Minimizing Completion Time for Loop Tiling with Computation and Communication Overlapping

Georgios Goumas, Aristidis Sotiropoulos and Nectarios Koziris
National Technical University of Athens
Dept. of Electrical and Computer Engineering
Computing Systems Laboratory
Zografou Campus, Zografou 15773, Greece
e-mail: {goumas, sotirop, nkoziris}@cslab.ece.ntua.gr

Abstract

This paper proposes a new method for the problem of minimizing the execution time of nested for-loops using a tiling transformation. In our approach, we are interested not only in tile size and shape according to the required communication to computation ratio, but also in overall completion time. We select a time hyperplane to execute different tiles much more efficiently by exploiting the inherent overlapping between communication and computation phases among successive, atomic tile executions. We assign tiles to processors according to the tile space boundaries, thus considering the iteration space bounds. Our schedule considerably reduces overall completion time under the assumption that some part from every communication phase can be efficiently overlapped with atomic, pure tile computations. The overall schedule resembles a pipelined datapath where computations are not anymore interleaved with sends and receives to non-local processors. Experimental results in a cluster of Pentiums by using various MPI_send primitives show that the total completion time is significantly reduced.

Index Terms—Loop tiling, communication overlapping, supernodes, hyperplanes, MPI send-receive primitives.

1 Introduction

One of the most difficult areas in the field of parallel computing is the automatic loop parallelization and efficient mapping onto different parallel architectures. The key issue in loop mapping is to mitigate communication overhead by efficiently controlling the computation to communication grain. In distributed memory machines, explicit message passing incurs extra time overhead due to message startup latencies and data transfer delays.

In order to eliminate the communication overhead, Shang [9], Hollander [5] and others, have presented methods for dividing the index space into independent sets of iterations, which are assigned to different processors. However, in many cases, independent partitioning of the index space is not feasible, thus data exchanges between processors impose additional communication delays. When fine grain parallelism is concerned, several methods were proposed to group together neighboring chains of iterations, while preserving the optimal hyperplane schedule [7], [3].

As far as coarse grain parallelism is concerned, researchers are dealing with the problem of alleviating the communication overhead by applying the supernode or tiling transformation. Under this scheme, neighboring iteration points are grouped together to built a larger computation node that can be atomically executed without any intervention. Data exchanges are also grouped and performed with a single message for each neighboring processor, at the end of each atomic supernode execution. Supernode partitioning of the iteration space was proposed by Irigoin and Triolet in [6]. In their paper Ramanujam and Sadayappan [8] showed the equivalence between the problem of finding a set of extreme vectors for a given set of dependence vectors and the problem of finding a tiling transformation H that produce valid, deadlock-free tiles. The use of a communication function that has to be minimized by linear programming approaches was used by Boulet et al. in [2]. They calculated the total communication produced by a tile as a function of its sides and shape and proved that the minimization can be done independently of the tile volume.

Nevertheless, all above approaches ignore the actual iteration space boundaries. Although tile shape is of great importance to communication reduction, the objective should be the overall tiled space completion time. Hodzic and Shang [4] proposed a method to correlate optimal tile size

and shape, based on overall completion time reduction. They consider supernode transformations where data exchanges are between neighboring successive tiles. In this context, the tiled space is considered as a new iteration space with unitary dependencies. They applied the hyperplane transformation to these loop tiles and generated a schedule where the objective is to reduce the overall time by adjusting the tile size and shape appropriately. Each processor executes all tiles along a specific dimension, by interleaving computation and communication phases. All processors first receive data, then compute and finally send result data to neighbors in explicitly distinct phases, according to the hyperplane scheduling vector.

In this paper we propose an alternative method for the problem of scheduling the tiles to processors. Each atomic tile execution involves a communication and a computation phase and this is repeatedly done for all time planes. We are compacting this sequence of communication and computation phases, by overlapping them for the different processors. The proposed method acts like enhancing the performance of a processor's datapath with pipelining, because a processor computes its tile at k time step and concurrently receives data from all neighbors to use them at $k+1$ time step and sends data produced at $k-1$ time step. Since data communications involve some startup latencies, we adjust the computation grain to make room for this overhead and try to overlap with all communication, which can be done in parallel. The time hyperplane that allows for such overlapping is determined by the bounds of the tiled space. Specifically, the dimension with the larger boundary defines the processor mapping, thus all tiles along this dimension are mapped to the same processor. Previous work in the field of UET-UCT scheduling of grid graphs in [1], has shown that this schedule is optimal when the computation to communication ratio is one.

The rest of the paper is organized as follows: Basic terminology used throughout the paper and definitions of loop tiling are introduced in Section 2. In Section 3 we analyze the properties of the non-overlapping optimal time schedule of tiles, whereas in Section 4 we introduce the pipelined approach of an overlapping time schedule.In Section 5 we present the experimental results by implementing both scheduling approaches to various problems using MPI primitives. Finally, we summarize our results and propose future work.

2 Models – Loop Tiling

2.1 The Model of the Algorithms

In this paper we consider algorithms with perfectly nested FOR-loops and constant loop carried data dependencies. That is, our algorithms are of the form:

FOR i_1=l_1 TO u_1 DO
  ...
  FOR i_n=l_n TO u_n DO
    $AS_1(i)$
    ...
    $AS_k(i)$
  ENDFOR
  ...
ENDFOR

where: (1) l_i and u_i are integer-valued constants, meaning that the iteration set is a parallelepiped/multidimensional rectangle, (2) $i = (i_1, ..., i_n)$ and (3) $AS_1, ..., AS_k$ are assignment statements of the form $V_0 = E(V_1, ..., V_l)$, where V_0 is an output variable indexed by i and produced by expression E operating on input variables $V_1, ..., V_l$, also indexed by i.

2.2 Notation

Throughout this paper the following notation is used: N is the set of naturals, Z is the set of integers, n is the number of nested FOR-loops of the algorithm and m is the number of dependence vectors of the algorithm. $J^n \subset Z^n$ is the set of indices: $J^n = \{j(j_1, ..., j_n) | j_i \in Z \wedge l_i \leq j_i \leq u_i, 1 \leq i \leq n\}$. Each point in this n-dimensional integer space is a distinct instantiation of the loop body. A dependence vector is denoted $d_i = (d_{i1}, ..., d_{in}), 1 \leq i \leq m$. The dependence set D of an algorithm A is the set of all dependence vectors of this algorithm: $D = \{d_{i1}, d_{i2}, ..., d_m\}$. Notice that all dependence vectors are considered uniform and constant, i.e. independent of the indices of computations.

