match positions whose associated computations require pixels from an adjacent PE, partial sums rather than unprocessed pixels, are received from the adjacent PEs. Specifically, PE  $J$  receives a partial sum from PE  $J + 1$  to complete the computations associated with `match_position[i, j]`, where  $0 \leq i < R_S - r$  and  $C_S - (c - 1) \leq j < C_S$ .

Figure 5 depicts an example of this process for an  $r = 5$  and  $c = 4$  template. For each of the above sums, the pixels marked by x's are used by PE  $J + 1$  to compute its partial sum and those pixels denoted by +'s are used by PE  $J$  to compute its partial sum. PE  $J + 1$  sends its x partial sum to PE  $J$  which adds it to its + partial sum to form the total sum for `match_position[i,  $C_S - 1$ ]` in PE  $J$ . Furthermore, for the  $y[\ ]$  and  $y[\ ]^2$  sums, PE  $J + 1$  uses its x partial sum (together with its pixels labeled o in Figure 5) to form its total sums for `match_position[i, 0]` in PE  $J + 1$ .

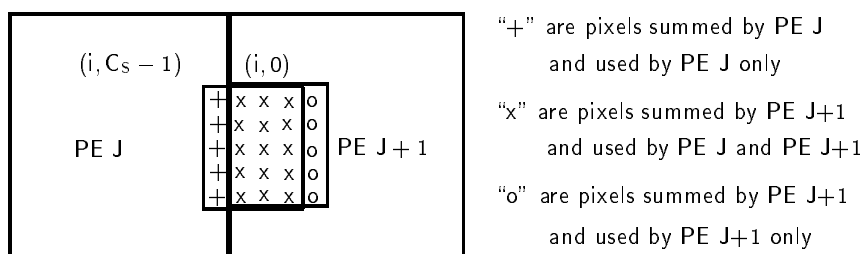


Figure 5. Pixels summed by PE  $J + 1$  and used by PE  $J$  as well as PE  $J + 1$  for computation of  $\sum_{l=0}^{r-1} \sum_{k=0}^{c-1} t[l, k]y[l, k]$  and  $\sum_{l=0}^{r-1} \sum_{k=0}^{c-1} y[l, k]$  for their respective match positions.

In general, the portion of the three sums that will be computed by PE  $J + 1$  is the part that uses the pixels in rows  $i$  to  $i + r - 1$  of columns 0 to  $j + c - C_S - 1$  of

PE  $J+1$ 's subimage. Using the  $y[\ ]$  notation of Section 2, the partial sums that are transferred from PE  $J+1$  for these match positions are  $\sum_{l=0}^{r-1} \sum_{k=C_S-j}^{c-1} t[l, k]y[l, k]$ ,  $\sum_{l=0}^{r-1} \sum_{k=C_S-j}^{c-1} y[l, k]$ , and  $\sum_{l=0}^{r-1} \sum_{k=C_S-j}^{c-1} y[l, k]^2$ . These partial sums are added in PE  $J$  to the corresponding partial sums (calculated locally in PE  $J$ )  $\sum_{l=0}^{r-1} \sum_{k=0}^{C_S-j-1} t[l, k]y[l, k]$ ,  $\sum_{l=0}^{r-1} \sum_{k=0}^{C_S-j-1} y[l, k]$ , and  $\sum_{l=0}^{r-1} \sum_{k=0}^{C_S-j-1} y[l, k]^2$ , respectively, to compute the desired sums  $\sum_{l=0}^{r-1} \sum_{k=0}^{c-1} t[l, k]y[l, k]$ ,  $\sum_{l=0}^{r-1} \sum_{k=0}^{c-1} y[l, k]$ , and  $\sum_{l=0}^{r-1} \sum_{k=0}^{c-1} y[l, k]^2$ , respectively.

Because there are three partial sums that are computed for each `match_position[i, j]`, for  $0 \leq i < R_S - r$  and  $C_S - (c - 1) \leq j < C_S$ , three values are transferred by the partial sums algorithm as opposed to only one raw pixel transfer in the dynamic complete sums algorithm. However, less computation is required by the partial sums algorithm for the following reason. When PE  $J+1$  computes the partial sums  $\sum_{l=0}^{r-1} \sum_{k=1}^{c-1} y[l, k]$  and  $\sum_{l=0}^{r-1} \sum_{k=1}^{c-1} y[l, k]^2$  for the computation associated with `match_position[i, C_S - 1]`,  $0 \leq i < R_S - r$ , in PE  $J$ , it has computed the partial sums for its own `match_position[i, 0]`. Thus, in the partial sums algorithm, the computation of  $\sum_{l=0}^{r-1} \sum_{k=0}^{c-2} y[l, k]$  and  $\sum_{l=0}^{r-1} \sum_{k=0}^{c-2} y[l, k]^2$  at `match_position[i, 0]`,  $0 \leq i < R_S - r$ , in PE  $J+1$  and `match_position[i, C_S - 1]`,  $0 \leq i < R_S - r$ , in PE  $J$  is done once (by PE  $J+1$ ). In the dynamic complete sums algorithm this computation is done by both PE  $J+1$  and PE  $J$ , creating redundant computation.

Next, consider match positions within PE  $J$  whose computations require data from PE  $J + \sqrt{P}$ , i.e., each `match_position[i, j]`, where  $R_S - (r - 1) \leq i < R_S$  and  $0 \leq j < C_S - c$ . The information transferred from PE  $J + \sqrt{P}$  to PE  $J$  is the partial sums  $\sum_{l=(R_S-i)}^{r-1} \sum_{k=0}^{c-1} t[l, k]y[l, k]$ ,  $\sum_{l=(R_S-i)}^{r-1} \sum_{k=0}^{c-1} y[l, k]$ , and  $\sum_{l=(R_S-i)}^{r-1} \sum_{k=0}^{c-1} y[l, k]^2$ . These are added in PE  $J$  to the partial sums (calculated locally in PE  $J$ )  $\sum_{l=0}^{R_S-i-1} \sum_{k=0}^{c-1} t[l, k]y[l, k]$ ,  $\sum_{l=0}^{R_S-i-1} \sum_{k=0}^{c-1} y[l, k]$ , and  $\sum_{l=0}^{R_S-i-1} \sum_{k=0}^{c-1} y[l, k]^2$  to get the desired sums. Once again three values are transferred for each `match_position[i, j]`,  $R_S - (r - 1) \leq i < R_S$  and  $0 \leq j < C_S - c$ , as opposed to one raw pixel transfer in the dynamic complete sums algorithm. However, in the partial sums algorithm there is no redundant computation by PE  $J$  and PE  $J + \sqrt{P}$  as there is in the dynamic complete sums algorithm for a similar reason as that described above.

Finally, because the partial sum quantities are transferred in the partial sums algorithm, communication must occur between PE  $J$  and PE  $J + \sqrt{P} + 1$ . Specifically, `match_position[i, j]`,  $R_S - (r - 1) \leq i < R_S$  and  $C_S - (c - 1) \leq j < C_S$ , in PE  $J$  requires from PE  $J + \sqrt{P} + 1$  the partial sums  $\sum_{l=(R_S-i)}^{r-1} \sum_{k=C_S-j}^{c-1} t[l, k]y[l, k]$ ,  $\sum_{l=(R_S-i)}^{r-1} \sum_{k=C_S-j}^{c-1} y[l, k]$ , and  $\sum_{l=(R_S-i)}^{r-1} \sum_{k=C_S-j}^{c-1} y[l, k]^2$ . As mentioned earlier, in the dynamic complete sums algorithm, PE  $J$  needs to communicate with only PE  $J+1$  and PE  $J + \sqrt{P}$ . In the partial sums algorithm, PE  $J$  communicates with PE  $J+1$ , PE  $J + \sqrt{P}$ , and PE  $J + \sqrt{P} + 1$ .

