Polyvalent Parallelizations for Hierarchical Block Matching Motion Estimation

Charalampos Konstantopoulos, Andreas Svolos and Christos Kaklamanis

Computer Engineering and Informatics Department, University of Patras and Computer Technology Institute, Athens, Greece

Block matching motion estimation algorithms are widely used in video coding schemes. In this paper, we design an efficient hierarchical block matching motion estimation (HBMME) algorithm on a hypercube multiprocessor. Unlike systolic array designs, this solution is not tied down to specific values of algorithm parameters and thus offers increased flexibility. Moreover, the hypercube network can efficiently handle the non regular data flow of the HBMME algorithm. Our techniques nearly eliminate the occurrence of "difficult" communication patterns, namely many-to-many personalized communication, by replacing them with simple shift operations. These operations have an efficient implementation on most of interconnection networks and thus our techniques can be adapted to other networks as well. With regard to the employed multiprocessor we make no specific assumption about the amount of local memory residing in each processor. Instead, we introduce a free parameter S and assume that each processor has $\Theta(S)$ local memory. By doing so, we handle all the cases of modern multiprocessors, that is fine-grained, medium-grained and coarse-grained multiprocessors and thus our design is quite general.

Keywords: motion estimation, block matching algorithms, multiresolution pyramid, programmable architectures, multiprocessors, interconnection networks, hypercube, mesh.

1. Introduction

Block matching motion estimation algorithms are widely used in video coding schemes [1, 2]. The basic idea is to divide the current frame into equally sized blocks, and then to find for each block the best matching block in an available previous frame. This can be done by full (exhaustive) search within a search window (optimal solution), or by using an intelligent non exhaustive search (suboptimal solution) in order to reduce the computation requirements. Additionally, a multiresolution representation of video frames can be used for achieving higher performance (hierarchical block matching algorithms). Next, the motion vector of each block in the current frame is determined by the relative displacement of the best matched block in the previous frame. As a measure of block similarity, the mean absolute difference between two blocks is typically used because it requires no multiplication and has similar performance to the mean square error.

Due to its high computational demands, video coding is usually implemented in hardware. Hardware architectures can be split into application-specific and programmable [3, 4]. In the first case, special purpose hardware is optimally designed. For example, a large number of architectures have appeared for block matching motion estimation algorithms especially for the full search algorithm [5]. Due to its highly regular data flow, most realizations of this algorithm are based on mesh-like systolic arrays. Despite their high efficiency, these application-specific designs lack flexibility. A change in algorithm parameters or improvements in the coding scheme may lead to a costly hardware redesign. On the other hand, programmable architectures offer higher flexibility at a cost of reduced efficiency [6, 7, 8, 9, 10]. The coding algorithms are developed in software and thus any change can be easily handled. In the literature, many designs have been reported that follow either of the two approaches (for a survey see [3, 4, 5]).

In this paper, following the programmable architecture approach, we study the software-based realization of the HBMME algorithm on a hypercube-based multiprocessor. We also present an efficient full search block matching motion estimation (FSBMME) algorithm which is used as a subroutine in the HBMME algorithm. A basic point in our study is the amount of local memory at each processor. Specifically, we assume that there is $\Theta(S)$ local memory at each processor where S is a free parameter. Most programmable architectures used in video coding are coarse-grained multiprocessors where each processor has enough memory to keep all data it will need throughout algorithm execution. Thus interprocessor communication is almost eliminated and processors can operate independently. In our study, by including the amount of local memory at each processor as a free parameter, we consider all classes of modern multiprocessors, that is fine-grained, medium-grained as well as coarse-grained parallel machines. As will become evident later, the smaller the value of the parameter S is, the harder it is to design an efficient algorithm. This is because interprocessor communication becomes inevitable when processors have limited local memory. In order to keep communication overhead low, we need to carefully arrange the necessary communication operations. In addition, the use of a powerful network such as the hypercube for the implementation of the HBMME algorithm is well justified, because this algorithm has a non regular data flow and its inherent communication is not local. Thus it cannot be easily implemented on systolic arrays. Dedicated hardware designs for the HBMME algorithm[11, 12, 13] require high external memory bandwidth or relieve these requirements by using large on chip memory.

However, we do not simply count on the large communication bandwidth of the hypercube network in order to efficiently execute data transfers required by our algorithms. We further enhance this performance by minimizing the occurrence of “difficult” communication patterns such as many-to many personalized communication. To this end, we devise efficient techniques which allow the utilization of simple communication operations, shifts, in place of complex communication operations most of the time. Since shifts have simple implementation on most of interconnection networks, our algorithms can be easily adapted to other interconnection networks as well, e.g the mesh network. The clear advantage of the hypercube over sparser interconnection networks such as the mesh lies in the faster execution of the difficult communication patterns which inevitably arise due to the inherent irregularity of the HBMME algorithm. Thus, although there exist nearly optimal algorithms which implement this kind of irregular communication on other networks as well, a comparative increase in the execution time should be expected when extending our algorithms to sparser networks.

The rest of this paper is organized as follows. In Sect. 2, we briefly review the block matching motion estimation algorithms. In Sect. 3 we discuss the basic assumptions and communication operations used in our algorithms. After having defined the basic assumptions and communication primitives, in the next two sections we present our parallel algorithms. In Sect. 4, we present the implementation of FSBMME on the hypercube-based multiprocessor whereas in Sect. 5 we describe the parallel algorithm for HBMME on the same multiprocessor. Then, in Sect. 6, we discuss how we can adapt our design to other interconnection networks as well. In the next section, Sect. 7, we present experimental results which confirm the main theoretical results of the paper. Finally, in section Sect. 8, we summarize our work in this paper.

2. Block Matching Algorithms

In the FSBMME algorithm, the current frame $N \times N$ is divided into blocks of size $M \times M$ and each block is compared with all the blocks of size $M \times M$ within a search window of size $(M + 2d) \times (M + 2d)$ in the previous frame (Fig. 1(a)). Here d denotes the maximum displacement in each direction. We also refer to a block $M \times M$ with its top left corner at the pixel (u, l) as block (u, l). For all displacements $(x, y)(x, y = -d, \cdots, d)$, the mean absolute difference (MAD) between the block (u, l) of the current frame X and the block $(u + x, l + y)$ in the search window of the previous frame Y is

given by:

$$MAD_{(u,l)}(x, y) = \frac{1}{M^2} \sum_{i=0}^{M-1} \sum_{j=0}^{M-1} | X(u+i, l+j) - Y(u+x+i, l+y+j) | \quad (1)$$

where $X(u+i, l+j)$, $Y(u+x+i, l+y+j)$ denote the intensity values of the corresponding pixels. Now, the motion vector $v(u, l)$ of the block (u, l) is given by

$$v(u, l) = arg \min_{(x,y)} MAD_{(u,l)}(x, y) \ . \quad (2)$$

The choice of the right block size is important for the FSBMME algorithm. With either too large or too small block size, the algorithm might well yield false motion estimates [11]. The HBMME algorithm solves this kind of problems by using a multiresolution (hierarchical) representation of video frames in the form of a Laplacian pyramid [1, 14, 15]. The basic idea is to start the estimation of motion field from the lowest resolution level. At this level, the block size is relatively large in comparison with the frame size at that resolution level and the estimated motion vectors capture the large-scale movements existing in the scene. Then, these vectors are passed onto the next higher resolution level as an initial estimate. The higher resolution levels refine on the motion vector estimates and thus smaller block size should be used. The lower resolution frames in the pyramid are obtained by a series of low-pass filtering and subsampling operations.

There are a number of variations of the basic algorithm. The first variation is to skip the subsampling between successive levels of the pyramid. Alternatively, we can use only subsampling without low pass filtering. A third possibility is the use of overlapping blocks at each level. In this scheme, the motion vector of each block at one level is initialized as a linear interpolation of the motion vectors of its adjacent blocks at the previous lower resolution level. Finally, we can use either the FSBMME algorithm or a non exhaustive search BMME algorithm for motion estimation at each level.

An example of motion vector estimation by using a 3–level hierarchy is shown in Fig. 1(b). First the motion vector d_3 of the largest block is estimated (the lowest resolution level). Then at the next higher resolution level the vector d_2 is calculated around the point which d_3 points to. In the third level the vector d_1 is estimated using the smallest size block. The final motion vector is the vector sum d of d_1, d_2, d_3.

3. Basic Assumptions and Communication Operations

Having presented the basic points of both FSBMME and HBMME algorithm, we are now ready to start the description of the parallel implementation of these two algorithms. In this section, we will first refer to the basic assumptions we make and then we will give the details of the communication primitives we use for implementing the algorithms.

As has already been mentioned, the parallel implementation of the FSBMME and HBMME algorithm is carried out on a hypercube-based multiprocessor. We assume that this multiprocessor consists of P^2 ($P = 2^p$) processors with each processor having $\Theta(S)$ local memory. In order to keep analysis simple, we assume that $S = s_r \times s_r$ where $s_r = 2^r$. For convenience too, we view the hypercube as a two dimensional $P \times P$ grid and thus when we use terms like row, column, block and all that, we will actually mean the corresponding subhypercube. Initially, the pixel values $X(i, j)$ and $Y(i, j)$ of the current and previous frame respectively are stored in processor $\left(\lfloor \frac{i}{s_r} \rfloor, \lfloor \frac{j}{s_r} \rfloor\right)$ where processor (i, j) is the processor with address $j + iP$.

An important consideration in our study is also whether or not processors can utilize all their communication links simultaneously [16]. When processors communicate with all their neighbors at the same time (all-port assumption), we can make use of the full communication bandwidth provided by the hypercube network. In contrast, when each processor can send to or receive from only one of its neighbors at a time (one-port assumption), we do not fully exploit the communication capabilities of the hypercube network. On the other hand, all-port assumption usually implies increased hardware complexity for the communication interface of hypercube nodes. In regard to our study, it will

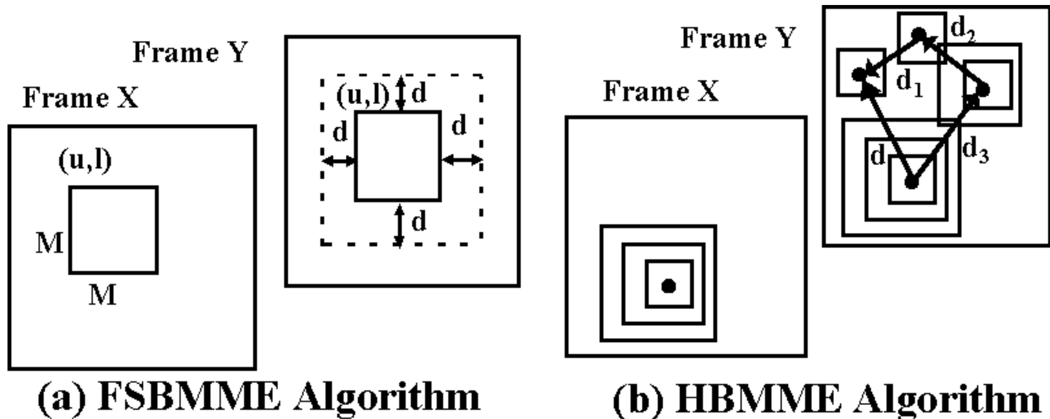

Fig. 1. BMME algorithms.

become clear later that our algorithms can be easily adapted to both these assumptions.

Before proceeding further, we must introduce the basic communication operations used in our algorithms. In deriving the communication complexity of these operations, we assume that sending a message of size M along one hypercube link incurs $\tau + M\beta$ delay where the τ is the fixed start-up cost for sending packets and β is the data transfer rate (bandwidth) of the link. As has been mentioned in the introduction, we will handle all the classes of modern multiprocessors: fine-grained, medium-grained and coarse-grained multiprocessors. Apart from the amount of local memory at each node, these classes of multiprocessors differ in the methods they use to route packets across their interconnection networks. For a detailed description of various routing methods see [17].

Most of fine/medium-grained multiprocessors employ the simple store-and-forward technique for routing packets. In this technique a packet is first stored in full in one processor, before this processor forwards the packet to the next processor en route. Thus sending a packet of size M on a P-node hypercube takes $O(\log P(\tau + M\beta))$ time at most. Notice that the size M of packets is relatively small in fine-grained machines. The same is also true for the start-up overhead τ and thus we can safely ignore this parameter in our time estimates. However, in the case of coarse-grained multiprocessors the start-up overhead τ is much larger and it should be taken into account. This is primarily due to the fact that most modern coarse-grained multiprocessors are manufactured with commodity multiprocessors which have not been optimized for use in high speed interconnection networks [18]. In addition, the size M of messages is usually large in this kind of machines and thus the use of a store-and-forward routing method requires a large buffer space, $O(M)$, at each processor.

Due to these serious drawbacks of store-and-forward routing method, most modern coarse-grained parallel machines use an alternative routing method: wormhole routing [19]. This kind of routing makes better use of the network bandwidth: each message is split into small basic units (called flits) and then these units are pipelined all along the route from the source to the destination of the message. With wormhole routing, every node needs only one flit buffer space per incident link. Further, the start-up overhead for sending a message is paid once at the source node of the message and not at each node along the route of the message. Another positive aspect of wormhole routing is that, due to pipelining the time of sending a message is almost independent of the distance between the sender and the recipient of the message. Thus under low load condition, the network gives the impression that there is a point-to-point link between each pair of processors. Even if the network is moderately or heavily loaded, the use of virtual channels [20] alongside wormhole routing as well as the use of high speed interconnection links significantly alleviate the problem of congestion in modern coarse-grained multiprocessors. The same effect of the very small variance in the time required for executing an arbitrary routing instance can also be achieved by some randomized algorithms. In this kind of algorithms, input packets are first sent to random intermediate destinations and then to their ultimate destination nodes [21]. Due to

this common property of modern routing methods, most of the algorithms that have been presented in the literature for coarse-grained machines [22, 23, 24] make the simplified assumption that the cost of sending a message between any pair of nodes is independent of the distance between the sender and the receiver of the message. Thus this cost is simply given by the expression $\tau_c + M\beta_c$ where τ_c is some fixed communication overhead and β_c is the mean data transfer rate through the network.

In our algorithms, we handle all methods of routing: store-and-forward and wormhole/randomized routing. However, in coarse-grained machines, processors usually have sufficient memory to keep all data they will need throughout algorithm execution. Thus processors can operate almost independently of each other and the interconnection network remains idle most of the time. In contrast, processors in fine-grained machines frequently exchange messages and the interconnection network is constantly utilized during the algorithm execution. Thus most of the following communication operations assume a store-and-forward routing model. Only the last operation is studied under the wormhole/randomized routing model too since there is a case, not very possible in practice, where processors of coarse-grained machine should frequently send and receive messages.