2.3 Supernode Transformation

In a supernode transformation the index space J^n is partitioned into identical n-dimensional parallelepiped areas (tiles or supernodes) formed by n independent families of parallel hyperplanes. Supernode transformation is defined by the n-dimensional square matrix H. Each row vector of H is perpendicular to one family of hyperplanes forming the tiles.

Dually, supernode transformation can be defined by n linearly independent vectors, which are the sides of the supernodes. Matrix P contains the side-vectors of a supernode as column vectors. It holds $P = H^{-1}$. Formally supernode transformation is defined as follows:

$$r : Z^n \longrightarrow Z^{2n}, r(j) = \begin{bmatrix} \lfloor Hj \rfloor \\ j - H^{-1} \lfloor Hj \rfloor \end{bmatrix},$$

where $\lfloor Hj \rfloor$ identifies the coordinates of the tile that index point $j(j_1, j_2, \ldots, j_n)$ is mapped to and $j - H^{-1}\lfloor Hj \rfloor$ gives the coordinates of j within that tile relative to the tile origin. Thus the initial n-dimensional index space is transformed to a $2n$-dimensional one, the space of tiles and the space of indexes within tiles. Indexes within tiles have to be sequentially executed, while tiles themselves can be assigned to processors and executed in parallel according to a valid hyperplane schedule as we will see in Sections 3 and 4. The tiled space J^S and the supernode dependence matrix D^S are defined as follows: $J^S = \{j^S | j^S = \lfloor Hj \rfloor, j \in J^n\}$, $D^S = \{d^S | d^S = \lfloor H(j_0 + d) \rfloor, d \in D, j_0 \in J^n | 0 \leq \lfloor Hj_0 \rfloor < 1\}$ where j_0 denotes the index points belonging to the first complete tile starting from the origin of the index space J^n. The tiled space can be also written as $J^S = \{j^S(j_1^S, \ldots, j_n^S) | j_i^S \in Z \wedge l_i^S \leq j_i^S \leq u_i^S, 1 \leq i \leq n\}$. Each point j^S in this n-dimensional integer space J^S is a distinct tile with coordinates $(j_1^S, j_2^S, \ldots, j_n^S)$.

Given an algorithm with dependence matrix D, for a tiling to be legal, it must hold $HD \geq 0$. This ensures that tiles are atomic and that the initial execution order is preserved [6], [8]. In the opposite case any execution order of tiles would result in a deadlock.

In this paper we assume that all dependence vectors are smaller than the tile size, thus they are entirely contained in each supernode's area, which means that $|HD| < 1$ [12] or alternatively that the supernode dependence matrix D^S contains only 0's and 1's. This assumption is quite reasonable since dependence vectors for common problems are relatively small, while tile sizes may result to be orders of magnitude greater in systems with very fast processors. So, for a computation to communication grain to be meaningful tiles are large enough to encapsulate all dependence vectors. In this case every tile needs to exchange data only with its nearest neighbors, one in each dimension of J^n.

2.4 Computation Cost - Communication Cost

The number of index points contained in a supernode expresses the respective computation cost of this supernode (tile), and is calculated by $det(P)$. Thus we have $V_{comp} = det(P)$. The communication cost of a tile is proportional to the number of iteration points that need to send data to neighboring tiles, in other words, the sum of dependence vectors cutting the supernode's boundaries. This can be calculated by the expression:

$$V_{comm}(H) = \frac{1}{|det(H)|} \sum_{i=1}^{n} \sum_{k=1}^{n} \sum_{j=1}^{m} h_{i,k} d_{k,j} \qquad (1)$$

Practically this formula computes and sums all possible $h_i d_j$, which express the contribution to communication of every dependence vector, to every tile boundary surface.

If tiles along the same dimension are mapped to the same processor, dependence vectors cutting the tile's boundary surface in the respective dimension impose no interprocessor communication. In that case, the communication cost is calculated by the expression:

$$V_{comm}(H) = \frac{1}{|det(H)|} \sum_{i \in \{1,\ldots,x-1,x+1,\ldots,n\}, j \in \{1,\ldots,m\}} (H_{-x} D)_{i,j'} \qquad (2)$$

where H_{-x} denotes the H matrix with the column vector vertical to the boundary surface in the dimension of processor mapping extracted. A technique, presented in [2] and [11], calculates the vector H that imposes the minimum amount of communication for a given supenode size.

2.5 Scheduling of Tiles

If $HD \geq 0$, tiles are atomic and preserve the initial execution order. Consequently the tiled index space J^S can be scheduled using similar techniques to the initial index space J^n. In this paper we use linear schedules. Recall ([10]) that a point $j \in J^n$ scheduled according to a linear time schedule Π, will be executed at $t_j = \lfloor \frac{\Pi j + t_0)}{disp\Pi} \rfloor$, where $t_0 = -min \Pi i : i \in J^n$ and $disp\Pi = min \Pi d_i : d_i \in D$. Thus, a tile $j^S \in J^S$ will be executed at $t_{j^S} = \lfloor \frac{\Pi j^S + t_0}{disp\Pi} \rfloor$

2.6 Architecture

We discuss tiling and scheduling techniques for systems using message-passing environments (e.g. MPI) for interprocessor communication. Every processor has instant access to its local memory and in order to communicate with other processors it sends messages through an interconnection network. The time needed for a single computation is denoted by t_c, the communication startup time by t_s (also denoted $t_{startup}$ in this paper), the transmission time per byte by t_t, and the number of bytes per node data by b. There are two important parameters in the underlying architecture affecting the tiling process: The processor's computation speed, which affects the optimal tile size and the system's communication startup time and transmission time which affects the optimal tile shape.

3 Non-overlapping Schedule

In [4], Hodzic and Shang have presented a scheme for scheduling loops that have been transformed through a supernode transformation. The optimal tile size g that minimizes total execution time is determined by the actual parallel architecture parameters i.e. communication to computation grain. Given the tile size, they calculate the optimal tile transformation H that reduces communication cost for

each tile. The rows of matrix H determine the actual tile shape. Relative sizes for tile sides and shape are defined by the dependence vectors of the algorithm, whereas tile volume (size g) is defined by the hardware parameters. Once H is fully determined, it is applied to the original index space. The resulting tiled space J^S is scheduled using a linear time hyperplane Π. All tiles along a certain dimension are mapped to the same processor. Total execution of tiles consists of successive computation phases interleaved with communication ones. A processor receives the data needed to execute a tile at time step i performs the computations and sends to its neighboring processors the boundary data, which will be used for tile calculations in time step $i+1$.