The fundamental trade-off between the partial sums algorithm and the dynamic complete sums algorithm is communication overhead versus computation cost. The total number of data items (partial sums) transferred by the partial sums algorithms

is three times the number of data items (pixels) transferred by the dynamic complete sums algorithm, i.e.,  $2R_S(c-1) + 2C_S(r-1) + 2(r-1)(c-1)$  more data items are transferred by the partial sums algorithm compared to the dynamic complete sums algorithm [31]. However, because the dynamic complete sums algorithm has some redundant computations when computing the sums  $\sum_{l=0}^{r-1} \sum_{k=0}^{c-1} y[l, k]$  and  $\sum_{l=0}^{r-1} \sum_{k=0}^{c-1} y[l, k]^2$  for match positions at the “edge” of each subimage, the partial sums algorithm performs  $R_S(c-1) + C_S(r-1) + (r-1)(c-1)$  less multiplications and  $8R_S + 8C_S + 3r + 3c - 6$  less additions [31]. Table 1, based on [31], contrasts the algorithm complexities of the dynamic complete sums and partial sums for square images ( $R = C$  and  $R_S = C_S$ ) and square templates ( $r = c$ ).

Table 1. Operation count comparison between the dynamic complete sums and partial sums algorithms [31].

Operation Count	Dynamic Complete Sums Algorithm	Partial Sum Algorithm
Number of Additions	$7C^2/P + 8Cc/\sqrt{P} + C^2c^2/P + 2c^2 - 2c$	$7C^2/P + 8Cc/\sqrt{P} - 16C/\sqrt{P} + C^2c^2/P + 2c^2 - 8c + 6$
Number of Multiplication	$(C/\sqrt{P} + c - 1)^2 + C^2c^2/P$	$C^2c^2/P + C^2/P$
Number of Value Transfers	$(c - 1)^2 + 2C(c - 1)/\sqrt{P}$	$3(c - 1)^2 + 6C(c - 1)/\sqrt{P}$

For all these algorithms, by increasing the size of the input image while the template size and the number of PEs remains fixed, the amount of computation increases much faster than the amount of inter-PE communication. This is because the number of data items to transfer is  $O(R_S \cdot c + C_S \cdot r + r \cdot c)$  and the computation is  $O(R_S \cdot C_S \cdot r \cdot c)$ .

## 5. PASM Implementations

### 5.1. Overview of PASM

The PASM small-scale proof-of-concept prototype is a distributed memory, partitionable, mixed-mode machine, with 16 (MC68000-based) PEs in the computational engine [27, 32]. In SIMD mode, it is assumed that a control unit (CU) broadcasts instructions and common data to the PEs. In MIMD mode, the PEs independently execute the programs located within their local memories. The inter-PE communication is performed via a circuit-switched “extra stage cube” multistage interconnection network [29]. The algorithms for PASM were coded using a combination of a C language compiler, AWK scripts (for pre- and post-processing), and library routines for data conditionals, inter-PE data transfers, and data transfers between the CU and PEs. The absolute execution times from the small-scale PASM prototype are very slow compared to current workstations; however, for this research comparative times among different PASM implementations are the focus.

The SIMD CU in PASM includes a fetch unit (*FU*). In SIMD mode, the CU CPU initiates the parallel computation by instructing the FU to send blocks of SIMD code from the FU memory (which contains the SIMD code) to the FU queue. Once in the FU queue, each SIMD instruction is broadcast to all PEs. While the FU is enqueueing and broadcasting SIMD instructions to the PEs and the PEs are executing instructions, the CU CPU can be performing its own computations – this property is called *CU/PE overlap* [14].

Switching between SIMD mode and MIMD mode on PASM is handled by dividing the PEs’ logical address space into an MIMD address space, where the PEs access their own local memory, and an SIMD address space, where the PE memory requests are satisfied by the FU broadcasting the SIMD instructions. Switching from SIMD to MIMD is implemented by broadcasting to the PEs a branch instruction to MIMD space, while switching from MIMD to SIMD is implemented by all PEs independently branching from MIMD to SIMD space. Changing execution modes changes the source of instructions to execute; all stored information (memory, registers, processor state, etc.) is unaffected.

The algorithms were run on 16 PEs but the results can be directly extrapolated for large-scale systems. The time to execute Phase II in SIMD mode would be exactly the same for larger system as long as the subimages are  $R_S$ -by- $C_S$  (which is a function of  $R$ ,  $C$ , and  $P$ ); in MIMD mode, the results would be approximately the same, because of the added synchronization overhead when more PEs are used. The execution times of Phases I and III are generally negligible,  $O(\log P)$ , compared to Phase II. Therefore, the results are applicable to large-scale systems.

## 5.2. Comparison of Algorithms

Figure 6(a) shows template size versus total *execution time* (computation time plus communication time) for the SIMD and MIMD mode complete sums and dynamic complete sums mappings. Figure 6(b) compares the communication times of the two algorithms (for each mode).

The complete sums algorithm requires additional nested loops to cycle through Phase II’s inter-PE transfers. The loop overhead incurred by these additional nested loops is  $O(R_S \cdot c + C_S \cdot r + r \cdot c)$ . This extra loop overhead in the complete sums algorithm is hidden in SIMD mode due to CU/PE overlap, but is apparent in MIMD mode. This is the reason why in MIMD mode the communication time for the dynamic complete sums is less than the communication time for the complete sums algorithm, while in SIMD mode the communication times are equal. For MIMD mode, the communication time difference between the complete sums and dynamic complete sums algorithms does not impact the total execution times of the two algorithms because the communication time is negligible.

Figure 7(a) compares the performance of the SIMD and MIMD mode implementations of the partial sums and dynamic complete sums algorithms. Figure 7(b) compares the communication time of these two algorithms (for each mode).

The total number of data items (partial sums) transferred by the partial sums algorithm is three times the number of data items (pixels) transferred by the dynamic

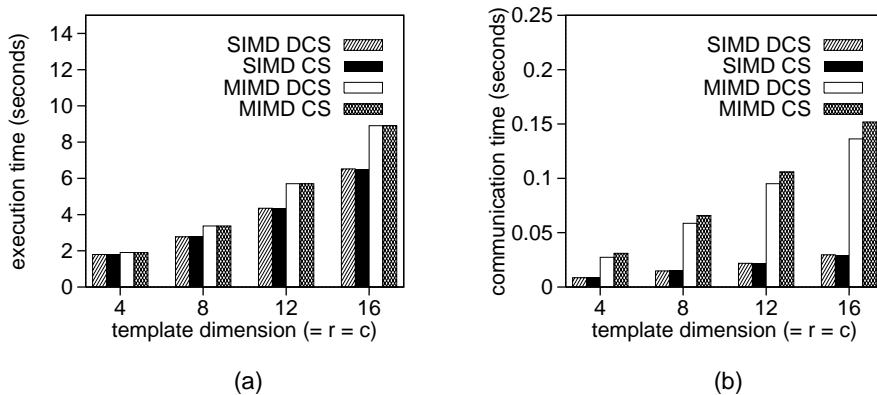


Figure 6. Varying template size on PASM with an 80-by-80 image for dynamic complete sums (DCS) and complete sums (CS) SIMD and MIMD mode mappings, where (a) compares execution time and (b) compares communication time.