For more details of the following operations see [25, 26]. The time estimates for the first three operations are obtained under one-port assumption.

- Shift(A,B,i,L,P): $B\,(j)\,[r] \longleftarrow A\,((j+i) \bmod P)\,[r]$ where $j = 0 \cdots P-1$, $r = 0 \cdots L-1$, P is the size of the hypercube, A and B are two L-element arrays stored locally in each processor and the notation $B\,(j)\,[r]$ denotes the rth element of array B stored in processor j. This operation moves the elements of array A of processor $(j+i) \bmod P$ to the first L positions of array B of processor j. This transfer is carried out by visiting the hypercube dimensions one at a time in a descending order. The complexity of the operation is $O\,(L \log P\ \beta)$ in general.
- Data_Sum(A,B,L,P): the sum of the corresponding elements of the L-element array A across all processors is stored into the L-element array B of processor 0. In other words, if $A(i)[j]$ is the jth-element of the array A of processor i then the result of the Data_Sum is: $B(0)[j] = \sum_{i=0}^{P-1} A(i)[j]$ and $j = 0 \cdots L-1$. The operation consists of $\log N$ steps. At the ith step ($i = 0 \cdots \log N - 1$), the processors whose i^{th} bit is 1 send their data to their neighbors along i^{th} dimension. These processors in turn add the incoming data to their own data and the whole process is repeated for the $(i+1)^{th}$ bit. Obviously, the Data_Sum operation can be easily modified to store the final sum not only in processor 0 but in any other processor. In addition, due to the associativity of the addition, it is clear that the algorithm can visit the hypercube dimensions in any order without altering the ultimate sum. As will be seen later, both these facts are exploited to a large extent in our algorithms. The operation complexity is $O\left(L \log P\ \beta + L \log P\ T_{op}\right)$ where T_{op} is the time for performing a single arithmetic operation.
- Broadcast(A,B,L,P): Processor 0 broadcasts the contents of its L-element array A and these are stored in the array B of each processor. The time complexity is also $O\,(L \log P\ \beta)$[1].
- Random Access Read RAR(A,B,L,P): In this operation each processor reads the contents of a L-element array A residing in some other processor and then stores the data into its array B. Note that it is not necessary for all processors that contact a particular processor to read the same L-element array.

  Each RAR operation consists of two phases where the second phase is the reverse of the first. The basic communication pattern in each phase is the many-to-many personalized communication with possibly high variance in the message size. In this kind of communication, each processor of a parallel machine sends distinct messages to

[1] When processors can send and receive messages through all their links at the same time (all-port capability), the communication complexity of the above three communication operations falls to $O\,((L + \log P)\,\beta)$.

only some of the processors of the machine. In addition, these messages do not necessarily have the same size. This communication pattern, which is also known as h-relation, is prevalent in parallel algorithms and its importance for efficient implementation of parallel algorithms has long been recognized [27]. For this reason, numerous proposals for implementation of the h-relation has appeared in the literature [28, 29, 30, 31, 32, 33]. The main objective of these algorithms is to decompose the irregular pattern of an h-relation into a number of more regular communication patterns such as all-to-all personalized communication with messages of nearly uniform size or even simple one-to-one permutation routing where all messages communicated have also nearly the same size. In doing so, some of these algorithms use randomization as a tool for minimizing the number of rounds of regular communication required overall [33]. On the other hand, deterministic algorithms have also been proposed [28, 30, 31, 32]. Most of these algorithms assume that there is a virtual point-to-point link between each pair of processors and hence are well suited for coarse-grained machines with wormhole or randomized routing. In our study, we use the algorithm in [31] for implementation of the RAR operation on coarse-grained machines. In this algorithm, there are two phases of communication. Data are first routed to intermediate destinations and then, during the second phase, they are routed to their final destination. By first sending data to intermediate destinations, the algorithm achieves a more balanced distribution of communication across the network.

For the hypercube network, implementation of the RAR operation has been presented in [25]. This algorithm assumes a store-and-forward routing model and that each processor has only one data element in its local memory. For coarser data distributions we can use the algorithm in [34]. This algorithm solves the problem of routing N packets on a P-processor hypercube such that each processor is the source of at most k_1 packets and the destination of at most k_2 packets. This routing can be accomplished in $O\left(\left(k_1 + k_2 + \log^2 P\right)\beta + \frac{N \log N}{P}\, T_{op}\right)$. The basic assumption of this algorithm is that each processor can send and/or receive along all its links at the same time (all-port capability).

Both algorithms in [25, 34] use similar techniques and employ sorting as a basic step for ordering packets according to their destination addresses. This sorting step also determines the complexity of these two algorithms.

In regard to parallel sorting algorithms, there is a vast amount of literature. These algorithms are divided into two major classes. The first class includes all these algorithms which are not based solely on comparisons in order to sort their inputs. This means that their performance depends on the specific values of input keys. Examples of this kind of sorting algorithms [35, 36, 37] are the sample sort, radix sort, flashsort etc. In contrast, the algorithms of the second class are based on sorting circuits and perform comparisons in a predetermined order in order to sort input elements. Thus these algorithms are oblivious to the values of their input keys and their performance is more predictable. The most known example of oblivious sorting algorithm is the odd-even merge sorting [26] which is one of the oldest, yet widely used parallel sorting algorithm. The complexity of this algorithm for sorting N elements on a N-node hypercube is $O\left(\log^2 N\, \beta + \log^2 N\, T_{op}\right)$. In our study however, it is possible to lower this complexity by exploiting the special structure of BMME algorithms. If the total number P of hypercube nodes is smaller than the number N of input elements, which algorithm gives the best results depends on the value of ratio $\frac{N}{P}$. In [35], a experimental study of various sorting algorithms was carried out on the CM-2 parallel machine which is a SIMD hypercube-based machine. This study showed that when the ratio $\frac{N}{P}$ is relatively small, the best results are given by bitonic sorting [38], a sorting algorithm very similar to the odd even merge sorting. This result has also been verified in [39]. The running time

of odd-even merge sorting algorithm is $O\left(\frac{N}{P}\log^2 P\,\beta + \frac{N}{P}\log^2 N\,T_{op}\right)$ under one-port assumption. If processors are capable of sending to or receiving from all their neighbors (all-port capability) the above complexity falls to $O\left(\left(\frac{N}{P}\log P + \log^2 P\right)\beta + \frac{N}{P}\left(\log^2 P + \log N\right) T_{op}\right)$.

When the number of elements at each processor is large, the most efficient algorithm is the sample sorting whereas for all values of the ratio $\frac{N}{P}$ in between the above two cases the radix sorting algorithm presents the best performance.

4. The Parallel FSBMME Algorithm

In this section we will study the realization of the FSBMME algorithm on the hypercube network. As has already been stated, in this algorithm the current frame (frame X) is partitioned into $\frac{N^2}{M^2}$ non overlapping $M \times M$ blocks. We assume that N and M are powers of 2, namely $N = 2^n$ and $M = 2^m$. The basic operations in the algorithm are given by (1), (2). In our study, we first consider the case $S < M^2$. Considering the values parameter M takes in practice, inequality $S < M^2$ implies that our multiprocessor is fine grained and each block $M \times M$ is assigned to bp processors where $bp = \frac{M^2}{S}$. Next we will examine the case of our multiprocessor being medium/coarse-grained, that is $S \geq M^2$. It will become clear later that the latter case is much easier to handle than the former.

4.1. Fine-Grained Multiprocessors

As Fig. 1(a) shows, in order to find the motion vector of a block (u, l), the MAD should be evaluated for all the $(2d+1)^2$ candidate vectors (x, y). All pixels of the previous frame Y required for these calculations are located inside the search window of the block (u, l). The parallel algorithm for this estimation consists of $O(\lceil \frac{d}{s_r} \rceil^2)$ steps. At each step, each processor fetches in its local memory a different square subregion of the previous frame Y. This subregion has size $2s_r \times 2s_r$ and is stored into four $s_r \times s_r$ local temporary arrays: temp_{00}, temp_{01}, temp_{10}, temp_{11}. Having this set of pixels of frame Y in their local memory, all processors of block (u, l) can now calculate the MAD for as much as S candidate motion vectors of the block. In what follows, we give more details of this scheme. For the moment also, we assume one-port capability. Next, we will see how our scheme can be adapted to the case of all-port capability.

One-port capability Fig. 2 shows one processor belonging to a block (u, l), processor (g, h) $(g \in [\frac{u}{s_r} \cdots \frac{u+M}{s_r} - 1],\ h \in [\frac{l}{s_r} \cdots \frac{l+M}{s_r} - 1])$, and all pixels of frame Y that will be needed by this processor throughout the execution of the FSBMME algorithm for the block (u, l). In general, this region of pixels has size $(2d + s_r) \times (2d + s_r)$ and is distributed among $(2\lceil \frac{d}{s_r} \rceil + 1) \times (2\lceil \frac{d}{s_r} \rceil + 1)$ processors. In this figure we assume that $\frac{d}{s_r} = 4$, that is s_r divides d exactly. The general case where d is not a multiple of s_r will be handled later in this section.

As a first step of motion estimation, each processor execute four shift operations: Shift(Y, temp00, $-\frac{d}{s_r} - \frac{d}{s_r}P$,$S$,$P^2$), Shift($Y$,temp01,$-\frac{d}{s_r} + 1 - \frac{d}{s_r}P$,$S$,$P^2$), Shift($Y$,temp10,$-\frac{d}{s_r} - (\frac{d}{s_r} - 1)P$, S, P^2), Shift(Y, temp11,$-\frac{d}{s_r} + 1 - (\frac{d}{s_r} - 1)P$,$S$,$P^2$) ($O(S \log P\ \beta)$ total delay). For processor (g, h), these four shifts transfer the pixels of frame Y stored in the 4 top left processors enclosed by the first dashed square of Fig. 2, namely processors $(g - \frac{d}{s_r}, h - \frac{d}{s_r})$, $(g - \frac{d}{s_r}, h - \frac{d}{s_r} + 1)$, $(g - \frac{d}{s_r} + 1, h - \frac{d}{s_r})$ $(g - \frac{d}{s_r} + 1, h - \frac{d}{s_r} + 1)$, into arrays temp00, temp01, temp10, temp11 respectively.

After these shifts, the processors of each block can estimate the MAD for S candidate vectors, namely the vectors $(-d+i, -d+j)$ where $i, j = 0, \cdots s_r - 1$. Specifically, each processor first computes S partial sums locally, each sum corresponding to one of the S MAD computations ($O(S^2\ T_{op})$ arithmetic complexity). After this set of local computations, which we name local Data_Sum operation for brevity, all partial sums belonging to the same MAD computation should be added together. This is carried out in $O\left(S \log bp\ \beta + S \log bp\ T_{op}\right)$ time by a Data_Sum operation inside each block. After this operation the top-left processor of each

block (processor $\left(\frac{u}{s_r}, \frac{l}{s_r}\right)$ for a block (u, l)) has in its local memory the values of the MAD for S candidate vectors . Next these processors estimate the minimum of these values and keep the candidate vector giving the minimum.

After calculating the best vector among the first S candidate vectors, processors of each block proceed to the estimation of the MAD for a new set of candidate vectors. Now, each processor executes two shift operations: Shift(Y,temp00, $-\frac{d}{s_r} + 2 - \frac{d}{s_r}P$,$S$,$P^2$)and Shift($Y$,temp10,$-\frac{d}{s_r} + 2 - (\frac{d}{s_r} - 1)P$,$S$,$P^2$). These shift operations overwrite the contents of arrays temp00 and temp10 whereas the contents of arrays temp01 temp11 remain intact. As a result, each processor has the pixels of a new $2s_r \times 2s_r$ subregion of the previous frame Y in its local memory. For example, processor (g, h) has the pixels $Y(i, j)$ of the previous frame Y where $i = -d + gs_r, \cdots, -d + gs_r + 2s_r - 1$ $j = -d + hs_r + s_r, \cdots, -d + hs_r + 3s_r - 1$, that is all pixels stored in processors enclosed by the second dashed square in Fig. 2. Thus all processors corresponding to a block can now estimate the MAD for a new set of candidate vectors, namely the vectors $(-d + i, -d + j)$ where $i = 0, \cdots, s_r - 1$ and $j = s_r, \cdots, 2s_r - 1$. In general, by following the route of Fig. 2, we can estimate the MAD for all candidate vectors and thus determine the motion vector of each block. The shape of the route is such that it ensures maximal temporal locality: at each step, except the first one, each processor needs to execute only two shift operations instead of four. As a result of these operations, two new $s_r \times s_r$ subregions of frame Y are transferred inside the local memory of each processor. Fig. 2 shows how these subregions are placed in the temp arrays of processor (g, h) at each iteration. These subregions together with two adjacent $s_r \times s_r$ subregions already stored in the local memory of the processor from the previous step form a new $2s_r \times 2s_r$ subregion of the previous frame Y. Thus the calculation of the MAD for a new set of S candidate vectors can now be carried out.

Apparently, under this scheme most of $s_r \times s_r$ subregions are fetched twice by each processor. However this is the best we can achieve under the assumption of $\Theta(S)$ local memory. More complex kinds of scanning of the region in Fig. 2 (e.g. Hilbert curve [40] based scanning) do not reduce this redundancy. Also, one may notice that most of the processors in charge of a block ask for the same data throughout the execution of the FSBMME algorithm. Thus instead of a series of shift steps, a reasonable solution would be to execute a number of multicasting steps where at each step all processors get a common $s_r \times s_r$ block of pixels. However, this scheme turns out to be less efficient than ours. Although all processors receive the same block of pixels, this block does not correspond to the same motion vector for all the procesors. Thus, initially processors must receive at least $\Omega(\frac{M^2}{S})$ blocks of pixels before they can start estimating MAD values. Clearly, this also raises the per processor memory requirement to at least $\Omega(\frac{M^2}{S})$ local memory. By contrast, our scheme respects the $O(S)$ local memory bound since MAD estimations can start without delay just after every two shift operations. This also leads to a smaller total delay. Recall that broadcast operations are not any faster than shift operations on the hypercube network. Both operations have $O(\log N)$ complexity on a N-node hypercube.

The total complexity of the parallel FSBMME algorithm can be easily estimated. Each step along the route of Fig. 2 takes $O\left(\left(S^2 + S\log bp\right) T_{op} + S\log P\ \beta\right)$ time. Since there are $O\left(\frac{d^2}{S}\right)$ steps overall, the total complexity of the FSBMME algorithm is $O\left(\left(d^2S + d^2\log bp\right) T_{op} + d^2\log P\ \beta\right)$ when assuming one-port capability.