Thus the total execution time is given by:

$$T = P(g)(T_{comp} + T_{comm}), \tag{3}$$

where $T_{comm} = T_{startup} + T_{transmit}$, $P(g)$ is the number of time hyperplanes needed to execute the algorithm, T_{comp} the execution time of a tile ($T_{comp} = gt_c$) and T_{comm} the communication time. T_{comm} can be expressed as the communication startup latency ($T_{startup}$), and a factor expressing the transmission time ($T_{transmit}$). Clearly the total execution time depends on tile size g, since it affects the number of time planes (increase of tile size g leads to reduction of total time planes), the computation cost (gt_c) and the communication volume (V_{comm}).

Let us now consider the implementation of the above schedule in a message passing environment. In this context the execution time of a computation and a communication phase consists of: the transmission time of the data to be received ($T_{transmit}$), the receive startup time $T_{startup}$, the computation time $T_{compute}$, the send startup time $T_{startup}$ and the send transmission time($T_{transmit}$) (Fig. 4).

The overall parallel loop execution consists of atomic computations of tiles interleaved with communication for the transmission of the results to neighboring processors. Since the tiled space J^S has only the unitary dependence vectors (see subsection 2.3), the optimal linear time schedule can be easily proved to be: $\Pi = [1 \ 1 \ldots 1]$. In Fig. 1, the nonoverlapping schedule is shown for a tiled space using six processors. Each time step between successive hyperplanes contains a triplet of receive-compute-send nonoverlapped subphases for each tile. All tiles along the same dimension are mapped to the same processor.

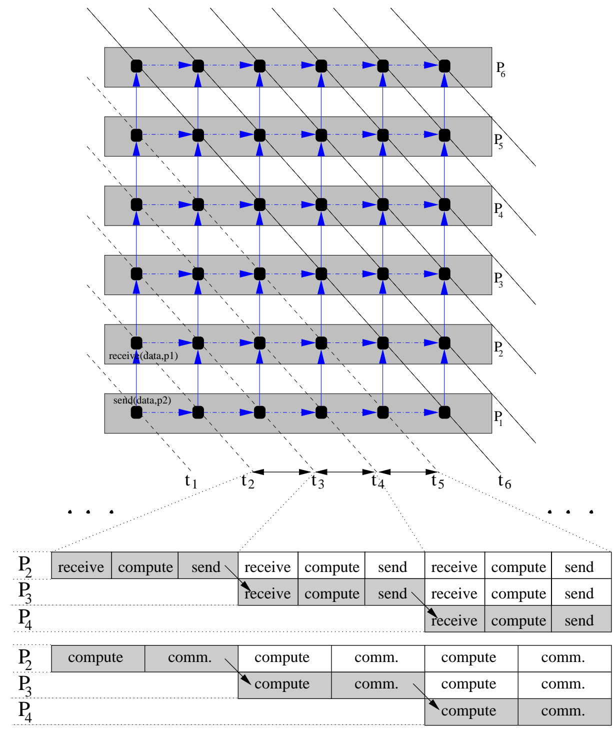


Figure 1. Nonoverlapping time schedule

Example 1

Consider the following algorithm:

for i_1=0 to 9999
  for i_2=0 to 999
    A(i_1,i_2)=A(i_1-1,i_2-1)+A(i_1-1,i_2)+A(i_1,i_2-1)
  endfor
endfor

$J^2 = \{(i_1, i_2) : 0 \le i_1 \le 9999, 0 \le i_2 \le 999\}, D = \{(1,1),(1,0),(0,1)\}$. Suppose that for the target architecture it holds $t_c \approx 1\mu sec$ (see experimental results in Section 5), $t_s = 100t_c$ (reasonable assumption since $t_c \ll t_s$, see Section 5) and $t_t = 0,8t_c/byte$ (i.e. Ethernet 10Mbps). Then according to expression (11) in [4], $g = \frac{ct_s}{t_c}$ which gives the optimal tile size in two dimensions, we have $g = 100$ ($c = 1$, the number of neighboring processors). Communication volume calculated by formula (2) is $V_{comm} = 20$ and each data size is $b = 4bytes$ (float). Consequently $T_{comp} = gt_c = 100t_c$, $T_{comm} = T_{startup} + T_{transmit}$. We have two startup latencies, one for each send and receive performed, thus $T_{startup} = 2 \times t_s = 200t_c$. $T_{transmit} = bV_{comm}t_t = 20 \times 4 \times 0,8t_c$. We optimally choose square tiles with side length 10 ($H = \begin{bmatrix} 0.1 & 0 \\ 0 & 0.1 \end{bmatrix}$). The tiled space will be: $J^S = \{(i_1^S, i_2^S) : 0 \le i_1^S \le 999, 0 \le i_2^S \le 99\}$. Since the maximum value for i_1^S is 999, thus greater than the maximum value for i_2^S, we map along i_1^S. The optimal scheduling vector for this algorithm is $\Pi = (1,1)$ and so the schedule length is $P = \Pi(999,99) - \Pi(0,0) + 1 = 999 + 99 + 1 = 1099$. The total execution time given by formula (3) is $T = 1099(100t_c + 200t_c + 20 \times 4 \times 0,8t_c) = 1099 \times 364t_c = 400036t_c = 0.4\ secs$. In this example, we assume $T_{transmit}$ as the overall transmission time for a complete send-receive pair. We could have splitted it into two pieces as well, without any effects on the results.

4 Overlapping Schedule

The linear schedule presented in the previous section achieves a moderate processor utilization. All processor nodes are concurrently either computing or communicating their results to their neighbors. However, what really imposes such inefficient processor utilization, is the data flow between succesive time steps. Specifically, it seems that computations and respective communication substeps for each time step should be serialized to preserve the correct execution order. Every processor should first receive data, then compute and finally send the results to be used at the next time step by its neighbor (Fig. 3).