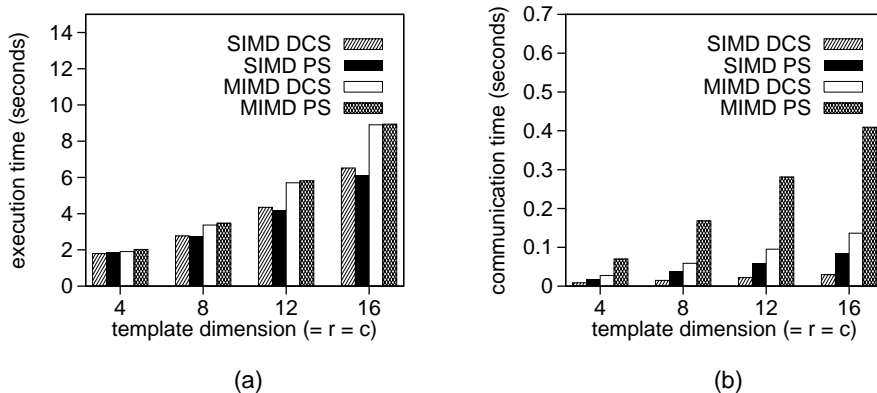


Figure 7. Varying template size on PASM with an 80-by-80 image for dynamic complete sums (DCS) and partial sums (PS) SIMD and MIMD mode mappings, where (a) compares execution time and (b) compares communication time.

complete sums algorithm. However, this increase in communication is offset by a decrease in arithmetic operations, as described in Subsection 4.2.2. In PASM, once a network path is established, it takes less time to transfer a data item between PEs than the time taken for an addition or multiplication operation. Thus, as the template size increases, the number of inter-PE transfers and arithmetic operations increases, which increases the performance difference between the two algorithms. Therefore, in SIMD mode, the partial sums algorithm performs better than the dynamic complete sums algorithm for larger template sizes. In MIMD mode, the cost of establishing the network path and transferring data items between PEs is more expensive than it is in SIMD mode, due to the synchronization overhead. However,

as with SIMD mode, once a path is established, it takes less time to transfer a data item between PEs than the time taken for an addition or multiplication operation. Thus, in MIMD mode the execution time difference (which is very small) reduces with increasing template sizes.

The results presented in Figure 7 show that there is a difference in the comparative performance of the various algorithms because of the mode of parallelism used; i.e., the crossover point in the performance of the dynamic complete and partial sums algorithms differ in the SIMD and MIMD mode of parallelism. Thus, the scalability of an algorithm is affected by the mode of parallelism employed.

The SIMD/MIMD trade-off for communication synchronization does not have a significant impact on the total execution time of the image correlation algorithms considered in this work for the machine size and data sizes used. Despite a small impact in total execution time, communication overhead can be seen to be significantly greater in MIMD mode than in SIMD mode. Figure 7(b) shows the communication time for the partial sums algorithm in the SIMD and MIMD modes. As can be seen from the graph, the overhead associated with the MIMD transfers is approximately five times greater than that associated with SIMD transfers for the template sizes shown. This extra overhead is associated with the software protocols and synchronization required for the PEs to communicate in MIMD mode.

Figure 8 shows the performance of the SIMD mode and mixed-mode implementations for the complete sums, dynamic complete sums, and partial sums algorithms. For the mixed-mode implementation of the dynamic complete sums and partial sums algorithms, Phase I and Phase III are performed in SIMD mode to avoid synchronization overhead during inter-PE communication and Phase II uses both SIMD mode and MIMD mode: MIMD mode for conditionals and SIMD mode for everything else. In the complete sums algorithm, the mixed-mode mapping performed Phase I, Phase II, and the transfers in Phase II in SIMD mode, but the rest of Phase II in MIMD mode. This mapping was chosen to measure the impact of CU/PE overlap in Phase II.

From Figure 8(a), it can be observed that the execution time of the mixed-mode dynamic complete sums is 4% better than the execution time of the SIMD dynamic complete sums algorithm. This performance difference is due to the conditional masking scheme used in PASM to perform data conditional statements in SIMD mode.

As can be seen from Figure 8(b), the SIMD complete sums algorithm outperforms the mixed-mode complete sums algorithm by over 14%. Recall that the mixed-mode complete sums mapping executes the computations in Phase II in MIMD. Hence, the 14% performance difference between the SIMD and mixed-mode mappings is due to the amount of CU/PE overlap. This SIMD mode performance advantage is gained by having the CU execute the loop index increment and compare operations as well as some array addressing computation, while the PEs execute PE data-dependent operations and other array addressing computations. A quantitative analysis of CU/PE overlap is given in the next subsection.

From the analysis presented in this subsection, the SIMD algorithms perform better than the MIMD algorithms. For the dynamic complete sums and partial sums

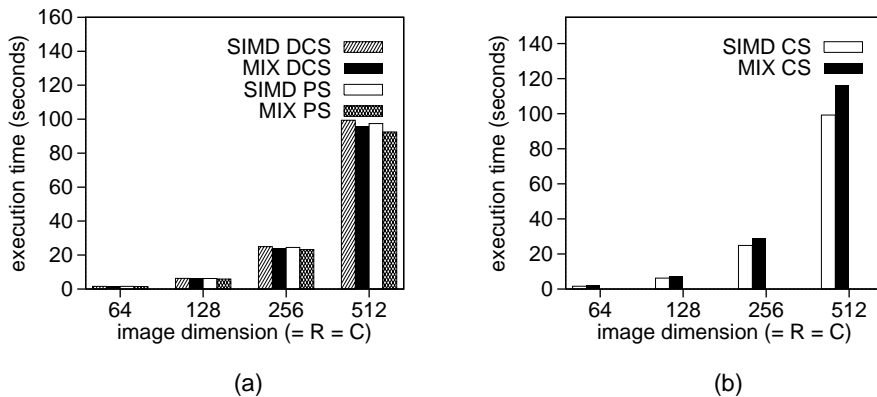


Figure 8. Varying image size on PASM with an 7-by-7 template for the SIMD mode and mixed-mode mappings, where (a) compares the execution times for the dynamic complete sums (DCS) and partial sums (PS) algorithms (b) compares the execution time for the complete sums (CS) algorithm.

algorithms, for  $r = c = 16$  and  $R_S = C_S = 20$ , SIMD implementations perform more than 26% better (Figure 7(a)). In addition, the dynamic complete and partial sums mixed-mode algorithms are representative of a SIMD mode implementation of the respective algorithms with an improved data-conditional masking scheme, which would further improve SIMD performance. Only three data-conditional statements are necessary (comparing the  $\rho'$  value with the current maximum  $\rho'$  value, checking if  $S_{yy}$  equals zero, and checking if PEs at the “edge” of the logical grid should process the match position), thus the advantage for MIMD mode owing to these conditionals are nominal. SIMD mode has the significant advantage of CU/PE computational overlap (approximately 14%) and more efficient inter-PE communication.

The mixed-mode partial sums algorithm of all the implementations tested had the best execution time. Recall that the mixed-mode partial sums algorithm executes the entire algorithm except the data conditionals in SIMD mode. The mixed-mode partial sums algorithm performs best due to two reasons. One is the low communication/computation ratio of PASM, making it beneficial to perform additional communications instead of additional computations. The other is that the mixed-mode partial sums algorithm benefits from the SIMD CU/PE overlap, SIMD communications, and MIMD data-conditional constructs.

### 5.3. An Empirical Examination of the Effects of CU/PE Overlap

When CU/PE overlap occurs, the total execution time for a program is measured from the start of execution to the time when both the PEs and the CU have completed their execution. There may be an unequal amount of work on the CU and PEs, causing one to become idle. This subsection uses simplified representative

code segments based on the kernel of the image correlation algorithm to illustrate how the CU/PE overlap can be quantified and optimized.

Consider the computation of  $\sum \sum t[i, j]y[i, j]$ , for  $0 \leq i < r$  and  $0 \leq j < c$ . Three possible code segment versions, for computing this two dimensional summation, with different workload distribution between the CU and PEs, are shown in Figures 9 through 11. For simplicity, it is assumed that the current match position is  $(0, 0)$ . The template and initial subimage arrays are represented as one dimensional arrays  $\mathbf{t}[\ ]$  (of length  $r \cdot c$ ) and  $\mathbf{I}_s[\ ]$  (of length  $R_S \cdot C_S$ ), respectively, and the result is stored in the scalar variable  $\mathbf{tysum}$ .