We have described the parallel FSBMME algorithm assuming that the displacement d is a multiple of parameter s_r. When this is not true, the basic technique of Fig. 2 can be applied again with the only difference that processor (g, h) in Fig. 2 will take pixel blocks of size smaller than $s_r \times s_r$ from the processors along the border of the search window, namely processors $(g + i, h + j)$ where $i = -\lceil\frac{d}{s_r}\rceil, \lceil\frac{d}{s_r}\rceil$ and $j = -\lceil\frac{d}{s_r}\rceil \cdots \lceil\frac{d}{s_r}\rceil$ or $i = -\lceil\frac{d}{s_r}\rceil \cdots \lceil\frac{d}{s_r}\rceil$ and $j = -\lceil\frac{d}{s_r}\rceil, \lceil\frac{d}{s_r}\rceil$. In particular, the processors at the four corners of the search window (processors $(g + i, h + j)$ $i = -\lceil\frac{d}{s_r}\rceil, \lceil\frac{d}{s_r}\rceil$, $j = -\lceil\frac{d}{s_r}\rceil, \lceil\frac{d}{s_r}\rceil$) should give a block of size $d \bmod s_r \times d \bmod s_r$. Processors on the upper and lower border (processors $(g + i, h + j)$ $i =$

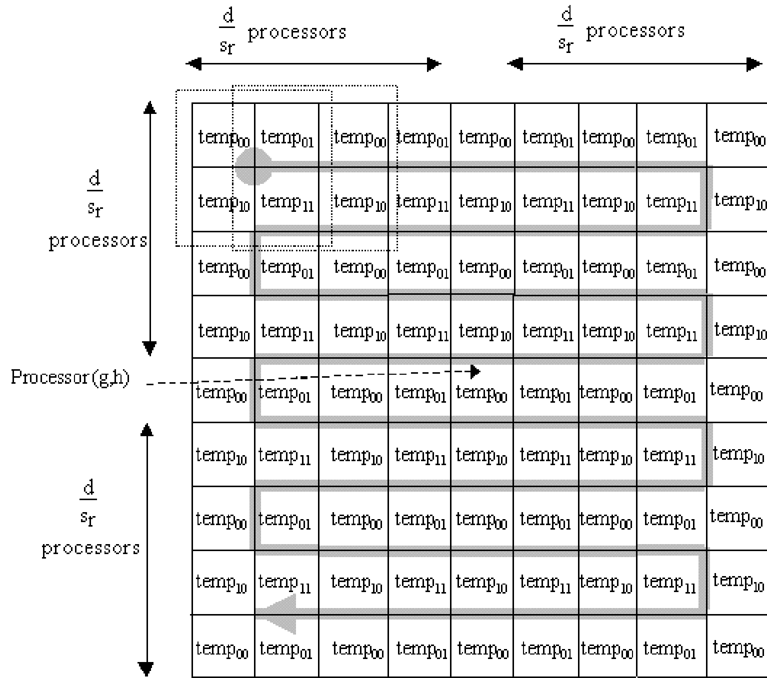

Fig. 2. Transfer of pixels of frame Y into the temp arrays of processor (g, h) by following a meander-like route.

$-\lceil \frac{d}{s_r} \rceil$, $\lceil \frac{d}{s_r} \rceil$, $j = -\lceil \frac{d}{s_r} \rceil + 1 \cdots \lceil \frac{d}{s_r} \rceil - 1)$ will provide a block of size $d \bmod s_r \times s_r$ whereas processors along the left and the right border (processors $(g+i, h+j)$ $i = -\lceil \frac{d}{s_r} \rceil + 1 \cdots \lceil \frac{d}{s_r} \rceil - 1$, $j = -\lceil \frac{d}{s_r} \rceil$, $\lceil \frac{d}{s_r} \rceil)$ should give a $s_r \times d \bmod s_r$ block of pixels. All other processors inside the search window will send a $s_r \times s_r$ block of pixels to processor (g, h) as before.

This special treatment of the processors along the border of the search window gives rise only to lower-order terms in the overall complexity. Thus the asymptotic complexity of the parallel FSBMME algorithm is $O\left(\left(d^2 S + d^2 \log bp\right) T_{op} + d^2 \log P \ \beta\right)$ again.

All port capability If processors are capable of sending and receiving along more than one link at the same time (all-port capability) we can reduce the communication overhead of the previous algorithm. The basic idea is to overlap in time the steps of the previous process. Except the first step, all other steps first execute two shift operations then a local Data_Sum operation and finally a Data_Sum operation. An important point in the above process is that rather than moving the previous frame Y around the hypercube by a series of horizontal and vertical shifts by $\pm$ 1,we maintain the initial placement of Y by transferring and storing copies of this frame into temporary arrays ($temp_{00}$, $temp_{01}$, $temp_{10}$, $temp_{11}$). In order to understand the difference, let us consider an example. Assume that a processor (i, j) needs all the $s_r \times s_r$ blocks of frame Y stored in processors (i, v) where $v = j \cdots j + q$. The straightforward solution would be to execute q horizontal shifts by -1 of the frame Y $(O(qS \log N)$ total delay). Note that these successive shifts cannot be overlapped in time. Since frame Y is moved around the hypercube, the net effect of these shifts (shift by $-q$) is achieved only when they are executed serially. However, in our scheme, since the frame Y maintains its initial placement throughout the execution, processor (i, j) can get the q $s \times s$ blocks by executing the following Shift operations: Shift by -1, Shift by -2 ... Shift by $-q$ $(O(qS \log N)$ total delay again). More importantly, these operations can be executed in any order. In other words, there is no dependence among them as in the case of shifts by -1.

From the previous example, it is now clear that the permanent storage of frame Y across the processors dramatically reduces the interdependence between the successive steps of the process depicted in Fig. 2 since from the beginning of each step we know where we can find the pixels of frame Y needed for this step without having to wait for the completion of the previ-

ous steps. As a result, each step can be executed almost independently of the other steps. Only at the point where the newly estimated MAD value is compared with the best MAD value estimated so far, the current step needs the results of the previous steps. But the scheduling of the operations is such that this information is always available on time.

We will now give the details of this scheduling. As mentioned before, except the first step[2], all other steps in the above iterative process perform two Shift operations and then calculate the MAD values for a certain set of S candidate vectors. The overlapping of these steps should be such that a minimal number of concurrent operations contend for the same node or link. The proposed scheduling guarantees that at any given moment each node executes arithmetic calculations of only two operations, namely a local Data_Sum operation and a Data_Sum operation. It is also guaranteed that at most three communication operations (two Shift and one Data_Sum operation) contend for the same hypercube link at the same time. All the above will be proved after giving the basic points of our scheduling. Our scheduling consists of three phases:

1. The first $2\log P$ steps in Fig. 2 are successively initiated every $2S\beta$ time units. Obviously, during this phase only Shift operations are in progress. The interval of $2S\beta$ time units is sufficient for transferring two messages of size S over the same link. These messages corresponds to the two Shift operations executed at each step.

2. In this phase, the remaining $\left(2\left\lceil\frac{d}{s_r}\right\rceil+1\right)^2$ $-2\log P - 1$ steps are initiated one after another with $3S\beta + (S+S^2)T_{op}$ time units separation between successive initiations. This time is necessary for completing a local Data_Sum operation (S^2T_{op} time units), the arithmetic operations of a accumulation step of a Data_Sum operation (ST_{op} time units) and the transfer of three messages of size S over the same link. These messages correspond to three communication operations contending for the same link, namely two shift and one Data_Sum operation. After initiating the last step, the above execution rate is sustained until the execution of the local Data_Sum operation of this step. After this point, all the remaining concurrent operations can be executed at a faster pace. Thus we are moving to the third phase.

3. In this phase, only Data_Sum operations are still in progress. These operations correspond to the last $\log bp$ steps of the parallel algorithm. Assuming there is no collision among these operations, this phase can be completed in $(S\beta + ST_{op})\log bp$ time units.

In fact, for this overlapping scheme to work, we should slightly modify the internal algorithm of the basic Data_Sum operation. Each Data_Sum operation in its simplest form stores its final result at the top left processor of each block (processor 0 of the corresponding bp-node subhypercube). During its execution, each Data_Sum operation visits hypercube dimensions in the order $0\cdots\log bp-1$. Thus partial sums are getting collected into successively smaller subhypercubes all containing processor 0. If we overlapped these operations in time without making any modification in the basic algorithm, some nodes would have to execute arithmetic operations for $O(\log bp)$ Data_Sum operations at each step of the second phase. But this contrasts with our assumption that at each step of the second phase each node is in charge of only one Data_Sum operation.

Clearly, in order to lower processing demands during the second phase of the proposed scheduling and hence the total arithmetic complexity, we should modify the Data_Sum operations executed inside each $M\times M$ block. As has been mentioned in Sec. 3, it is almost straightforward to alter the Data_Sum algorithm so as to collect the final sum at an arbitrary hypercube node. At that point, we have also noted that due to the associativity of the addition, the correctness of the Data_Sum algorithm is independent of the specific order in which Data_Sum operations visit hypercube dimensions. Now using these modified Data_Sum operations we obtain the following efficient overlapping scheme. In each $M\times M$ block, the first Data_Sum operation stores its result at node 2^0, the second one

[2] For convenience, we assume that this specific step is executed alone, without overlapping.

at node 2^1 and generally the jth Data_Sum operation ($j = 1 \cdots (2d+1)^2$) stores its result at node $2^{(j-1) \bmod \log bp}$ of the bp-node subhypercube corresponding to the $M \times M$ block. In addition, each Data_Sum operation visits hypercube dimensions cyclically with the jth operation starting from the hypercube dimension $(j-1) \bmod \log bp$. In this way, at any given moment all concurrent Data_Sum operations use the same hypercube dimension for their message transfers. What remains is to prove that this series of Data_Sum operations can be executed with neither link nor node contention.

Lemma 4.1. *Concurrent Data_Sum operations do not compete for the same links. In addition, at any time step each node calculates at most S partial sums all coming from the same Data_Sum operation.*

Proof. We prove the lemma by induction on the number of elapsed time steps. Clearly, the lemma is trivially true for the first step. Assume now that the lemma is true for all Data_Sum operations initiated up to time step i. Consider the Data_Sum operation starting at time step $i+1$. This Data_Sum operation will store its result at node $2^{i \bmod \log bp}$. The first message transfer of this operation is carried out along hypercube dimension $i \bmod \log bp$ in direction $0 \rightarrow 1$. At the same time all previously initiated concurrent Data_Sum operations use the same hypercube dimension but in direction $1 \rightarrow 0$ and thus there is no link collision. Clearly after this step the newly initiated Data_Sum operation works alone in the $(\log bp - 1)$-dimension subhypercube $xxx \cdots x1\underbrace{x \cdots x}_{i \bmod \log bp}$ whereas all other Data_Sum operations work inside the complementary subhypercube $xxx \cdots x0\underbrace{x \cdots x}_{i \bmod \log bp}$. From both this fact and the induction hypothesis, we can now easily see that all concurrent Data_Sum operations work on different hypercube nodes and links and thus we have proved the lemma. □

The previous lemma suggests that all-port capability is not essential for pipelining consecutive Data_Sum operations. This fact has also been showed in [16] where efficient pipelining of consecutive operations was achieved by using a 2-dilation embedding of a complete binary tree on the hypercube. However, the algorithm in [16] is not as regular as ours.

We have described how we can efficiently overlap Data_Sum operations without all-port capability. Unfortunately, for overlapping Shift operations in time we need this capability. As has already been mentioned in Sec. 3, in a single invocation of a Shift operation, hypercube dimensions are visited one at a time in a descending order. In our algorithm, at each step we use two Shift operations for transferring a new region of pixels of the previous frame Y inside the local memory of each processor. These two shifts can be easily combined into one operation and hence performed at the same time. This compound operation visits hypercube dimensions in exactly the same way as simple Shift operations. The only difference is that now the size of messages is double. Clearly, if each of these composite operations is initiated with one step delay from its previous one, all concurrent operations at any given moment use different hypercube dimensions and thus there is no link collision.

Link collision arises only among Data_Sum and Shift operations. However, at most three messages of size S try to pass through the same link at the same time and thus communication complexity increases only by a factor of 3, a constant factor. Two of these messages belong to a composite Shift operation whereas the third one is from a Data_Sum operation.

Finally, it can be easily seen that node contention among different local Data_Sum operations cannot possibly arise. Since each step in Fig. 2 execute a Local Data_Sum operation only once and there is also sufficient delay between successive step initiations in our scheduling, local Data_Sum operations corresponding to different steps are all executed at different moments.

After this sequence of overlapping Shift and Data_Sum and local Data_Sum operations, we have not yet determined the motion vectors of $M \times M$ blocks, but we are close to it; for each $M \times M$ block we know those candidate vectors which give the smallest $\log bp$ MAD values. Each such vector and its corresponding MAD value has been stored in one of the nodes 2^i

$(i = 0 \cdots \log bp - 1)$ of the subhypercube corresponding to the $M \times M$ block; recall that each Data_Sum operation has stored its result in one of these nodes. In order to estimate the final best candidate vector, we gather these MAD values to the node 0 of the above subhypercube $(O(\beta)$ delay) and then this node estimates the minimum of these MAD values $(O(\log bp\ T_{op})$ arithmetic complexity). The vector which gives this minimum determines the motion vector of the $M \times M$ block.

We have described how the operations of the FSBMME algorithm can be overlapped in time. The overall time complexity can be easily estimated by summing the running time of the three phases of the scheduling. Clearly, the first phase takes $4S \log P\beta$ time units to complete whereas the second one requires $\left(\left(\left(2\left\lceil\frac{d}{s_r}\right\rceil+1\right)^2-1\right)\left(3S\beta+(S+S^2)T_{op}\right)\right)$ time units. Finally, all the Data_Sum operations of the third phase can be completed in $\log bp\left(ST_{op}+S\beta\right)$ time units. Summing all these complexities, we can easily see that the total time for this pipelined FSBMME algorithm is $O((d^2S + S\log bp)T_{op} + (S\log P + d^2)\beta)$. This complexity does not change if we also take into account the last step of finding the minimum among the smallest $\log bp$ MAD values; this step has only $O(\beta + \log bp\ T_{op})$ complexity.

Before concluding the discussion about the implementation of the FSBMME algorithm on the hypercube-based fine grained machine, we should refer briefly to the systolic array designs that have been proposed in the literature for this algorithm. Most of these designs were derived by following the systematic approach of mapping the dependence graph of the basic operations of the FSBMME algorithm onto lower dimension systolic arrays[3]. The main difference between these designs and ours is that input frames in these designs are fed online during the algorithm execution whereas in our design the input frames have been already stored in the local memory of processors before the execution starts. However, among other proposals, type-1 array in [41], architecture AB2 in [42] and the design proposed by [43] are the most relevant to our design. Similarly to our parallel algorithm, in these systolic array architectures all arithmetic operations of a MAD calculation relevant to a particular motion vector are executed in parallel whereas the MAD calculations for different motion vectors are executed serially one after another. Another common point between our algorithm and the above mentioned proposals is that frame X maintains its initial position throughout the algorithm execution whereas in contrast the pixels of frame Y are repeatedly shifted. Especially type-1 array in [41] uses a meander-like data flow similar to that of Fig. 2 for moving the pixels of frame Y.