It would be ideal if a node was able to receive, compute and send data at the same time. Modern computers have DMA engines and network interfaces (NICs) that can work in parallel with the CPU. This means that some communication work can be overlapped with actual CPU cycles. In addition to this, non blocking message passing primitives mitigate processor waits for the completion of the respective messaging operations. In fact, even some non-blocking work needs the CPU, but most of kernel buffering (TCP/IP stacks) and the transmission phase can be ideally overlapped with other useful computation. A much more thorough look at the correct data flow in the nonoverlapping case, reveals the following interesting property: If we slightly modify the initial linear schedule, then we could overlap some communication time with computations. This means that, in each time step, the processor should send and receive data that is not directly dependent to the data computed at this step. A valid time execution scheme would be for a processor to receive data from all neighbors to use them at $k+1$ time step, send data produced at previous time step $(k-1)$ and compute its results (Fig. 3 and Fig. 2).

In [1] a linear hyperplane for the optimal time scheduling of Unit Execution Times-Unit Communication Times grid task graphs was presented. Grid graphs are like iteration spaces with unitary dependence vectors. Considering UET-UCT model, it is like having communication phases that need equal time to computation ones. In [1], it was also proven that the optimal space schedule for UET-UCT was to assign all points along the maximal dimension to the same processor. The analogy of equal computation to communication times with our case is obvious. If we could achieve a computation to communication grain g, so that the time needed to communicate with the others is equal to the time needed for the CPU to compute, then we could apply this slightly modified linear schedule and the respective space schedule. The optimal time schedule for tile $j^S(j_1^S, j_2^S, \ldots, j_n^S)$ in this case is $2j_1^S + 2j_2^S + \ldots + 2j_{i-1}^S + 2j_{i+1}^S + \ldots + 2j_n^S + j_i^S$, where i is the dimension along which all tiles are mapped to the same processor.

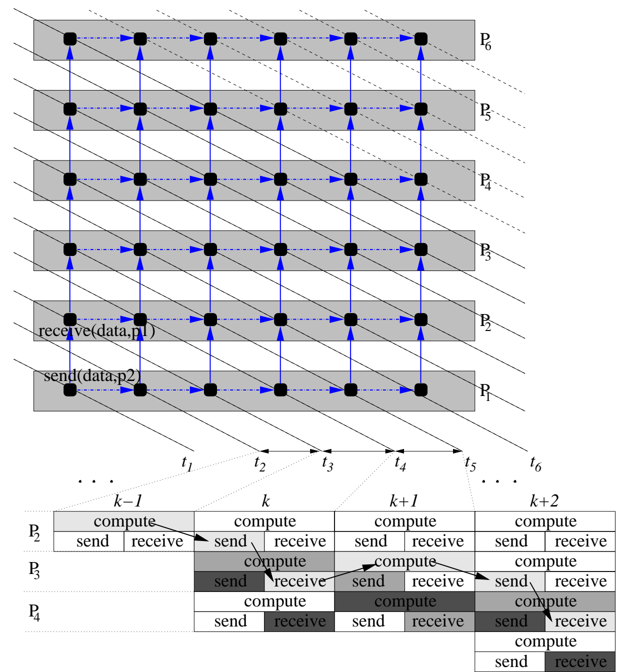


Figure 2. Overlapping time schedule

In Fig. 2 the overlapping scheduling is shown. Consider, for example, processor P_3 at k time step: while it makes the computation for a tile, it concurrently performs the following: sends the results produced during $k-1$ time step and receives data from neighbors, to be used during the computation of the next tile at $k+1$ time step. Note the arcs shown in Fig. 2. They depict the actual flow of data between successive time steps (computes-sends-receives) in pipelined way. The outcome of this schedule is to have successive computations overlapped with communication phases, thus theoretically 100% processor utilization.

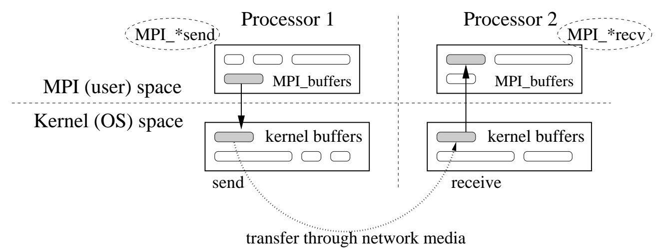


Figure 5. MPI (user) and kernel (OS) space

4.1 Implementation on a Message-Passing Environment

In a message-passing environment like MPI, a processor first initiates all nonblocking receive operations, then per-