The following analysis assumes that the instruction queue is empty when the CU starts execution and that it is large enough to never saturate during execution. The numbers along the left and right sides of each figure provide the approximate statement execution times for the CU and PEs, respectively. All times in this section are in microseconds, and are derived empirically from the PASM prototype. The comment for the `for` statement gives the time to initialize ( $4\mu s$ ), to test for end-of-loop condition ( $8\mu s$ ), and to increment ( $6\mu s$ ) the loop control variable. For each iteration after the first, the test and increment are performed, which takes  $14\mu s$ . The numerical results presented here use  $r = c = 7$ .

The total execution time of a processor consists of two components: the active computation time and the idle time. To evaluate the time taken by the for-loops in the code segments of Figures 9 through 11, one needs to consider the CU active computation time ( $\tau_{CU}$ ), PE active computation time ( $\tau_{PE}$ ), amount of time the PEs continue to execute after the CU has finished execution ( $\tau_{end}$ ), and PE idle time at the beginning of the iteration ( $\tau_{start}$ ).

For the code segment in Figure 9,  $\tau_{PE} > \tau_{CU}$  and the PEs are active except at the beginning of the first iteration, where the PEs wait for the CU to enqueue the SIMD instructions. Therefore, the overall execution time is given by  $\tau_{PE} + \tau_{start}$ . Specifically,  $\tau_{CU} = (14 + 17)r + (4 + 8) + ((14 + 17 + 6)c + (4 + 8))r$  and  $\tau_{PE} = 8r + (90 + 8)rc$ . If  $r = c = 7$ , then  $\tau_{CU} = 2126$  and  $\tau_{PE} = 4858$ .

<u>CU</u>	<u>PE</u>
14 <code>for (i=0; i&lt;r; i=i+1) { /* 4, 8, 6 */</code>	
17 <code>  broadcast i; /* send i from CU to PEs */</code>	8
14 <code>  for (j=0; j&lt;c; j=j+1) { /* 4, 8, 6 */</code>	
17 <code>    broadcast j; /* send j from CU to PEs */</code>	8
6 <code>    simdbegin /* broadcast SIMD block */</code>	
<code>      tysum = tysum + t[c*i+j] * I_s[C_s * i + j];</code>	90
<code>    simdend</code>	
<code>  }</code>	
<code>}</code>	

Figure 9. SIMD pseudo code segment that overworks the PEs.

Except for  $8\mu s$  spent receiving variable  $i$  from the CU, the PEs remain idle until the CU has finished enqueueing the instructions for the **broadcast j** statement during the first iteration, so  $\tau_{start} = 4 + 8 + 17 - 8 + 4 + 8 + 17 = 50\mu s$ .

For the code segment in Figure 10,  $\tau_{CU} > \tau_{PE}$  and the CU is active except for the  $\tau_{end}$  time period during the last iteration of the **for**-loop. Hence, the overall execution time is  $\tau_{CU} + \tau_{end}$ . Specifically, assuming that execution time is measured beginning upon entry of the outer loop (i.e., not including the pointer initialization),  $\tau_{CU} = (74c + 90)r + 12$  and  $\tau_{PE} = 50cr$ . If  $r = c = 7$ , then  $\tau_{CU} = 4268$  and  $\tau_{PE} = 2450$ .

<u>CU</u>	<u>PE</u>
-    tbase = t[]; /* initialize pointers */	
-    Ibase = I <sub>s</sub> [];	
14   for (i=0; i<r; i=i+1) { /* 4, 8, 6 */	
32       tptr = tbase + c*i; /* increment row ptrs */	
32       Ip <sub>tr</sub> = Ibase + C <sub>s</sub> *i;	
14       for (j=0; j<c; j=j+1) { /* 4, 8, 6 */	
10           tptr = tptr + 1; /* increment column ptrs */	
10           Ip <sub>tr</sub> = tptr + 1;	
17           broadcast tptr; /* send ptrs from */	8
17           broadcast Ip <sub>tr</sub> ; /* CU to PEs */	8
6           simdbegin /* broadcast SIMD block */	
tysum = tysum + t[tptr]*I <sub>s</sub> [Ip <sub>tr</sub> ];	34
simdend	
}	
}	

Figure 10. SIMD pseudo code segment that overworks the CU.

Consider the final iteration of the code segment given in Figure 10. The final PE activity begins when the CU has finished placing the **broadcast j** PE instructions on the instruction queue. After this point, the PEs require  $8\mu s$  to read and execute the **broadcast i** instructions (to receive  $i$  from the CU). Concurrently, the CU needs  $6\mu s$  to place the next block of SIMD instructions into the instruction queue. Once the PEs have completed the **broadcast j** instructions, they execute the next block of SIMD instructions. Hence, they remain active for another  $34\mu s$ . Meanwhile, once the CU finishes placing the next block of SIMD instructions in the instruction queue, it increments the inner loop control variable,  $j$ , and tests true for the end-of-loop test (this is the final iteration); likewise for the outer loop. The CU is then finished with the code segment. Thus, for Figure 10,  $\tau_{end} = 8 - 6 + 34 - (6 + 8 + 6 + 8) = 8$ .

From the above results the execution time of the code segment in Figure 9 is given by  $\tau_{PE} + \tau_{start} = 4858 + 50 = 4908$  and the execution time of the code segment in



Phase I, Phase II, and the transfers in Phase II in SIMD mode, but the rest of Phase II in MIMD mode). The SIMD implementation was faster, but there was less than a 28% difference. The discrepancy is due to factors such as the conditionals being performed better in MIMD mode than in SIMD mode in parts of the code outside that presented in Figure 11.

This subsection showed that the effects of CU/PE overlap can be optimized and can have an important impact on execution time, especially when array-intensive and loop-intensive computations are involved. To obtain the best SIMD implementation for the results in this paper, the programmer had to hand optimize the CU/PE overlap in many different portions of the code. This is a tedious process but necessary for peak performance. One objective for a SIMD compiler is to achieve a balance in the CU/PE computational load [25]. Thus, the potential advantages of CU/PE overlap must be considered when comparing the SIMD, MIMD, and mixed-mode parallelism.

## 6. MasPar MP-1 Implementations

The Purdue MasPar MP-1 is an SIMD machine with 16,384 PEs [3, 4, 21]. Each PE's arithmetic and logic unit (*ALU*) is 4-bits wide. There are two bit-serial networks in the MP-1: the X-Net is a 128-by-128 mesh that connects a PE to each of its eight nearest neighbors, and the multistage router provides a way to connect a PE to any other PE in three router stages (with possible conflicts). Matching the data layout to the X-Net mesh allowed exploitation of locality and quick transfers of the X-Net, opposed to the slower multistage interconnect. The MP-1 also has a 4-bit wide Global-Or bus that is capable of performing recursive doubling across all PEs at a substantial time improvement over a software implementation. The three parallel algorithms presented in Section 4 were implemented in MPL, a C-based language.

The MP-1 X-Fetch command is provided to allow the user to send large blocks of data over the X-Net, but the time spent performing the operation is longer than the time to send each piece in a loop using X-Net. This was verified experimentally by transferring up to 1000 integers using a loop with an X-Net call and then doing the same by using an X-Fetch call. Thus, the loop approach was used.

Because of the inherent synchronization that SIMD machines have, communication among PEs is fast. All enabled PEs send at the same time, and thus receive at the same time. Because the MP-1 has a relatively low communication/computation ratio, computed from experimental results, an optimal implementation could use excess data transfers while trying to minimize computations.

The three algorithms discussed in Section 4 were each implemented on the MP-1 with a 128-by-128 PE grid. In Figures 12 and 6.2(a), a 2048-by-2048 pixel image was used with a template that was varied from 2-by-2 pixels to 16-by-16 pixels, which is the maximum template size for a 2048-by-2048 image on 128-by-128 PEs (given the simplifying restriction  $r < R_S$  and  $c < C_S$ ). Figures 12(a) and 12(b) show the communication time and computation for the three algorithms (in Figure 12(b), dynamic complete sums and complete sums coincide). Figure 13(a) presents

the percentage of the total communication time spent performing recursive doubling across the PEs, which was implemented using the Global-Or bus. This figure only contains information for the partial sums implementation because it had the best total execution time of the three implementations. As can be noted from Figure 12, the communication time is a small portion of the total execution time, thus the algorithm that minimizes the number of computations performed will achieve the best execution time.