4.2. Medium/Coarse-Grained Multiprocessors

In this case $S \geq M^2$ and thus each processor is assigned the motion vector estimation of more than one block $M \times M$, namely $\frac{S}{M^2}$ blocks. The parallel algorithm for coarse-grained multiprocessors is very similar to that presented in the previous paragraph. Each processor will need again all the pixels inside the $(2d+s_r)\times(2d+s_r)$ subregion of Fig. 2(b). These pixels are fetched in a series of $O\left(\lceil\frac{d}{s_r}\rceil^2\right)$ steps of shift operations, following the route of Fig. 2(b). After each step, each processor is able to calculate the MAD criterion for all its blocks and for a particular set of S candidate vectors . The communication complexity of each step is $O(\tau_c + S\beta_c)$ assuming wormhole/randomized routing $(O(S \log P\ \beta)$ under a store and forward routing model and one-port assumption) and the arithmetic complexity is equal to the time required to calculate the MAD values for $\frac{S}{M^2}$ blocks and S candidate vectors, that is $O(S^2\ T_{op})$ overall. Under a store-and-forward routing model these steps can be easily overlapped in time again by using the previous pipeline scheme for fine-grained multiprocessors.

5. The Parallel HBMME Algorithm

In this section we will study the realization of the HBMME algorithm on the hypercube network. For the sake of presentation, we first assume that neither subsampling nor low pass filtering is performed between successive layers of the pyramid (basic scheme). Apparently, the basic scheme is not a "good" algorithm from image processing point of view. However, for our purposes, this simplification helps to more

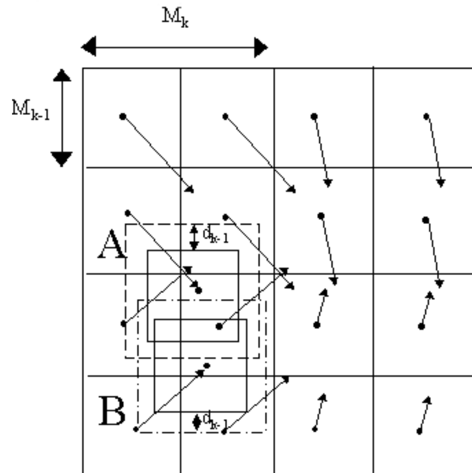

Fig. 3. Different motion vectors for each block $M_k \times M_k$ result in irregular communication patterns.

easily explain the basic difficulties arising in our effort to parallelize the HBMME algorithm. As the complete HBMME algorithm, with subsampling and low pass filtering included, is a fairly complicated algorithm by itself, the presentation of our basic techniques directly on this algorithm would be obscured by unnecessary technical details. In contrast, the basic scheme serves for presenting the key techniques of our parallel algorithm in a more manageable way. Then we will show how to adapt these techniques in order to handle the complete HBMME algorithm. Apart from the increased complexity due to the inclusion of low-pass filtering and subsampling, our main techniques remain the same in the abstract level needing only some modifications in the low implementation level.

Finally, due to space limitation, we will not consider overlapping blocks and non exhaustive search algorithms. However, the techniques which we develop in the following subsections can be easily adapted to these cases. An increase in time complexity should be expected in the case of overlapping blocks as processors in overlapping regions must execute all the operations relevant to the overlapping blocks. The slowdown factor is proportional to the degree of block overlapping.

5.1. The Basic Scheme

We assume that there are k levels in the hierarchy, k being the lowest and 1 the highest (initial) resolution level. Due to the absence of subsampling, the dimensions of video frames at each level remain the same and equal to $N \times N$. We introduce some useful notation. $M_i \times M_i$ will denote the dimensions of the blocks at the level i where $i = 1, \cdots, k$ and M_i is a power of 2 $(M_i = 2^{m_i})$. bp_i will denote the number of processors in charge of a block $M_i \times M_i$ at level i $(bp_i = \frac{M_i^2}{S})$. Finally d_i will denote the maximum vertical and horizontal displacement that a block can have at level i. We also make the realistic assumption that $M_i > M_{i-1}$.

The simplest approach in implementing the HBMME algorithm would be to naively apply the previous algorithm for FSBMME once at each pyramid level. Soon, we would realize that this approach incurs large communication overhead. Let us see in more detail how this overhead arises. The motion field at level k can efficiently be estimated without any problem by simply using the FSBMME algorithm of Sect. 4. Figure 3 shows the motion vectors just after the execution at this level; a set of motion vectors has been produced one for each block $M_k \times M_k$. These vectors probably have different size and direction as Fig. 3 shows. For instance, at level $k-1$, block A will need the pixel values enclosed by the line with pattern"— — —" whereas block B will need the pixel values enclosed by the line with pattern "— · —". Clearly, these two search windows are located at a different distance and direction from their corresponding blocks. This variation in the relative displacements is getting larger and larger as the algorithm moves to higher resolution levels. Since there does not exist a uniform displacement for all blocks of the frame, a large number of shift operations are required overall in order to fetch the pixels of each search window to the corresponding block

of the current frame.

Obviously, the basic FSBMME algorithm should be enhanced with techniques which are able to keep communication overhead low. The employed techniques depend very much on the values of parameters d_i. First, we will study the case $d_i \leq \frac{M_i}{2}$ $(i = 1, \cdots, k)$ which is often met in practice and then we will deal with the general case where there is no restriction on the values of d_i. As will become clear in the next paragraphs, the design of an efficient algorithm in the first case requires much less effort than in the general case.

5.1.1. The Case $d_i \leq \frac{M_i}{2}$.

The execution of the algorithm starts from the kth level and ends up at the first level. The calculations at a particular level cannot start before the motion estimates from the lower resolution levels are available. So, there is no parallelism across the levels. On the other hand, the computations inside each level can be easily parallelized by following a data parallel approach. Again, we draw a distinction between fine-grained $(S < M_k^2)$ and medium/coarse-grained multiprocessors $(S \geq M_k^2)$.

Fine-grained multiprocessors. Since the size of $M_i \times M_i$ blocks are getting smaller and smaller as we are moving to higher resolution levels, there may be a level i^* after which the size of these blocks is smaller than $\Theta(S)$, that is the size of local memory of each processor. If this is actually the case, then after level i^* we execute the variant of the algorithm for medium/coarse-grained multiprocessors described later in the paper.

For the moment, we will describe the execution of the algorithm at the lowest resolution level k. Exactly the same techniques are used for all other levels up to level i^*. Figure 4(a) shows a block at level k. The area enclosed by the dashed line contains all pixels of the previous frame Y that could be possibly required by the processors of this block at all levels of the pyramid. The basic steps at level k are:

- transfer of pixels of the shaded region of frame Y inside the block $M_k \times M_k$. Two horizontal shifts by $\sum_{i=1}^{k} d_i$ and two vertical shifts by the same displacement can perform this transfer in $O(S \log P\, \beta)$ $(O((S + \log P)\,\beta))$ time under one-port (all-port) assumption. Obviously, these operations are performed in parallel for all the blocks of the frame. After these movements each processor will hold $9S$ $(= \Theta(S))$ pixels at most. This is due to the values of parameters d_i $(d_i \leq \frac{M_i}{2}$, $i = 1, \cdots, k)$. Later, when we examine the general case, it will become clear that this initial concentration greatly reduces the communication cost and thus it is very important for achieving an efficient algorithm.
- execution of the FSBMME algorithm. After the completion of the previous step, each block has all the required information for the estimation of its motion vector

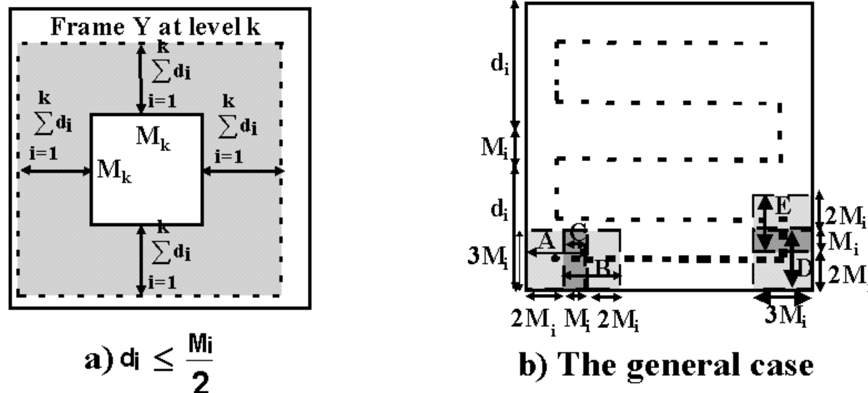

Fig. 4. Efficient techniques for the basic scheme of the HBMME algorithm.

and thus it needn't communicate with its adjacent blocks any longer. The algorithm in each subhypercube $M_k \times M_k$ is similar to the algorithm presented in Sect. 4.1. Since now each processor may be in charge of 9 $s_r \times s_r$ blocks, one may expect that processors should be sending more than one messages of size S at each step of a Shift operation, even under one port assumption. However, this is not the case. As the Shift operation belongs to the class of data permutations , each processor sends and receives only one block of size S during this operation. In our algorithm, a processor needs to send only one of the nine blocks of size S stored in its local memory. Each processor can easily determine this block by examining the current shifting distance.

Apparently, the number of messages sent at each step of Data_Sum operation does not change too. This operation is not affected by the different placement of frame Y, since it has as input the partial sums estimated by the Local Data_Sum operation at each processor.

As now all the Shift operations are executed inside $\log bp_k$-node subhypercubes, the total communication complexity of the FSBMME algorithm falls to $O\left(\left(d_k^2 + S\log bp_k\right)\beta\right)$ under all-port assumption whereas its arithmetic complexity remains the same, namely $O\left(\left(d_k^2 S + S\log bp_k\right)T_{op}\right)$.

- In each block $M_k \times M_k$, broadcasting of the estimated motion vector to all the processors of the block ($O\left(\log bp_k\ \beta\right)$ delay).

After the estimation of the motion field at level k, the algorithm visits the other levels of the pyramid and it repeatedly executes the three steps above up to level i^*. For example, at level $k-1$ the first step is the collection in each block $M_{k-1} \times M_{k-1}$ of all the pixel values of the previous frame Y which are within $\sum_{i=1}^{k-1} d_i$ pixels around the new position of the block. Clearly, the new position of the block at level $k-1$ is determined by the motion vector of its "parent" block $M_k \times M_k$ at level k. Notice that this motion vector is also the same for the other three $M_{k-1} \times M_{k-1}$ "child" subblocks of the $M_k \times M_k$ block. Thus the first step can be executed with simple shift operations without encountering any of the problems of Fig. 3. The complexity of the first step is $O\left(S\log bp_k\ \beta\right)$ ($O\left(S\log bp_{i+1}\ \beta\right)$ at level i) and not $O\left(S\log P\ \beta\right)$ as at the level k. This is because shift operations can now be performed inside subhypercubes $M_k \times M_k$ in contrast to the level k where the whole hypercube $P \times P$ is used. All the above complexities are obtained under one port assumption. Under all-port assumption, the first step at level i can be executed in $O\left(\left(S + \log bp_{i+1}\right)\beta\right)$ time. Under the same assumption, the second and third steps at level i have complexities $O\left(\left(d_i^2 + S\log bp_i\right)\beta + \left(d_i^2 S + S\log bp_i\right)T_{op}\right)$ and $O\left(\log bp_i\ \beta\right)$ respectively.

After level i^* we use the algorithm variant for medium/coarse-grained multiprocessors. Based on the results of the next paragraph, the execution of HBMME algorithm at the last $i^* - 1$ levels of the pyramid incurs $O\left(\left(S + \log bp_{i^*}\right)\beta\right)$ communication and $O((S\sum_{j=1}^{i^*-1} d_j^2)T_{op})$ arithmetic delay assuming all-port capability. Thus, for the case $d_i \leq \frac{M_i}{2}$ $(i = 1 \cdots k)$ the total communication and arithmetic complexity of the HBMME algorithm is $O\Big(\log P + \sum_{j=i^*}^{k}\Big(\Big(d_j^2 + S\log bp_j\Big)\beta + S\left(d_j^2 + \log bp_j\right)T_{op}\Big) + S\left(\sum_{j=1}^{i^*-1} d_j^2\right)T_{op}\Big)$ under all-port assumption. The complexity for one-port assumption can be easily derived in the same way by summing the delays of the algorithm steps under this assumption. We leave the details to the reader.

Medium/Coarse-grained multiprocessors. In this case, the size of blocks at level k is smaller than the size of local memory of each processor, that is $M_k^2 = O(S)$. Like the previous algorithm, each processor first gathers all pixels of the previous frame Y which are within $\sum_{i=1}^{k} d_i$ pixels around its own portion. After this step, each processor has all necessary information for applying the HBMME algorithm to its blocks and thus no further communication is required among processors. Hence the communication complexity in this case is only $O\left(\tau_c + S\beta_c\right)$ assuming wormhole or randomized routing and $O(S\log P\ \beta)$ under a store-and-forward routing model and one-port assumption. The arith-

metic complexity of the algorithm is as much as $O\left(S\left(\sum_{j=1}^{k} d_j^2\right) T_{op}\right)$.

5.1.2. The General Case

In the general case the parameters d_i have arbitrary values, thereby complicating the design of an efficient algorithm. If we try to apply the methods of the previous subsection, we will soon find out that after the first step of data collection for each block, the load of some processors is not necessarily bounded. This violates our initial assumption that the memory of each processor is $\Theta(S)$. Thus, we have to devise different techniques in order to handle the general case.

As has been shown in Fig. 3, except the lowest resolution level, all other levels require communication that does not have a special pattern. As the algorithm execution moves to the highest resolution level of the multiresolution pyramid this pattern is getting more and more irregular and thus more general communication operations such as RAR operations are clearly needed. As has already been mentioned, under a store-and-forward routing model each RAR operation is realized using two sorting steps one at the beginning and one at the end of the operation. These two steps are the most expensive in each RAR operation and thus determine the whole complexity of the operation too. On a N-node hypercube , sorting N elements, one element per node, can be executed in near optimal time, that is close to $O(\log N)$ [26]. However, this kind of sorting algorithms is rather theoretical with large constant factors hidden in the O-notation. The most known practical sorting algorithm for the network of hypercube is the odd-even merge sorting algorithm which has $O\left(\log^2 N\ \beta + \log^2 N\ T_{op}\right)$ complexity assuming the same data allocation as above, that is one data element per node. This algorithm first splits the input elements into $\frac{N}{2}$ lists of 2 elements each and then recursively merges larger and larger sorted lists until the N elements turn up sorted. Unfortunately, the $O(\log^2 N)$ complexity mentioned above cannot be hidden by overlapping RAR operations. The practical sorting algorithms usually employed in RAR operations, when pipelined, cause a large and non constant number of packets to contend for the same links, and thus greatly increase the local memory requirements.