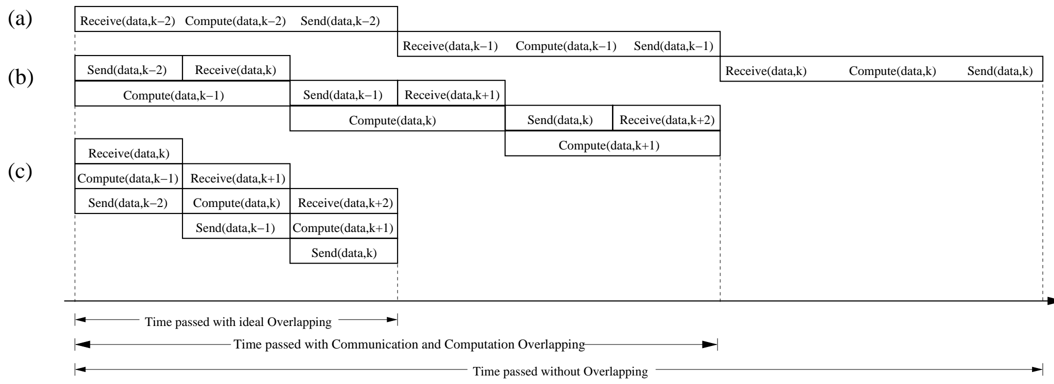


Figure 3. Various levels of computation to communication overlapping

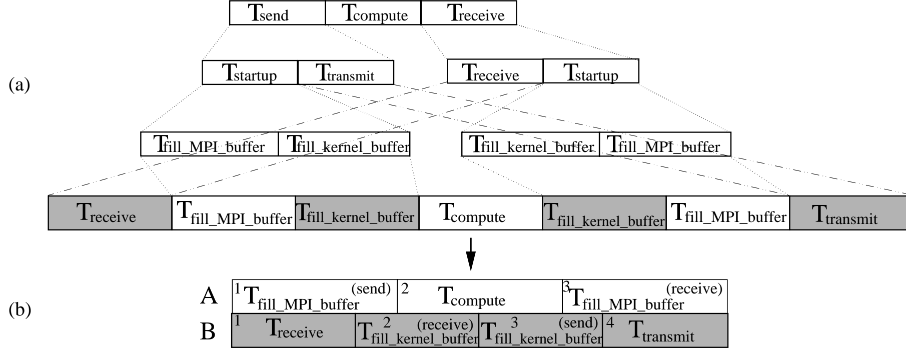


Figure 4. Analysis of a time step

forms the actual atomic tile computation and finally initiates all the nonblocking send operations. While it computes the tile iterations, it may receive data from neighbors and send previously computed data to others as well. Since all primitives are nonblocking, the issue of a send call, for example, requires for processor attention, only to fill the MPI system send buffer. After that, the control returns to the processor which executes the rest of the program, thus the computation of the tile. The same goes for the nonblocking receive. Once such a call is issued, an MPI receive buffer is prepared and the control returns immediately to the program to continue its execution with the next command. As long as the message arrives to the kernel, it is copied from the kernel buffer to the receive buffer and then it is ready to be used (Fig. 5). Actually, since the underlying layers receive the message before the actual issue of the receive call in user program, we put all receives at the end of each send-compute-receive triplet. In a similar way, we perform all sends at the begining of the above triplet, so that all sends are initiated the earliest possible time.

It seems that the initial preparation of both receive and send MPI system buffers is an unavoidable computation time. However, with the aid of a DMA engine, or so, the copy of the data to be sent to the kernel buffer, or the copy of the received data from the kernel buffer to the receive MPI buffer can be overlapped with computations. In addition to this, for long messages, transmission time is also of additional overhead, which can be also overlapped. An ideal scheme is shown in Fig. 3b and the respective analysis for each time step in Fig. 4b. If we ideally assume send and receive overlapping too (e.g. DMA support for multichannel I/O), then Fig. 3c shows the time compaction achieved.

According to the above, we have:

$$T = P(g)\max(A_1 + A_2 + A_3, B_1 + B_2 + B_3 + B_4) \quad (4)$$

since the time hyperplane is so, that either computation or overlapped communication prevails. As shown in Fig. 4,

A_1: time for the MPI system to fill the MPI buffer for send operation ($T^{(send)}_{fill_MPI_buffer}$),

A_2: time for computation ($T_{compute}$),

A_3: time for the MPI system to fill the MPI buffer for receive operation ($T^{(receive)}_{fill_MPI_buffer}$),

B_1: time to receive data (receive side) ($T_{receive}$),

B_2: time for the OS kernel to fill a kernel buffer for receive operation ($T^{(receive)}_{fill_kernel_buffer}$),

B_3: time for the OS kernel to fill a kernel buffer for send operation ($T^{(send)}_{fill_kernel_buffer}$),

B_4: time to transmit data (send side) ($T_{transmit}$).
We assume that the overall transmission is splitted into the sender side transmission time and the receiver side receive time, B_4 and B_1, respectively. From experimental results using MPICH (see Section 5), we derived that all A_i, B_i depend on the size g, thus $A_i(g), B_i(g)$. In the overlapping case, the optimal $P(g)$ is given by $2u_1^S + 2u_2^S + \ldots + 2u_{i-1}^S + 2u_{i+1}^S + \ldots + 2u_n^S + u_i^S$, where $(u_1^S, u_2^S, \ldots, u_n^S)$ are the coordinates of the "last tile" of the tiled space J^S, assuming $(0, 0, \ldots, 0)$ are the coordinates of the first tile and i is the largest dimension (see [1]). We have the following two cases:

1. the non-avoidable initial startup time for all sends and receives plus the net computation time are bigger than the rest communication and transmission time:

   If $A_1 + A_2 + A_3 > B_1 + B_2 + B_3 + B_4$ then (4) becomes:

   $$T(g) = P(g)(A_1 + A_2 + A_3) \tag{5}$$

   From LEMMA 1 in [4], it holds $P(g) = P_0 g^{-1/n}$, thus we have $T(g) = P_0 g^{-1/n}(A_1 + A_3 + gt_c) \Rightarrow T(g) = P_0(A_1(g) + A_3(g))g^{-1/n} + P_0 t_c g^{\frac{n-1}{n}}$. We obtain the optimal overall time when $T'(g) = 0$. In Section 5 we use the experimental values for g, since there isn't any analytical formula for $A_1(g)$, $A_3(g)$.

2. the non-avoidable initial startup time for all sends and receives plus the net computation time are less than than the rest communication and transmission time:

   If $A_1 + A_2 + A_3 > B_1 + B_2 + B_3 + B_4$ then (4) becomes:

   $$T(g) = P(g)(B_1 + B_2 + B_3 + B_4)$$

   As in [4], transmission time is $B_1 = B_4 = bt_t V_0 g^{\frac{n-1}{n}}$ and thus $T(g) = P_0 g^{-1/n}(B_2(g) + B_3(g) + 2bt_t V_0 g^{\frac{n-1}{n}}) \Rightarrow T(g) = P_0(B_2 + B_3)g^{-1/n} + 2P_0 bt_t V_0 g^{\frac{n-2}{n}}$. We obtain the optimal overall time when $T'(g) = 0$.

Example 2

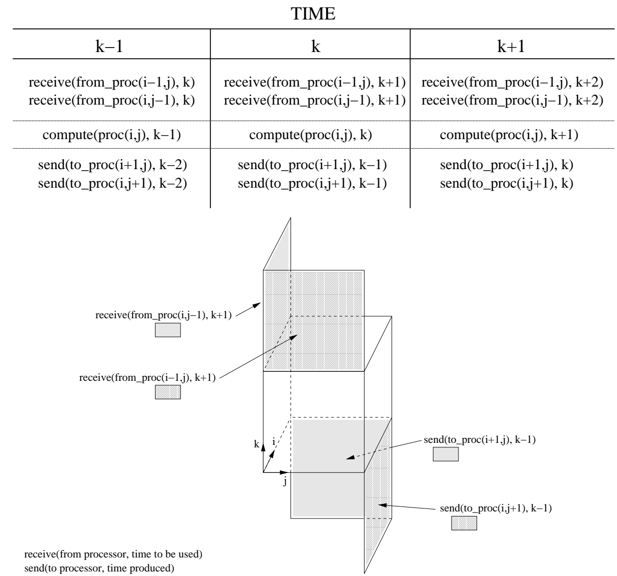
The pipelined data flow in the overlapping case works as follows (Fig. 2): Data computed from processor P_2 at $k-1$ time step, are send to P_3 during k time step, received by P_3 in the same k time step, and then computed during $k+1$ step, from the same processor. Next, at $k+2$ time step, processor P_3 sends the previously computed results to P_4, to be received until the end of the $k+2$ step.