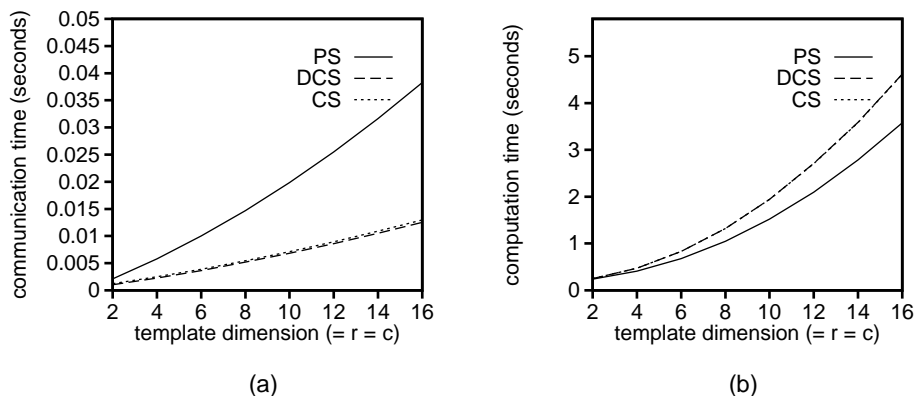


Figure 12. Varying template size on the MasPar MP-1, where (a) compares communication time and (b) compares computation time for the partial sums (PS), dynamic complete sums (DCS), and complete sums (CS) algorithms for a 2048-by-2048 image on 128-by-128 PEs.

The only difference between the dynamic complete sums, and the complete sums approaches is when the pixels are transferred. As shown in Figure 12(a), the complete sums communication time is only slightly larger than that of the dynamic complete sums because of loop overhead effects (even though the loop control itself was executed on the CU). The computation times for these two approaches are the same (Figure 12(b)), as is expected (see Subsection 4.2.2).

As discussed in Subsection 4.2.2, the communication time for the partial sums algorithm should be three times the communication time for the dynamic complete sums algorithm. This is verified in Figure 12(a). It should be noted that had character data been used for pixel values and integer data for the partial sums, the communication time for the partial sums algorithm would be between three and twelve times the communication time for the dynamic complete sums approach. This result comes from the fact that the communication time in the MP-1 is dependent on the size of the data sent, and the fact that an integer is two or four bytes, while a character is only one. To understand why integer data needs to be sent for the partial sums implementation, consider as an example the summation of the square of the 8-bit pixel values. Using an 8-bit character value for the pixel values, a 16-bit squared pixel value is obtained, which is then summed. Summing enough 16-bit values will overflow to a 17-bit number. A 32-bit integer was then chosen as the value to transfer to represent this sum of squared pixel values. To

compare communication time fairly across all algorithms, 32-bit integer pixel values were transferred in the dynamic complete sums and the complete sums methods.

The computation time for the dynamic complete sums algorithm is larger than that of the partial sums algorithm (see Figure 12(b)) because it performs more computations (see Subsection 4.2.2). Because the communication time is such a small portion of the overall execution time, the partial sums algorithm has the lowest total execution time.

Figure 13(a) shows that the percentage of time spent using the Global-Or bus decreases as the template size increases. The Global-Or bus always collects one value from each of the 16,384 PEs, so the time to collect the results from all PEs is constant for all experiments. However, as the template size increases, number of pixels or partial sums transferred increases. Thus, the percentage of communication time spent using the Global Or bus decreases.

Figures 13(b) and 14 show how varying the image size affects the three algorithms. The template size was fixed at 6-by-6 pixels, and the subimage was varied from 6-by-6 pixels to 16-by-16 pixels. This represents an image size of 768-by-768 pixels to an image of 2048-by-2048 pixels. Figures 14(a) and 14(b) presents the communication time and computation time for the three algorithms, respectively. Figure 13(b) shows the percent of the total communication time spent collecting results from all PEs using the algorithm that had the lowest execution time, which was the partial sums implementation. The explanations for these results are similar to the explanations for Figures 12 and 13(a).

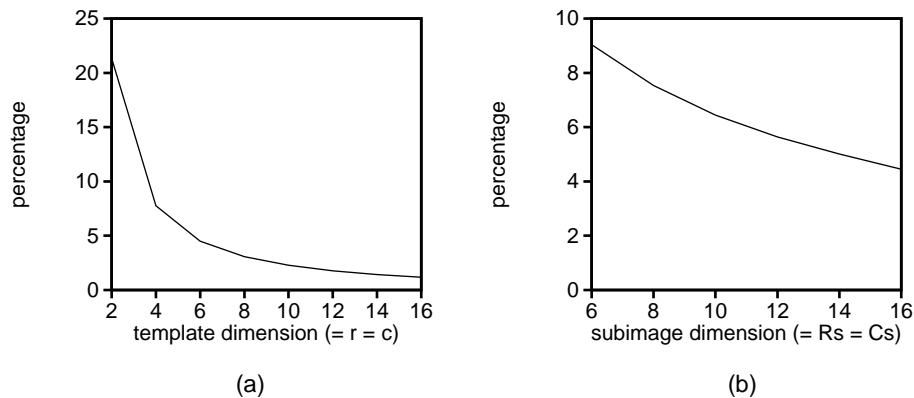


Figure 13. Percent of total communication time spent performing recursive doubling across the PEs of the MasPar MP-1 using the partial sums algorithm for (a) 2048-by-2048 image on 128-by-128 PEs and (b) 6-by-6 template on 128-by-128 PEs.

It was noted earlier that the MP-1 has a low communication/computation ratio, and as such the optimal algorithm for the MP-1 could perform additional inter-processor transfers while minimizing computations. Examining Figure 12(a), the partial sums algorithm has the largest communication time versus the dynamic complete sums and the complete sums algorithms. From Figure 12(b), the partial

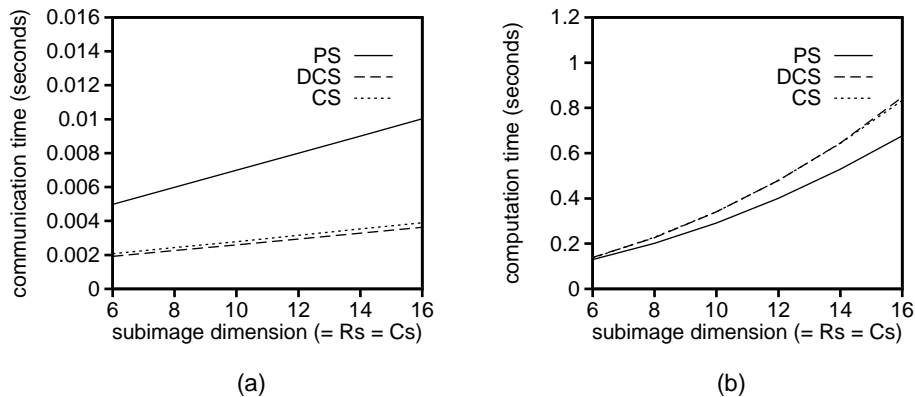


Figure 14. Varying image size on the MasPar MP-1, where (a) compares communication time and (b) compares computation time for the partial sums (PS), dynamic complete sums (DCS), and complete sums (CS) algorithms for a 6-by-6 template on 128-by-128 PEs.

sums algorithm has the lowest computation time. Because of the MP-1's low communication/computation ratio, the lowest computation time is the crucial part of the best implementation. These results, which were for varying the template size, are similar for varying the image size, as shown in Figure 14. Some of the features that lend themselves to making the dynamic complete sums or the complete sums an optimal algorithm are not available in the MP-1. For example, if block transfers were faster than sending individual pieces of data, or the MP-1 had a high communication/computation ratio, then the complete sums approach might perform better than the partial sums approach.