Having defined the RAR operation as the basic communication primitive in the general case of HBMME algorithm, we now describe in more detail how motion estimation is performed at a pyramid level i of the hierarchy. Once more, we first handle the case $S < M_i^2$ and then the case $S \geq M_i^2$. Since the values of M_i are not very large in practice, when the first case is true then we are almost sure that the employed multiprocessor is fine-grained. When the opposite is true, what kind of multiprocessor we assume depends on the relative values of S and M_i^2. If $S \gg M_i^2$ then our multiprocessor is assumed to be coarse-grained whereas when the values of S and M_i^2 are comparable we can assume that our multiprocessor is medium-grained or even fine-grained. In regard to the employed routing method, we assume wormhole or randomized routing for coarse-grained machines and store-and-forward routing for fine-grained, medium-grained machines.

Fine grained multiprocessors. The simplest approach to implementing the HBMME algorithm in the general case is to perform $(2d_i+1)^2$ RAR operations at each level i , one operation per candidate vector. Given T_{RAR} the complexity of a single RAR operation, this simple approach has $O(d_i^2 T_{RAR})$ complexity for level i. Obviously, there are two ways of reducing this complexity: a) by decreasing the number of RAR operations required at level i and b) by reducing the time T_{RAR} required by a single RAR operation.

We will first describe a technique for decreasing the number of RAR operations at each pyramid level. The basic idea is to take advantage of the special structure of the HBMME algorithm. Although it is not possible to initially transfer all pixel values needed by the processors of a block inside this block, these values can be transferred in batches. Figure 4(b) shows the search window of a block $M_i \times M_i$ at the level i. With at most 9 RAR operations, the pixels of the region A are transferred inside the block $M_i \times M_i$. After this transfer each processor will have $9S$ pixel values in its local memory. Clearly, each block $M_i \times M_i$ has now all the necessary information for the estimation of the

MAD at $(2M_i + 1)^2$ possible displacements. As has been shown in Sec. 4.1, the communication operations required by these estimations (Shift and Data_Sum operations) can be easily overlapped and thus these calculations can be completed with only $O\left(\left(S \log bp_i + M_i^2\right)\beta\right)$ communication and $O\left(S\left(M_i^2 + \log bp_i\right) T_{op}\right)$ arithmetic delay.

After this set of computations, the next batch of pixel values should be transferred inside each block $M_i \times M_i$. In Fig. 4(b), these pixels are located inside the region B. In fact, since the regions A and B intersect at region C, only pixels outside this region need to be transferred. In general, following the meandering route of Fig. 4(b), we can calculate the MAD for all the candidate vectors inside the search window using only $O\left(\lceil \frac{d_i}{M_i} \rceil^2\right)$ RAR operations in total. The right part of Fig. 4(b) shows how we move from one row to the next higher row of the route (blocks D and E). In the figure we assumed that d_i is a multiple of M_i; otherwise, the width of both the blocks D and E would be smaller than $3M_i$, namely $M_i + 2\,(d_i \bmod M_i)$.

Besides the above improvement, motion estimation at level i can be further sped up by keeping low the time complexity (T_{RAR}) of each RAR operation. In a RAR operation, each processor i reads a $s_r \times s_r$ block of pixels whose the top-left corner is, say, at the pixel (t_i, l_i) of the previous frame Y. In what follows, we briefly mention the main steps of a RAR operation in our algorithm [25]:

1. Each processor i creates a quadruple $Q_i = (i, t_i, l_i, \lfloor \frac{l_i}{s_r} \rfloor + P \lfloor \frac{t_i}{s_r} \rfloor)$. The fourth entity gives the address of the processor holding the top left pixel (t_i, l_i).

2. Sort the quadruples into non-decreasing order of the fourth entity. After this step, quadruples destined for the same processor appear consequently in the sorted order. Let $G_j = \{Q_0^j, Q_1^j, \cdots Q_{r_j-1}^j\}$ be such a group of quadruples residing in processors $i_j, i_j+1, \cdots i_j+r_j-1$ and whose common destination is the processor j.

3. For each group G_j, the expressions $(T_j, L_j) = (\min_{i=0\cdots r_j-1} t_i^j, \min_{i=0\cdots r_j-1} l_i^j)$ and $(B_j, R_j) = (\max_{i=0\cdots r_j-1} t_i^j + s_r, \max_{i=0\cdots r_j-1} l_i^j + s_r)$ are estimated and stored in processor i_j. This estimation can be performed in $O(\log P\ \beta)$ time by using a segmented prefix operation [26]. Notice that the coordinates $\left(T_j, L_j\right)$ and $\left(B_j, R_j\right)$ are the top left and bottom right corners respectively of the minimal rectangle that encloses all the $s_r \times s_r$ blocks corresponding to the quadruples of the group G_j. It can also be easily seen that the size of this rectangle is at most $4S$.

4. The first processor of each group G_j, processor i_j, sends the tuple $(i_j, T_j, L_j, B_j, R_j)$ to processor j. Since $i_j < i_l$ for any j, l where $j < l$, this step can be easily done by using monotone routing. As each tuple has $O(1)$ size, the communication delay of this transfer is $O(\log P\ \beta)$ at most.

5. Now each processor j that received a tuple at the previous step knows which pixels of its local portion are requested by the processors of group G_j. Notice that the rectangle (T_j, L_j, B_j, R_j) may span more than one processors, at most 4. In this case, processor j $(= (\lfloor \frac{j}{P} \rfloor, j \bmod P))$ must get pixels from some of its adjacent processors, namely processors $(\lfloor \frac{j}{P} \rfloor, j \bmod P + 1)$, $(\lfloor \frac{j}{P} \rfloor + 1, j \bmod P)$ and $(\lfloor \frac{j}{P} \rfloor + 1, j \bmod P + 1)$. These transfers can be easily done in $O((S + \log P)\beta)$ time using only simple shift operations and assuming all-port capability.

6. Having collected the required pixels in its local memory, each processor j multicasts all pixels of the previous frame Y inside the rectangle (T_j, L_j, B_j, R_j) to all processors of group G_j. This transfer can be realized in $O\left((S + \log P)\,\beta\right)$ time under all-port assumption by executing a concentrate operation followed by a generalized operation (see [25] for more details of these operations).

7. Now each processor examines the received rectangle and keeps only the pixels that it actually needs, discarding all other pixels. The remaining pixels form a $s_r \times s_r$ block that should be returned to the processor which initially asked for it. Thus a sorting step is executed where these blocks are

sorted according to the addresses of their final destinations. The last information has been kept in the local memory of each processor from the second step.

It is worth mentioning that by sending only one packet from each group G_j to the corresponding destination j we avoid serious hot spots at these nodes. The packet leaving each group contains the coordinates of the minimal rectangle enclosing all the requests of the group. As a result, the region of pixels returned to processors at step 6 is somewhat larger than that they initially asked for. Specifically, each processor receives a region of at most $4S$ pixels instead of S pixels. This difference is not large, because the parameter S is usually small in the case of fine grained multiprocessors. If processors were to receive exactly the pixels they asked for, then processors of each group G_j should have sent separate request-packets to their common destination j. Each such packet would contain the coordinates of the $s_r \times s_r$ region required by the sender of the packet. After collecting these requests, destination processors should have sent different reply-packets to each of the processors of their group. It is not hard to see that in this case the number of packets a processor should receive at step 4 and send at step 6 is not necessarily small since in the worst case a processor must receive and then send $\frac{N^2}{M_i^2}$ packets at level i of the pyramid. Apparently, this could create serious hot spots at some nodes, thereby raising the complexity of the RAR operation to at least $\Omega\left(S\frac{N^2}{M_i^2}\beta\right)$ time steps.

As has been mentioned previously, the two sorting steps in the RAR operation dominate the cost of this operation. Now we will give details of how these two sorting steps can be efficiently executed. We prove the following lemma:

Lemma 5.1. *The first sorting step of a RAR operation at level i can be performed with* $O\left(\left(\log^2 P - \log^2 bp_i\right) T_{op} + \left(\log^2 P - \log^2 bp_i\right)\beta\right)$ *delay.*

Proof. In each invocation of RAR operation, each processor $\left(\lfloor\frac{u}{s_r}\rfloor + i, \lfloor\frac{l}{s_r}\rfloor + j\right)$ $(i, j = 0 \cdots \frac{M_i}{s_r} - 1)$ of a $M_i \times M_i$ block (u, l) must read the pixels of a $s_r \times s_r$ block of the previous frame Y whose top left corner is at pixel $\left(t_{(i,j)}, l_{(i,j)}\right) = (x+i\ s_r+u, y+j\ s_r+l)$ where (x, y) is a displacement vector. Hence, it must communicate with the processor which stores the pixel $\left(t_{(i,j)}, l_{(i,j)}\right)$, namely the processor $\left(\lfloor\frac{u+x}{s_r}\rfloor + i, \lfloor\frac{l+y}{s_r}\rfloor + j\right)$. Due to this special reading pattern, the input of the sorting step of each RAR operation consists of $\frac{N^2}{M_i^2}$ sorted lists of at most bp_i quadruples each. Each of these lists corresponds to a $M_i \times M_i$ block of the current frame and the quadruples of each list are already sorted by the coordinates of the processors to be contacted. This ordering becomes more apparent after performing the following permutation: processor $(i, j) = (i_{p-1}i_{p-2}\cdots i_{m_i-r}i_{m_i-r-1}\cdots i_0, j_{p-1}j_{p-2}\cdots j_{m_i-r}j_{m_i-r-1}\cdots j_0)$ sends its quadruple to processor $(i_{p-1}i_{p-2}\cdots i_{m_i-r}j_{p-1}j_{p-2}\cdots j_{m_i-r}, i_{m_i-r-1}\cdots i_0 j_{m_i-r-1}\cdots j_0)^3$. This permutation transforms each $M_i \times M_i$ block into a linear array by "scanning" the $2^{2(m_i-r)}$ processors of the block in row-major order. It also belongs to the well studied class of the Bit-Permute-Complement permutations and can be executed in $O(\log P\ \beta)$ time [25]. After this communication step, the emerging set of sorted lists can be easily merged in $O\left(\left(\log^2 P - \log^2 bp_i\right) T_{op} + \left(\log^2 P - \log^2 bp_i\right)\beta\right)$ time by applying the odd-even merge sorting algorithm.

□

The second sorting step completes the RAR operation by moving each $s_r \times s_r$ pixel block to its ultimate destination. This can be simply done in $O\left(\left(S\log P + \log^2 P\right)\beta + \log^2 P\ T_{op}\right)$ time by using the odd-even merge sorting algorithm and assuming all-port capability. But we can actually do better if we just reverse the two phases of the first sorting step. Thus as a first phase, each $s_r \times s_r$ pixel block destined for processor $(i, j) = (i_{p-1}i_{p-2}\cdots i_{m_i-r}i_{m_i-r-1}\cdots i_0, j_{p-1}j_{p-2}\cdots j_{m_i-r}j_{m_i-r-1}\cdots j_0)$ is moved to processor $(i_{p-1}i_{p-2}\cdots i_{m_i-r}j_{p-1}j_{p-2}\cdots j_{m_i-r}, i_{m_i-r-1}\cdots i_0 j_{m_i-r-1}\cdots j_0)$. As is proved in lemma 5.2, this phase can be completed in $O\left(S\left(\log^2 P - \log^2 bp_i\right)\beta + \left(\log^2 P - \log^2 bp_i\right) T_{op}\right)$ under one-port assumption and in $O((S(\log P -$

[3] Recall that $M_i = 2^{m_i}$, $s_r = 2^r$ and $P = 2^p$.

$\log bp_i) + \log^2 P - \log^2 bp_i)\beta + \left(\log^2 P - \log^2 bp_i\right)$ $T_{op})$ time under all-port assumption. After this phase, the permutation $\left(i_{p-1}i_{p-2}\cdots i_{m_i-r}i_{m_i-r-1}\cdots i_0, j_{p-1}j_{p-2}\cdots j_{m_i-r}j_{m_i-r-1}\cdots j_0\right) \rightarrow \left(i_{p-1}i_{p-2}\cdots i_{m_i-r}j_{p-1}j_{p-2}\cdots j_{m_i-r}, i_{m_i-r-1}\cdots i_0 j_{m_i-r-1}\cdots j_0\right)$ is executed again. But now it takes $O\left(S \log P\ \beta\right)$ time under one-port assumption since all messages sent or received at each communication step have $\Theta\left(S\right)$ size. However, under all-port assumption this complexity can be lowered to $O\left(\left(S + \log P\right)\beta\right)$ time by overlapping the communication steps of the algorithm implementing the bit-permute-complement permutation. After the end of the second phase, each $s_r \times s_r$ block has reached its final destination. Now we prove the following lemma:

Lemma 5.2. *The first phase of the second sorting can be completed in* $O\left(S\left(\log^2 P - \log^2 bp_i\right)\beta + \left(\log^2 P - \log^2 bp_i\right) T_{op}\right)$ *time under one-port assumption and in* $O\left(\left(S\left(\log P - \log bp_i\right) + \log^2 P - \log^2 bp_i\right)\beta + \left(\log^2 P - \log^2 bp_i\right) T_{op}\right)$ *time under all-port assumption.*

Proof. For the implementation of this stage, we use a variant of the radix sort algorithm. This kind of sorting algorithm does not belong to the class of comparison only sort algorithms, since it exploits the binary representation of keys in order to determine their relative ordering. For the radix sort algorithm to be an efficient algorithm, the input keys should be integers taken from a limited range of values. In our case, the keys are the $2\log P$-bit integers corresponding to the addresses of processors. The algorithm starts its execution by first examining the most significant bit of each key ($2\log P - 1$ bit). Depending on the value of this bit, each key is moved to the corresponding $(2\log P - 1)$-dimensional subhypercube. Using a packing operation [26], this move can be easily implemented in $O\left(S \log P\ \beta + \log P\ T_{op}\right)$ time under one-port assumption. If the all-port assumption is true, the steps of the packing operation can be easily interleaved in time [44] and thus the time for this operation is reduced to $O\left(\left(S + \log P\right)\beta + \log P\ T_{op}\right)$. Now the execution of the algorithm continues recursively within the two $(2\log P - 1)$-dimensional subhypercubes. At the ith step the i^{st} most significant bit is examined and the keys are moved to the corresponding $(2\log P - i)$-dimensional subhypercubes. Notice also that at the ith step the cost of the packing operation is at most $O\left(\left(S + \log P - i\right)\beta + \left(\log P - i\right) T_{op}\right)$ under all-port assumption.