Example 3

Consider the algorithm of Section 3 where now the optimal scheduling vector is $(1, 2)$. As we will see in Section 5, a realistic assumption can be that of $T_{fill_MPI_buffer} = \frac{1}{2}t_s$, and $T_{fill_MPI_buffer} + T_{fill_kernel_buffer} = T_{startup}$ we have one send and one receive in each time step along i_2 dimension. The schedule length now is $P = \Pi(999, 99) - \Pi(0, 0) + 1 = 999 + 2 \times 99 + 1 = 1198$. Since $B_1 + B_2 + B_3 + B_4 = 50t_c + 50t_c + 20 \times 0.4 \times 0.8t_c < A_1 + A_2 + A_3 = 50t_c + 50t_c + 100t_c$, the total execution time is now $1198(25t_c + 25t_c + 100t_c) = 179700t_c = 0.24\ secs$, much less than the nonoverlapping case ($0.4\ sec$). If we adjust g so that $A_1 + A_2 + A_3 = B_1 + B_2 + B_3 + B_4$, thus complete overlapping, we could achieve a much better result. It is obvious that a greater g would increase the ammount of data needed to be communicated and reduce the number of hyperplanes $P(g)$, while also increasing gt_c. On the other hand, a smaller g would decrease the ammount of data needed to be communicated and increase the number of hyperplanes $P(g)$, while also decreasing gt_c. In Section 5 we experimentally tune tile size g to reach optimal result for the overall completion time.

5 Experimental results

We run our experiments on cluster with 16 identical 500MHz Pentium nodes. Each node has 128M of RAM and 10G hard drive and runs Linux with 2.2.14 kernel version. We used MPI (MPICH) to run the experiments over the FastEthernet.



TIME

k−1	k	k+1
receive(from_proc(i−1,j), k) receive(from_proc(i,j−1), k)	receive(from_proc(i−1,j), k+1) receive(from_proc(i,j−1), k+1)	receive(from_proc(i−1,j), k+2) receive(from_proc(i,j−1), k+2)
compute(proc(i,j), k−1)	compute(proc(i,j), k)	compute(proc(i,j), k+1)
send(to_proc(i+1,j), k−2) send(to_proc(i,j+1), k−2)	send(to_proc(i+1,j), k−1) send(to_proc(i,j+1), k−1)	send(to_proc(i+1,j), k) send(to_proc(i,j+1), k)

Figure 6. Timing and extra buffering for the overlapping case

Our test application is a simple loop with only one assignment statement i.e. $A(i, j, k) = \sqrt{A(i-1, j, k)} + \sqrt{A(i, j-1, k)} + \sqrt{A(i, j, k-1)}$. We used square roots and floats to increase t_c at a reasonable value. The optimal tiling is in rectangular tile shapes. Each tile is a cube

with ij, ik and kj sides. We selected k dimension to be the largest one, so all tiles along k-axis are mapped to the same processor $P_i, i = (0, \ldots, 15)$. During each time step, every processor in the ij plane with coordinates (i, j) receives from neighboring processors $(i-1, j)$ and $(i, j-1)$, computes and sends to processors $(i+1, j)$,$(i, j+1)$ In order to achieve overlapping of computation and communication, we need extra space, besides the tile space, on each node in order to buffer the surfaces that are received or being sent to every neighboring node, while changing the data during the computation of (i, j, k) tile (Fig. 6).

The most common message-passing primitives are called blocking primitives (synchronous primitives). When a process calls a send routine, it specifies a buffer and a destination to send it to. While the message is being sent, the sending process is blocked (i.e. suspended). The instruction following the call to send is not executed until the message has been completely sent, as shown in Fig. 7. Similarly, a call to receive does not return control until a message has actually been received and put in the message buffer, pointed to by the parameter of the receive call.

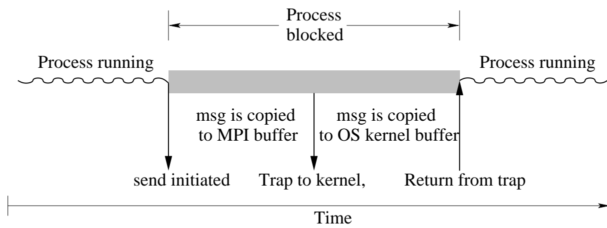


Figure 7. Blocking send primitive

An alternative to blocking primitives are nonblocking primitives (sometimes called asynchronous primitives). If send is nonblocking, it returns control to the caller immediately, before the message is sent. The advantage of this scheme (Fig. 8) is that the sending process can continue computing in parallel with the message transmission, instead of having the CPU go idle

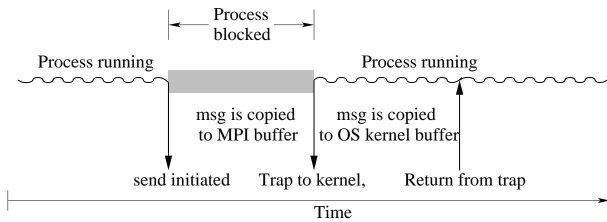


Figure 8. Nonblocking send primitive

The send part of the receive-compute-send triplet (Fig. 4) is divided to the startup part and the transmission part. The startup part itself can be divided to the writing of MPI buffer on behalf of the MPI_Send command and the

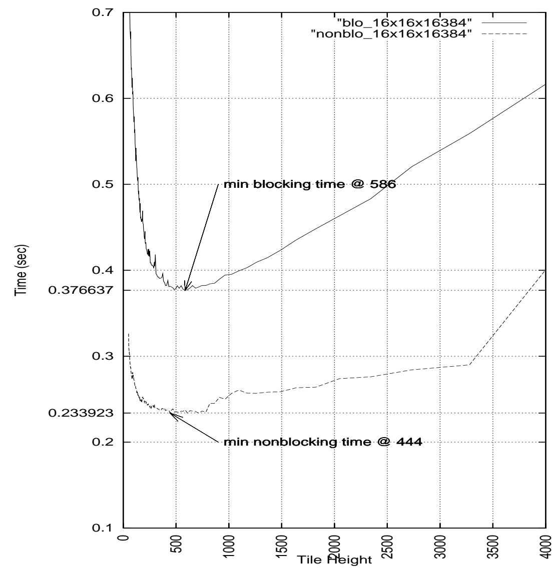


Figure 9. Results for 16x16x16384 space

reading of the MPI buffer on behalf of the kernel. That means that the white part of the unfolded triplet can be overlaped. So we increase the $T_{compute}$ part and try to fit the $T_{receive}, T^{(receive)}_{fill_kernel_buffer}, T^{(send)}_{fill_kernel_buffer}$ and $T_{transmit}$ parts under the $T^{(send)}_{fill_MPI_buffer}, T_{compute}$ and $T^{(receive)}_{fill_MPI_buffer}$ parts.