## 7. The Intel Paragon and nCUBE Implementations

### 7.1. Intel Paragon XP/S 10 Implementations

The Purdue Intel Paragon XP/S 10 [1, 10] is a distributed memory MIMD machine with 140 nodes in the computing partition. Each node has two Intel i860XP processors: one is dedicated to handling message-passing operations and is called the *message coprocessor*; the other is called the application processor (or PE in this paper). The message coprocessor, which can access the PE memory, handles message protocol processing for the application program and implements global operations such as synchronization, broadcasting, and global reduction operations (e.g., global sums). The nodes are interconnected via a wormhole routed two-dimensional rectangular mesh with each node connected only to its four nearest neighbors.

The C-language and the “nx” message-passing library were used to implement the three image correlation algorithms. The experiments were conducted in an 8-by-8 subpartition of nodes. On the Paragon, the communication/computation ratio is high mainly due to the overhead involved in establishing an inter-node data

transfer. Once the path is established, high data rates can be sustained for data transfers between the source and destination PEs. Because the interconnection network on the Paragon is a four nearest neighbor mesh, all inter-PE data transfers required by template matching involve only adjacent PEs.

In the complete sums algorithm, all data required by a PE is acquired before commencing any computations on the pixel data. The complete sums algorithm can be implemented in two different ways. In the first implementation, the pixels to be sent to another PE are packed into a contiguous memory area (*message blocking*) and a single “send” call is used to send this data to the destination PE. The second implementation is obtained by transferring each pixel in a single “send” call. In the dynamic complete sums algorithm, the inter-PE data transfers are performed on demand, i.e., the pixels are transferred when they are needed for computation. Thus, the “send” and “receive” routines are called within a nested loop. This increases the software and hardware overhead due to the inter-PE transfers compared to the implementation of the complete sums algorithm with message blocking. In partial sums algorithm, three partial sums are transferred from the source PE to the destination PE for each match position. This produces more inter-PE data transfer overhead compared to dynamic complete sums and complete sums algorithm.

Figure 15(a) shows the total execution times of the three methods for different template sizes with an image size of 320-by-320 on an 8-by-8 array of PEs. It can be observed that complete sums algorithm (with message blocking) performs the best. Due to message blocking, the complete sums algorithm minimizes the inter-PE transfer overhead. The partial sums algorithm has the worst total execution time, as expected due to its higher inter-PE transfer overhead.

In Figure 15(b), timings for different versions of the complete sums algorithm are compared. On the Paragon, due to the availability of a message coprocessor to handle the inter-PE communication requests, a “modified” complete sums algorithm was also implemented. In this algorithm, the communication phase is overlapped with the local computation phase. The asynchronous receive routine provided in the “nx” library is used to post a receive at the beginning of the program, so that the received data is directly copied onto the program variables from the network buffers. Then the data is sent to the destination PEs with message blocking. This is followed by a phase that performs computation involving only local pixel values. The buffers are then checked for the non-local pixel values. During the local computation phase the message coprocessor is active and is transferring the data through the network. Once the pixel data from adjacent PEs are in the message buffers of the local PE, the computation involving the non-local pixel values is initiated. The match positions for which matching can be performed using only the local pixel values decreases with increasing template size for a given image size. Even though a larger template requires more calculation, this decrease in match positions decreases the total time consumed by the local computation phase. Hence the amount of performance gain obtained by overlapping local computation with communication decreases with increasing template size. This is evident from the experimental results presented in Figure 15(b).

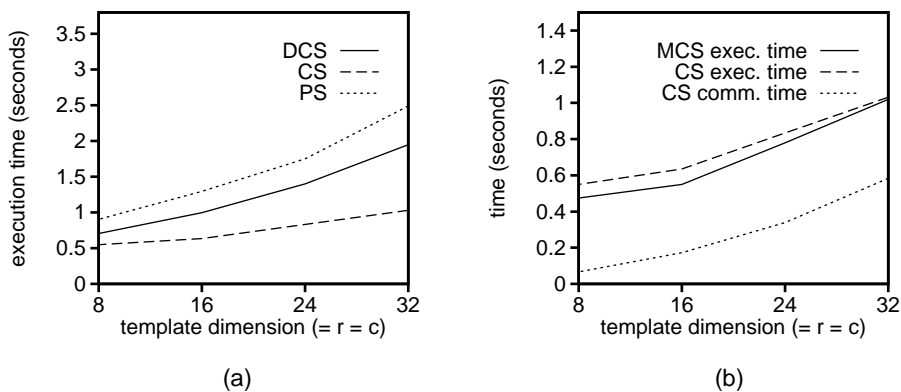


Figure 15. Varying the template size on 64 PEs of the Intel Paragon for a 320-by-320 image, where (a) compares total execution times for the dynamic complete sums (DCS), complete sums (CS), and partial sums (PS) algorithms and (b) compares the timings for the modified complete sums (MCS) and CS algorithms.

## 7.2. nCUBE 2 Implementations

The Purdue nCUBE 2 is a distributed memory MIMD machine with 64 PEs. The CPU in each PE is a 64-bit RISC-based processor. The PEs are interconnected via a wormhole-routed six-dimensional binary hypercube [9]. For all algorithm implementations, it is assumed that the PEs are arranged in a 8-by-8 logical grid. In the *embedded* implementations, the PEs were logically renumbered using an inverted gray-code sequence to map a four-nearest neighbor mesh onto the hypercube. This ensures that all communications other than those generated by the global summing operations are between PEs physically adjacent to each other. In the *direct* implementations, no logical renumbering is used. The C language augmented with a message-passing library was used to program the nCUBE 2.

On the nCUBE 2, the communication/computation ratio is high compared to an SIMD machine such as the MasPar MP-1. The communication time is dominated by the overhead involved in setting up the path. Once the path is established, high data rates can be sustained for data transfers between the source and destination PEs.

Figure 16(a) shows the relative execution times of the different direct implementations. The complete sums algorithm with message blocking has the best execution time, due to the minimal inter-PE data transfer overhead. A similar comparison is shown in Figure 16(b) for the different embedded implementations. The performance of the three algorithms remain almost the same with or without embedding the logical mesh.

From Figure 16(a) it can be noted that using message blocks improves the performance of the direct implementation of the complete sums algorithm, and the performance benefit increases with increasing template size. This is because the

volume of data transferred among the PEs increases with increasing template size. Therefore, the impact of using message blocks is more prominent as the template size increases.

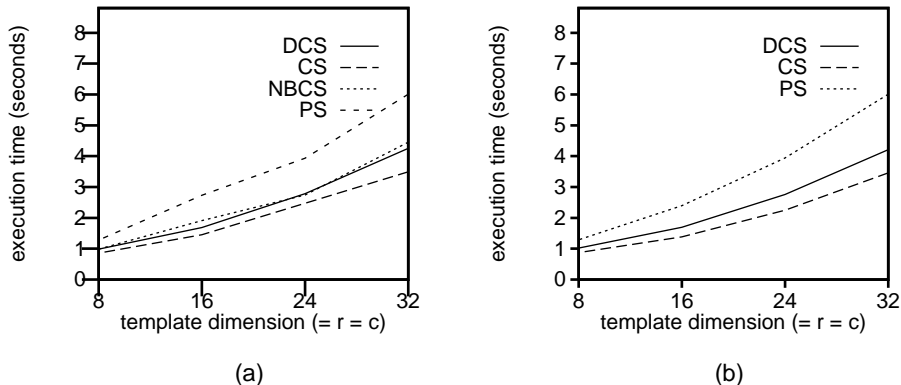


Figure 16. Varying the template size on 64 PEs of the nCUBE 2 for a 320-by-320 image, where (a) compares total execution times for the dynamic complete sums (DCS), complete sums (CS), non-blocking complete sums (NBCS), and partial sums (PS) algorithms with direct implementations and (b) compares the total execution times for the DCS, CS, and PS algorithms with embedded implementations.