In fact, we do not need to examine the last $2(m_i - r)$ bits of the keys. When the algorithm reaches the $2(m_i - r)^{st}$ least significant bit, all $s_r \times s_r$ blocks destined for processors of the same $M_i \times M_i$ block are within the same $2(m_i - r)$-dimensional subhypercube. In addition, these blocks turn up sorted. Recall from lemma 5.1 that all the request-packets coming from the same $M_i \times M_i$ block have already been sorted at the beginning of each RAR operation. Since this relative ordering is respected in all steps of each RAR operation, the $s_r \times s_r$ blocks asked for by these request-packets end up sorted after they have been collected inside their corresponding $2(m_i - r)$-dimensional subhypercubes.

Now it is clear that the number of packing operations required overall is $O\left(\log P - \log bp_i\right)$. Summing the delays of these operations, we can easily obtain the complexities stated in the lemma.

□

The second sorting step of a RAR operation dominates the cost of this operation. Thus the total complexity of RAR operations at level i is $O\left(S\left(\log^2 P - \log^2 bp_i\right)\beta + \left(\log^2 P - \log^2 bp_i\right) T_{op}\right)$ under one-port assumption and $O\left(\left(S\left(\log P - \log bp_i\right) + \log^2 P - \log^2 bp_i\right)\beta + \left(\log^2 P - \log^2 bp_i\right) T_{op}\right)$ under all-port assumption.

Medium/Coarse Grained Multiprocessors. In this case, each processor has at least one $M_i \times M_i$ block in its local memory, namely $\frac{S}{M_i^2}$ blocks. If this fraction is not very large then considering the values of M_i in practice, we can assume that our multiprocessor is medium-grained or even fine-grained. This kind of multiprocessors usually employs store-and-forward routing. Thus each RAR operation can be performed in $O\left(\left(S \log P + \log^2 P\right)\beta + \frac{S}{M_i^2} \log \frac{N^2}{M_i^2}\ T_{op}\right)$ time

using the algorithm in [34]. This algorithm assumes all-port capability at each node and implements the many-to-many personalized communication pattern on the hypercube network. In order to reduce the number of packets received by destination nodes, i.e. nodes holding pixel regions to be read by other processors, we use a technique similar to that of the previous paragraph. Specifically, we can ensure that the number of messages received by each destination node is at most $\frac{S}{M_i^2}$ by allowing each block $M_i \times M_i$ to receive somewhat more pixels than it actually needs at the end of each RAR operation. Specifically, it receives at most $4M_i^2$ whereas it needs only M_i^2.

Now if the fraction $\frac{S}{M_i^2}$ takes very large values ($S \gg M_i^2$), then we can assume that the employed multiprocessor is coarse-grained. As a matter of fact, it is very likely that the above inequality is true for all other pyramid levels too. This in turn implies that the sum of maximum displacements d_i for all pyramid levels ($\sum_{i=1}^{k} d_i$) is much smaller than $O(\sqrt{S}) = O(s_r)$ in all probability. Clearly, if these assumptions are true, then each processor can initially perform a gathering of all the pixels of the previous frame Y that it will need throughout the execution of the HBMME algorithm ($O(\tau_c + S\beta_c)$ delay). After this step, the load of each processor remains $\Theta(S)$. In addition, processors can now work independently of each other and thus no further interprocessor communication is necessary.

On the rare occasion that either the size of blocks or the maximum displacements at each level are so large that the initial gathering cannot be performed without increasing the load of each processor beyond the bound $\Theta(S)$, we can use the algorithm in [31] for implementing the RAR operation. As has been mentioned in Sec. 3 , this algorithm assumes a virtual point-to-point link between each pair of processors and thus transferring a message of size M between any pair of processors takes $O(\tau_c + M\beta_c)$ time. Using this algorithm we can perform each RAR operation at pyramid level i in $O\left(P^2\ \tau_c + S\beta_c + \frac{S}{M_i^2}\ T_{op}\right)$ time.

Besides the improvements we can achieve in the execution time of a single RAR operation, we can also reduce the total number of these operations required at each pyramid level by using again the technique of Fig. 4(b); each $M_i \times M_i$ block at pyramid level i reads the pixels of its search window in $O\left(\lceil \frac{d_i}{M_i} \rceil^2\right)$ steps following the route of this figure. Thus the total number of RAR operations required at level i is $O\left(\lceil \frac{d_i}{M_i} \rceil^2\right)$ again.

5.2. Subsampling and Low-Pass Filtering

So far we have presented a simplified scheme for the multiresolution motion estimation algorithm where low pass filtering and subsampling between successive pyramid levels has been left out. When using low pass filtering and subsampling, the dimensions of video frames are getting increasingly smaller at higher levels of the pyramid and the values of the pixels of each frame are changing from one level to another. Despite these complications, our parallel algorithm can be easily adapted to the case of low pass filtering and subsampling.

The most commonly used low pass filters take the form of small two dimensional arrays where the value of each pixel is given by a linear function of the values of the pixels in its neighborhood [45]. This kind of operation can be implemented on the hypercube by using a parallel template matching algorithm (see for example the parallel algorithms in [46, 47]). Different though the purpose of these algorithms is, their basic step is a convolution-like operation and thus they can also serve to implement low pass filtering. After this operation the low pass filtered frame is subsampled. Many subsampling patterns have been proposed in the literature but one of the most commonly used is to keep every other pixel along each row and column [45]. For example, the frame at level 2 results from the low pass filtered frame at level 1 by keeping the pixels $(2k, 2l)$ where $k, l = 0 \cdots \frac{N}{2} - 1$. In general, the frame at level i results from the frame at level $i-1$ by keeping the pixels $\left(2^{i-1}k, 2^{i-1}l\right)$ where $k, l = 0 \cdots \frac{N}{2^{i-1}} - 1$. The well-known Gaussian pyramid representation by Burt et al.[48] uses the same kind of subsampling in combination with low pass filtering based on Gaussian filter. Other multiresolution schemes[11, 15] use also similar subsampling

along with simple averaging of neighbouring pixels as a low pass filter.

This subsampling operation reduces the number of pixels of input frames as we are moving to higher pyramid levels. Thus after a certain level, level $\log S + 1$, some processors begin not to have pixels to process, as all their stored pixels have been discarded due to subsampling process and thus these processors remain inactive. Further, after this particular level, the number of inactive processors increases by a factor of 4 as we are moving from one level to the next higher one. At the same time, each processor which remains active after level $\log S + 1$ needs more than $\Theta(S)$ local memory, namely $\Theta(S + k)$, in order to keep the different values its pixels assume due to low-pass filtering across pyramid levels. Fortunately, it is possible to balance the load of processors by distributing the extra load to inactive processors. Specifically, the values of a pixel in processor (i, j) at all levels after level $\log S + 1$ can be stored in those inactive processors which are also neighbors in the hypercube with processor (i, j). At the beginning of the execution at each pyramid level higher than level $\log S + 1$, this processor can take the value of its pixel for that level from one of these neighbors in the hypercube with only $O(1)$ delay.

On the practical side, the algorithm we described above is more suitable for fine-grained machines. In coarse-grained machines, the value of parameter S is so large that it is very unlikely the employed multiresolution pyramid has more than $\log S + 1$ levels. This simply means that there are not inactive processors in coarse-grained machines and thus problem of load imbalance cannot normally arise. However, in coarse-grained machines we face a different difficulty. As is mentioned before, due to subsampling the size of input frames from one level to the next higher one is reducing by a factor of 4. Accordingly, the number of pixels in the local memory of each processor is reducing by the same factor. Thus at higher pyramid levels our algorithm begins to assume fine-grained characteristics: increasingly fewer arithmetic operations are performed before executing a communication operation. But in coarse-grained machines, communication operations are rather expensive in general and thus frequent execution of these operations leads to high overhead. Thus in order to alleviate this overhead, it would be better if the pixels of input frames at one pyramid level were gathered in a specific subhypercube before beginning the execution of the FSBMME algorithm for that level. In this way, more pixels correspond to each processor and thus the frequency of communication operations is decreased proportionally. If the number of pixels per processor increases from S_1 to S_2 then we can execute the above gathering step in $O(\frac{S_2}{S_1}\tau_c + S_2\beta_c + T_{op})$[4] time by using the concentrate algorithm in [30]. The size of the subhypercube used for storing the frames at one pyramid level depends on how many times a communication operation is more expensive than a single arithmetic operation. The larger this difference is the smaller the subhypercube should be. The optimal size at a pyramid level can be easily determined by comparing the execution time at this level of the algorithm enhanced with this gathering step and the execution time of the algorithm without this step. The optimal subhypercube size is that which gives the largest time savings. Based on the formulas derived in the previous paragraphs of the paper, we can easily perform this optimization. We leave the details to the interested reader.

We have described the basic initialization steps before the execution of the HBMME algorithm at a particular pyramid level. Apart from these initial steps, the rest of the algorithm at each level uses the same techniques we developed for the basic scheme of our parallel algorithm. Apparently now, the total complexity of the HBMME algorithm is greatly lowered, because the size of both the frames and subhypercubes involved in the algorithm execution are getting smaller and smaller as we are moving to higher pyramid levels. At this point, one may reasonably argue that since the size of frames are decreasing at higher pyramid levels, our techniques are not as important at these levels. Naive straightforward techniques could be acceptable in this case, since at higher levels the hypercube network does not handle large data volume. However, under this simplified strategy, we end up having much larger communication overhead in total. Recall from the discussion in Sec 5.1 that the basic problem in using straightforward techniques is that as the

[4] Wormhole or randomized routing model is assumed.

algorithm proceeds towards lower pyramid levels irregular communication patterns make their appearance. The irregularity of these patterns is ever increasing and thus when we arrive near the bottom of the pyramid where the size of frames is rather large, we encounter quite large communication overhead, which eventually dominates the total execution time. In contrast, our techniques ensure that we can count on simple shift operations most of the time in order to perform all the necessary data transfers at the lower levels of the pyramid, where efficient communication is highly desirable.

Another difference between the basic scheme and the complete HBMME algorithm is that whereas in the basic scheme the size $M_i \times M_i$ of block at level i is increasing with the height of level i, in the actual algorithm the size of blocks remains basically the same across the different pyramid levels. However, as mentioned above, the frames at higher levels are distributed more sparsely among the processors, and thus any two neighboring pixels are now wider apart. This in turn implies that a block at a particular level spreads over a larger portion of the processor array than the blocks at next lower level. Thus once again, each block at any pyramid level "contains" all its child blocks at the next level. Since our parallel algorithm for the practical case $d_i \leq \frac{M_i}{2}$ was based exactly on this property, this algorithm can now be used again for handling the same range of values of displacements d_i in the complete HBMME algorithm.

Finally, with regard to the general case of basic scheme, the algorithm does not make any specific assumption about the size of blocks at various levels and thus is directly applicable to the complete HBMME algorithm. The employed sorting step in conjunction with the set of shifts performed "inside" each block can cope successfully with the irregular patterns arising at the lower levels of the pyramid.

6. Extensions to other Interconnection Networks

So far we have developed parallel algorithms for motion estimation techniques on a hypothetical multiprocessor which uses a hypercube network for interprocessor communication. Yet most of our techniques are actually independent of the specific interconnection network we use. This is certainly true for coarse-grained multiprocessors. Our algorithms for this kind of multiprocessors either make a very restricted use of the interconnection network or are based on a virtual crossbar communication model where the cost of each communication operation is assumed to be independent of the distance of the processors involved. This model has been verified in most modern coarse-grained multiprocessors where the role of the employed interconnect is ever diminishing.

Even for fine-grained multiprocessors, where store-and-forward routing is usually employed, our algorithms can be easily extended to other interconnection networks as well. As should have become apparent in our analysis, shift operations play an important role in our design. In the FSBMME algorithm as well as in the case $d_i \leq \frac{M_i}{2}$ of HBMME algorithm, all data transfers are carried out through shift operations. We also use Data_Sum and broadcast operations for the remaining steps. These three operations are fairly simple and thus can be efficiently implemented on most of interconnection networks, e.g. the mesh network.

With regard to the general case of HBMME algorithm, our main objective is to use simple shift operations as much as possible in place of complex RAR operations. As has already been mentioned, the emerging communication pattern in RAR operations is the many-to-many personalized communication. This kind of communication has been extensively studied in most communication networks. For instance, on the mesh network a number of algorithms have been proposed which realize this communication pattern in near optimal time[49, 50]. The main drawback of these algorithms is that they are rather theoretical with large low-order terms in their complexity. Thus they are not very practical for medium and small size parallel machines, i.e. machines frequently met in practice. Alternatively, we could use practical sorting algorithms again for implementing RAR operations on the mesh. A number of practical algorithms for sorting on the mesh has already been presented in the literature[37, 51, 52, 53]. Most of these references conclude that bitonic sorting, an algorithm closely related to odd-even merge sorting, outperforms all other sorting algorithms for small values of ratio $\frac{N^2}{P^2}$ (number

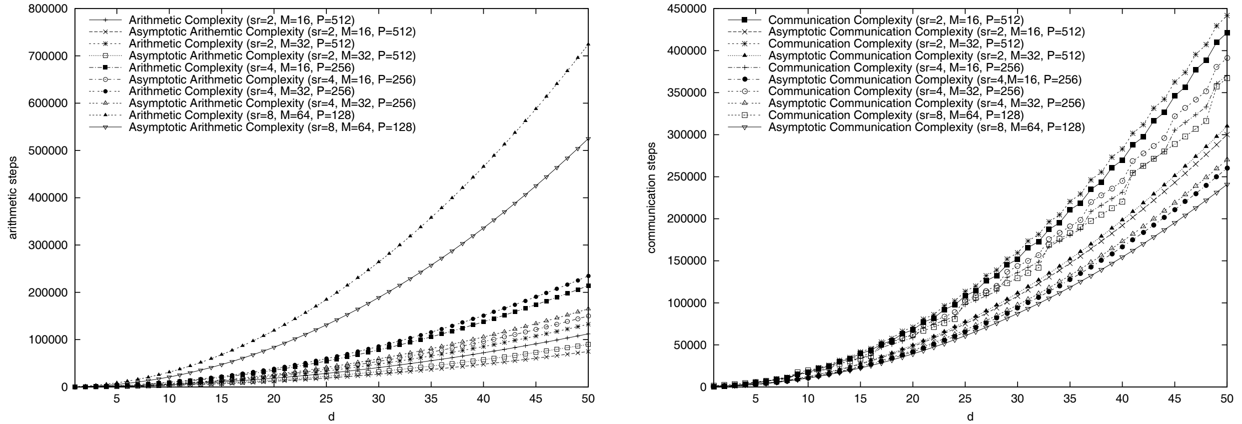

Fig. 5. Number of arithmetic and communication steps of the FSBMME algorithm.

of elements per processor). This range of values of ratio $\frac{N^2}{P^2}$ normally arises on fine-grained machines. Thus, for this kind of parallel machines, it would be better to use the odd-even merge sorting algorithm for implementing RAR operations on the mesh network. This also implies that we can use again the technique of lemma 5.1 for further reduction of the execution time of each RAR operation. Summing up, the use of shift operations as frequently as possible in combination with the efficient realization of RAR operations on the mesh network guarantees fast execution of the general case of HBMME algorithm on this network.