The pseudocode for the blocking case is:

```
for i = 0 to max_i_tile-1
 for j = 0 to max_j_tile-1
   ProcB(i,j)
```

where: `ProcB(i, j)` is

```
for k = 0 to max_k_tile-1
{
 MPI_Recv(T(i-1, j), results(T(i-1, j), k))
 MPI_Recv(T(i, j-1), results(T(i, j-1), k))
 compute();
 MPI_Send(T(i+1, j), results(T(i, j), k))
 MPI_Send(T(i, j+1), results(T(i, j), k))
}
```

While the pseudocode for the nonblocking case is:

```
for i = 0 to max_i_tile-1
 for j = 0 to max_j_tile-1
   ProcNB(i,j)
```

where: `ProcNB(i,j)` is

```
for k = 0 to max_k_tile-1
{
 MPI_Isend(T(i+1, j), results(T(i, j), k-1), &s1)
 MPI_Isend(T(i, j+1), results(T(i, j), k-1), &s2)
 MPI_Irecv(T(i-1, j), results(T(i-1, j), k+1), &r1)
 MPI_Irecv(T(i, j-1), results(T(i, j-1), k+1), &r2)
 compute();
 MPI_Wait(s1);
```

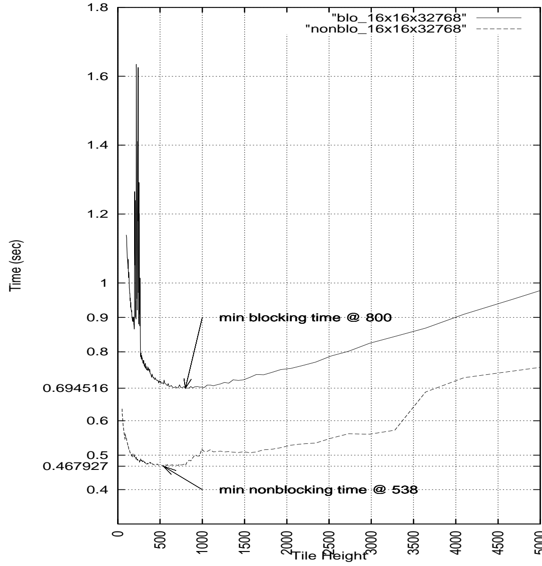


Figure 10. Results for 16x16x32768 space

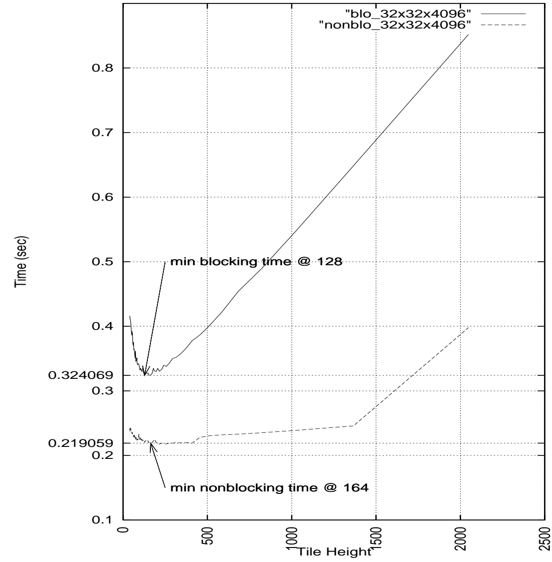


Figure 11. Results for 32x32x4096 space

```
 MPI_Wait(s2);
 MPI_Wait(r1);
 MPI_Wait(r2);
}
```

The experiments were concerning a $16 \times 16 \times 16384$ space, a $16 \times 16 \times 32768$ and a $32 \times 32 \times 4096$ space, where $A \times B \times C$ represent the boundaries of i, j, k axes respectively. The tiled space will have k as larger dimension, so mapping all tiles to the same processor is performed along the k-axis. For every of the above three problems, we were using all 16 processors, that is 4 processors for each i, j dimension. This means that all tiles, for example in the first case, were having sizes of $4 \times 4 \times V$ where V was a variable (V is denoted as tile height, since it is the size of tile along axis k). For all possible values of V, ranging from 4 to $\frac{32768}{4}$, we ran both complete non overlapping and overlapping MPI programs, and calculated the size of $V_{optimal}$ for which the minimum completion time ($t_{optimal}$) is achieved. Figures 9, 10 and 11 summarize our results.

We compare the experimental results with the theoretical ones calculated from formula (5). From the analysis of Section 4, the optimal grain for the overlapping case depends on the t_c for each iteration of the initial J^n and $T_{startup}$. To calculate the t_c we ran 1000 iterations of the loop in a single node, and calculated the overall time. By dividing it to the number of iterations, we calculated $t_c = 0.441\,\mu sec$. Actualy, from formula (5), we also need the $T_{fill_MPI_buffer}$ for both MPI non_blocking sends and receives. For the calculation of $T_{fill_MPI_buffer}$ we wrote a simple program with 10.000 successive non_blocking sends from the one node to another using immediate (MPI_Ireceive), so that the receiver posts the receives without causing any delays. Each time, the size of the message sent was equal to the data transmitted from one tile to another, for tile sizes 4x4x$V_{optimal}$. We used (MPI_Isend) for the send primitive. This was done to simulate, as close as possible, the behavior of the $T_{fill_MPI_buffer}$ in the complete program. As far as the tile size is concerned, we use the optimal size we obtained from the experiments. At last, we need the number of hyperplanes for the particular tile size. This is calculated by the expression: $P(g) = 2 \times i_{max} + 2 \times j_{max} + \frac{k_{max}}{V_{optimal}}$.