The partial sums results verify that the trade-off of reducing redundant computation by increasing redundant communication in the partial sums algorithm increases the total execution time in a machine with a high communication/computation ratio like the nCUBE 2. As seen from results presented in Section 6, this trade-off is beneficial in a machine with a low communication/computation ratio like the MasPar MP-1.

Figure 17 shows various timings for the complete sums algorithm. It shows that for the nCUBE 2 communication times for the complete sums algorithm is not negligible for both the direct and embedded implementations. Figure 17(b) shows the advantage of message blocking for the embedded implementation, as was discussed previously for the direct implementation.

In Figure 18(a), the variation of the total execution times of direct implementations of the three algorithms with the image size is shown. Figure 18(b) presents the variation of different timings for direct implementations of the complete sums algorithms with the image size. These results indicate that the complete sums algorithm with message blocking is the best approach on the nCUBE 2.

In an MIMD machine such as nCUBE 2, it is very difficult to time individual code segments (e.g., to find the time for a segment of code that performs the inter-PE data transfers), due to the lack of a synchronizing primitive (e.g., barrier [5]). To determine the communication time, the program is executed without the “send” and “receive” calls and the execution time is noted. This time is subtracted from the total execution time to obtain the communication time. This procedure is valid

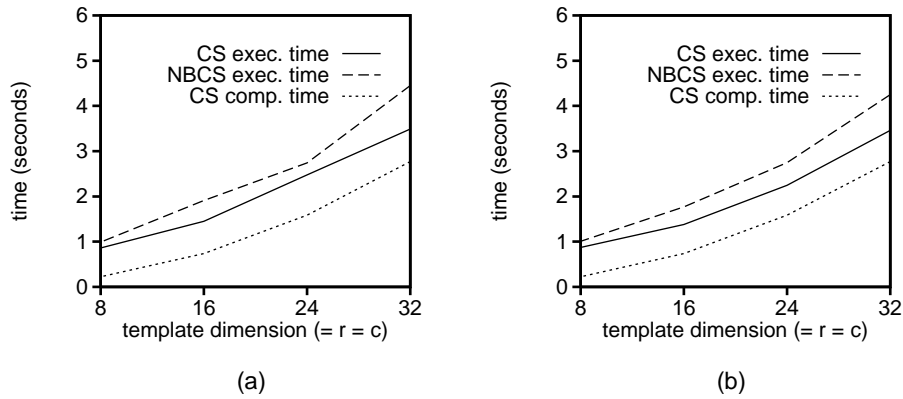


Figure 17. Varying the template size on 64 PEs of the nCUBE 2 for a 320-by-320 image, where (a) compares different timings for the complete sums (CS) and non-blocking complete sums (NBCS) algorithms with direct implementations and (b) compares different timings for the CS and NBCS algorithms with embedded implementations.

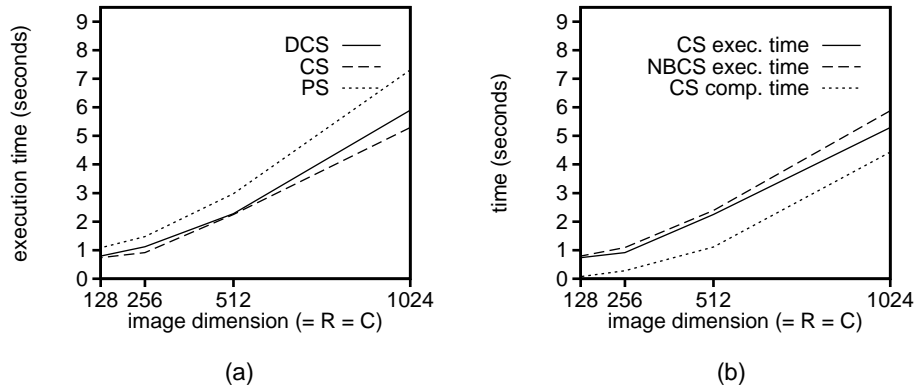


Figure 18. Varying the image size on 64 PEs of the nCUBE 2 for a 12-by-12 template using direct implementations, where (a) compares execution times for the dynamic complete sums (DCS), complete sums (CS), and partial sums (PS) algorithms and (b) compares the different timings for the CS and non-blocking complete sums (NBCS) algorithms.

because the execution time of the three image correlation algorithms do not depend on the actual pixel values, but instead depend upon the image and template sizes.

### 7.3. Summary

From the implementations on the nCUBE 2 and Intel Paragon, it can be observed that algorithms with message blocking perform well on both machines. This is primarily due to the high communication/computation ratio. Also, when there is

a trade-off between redundant communication and redundant computation, it is better to choose the algorithms that employ redundant computation on these machines. In a machine with a low communication/computation, the situation would be different, e.g., the MasPar MP-1. Another factor that improved the algorithm performance on the Intel Paragon is the communication and computation overlap. Specifically, the modified complete sums algorithm performed better than the other algorithms due to overlapping the communication with the local computation.

## 8. Conclusions

The experiments in this study have shown that different algorithms perform best on the PASM prototype, the MasPar MP-1, the nCUBE, and the Intel Paragon. The analyses of the results presented explain how these performance differences are due to the match of algorithm characteristics to the system features of each machine. Algorithm characteristics considered included the use of additional communication versus redundant computation, data block transfers, and communication/computation overlap. Machine features examined included mode of parallelism, communication/computation ratio, cost of setting up a network transfer, and overlapping communication and computation. When preparing to design an algorithm, the programmer should keep in mind the characteristics of possible alternative algorithms and the machine features that will effect the execution time. Only then can the programmer select the most appropriate machine and the best implementation for that machine.

Understanding the characteristics of an application's computational structure and the attributes of a machine's architecture are, of course, important for effectively mapping an application task onto a parallel machine. Case studies, such as the one presented here and ones for other types of applications (e.g., [34]), are a necessary step in developing techniques for using these characteristics and attributes in software tools that will facilitate the mapping process [12]. Furthermore, understanding and quantifying the interaction of important application compute characteristics [13] with important machine attributes is a key enabling technology needed for facilitating software tools for mixed-machine heterogeneous computing, where a task is decomposed into subtasks and executed across a suite of different parallel machines [28]. Thus, case studies such as this one will aid the development of future methods for facilitating mapping a task onto a single parallel machine or mapping subtasks or a set of independent tasks onto a heterogeneous suite of parallel machines.

## Acknowledgments

The authors thank Prof. Leah H. Jamieson for her comments and ideas during many useful discussions. Preliminary versions of portions of this material were presented at the 3rd IEEE Symposium on Parallel and Distributed Processing and the 23rd International Conference on Parallel Processing. The authors also thank the referees for their suggestions.