7. Experimental Results

In order to confirm the theoretical results of the paper, we conducted a number of experiments. We focused on fine-grained machines, because most of the analysis in the paper concerns this kind of parallel machines. As it is difficult to find a hypercube-based parallel machine with fine-grained characteristics, we had two options for our experiments; first, to run the experiments on a coarse-grained parallel machine with mesh interconnection network and second, to write a parallel program on a software simulator. The first solution was rejected because the embedding of the hypercube on the mesh and the coarse-grained characteristics of the employed machine would cause large inconsistency between theoretical and experimental results. Thus, we decided to run the experiments on a software simulator. Specifically, we used the Parallaxis-III [54, 55], a simulator designed at University of Stuttgart. In fact, Parallaxis-III is a language for data-parallel programming which is also machine-independent across different SIMD computer systems. However, when the code written in this language is compiled by a conventional C compiler, simulation code is obtained. Another interesting feature of this language is that programmers can easily determine the interconnection network on which communication takes place.

Our programs were written in SIMD control style and assumed one port capability. The all-port capability was not tested because the simulator does not provide such a capability. Neither does the simulator provide absolute timing results in terms of milliseconds. Thus, in our experiments we measured the number of communication and arithmetic steps required overall. We assumed that sending or receiving a word over a hypercube link takes one communication step, whereas one basic arithmetic operation (addition, subtraction, comparison) takes one arithmetic step.

In all our experiments we used frames of size 1024×1024. The number of processors P^2 and the amount of local memory $s_r \times s_r$ at each processor were adjusted in such a way that the equation $s_r \cdot P = 1024$ always holds. It is also important to notice that our theoretical results are not affected by the specific content of the video frame since all the complexities presented in the paper are worst case complexities, that is they remain the same for the same values of the basic parameters (d, M, sr, P). Thus in most of our experiments we used synthetic video frames. Also this kind of video frames help us to debug the simulation code more easily.

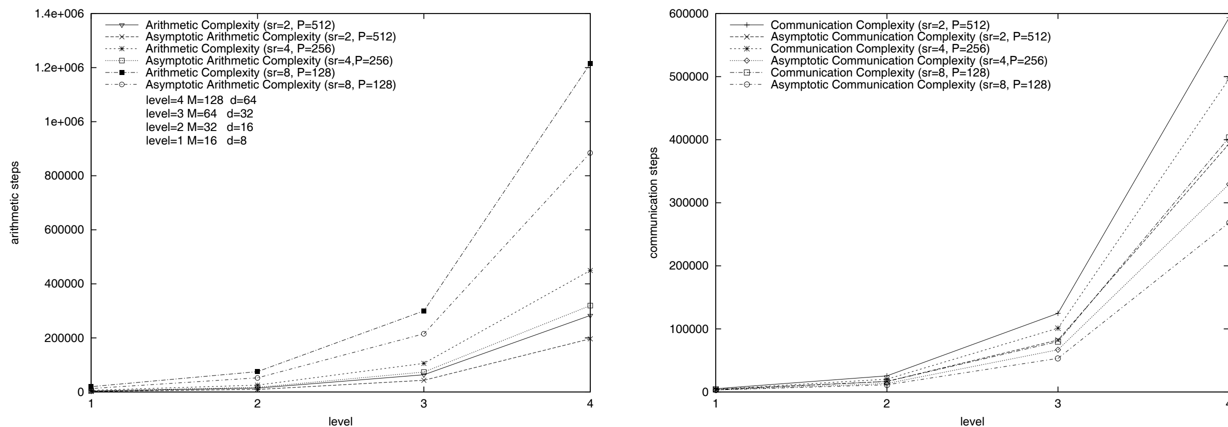

Fig. 6. Number of arithmetic and communication steps for the case $d_i \leq \frac{M_i}{2}$ of the basic scheme of the HBMME algorithm.

The first set of experiments concerns the FSBMME algorithm. Figure 5 shows the variation of the number of arithmetic and communication steps with the parameter d. For comparison reasons, we also included the corresponding asymptotic complexities estimated in the paper. In addition, we considered various combinations of the basic parameters including also the case where the maximum displacement d is not a multiple of s_r. This case complicates the programming as special handling is required along the border of the search window. From this set of experiments, we can easily notice that the arithmetic complexity increases with the parameter s_r since each processor should execute more computations as the amount of local memory $s_r \times s_r$ increases. In contrast, communication complexity is decreasing since fewer shift and Data_Sum operations are now needed overall. The size $P \times P$ of the hypercube is also smaller now due to the assumption that $s_r \cdot P = 1024$. However, larger messages are now transferred at each step and thus eventually the overall communication overhead does not significantly fall with this rise in the amount of the local memory at each processor. This fact is also consistent with the asymptotic results presented in the paper.

Another important point in Fig. 5 is that the actual communication complexity presents some kind of periodic discontinuity, in contrast to the remaining complexity curves. Specifically, discontinuity occurs at the points where the parameter d is a multiple of the parameter s_r. These are exactly the points where there is an increase in the number of communication rounds required overall in the technique of Fig. 2. At any other point, this number does not change and is equal to that of the previous discontinuity point. However, this form of discontinuity corresponds only to lower order terms of the total complexity and thus it does not appear again along the curve of the asymptotic communication complexity of the FSBMME algorithm. Except for this difference, all the curves in Fig. 5 have basically the same shape, namely parabolic. This should be expected since the highest order term in both arithmetic and communication complexity contains the factor d^2. In addition, the fact that the curves of asymptotic and actual complexities are both parabolic indicates that the asymptotic complexities derived in the paper well capture the basic rate of growth of the complexities occurring in practice. However, the apparent difference in the height of the corresponding curves is due to the $O(1)$ constant factors of the higher order terms of the complexities; in contrast to the experimental results, these factors have been omitted in the asymptotic notation.

At the same conclusion we reached with the other sets of experiments too. The second set concerns the basic scheme of HBMME algorithm when $d_i \leq \frac{M_i}{2}$ (Fig. 6). For this scheme we provide analytical results reporting the number of steps at each level of the pyramid along with the corresponding asymptotic complexities. As is expected, higher pyramid levels incur larger communication and arithmetic complexity since larger block sizes and displacements are used at these levels.

We also implemented the complete HBMME algorithm (low-pass filtering and subsampling

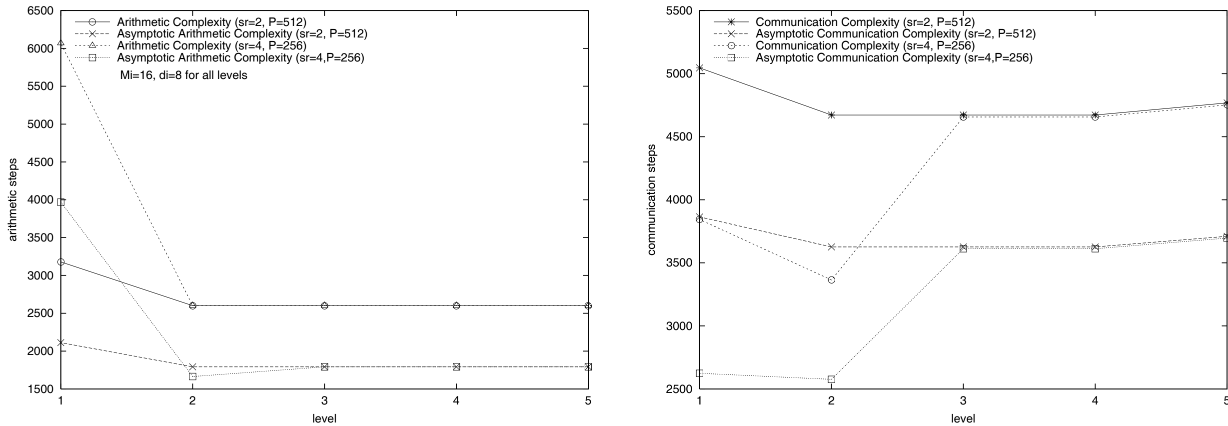

Fig. 7. Number of arithmetic and communication steps of the complete HBMME algorithm $(d_i \leq \frac{M_i}{2})$.

included) for the case $d_i \leq \frac{M_i}{2}$. In particular, we considered two different cases for the values of parameters sr and P (see Fig. 7). We also assumed that parameters M_i and d_i have the same values for all the levels of the pyramid; this assumption is frequently used in practice. One may notice that the communication time at higher levels is comparable with that of lower levels and in the case $(s_r = 4, P = 256)$ is actually higher. This is mainly due to the fact that at higher levels large subhypercubes are used $(M_i \times M_i = 256$ nodes) for shift and data sum operations while at lower levels these subhypercubes are getting smaller and smaller and at the lowest level, level 1, the employed subhypercubes are of size $\frac{M_i}{sr} \times \frac{M_i}{sr} = 16$ nodes. However, the small size of messages at higher levels $(O(1)$ instead of $O(s_r^2)$ at lower levels) compensate well for much of the performance loss caused by the use of larger subhypercubes at these levels. Notice also that the communication complexity of the highest level is somewhat greater than that of the two or three next lower levels. The main reason for this difference is that at the highest level we perform shift operations using the entire hypercube in order to initially transfer pixels of frame Y inside each block (Fig. 4(a)). At all other levels these shifts are carried out inside much smaller subhypercubes and hence this initialization step incurs much less overhead.

With regard to the arithmetic complexity, it can be easily seen that the complexity remains the same for the higher pyramid levels since parameters M_i and d_i have the same values at all these levels and each processor is in charge of only one pixel. However, at lower levels processors are responsible for more than one pixels and thus a proportional increase in the arithmetic complexity should be expected.

As a last set of experiments, we tested the general case of the basic scheme (see Fig. 8). Specifically, we measured the communication and arithmetic steps of the algorithm at a particular pyramid level. Since in the general case there is no initial data collection step, as in the case $d_i \leq \frac{M_i}{2}$, the experimental results do not depend on that particular level but instead remain the same for the same values of the basic parameters. One may also easily notice the increased communication and arithmetic complexity of the general case in comparison to the case $d_i \leq \frac{M_i}{2}$. This is due to the relatively large values of displacements d_i and to the employment of sorting for realizing the necessary data transfers. It is also apparent that communication complexity presents some discontinuity at periodic intervals, namely every M_i units; at these specific points we have an increase in the number of RAR operations required overall whereas at all other points this number remains fixed. This steep rise in the communication complexity at the points where the number of RAR operations increases clearly shows that RAR operations take up a significant portion of the overall communication time. This is also verified by the fact that the number of communication steps in Fig. 8 is getting smaller when the size of blocks $M_i \times M_i$ is increasing and the other basic parameters (d_i, s_r, P) remain constant. As $M_i \times M_i$ blocks are getting larger, the complexity of the sorting steps of each RAR operation is falling since an increasingly larger portion of input data are already sorted in this

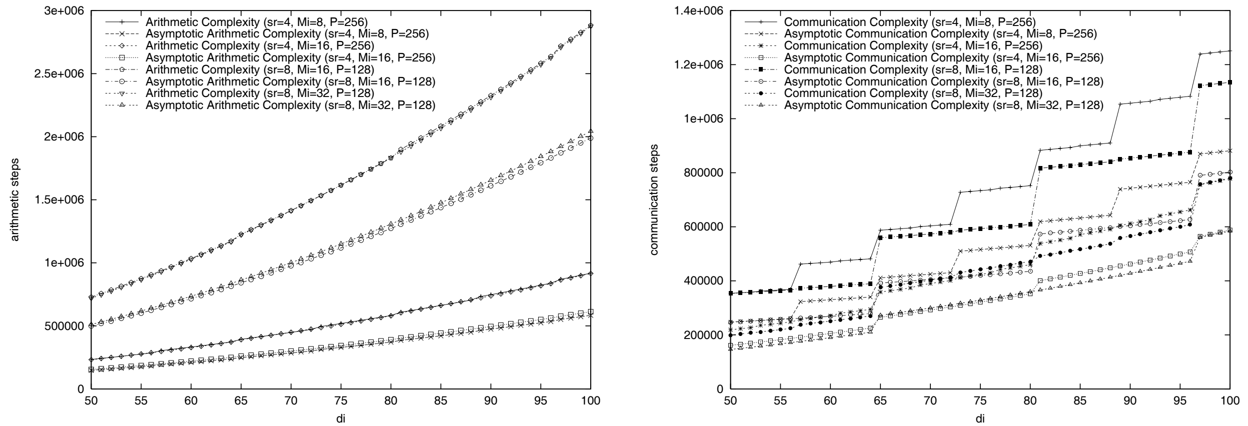

Fig. 8. Number of arithmetic and communication steps for the general case of the basic scheme of the HBMME algorithm.

case (recall lemmas 5.1, 5.2). In addition, fewer communication rounds are now required overall for transferring all the pixels of the search window of Fig. 4(b).

On the other hand, arithmetic complexity is almost the same for different values of parameter M_i since curves which differ only in the value of M_i nearly coincide in the left diagram of Fig 8. The same holds for the corresponding asymptotic results. This very small effect of parameter M_i on the arithmetic complexity can be explained as follows. The decrease in the complexity of RAR operations due to the increase in the value of parameter M_i does not seriously affect the overall arithmetic complexity of the general case because the total arithmetic delay of RAR operations is very small in comparison to that of FSBMME computations. Furthermore, reduction in the number of the communication rounds needed overall does not proportionally decrease the total time spent by the FSBMME arithmetic operations since now the FSBMME algorithm works on a larger search window (of size $3M_i \times 3M_i$) at each step along the route of Fig. 4(b). So, it turns out that the overall complexity of FSBMME computations depends only on the logarithm of M_i, a very slowly growing function.

Finally, it should be mentioned that we did not test the general case for the complete HBMME algorithm. This case is mainly of theoretical interest since arbitrary large block displacements are not considered in practice.