For the first experiment (Fig 12i) the iteration space J^n is $\{(i,j,k)|0 \leq i < 16,\ 0 \leq j < 16,\ 0 \leq k < 16384\}$, the respective tiled space J^S is $\{i^S, j^S, k^S|\ 0 \leq i^S < 4,\ 0 \leq j^S < 4,\ 0 \leq k^S < V\}$, where V ranges from 4 to $\frac{16384}{4}$. From the experimental results (see Fig. 9), the optimal overall completion time is achieved for $g_{experimental} = 4 \times 4 \times 444$, thus $V_{experimental} = 444$. For the first experiment, we use a packet size of 7104 bytes for ij or ik tile sides when $V = 444$, to calculate $T_{fill_MPI_Buffer} = 0.627\,msec$, and t_c is equal to $0.441\,\mu sec$. The respective optimal completion time was found to be $0.233923\,sec$. On the other hand, if we calculate the number of hyperplanes $P(g)$ corresponding to $g_{experimental} = 4 \times 4 \times 444$, it is $P(g_{experimental}) = 2 \times 4 + 2 \times 4 + \frac{16384}{444} \approx 53$. We assume that $T_{fill_MPI_buffer}$ for MPI_Irecv is the same as for MPI_Isend. If we apply the experimental values for the parameters $g, t_c, T_{fill_MPI_buffer}$ to (5), the theoretical overall completion time for the overlapping case is $P(g_{experimental}) \times (4 \times 0.617 + g_{experimental} \times 0.441 \times 10^{-3}) msec = 0.24 sec$, which differs to the overall experimental measured completion time 2.5%. Notice in Fig 9 that the optimal completion time for the non-overlapping case is $0.376637 sec$. Thus the overlapping technique offers

a 30% improvement in total execution time. The results for the other two experiments are shown in Fig 12 ii, iii.

	i	ii	iii
index set size $(i \times j \times k)$	$16 \times 16 \times 16384$	$16 \times 16 \times 32768$	$32 \times 32 \times 4096$
$V_{optimal}$	444	538	164
$g_{optimal}$	7104	8608	10996
$t_{optimal}$ overlapping experimental	0.233923 sec	0.467929 sec	0.219059 sec
$T_{fill_MPI_buf}$	0.627 msec	0.745 msec	0.37 msec
$P(g)$	53	76	41
$t_{optimal}$ overlapping theoretical	0.24 sec	0.507 sec	0.25 sec
difference experimental vs. theoretical	2.5%	7%	12%
$t_{optimal}$ non-overlapping experimental	0.376637 sec	0.694516 sec	0.324069 sec
improvement overlapping vs. non-overlapping	38%	33%	32%

Figure 12. Experimental Results

6 Conclusions – Future Work

In this paper we proposed a novel approach for the problem of minimizing the completion time for loop tiles by overlapping computation and communication for each tile execution. Experimental results have shown that the theoretically calculated overall time, following the optimal hyperplane transformation, is very similar to the experimental results. What remains open is an analytical expression for $A_i(g)$ and $B_i(g)$ so that we can calculate $g_{optimal}$ from the parallel architectures internal characteristics (t_c, t_t) and MPI internal communication latencies. Furthermore, modern hardware capabilities (DMA engines, parallel I/O, NICs) are not fully exploited by the overlying software layers (OS drivers). We plan to use a DMA enabled driver with SCI to concurrently send and receive while the CPU computes.

7 Acknowledgements

This work was partially funded by the Ministry of Development, General Secretariat for Research and Technology, project PENED 99ED308.

References

[1] T. Andronikos, N. Koziris, G. Papakonstantinou, P. Tsanakas, *Optimal Scheduling for UET/UET-UCT Generalized N-Dimensional Grid Task Graphs,* Journal of Parallel and Distributed Computing, vol. 57, no. 2, pp. 140–165, May 1999.

[2] P. Boulet, A. Darte, T. Risset, Y. Robert, *(Pen)-ultimate tiling?,* INTEGRATION, The VLSI Jounal, volume 17, pp. 33-51, 1994.

[3] I. Drossitis, G. Goumas, N. Koziris, G. Papakonstantinou, P. Tsanakas, *Evaluation of Loop Grouping Methods based on Orthogonal Projection Spaces,* in Proceedings of the 2000 Int'l Conference on Parallel Processing, pp. 469–476, Toronto, Canada, Aug. 2000.

[4] E. Hodzic, W. Shang, *On Supernode Transformation with Minimized Total Running Time,* IEEE Trans. on Parallel and Distributed Systems, vol. 9, no. 5, pp. 417–428, May 1998.

[5] E. H. Hollander, *Partitioning and Labeling Loops by Unimodular Transformations,* IEEE Trans. on Parallel and Distributed Systems, vol. 3, no. 4, pp. 465–476, Jul. 1992.

[6] F. Irigoin, R. Triolet, *Supernode Partitioning,* Proc. 15th Ann. ACM SIGACT-SIGPLAN Symp. Principles of Programming Languages, pp. 319–329, San Diego, California, Jan 1988.

[7] P. Tsanakas, N. Koziris, G. Papakonstantinou, *Chain Grouping: A Method for Partitioning Loops onto Mesh-Connected Processor Arrays,* IEEE Trans. on Parallel and Distributed Systems vol. 57, no. 2, pp. 941–955, Sep. 2000.

[8] J. Ramanujam, P. Sadayappan, *Tiling Multidimensional Iteration Spaces for Multicomputers,* Journal of Parallel and Distributed Computing, vol. 16, pp.108–120, 1992.

[9] W. Shang, J.A.B. Fortes, *Independent Partitioning of Algorithms with Uniform Dependencies,* IEEE Trans. Comput., vol. 41, no. 2, pp. 190–206, Feb. 1992.

[10] W. Shang, J.A.B. Fortes, *Time Optimal Linear Schedules for Algorithms with Uniform Dependencies,* IEEE Trans. Comput., vol. 40, no. 6, pp. 723–742, June 1991.

[11] J. Xue, *Communication-Minimal Tiling of Uniform Dependence Loops,* Journal of Parallel and Distributed Computing, vol. 42, no.1, pp. 42–59, 1997.

[12] J. Xue, *On Tiling as a Loop Transformation,* Parallel Processing Letters, vol.7, no.4, pp. 409–424, 1997.