## Notes

1. In [31] dynamic complete sums is referred to as complete sums.

## References

1. G. S. Almasi and A. Gottlieb. *Highly Parallel Computing*, 2nd ed. Benjamin/Cummings, Redwood City, CA, 1994.
2. H. R. Arabnia and S. M. Bhandarkar, Parallel stereocorrelation on a reconfigurable multi-ring network. *The Journal of Supercomputing*, 10:243-269, 1996.
3. T. Blank. The MasPar MP-1 architecture. *IEEE Compccon*, pp. 20-24. IEEE Computer Society Press. 1990.
4. P. Christy. Software to support massively parallel computing on the MasPar MP-1. *IEEE Compccon 1990*, pp. 29-33. IEEE Computer Society Press. 1990.
5. H. Dietz, T. Schwederski, M. O'Keefe, and A. Zaafrani. Static synchronization beyond VLIW. *Supercomputing '89*, pp. 416-425. IEEE Computer Society Press. 1989.
6. P. Duclos, F. Boeri, M. Auguin, and G. Giraudon. Image processing on a SIMD/SPMD architecture: OPSILA. *9th Int'l Conf. Pattern Recognition*, pp. 430-433. 1988.
7. Z. Fang, X. Li, and L. M. Ni. Parallel algorithms for image template matching on hypercube SIMD computers. *IEEE Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, pp. 33-40. IEEE Computer Society Press. 1985.
8. Z. Fang, X. Li, and L. M. Ni. Parallel algorithms for 2-D convolutions. *1986 Int'l Conf. Parallel Processing*, pp. 262-269. IEEE Computer Society Press. 1986.
9. J. P. Hayes and T. N. Mudge. Hypercube supercomputers. *Proceedings of the IEEE*, 77:1829-1841, 1989.
10. Intel Corp. Supercomputing Systems Division. Intel Paragon User's Guide. Beaverton, OR, 1993.
11. Integrated Computing Engines, Inc. The MeshSP Technical Report. ICE, Inc., 1995.
12. L. H. Jamieson. Characterizing parallel algorithms. In L. H. Jamieson, D. G. Gannon, and R. J. Douglass, eds., *The Characteristics of Parallel Algorithms*, The MIT Press, Cambridge, Massachusetts, 1987.
13. T. Kidd, D. Hensgen, R. Freund, and L. Moore. SmartNet: A scheduling framework for heterogeneous computing. *2nd Int'l Symp. Parallel Architectures, Algorithms, and Networks*, pp. 514-521. IEEE Computer Society Press. 1996.
14. S. Kim, M. Nichols, and H. J. Siegel. Modeling overlapped operation between the control unit and processing elements in an SIMD machine. *J. Parallel and Distributed Computing*, 12:329-342, 1991.
15. P. M. Kogge. EXECUBE - a new architecture for scalable MPPs. *1994 Int'l Conf. Parallel Processing*, pp. 77-84. IEEE Computer Society Press. 1994.
16. P. Kumar and V. Krishnan. Efficient image template matching on SIMD hypercube machines. *1987 Int'l Conf. Parallel Processing*, pp. 765-771. IEEE Computer Society Press. 1987.
17. S. Lee and J. Aggarwal. Parallel 2-D convolution on a mesh-connected array processor. *IEEE Trans. Pattern Analysis and Machine Intelligence*, PAMI-9:590-594, 1987.
18. G. Lipovski and M. Malek. *Parallel Computing: Theory and Comparisons*, John Wiley and Sons, New York, NY, 1987.
19. M. Maresca and H. Li. Morphological operations on mesh-connected architecture: A generalized convolution algorithm. *IEEE Workshop on Computer Vision and Pattern Recognition*, pp. 299-304, IEEE Computer Society Press. 1986.
20. R. H. Myers. *Classical and Modern Regression with Applications*. Duxbury Press, Boston, MA, 1986.
21. J. R. Nickolls. The design of the MasPar MP-1: A cost effective massively parallel computer. *IEEE Compccon 1990*, pp. 25-28. IEEE Computer Society Press. 1990.
22. M. Philippsen, T. Warschko, W. Tichy, and C. Herter. Project Triton: Towards improved programmability of parallel machines. *26th Hawaii Int'l Conf. System Sciences*, pp. 192-201. 1993.

23. S. Ranka and S. Sahni. Image template matching on SIMD hypercube multicomputers. *1988 Int'l Conf. Parallel Processing*, pp. 84-91. IEEE Computer Society Press, 1988.
24. S. Ranka and S. Sahni. Image template matching on MIMD hypercube multicomputers. *J. Parallel and Distributed Computing*, 10:79-84, 1990.
25. G. Saghi and H. J. Siegel. Compiler techniques for increasing CU/PE overlap in SIMD machines. *9th Int'l Parallel Processing Symp.*, pp. 369-375. IEEE Computer Society Press, 1995.
26. H. J. Siegel, J. B. Armstrong, and D. W. Watson. Mapping computer-vision-related tasks onto reconfigurable parallel-processing systems. *IEEE Computer*, 25:54-63, 1992.
27. H. J. Siegel, T. D. Braun, H. G. Dietz, M. B. Kulaczewski, M. Maheswaran, P. H. Pero, J. M. Siegel, J. J. E. So, M. Tan, M. D. Theys, and L. Wang. The PASM project: A study of reconfigurable parallel computing. *2nd Int'l Symp. Parallel Architectures, Algorithms, and Networks*, pp. 529-536. IEEE Computer Society Press, 1996.
28. H. J. Siegel, H. G. Dietz, and J. K. Antonio. Software support for heterogeneous computing. In A. B. Tucker, Jr., ed., *The Computer Science and Engineering Handbook*, pp. 1886-1909. CRC Press, Boca Raton, FL, 1997.
29. H. J. Siegel, W. T. Hsu, and M. Jeng. An introduction to the multistage cube family of interconnection networks. *The Journal of Supercomputing*, 1:13-42, 1987.
30. H. J. Siegel, M. Maheswaran, D. W. Watson, J. K. Antonio, and M. J. Atallah. Mixed-mode system heterogeneous computing. In M. M. Eshaghian, ed., *Heterogeneous Computing*, pp. 19-65. Artech House, Norwood, MA, 1996.
31. L. J. Siegel, H. J. Siegel, and A. Feather. Parallel processing approaches to image correlation. *IEEE Trans. Computers*, C-31:208-218, 1982.
32. 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 A. Y. Zomaya, ed., *Parallel Computing: Paradigms and Applications*, pp. 78-114. International Thomson Computer Press, London, U.K., 1996.
33. H. J. Siegel, L. Wang, J. J. So, and M. Maheswaran. Data parallel algorithms. In A. Y. Zomaya, ed., *Parallel and Distributed Computing Handbook*, pp. 466-499. McGraw-Hill, New York, NY, 1996.
34. 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*, 21:124-139, 1994.

## Contributing Authors

**James B. Armstrong** is Manager of Advanced Product Development for Software at Sarnoff Real Time Corporation. He received a B.S. degree and a certificate in management systems from Princeton University, and the M.S.E., and Ph.D. degrees from Purdue University. He has coauthored over 20 technical papers, and received one best paper award. Currently, his interests are in the area of mapping multi-media applications (e.g., video-on-demand, internet services) onto parallel machines.

**Muthucumaru Maheswaran** is a Ph.D. student and research assistant in the School of Electrical and Computer Engineering at Purdue University. He received a B.Sc. degree in electrical engineering from the University of Peradeniya, Sri Lanka and a M.S.E.E. degree in electrical engineering from Purdue University. He received a Fulbright scholarship to pursue his M.S.E.E. degree at Purdue University. His research interests include heterogeneous computing, parallel iterative algorithms, and computer architecture.

**Mitchell D. Theys** is a Ph.D. student in the School of Electrical and Computer Engineering at Purdue University. He received a B.S.C.E.E. and an M.S.E.E. from Purdue University in 1993 and 1996, respectively. He has received a Benjamin Meisner Fellowship from Purdue University for the 1996-1997 academic year, and an Intel Graduate Fellowship for the 1997-1998 academic year. He has held positions with Compaq Computer Corporation, S&C Electric Company, and Lawrence Livermore National Laboratory.

**Howard Jay Siegel** is a Professor in the School of Electrical and Computer Engineering at Purdue University. He received two B.S. degrees from MIT, and the M.A., M.S.E., and Ph.D. degrees from Princeton University. He has coauthored over 240 technical papers and is a Fellow of the IEEE. He was a Coeditor-in-Chief of the Journal of Parallel and Distributed Computing, and has served on the Editorial Boards of both the IEEE Transactions on Parallel and Distributed Systems and the IEEE Transactions on Computers.

**Mark A. Nichols** received the B.S. degree with triple major of electrical engineering, computer engineering, and mathematics (computer science) from Carnegie-Mellon University, the M.S.E.E. degree from the Georgia Institute of Technology, and the Ph.D. degree in electrical engineering from Purdue University. Having been employed by NCR, San Diego, CA, he is currently a consultant in the same city. His research interests include parallel language/compiler design, parallel architecture modeling, and interconnection networks.

**Kenneth H. Casey** biography not available.