8. Conclusions

We have presented efficient parallel algorithms for full search and hierarchical block matching motion estimation on a hypercube based multiprocessor. Our solutions cover the whole range of modern parallel machines, i.e. fine-grained, medium-grained as well as coarse-grained machines. For fine-grained machines, the rich interconnection structure of the hypercube network ensures efficient execution of complex communication operations such as RAR operations. In addition, our main technique of maximal utilization of simple shift operations in place of complex RAR operations makes our algorithms versatile, easy to execute in other interconnection networks as well. In regard to coarse-grained machines, routing methods such as wormhole and randomized routing greatly facilitates the algorithm design on these machines since most of the details of the employed interconnection network are nearly hidden from the programmer. Thus our algorithms for coarse-grained machines are not specific to the hypercube network but instead they can be applied to any interconnection network which uses wormhole or randomized routing. Finally, a positive aspect of our design is also that it remains valid for the whole range of the values of block matching algorithm parameters.This is very important in such a rapidly evolving research field as video coding where optimal parameters of coding algorithms have not been fixed yet.

Acknowledgements

This work is supported in part by the General Secretariat of Research and Technology of Greece under Project ΠΕΝΕΔ 95 ΕΔ 1623.

References

[1] M. TEKALP, *Digital Video Processing*, Prentice Hall Signal Processing Series, 1995.

[2] B. HASKELL, P. HOWARD, Y. LECUN, A. PURI, J. OSTERMANN, R. CIVANLAR, L. RABINER, L. BOTTOU AND P. HAFFNER, Image and Video Coding - Emerging Standards and Beyond, *IEEE Trans. on Circuits and Systems for Video Technology*, 8(7), 814–837, November 1998.

[3] P. PIRSCH, N. DEMASSIEUX AND W. GEHRKE, VLSI Architectures for Video Compression- A Survey, *Proceedings of the IEEE*, 83(2), 220–246, February 1995.

[4] P. PIRSCH AND H. STOLBERG, VLSI Implementations of Image and Video Multimedia Processing Systems, *IEEE Trans. on Circuits and Systems for Video Technology*, 8(7), 878–891, November 1998.

[5] S. CHENG AND H. HANG, A Comparison of Block-Matching Algorithms Mapped to Systolic-Array Implementation, *IEEE Transactions on Circuits and Systems for Video Technology*, 7(5), 741–757, October 1997.

[6] H. YOSHIMURA AND Y. SUZUKI, Multiprocessor DSPs for Low Bit Rate Video Codec, In P. Pirsch, editor, *VLSI Implementations for Image Communications*, chapter 4, pages 117–148, Elsevier Amsterdam-London-New York-Tokyo, 1993.

[7] S. AKRAMULLAH AND I. AHMAD AND M. LIOU, Performance of Software-Based MPEG-2 Video Encoder on Parallel and Distributed Systems, *IEEE Transactions on Circuits and Systems for Video Technology*, 7(4), 687–695, August 1997.

[8] K. SHEN AND E. DELP, A parallel implementation of a MPEG encoder: Faster than real-time!, In *Proc. of SPIE on Digital Video Compression: Algorithms and Technologies*, pages 407–418, February 1995.

[9] M. TAN AND J. SIEGEL AND H. SIEGEL, Parallel implementation of block-based motion vector estimation for video compression on the MasPar MP-1 and PASM, In *1995 International Conference on Parallel Processing*, 21–24, August 1995.

[10] A. DOWNTON, Speed-up trend analysis for H.261 and model-based image coding algorithms using a parallel-pipeline model, *Signal Processing: Image Communication*, 7, 489–502, 1995.

[11] G. GUPTA AND C. CHAKRABARTI, Architectures for Hierarchical and Other Block Matching Algorithms, *IEEE Transactions on Circuits and Systems for Video Technology*, 5(6), 477–489, December 1995.

[12] T. KOMAREK AND P. PIRSCH, VLSI Architectures for Hierarchical Block Matching Algorithm", In *IFIP Workshop*, pages 168–181, December 1989.

[13] L. DE VOS, VLSI Architectures for the Hierarchical Block Matching Algorithms for HDTV Applications, In *Proc. SPIE Visual Commun. Image Processing, vol. 1360*, pages 398–409, 1990.

[14] Q. WANG AND R. CLARKE, Motion estimation and compensation for image sequence coding, *Signal Processing: Image Communication*, 4, 161–174, 1992.

[15] M. BIERLING, Displacement estimation by hierarchical block-matching, In *Proc. SPIE Visual Commun. and Image Processing vol. 1001*, pages 942–951, 1988.

[16] C. PLAXTON, *Efficient Computation on Sparse Interconnection Networks*, PhD thesis, Department of Computer Science, Stanford University, 1989.

[17] D. CULLER, J. SINGH AND A. GUPTA, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kauffman Publishers, August 1998.

[18] B. MAGGS, A Critical Look at Three of Parallel Computing's Maxims, In *Proceedings of the 1996 International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN '96)*, pages 1–7", June 1996.

[19] M. LIONEL AND K. MCKINLEY, A Survey of Wormhole Routing Techniques in Direct Networks, *IEEE Computer*, 26(2), 62–76, February 1993.

[20] W. DALLY AND C. SEITZ, Deadlock-Free Message Routing in Multiprocessor Interconnection Networks, *IEEE Transactions on Computers*, 36(5), 547–553, May 1987.

[21] C. LEISERSON, Z. ABUHAMDEH, D. DOUGLAS, C. FEYNMAN, M. GANMUKHI, J. HILL, W. DANIEL HILLIS, B. KUZMAUL, M. ST.PIERRE, D. WELLS, M. WONG-CHAN, S. YANG AND R. ZAK, The Network Architecture of the Connection Machine CM-5, *Journal of Parallel and Distributed Computing*, 33(2), 145–158, March 1996.

[22] I. AL-FURIAH, S. ALURU, S. GOIL AND S. RANKA, Practical Algorithms for Selection on Coarse-Grained Parallel Computers, *IEEE Transactions on Parallel and Distributed Systems*, 8(8), 813–824, 1997.

[23] D. BADER, J. JÁJÁ AND RAMA CHELLAPPA, Scalable Data Parallel Algorithms for Texture Synthesis and Compression using Gibbs Random Fields, CS-TR-3123, Department of Electrical Engineering, and Institute for Advanced Computer Studies, University of Maryland, 1993.

[24] D. BADER AND J. JÁJÁ, Practical Parallel Algorithms for Dynamic Data Redistribution, Median Finding, and Selection, In *Proceedings of International Parallel Processing Symposium*, pages 292–301, 1996.

[25] S. RANKA AND S. SAHNI, *Hypercube Algorithms with Applications to Image Processing and Pattern Recognition*, Springer-Verlag, 1990.

[26] T. LEIGHTON, Introduction to Parallel Algorithms and Architectures: Arrays-Trees-Hypercubes, Morgan Kauffman Publishers, San Mateo, California, 1992.

[27] L. VALIANT, A Bridging Model for Parallel Computation, *Communications of the ACM*, 2(8), 103–111, 1990.

[28] S. RANKA, J. WANG AND G. FOX, Static and Runtime Algorithms for All-to-Many Personalized Communication on Permutation Networks, *IEEE Transactions on Parallel and Distributed Systems*, 5(12), 1266–1274, December 1994.

[29] L. VALIANT, General Purpose Parallel Architectures, In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, Elsevier Science Publishers B.V., Amsterdam, The Netherlands, 943–972, 1990.

[30] R. SHANKAR AND S. RANKA, Random Data Accesses on a Coarse-Grained Parallel Machine I: One-to-one mappings, JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING, 44(1)(10), 14–23, July 1997.

[31] R. SHANKAR AND S. RANKA, Random Data Accesses on a Coarse-Grained Parallel Machine II: One-to-Many and Many-to-one Mappings, *Journal of Parallel and Distributed Computing*, 44(1)(10), 24–34, July 1997.

[32] D. HELMA, D. BADER AND J. JÁJÁ, Parallel Algorithms for Personalized Communication and Sorting with an Experimental Study (Extended Abstract), In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 211–222, June 1996.

[33] M. ADLER, J. BYERS AND R. KARP, Scheduling Parallel Communication: The h-relation Problem, In *MFCS*, pages 1–20, 1995.

[34] J. JÁJÁ AND K. RYU, Load balancing and routing on the hypercube and related networks, *Journal of Parallel and Distributed Computing*, 14, 431–435, 1992.

[35] G. BLELLOCH, C. LEISERSON, B. MAGGS, C. PLAXTON, S. SMITH AND M. ZAGHA, A Comparison of Sorting Algorithms for the Connection Machine CM-2, In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 3–16, July 1991.

[36] A. DUSSEAU, D. CULLER, K. SCHAUSER AND R. MARTIN, Fast parallel sorting under LogP: Experience with the CM-5, *IEEE Transactions on Parallel and Distributed Systems*, 7, 791–805, 1996.

[37] W. HIGHTOWER, J. PRINS AND J. REIF, Implementations of Randomized Sorting on Large Parallel Machines, In *Proceedings of 3rd Symposium on Parallel Architectures and Algorithms*, pages 158–167, ACM, 1992.

[38] K. BATCHER, Sorting networks and their applications, In *Proceedings of the AFIPS Spring Joint Computing Conference*, pages 307–314, 1968".

[39] A. WACHSMANN AND R. WANKA, Sorting on a Massively Parallel System Using a Library of Basic Primitives: Modeling and Experimental Results, In *Proceedings of 3rd European Conference in Parallel Processing (Euro-Par)*, pages 399–408, 1997.

[40] D. HILBERT, Uber die steitige Abbildung einer linie auf ein Flachenstuck, *Math. Ann.*, 38, 1891.

[41] L. VOS AND M. STEGHERR, Parameterizable VLSI Architecture for the Full-Search Block-Matching Algorithm, *IEEE Transactions on Circuits and Systems*, 36(10), 1309–1316, October 1989.

[42] T. KOMAREK AND P. PIRSCH, Array Architectures for Block Matching Algorithms, *IEEE Transactions on Circuits and Systems*, 36(10), 1301–1308, October 1989.

[43] C. HSIEH AND T. LIN, VLSI Architecture for Block-Matching Motion Estimation Algorithm, *IEEE Transactions on Circuits and Systems for Video Technology*, 2(2), 169–175, June 1992.

[44] P. VARMAN AND K. DOSHI, Sorting with linear speedup on a pipelined hypercube, *IEEE Transactions on Computers*, C-41(1), 97–105, 1992.

[45] M. VETTERLI AND J. KOVACEVIC, Wavelets and Subband Coding, Prentice Hall PTR, Englewood Cliffs New Jersey 07632, 1995.

[46] V. KUMAR AND V. KRISHNAN, Efficient template matching on SIMD arrays, In *1987 International Conference on Parallel Processing*, pages 765–771, 1987.

[47] S. RANKA AND S. SAHNI, Image Template Matching on MIMD Hypercube Multicomputers, *Journal of Parallel and Distributed Computing*, 10, 1990, 79–84.

[48] P. J. BURT AND E. H. ADELSON, The Laplacian pyramid as a compact image code, *IEEE Trans. on Communications*, COM-31, 532–540, April 1983.

[49] M. KUNDE, Block gossiping on grids and tori: deterministic sorting and routing match the bisection bound, In *Proceedings European Symp. Alg., Lect. Notes Comput Sci. 726*, pages 272–283, Springer-Verlang, 1993.

[50] J. SIBEYN AND M. KAUFMANN, Deterministic 1-k routing on meshes, In *Proceedings 11th Symp. Theoret. Asp. Comput. Sci., Lect. Notes Comput. Sci. 775*, pages 237–248, Springer-Verlag, 1994.

[51] R. DIEKMANN, J. GEHRING, R. LÜLING, B. MONIEN, M. NÜBEL AND R. WANKA, Sorting Large Data Sets on a Massively Parallel System, In *Proc. 6th IEEE Symposium on Parallel and Distributed Processing*, pages 2–9, 1994.

[52] TH. STRICKER, Supporting the hypercube programming model on mesh architectures (A fast sorter for iWarp tori), In *Proceedings of 4th ACM-SPAA*, pages 148–157, 1992.

[53] K. BROCKMANN AND R. WANKA, Efficient Oblivious Parallel Sorting on the MasPar MP-1, In *IEEE Proc. HICSS-30 I*, pages 200–208, 1997.

[54] T. BRÄUNL, S. FEYRER, W. RAPF AND M. REINHARDT, Parallele Bildverarbeitung, Addison-Wesley, Bonn, 1995.

[55] http://www.informatik.uni−stuttgart.de/ipvr/bv/p3.

Received: May 15, 1999
Accepted in revised form: January 21, 2000

Contact address:
Charalampos Konstantopoulos
Computer Engineering and Informatics Department
University of Patras and
Computer Technology Institute
11 Aktaiou & Poulopoulou Str.
GR-118 51 Thiseio, Athens
Greece
phone: (00301) 3416220
Fax: (00301) 3416700
e-mail: konstant@cti.gr

Andreas Svolos
Computer Engineering and Informatics Department
University of Patras and
Computer Technology Institute
11 Aktaiou & Poulopoulou Str.
GR-118 51 Thiseio, Athens
Greece
phone: (00301) 3416220
Fax: (00301) 3416700
e-mail: svolos@cti.gr

Christos Kaklamanis
Computer Technology Institute and
Department of Computer Engineering and Informatics
University of Patras
26500 Rio
Greece
phone: (003061) 997868
Fax: (003061) 993973
e-mail: kakl@cti.gr.

CHARALAMPOS G. KONSTANTOPOULOS received the "Diploma" (five-year first degree) in computer engineering from the Department of Computer Engineering and Informatics of the University of Patras, Greece (1993). He is currently a PhD candidate at the same department. He is also a researcher at the Computer Technology Institute, Patras, Greece. His research interests include parallel algorithms and architectures, efficient algorithms for image processing especially for image and video coding/decoding.

ANDREAS I. SVOLOS was born in Athens, Greece, on January 29, 1971. He received the "Diploma" (five-year first degree) in computer engineering from the University of Patras, Greece, in 1993. He is currently pursuing the Ph.D. degree from the University of Patras, where he is working on his dissertation on efficient algorithms in parallel image processing. His research interests are in the areas of image processing, data structure theory, and parallel and distributed processing. He is a member of IEEE and SPIE.

CHRISTOS KAKLAMANIS received his S.B. (1986) in Computer Engineering from the EECS Department at Massachusetts Institute of Technology, his S.M. (1989) and PhD (1992) in computer science from Harvard University, Cambridge, USA. Currently he is Associate Professor in the Department of Computer Engineering and Informatics at the University of Patras, Greece. He is also a Senior Researcher at the Computer Technology Institute, Patras, Greece. His research interests include parallel algorithms and architectures, distributed computing and communications, theory of computation.