Minimizing Completion Time for Loop Tiling with Computation and Communication Overlapping

N. Koziris, A. Sotiropoulos and G. Goumas

National Technical University of Athens,
Dept. of Electrical and Computer Engineering,
Computing Systems Laboratory,
Zografou Campus, 15773, Greece

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. We survey the application of our schedule to modern communication architectures. We performed two sets of experimental results, one using MPI primitives over FastEthernet and one using the SISCI API over an SCI network. In both cases, the total completion time is significantly reduced.

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 [1], Hollander [2] 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 [3,4].

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 [5]. In their paper Ramanujam and Sadayappan [6] 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 [7]. 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 [8] 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 [9], has shown that this schedule is optimal when the computation to communication ratio is one.

We investigate the application of our scheduling method to modern communication architectures. At first, a general message passing environment over a packet based communication layer (e.g. MPI over Ethernet) is considered. In this environment the overlapping schedule is achieved by appropriately using nonblocking communication primitives and DMA. Second, we consider a more advanced communcation technology like SCI. SCI NICs support shared memory programming either through PIO (Programmed-IO) messaging or through DMA. We are using their kernel-level DMA support for messaging. Invoking kernel system calls, causes extra CPU cycles overhead. However, we can avoid extra copying from user space to kernel space (physical memory) when using DMA. We allocate user level pages which correspond to physical pre-reserved memory regions, for DMA communications. If user level (virtual) memory, reserved for data communication between neighboring tiles, is directly mapped to hardware (physical), we have no extra copying [10]. The zero copy transfer mechanism requires that both sender and receiver memory areas must be pinned down to physical memory during transfer, since NICs and most DMA engines access only physical memory addresses. Zero-copy is considerably reducing the initial latency for data transfers (for the effects of a zero-copy implementation of a common communication layer see [11]). In addition to this, SCI shared memory communication mode (either PIO or DMA based) is dramatically reducing communication time, compared to other software packet-based communication layers (i.e. Ethernet ones like MPI). In fact, when using SCI as communication medium, processes only declare their communication end-points (reserve memory areas through kernel syscalls) and then all data exchanges can be just implemented as very fast remote writes ("send") and local reads ("receive"). No other software protocol layering is needed (packetization etc.) which increases initial latencies. SCI packetization and flow control is completely in hardware.

We propose the use of DMA to remote write (send) data to neighboring nodes, while the CPU is computing each tile. Every node reserves special (pinned down) regions of memory as message buffers, exported to the SCI global address space. These memory areas are used to store data needed to be sent to neighbors. All nodes that have to "send" data, import these regions and perform SCI remote writes, where data are being written (transferred) with the aid of DMA. We use the SISCI API [12–14] for all system calls related to SCI. The CPU is doing all computations, calculating all results per tile. It stores the results directly into reserved physical memory areas where the DMA can directly read them. DMA reads the results (locally) and sends them to the neighboring nodes by performing remote writes to the respective exported-areas. Each node is importing the exported segments so that it can perform remote writes to them. The CPU of the neighboring node reads from its local (exported) segment the results that the DMA post call of the previous node has written to. Each node performs a enqueue-post DMA sequence to prepare the DMA remote writing, then concurrently issues the `compute()` call and finally, waits for DMA to finish the transfer. Synchronization between neighboring nodes is done through SCI interrupts. Under the above implemented scheme, we avoid most of communication overhead and allow for actual computation to communication overlapping. All experimental results show that when the overlapping schedule is applied, the overall completion time is considerably reduced, under the term of controlling the computation to communication grain.

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 discuss the application of our scheduling method to specific communication architectures.

In section 6 we present experimental results by simulating both scheduling approaches to various problems using MPI primitives and by implementing them on SCI. 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. If A is a $\kappa \times \lambda$ matrix, we denote a_{ij} the matrix element in the i-th row and j-th column, thus $A = \{a_{ij}\}$, where $1 \leq i \leq \kappa$ and $1 \leq j \leq \lambda$. A dependence vector is denoted $d_j = (d_{1j}, \ldots, d_{nj}), 1 \leq j \leq m$. The dependence set DS of an algorithm A is the set of all dependence vectors of this algorithm: $DS = \{d_1, d_2, ..., d_m\}$. Similarly, the dependence matrix D is defined as $D = \{d_{ij}\}, 1 \leq i \leq n, 1 \leq j \leq m$, where d_{ij} denotes the i-th coordinate of the j-th dependence vector. 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 $H = \{h_{ij}\}$ [6]. 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:

$$r : Z^n \longrightarrow Z^{2n}, r(j) = \left[\begin{array}{c} \lfloor Hj \rfloor \\ j - H^{-1}\lfloor Hj \rfloor \end{array} \right],$$

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 [5], [6]. 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 $|HD| < 1$ [15] 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:

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

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:

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

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 [7] and [16], 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 ([17]) 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 $t_{j^S} = \lfloor \frac{\Pi j^S + t_0}{disp\Pi} \rfloor$

3. Non-overlapping Schedule

In [8], 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:

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

where $t_{comm} = t_{startup} + t_{transmit}$ is the communication time, $P(g)$ is the number of time hyperplanes needed to execute the algorithm, t_{comp} the execution time of a tile ($t_{comp} = gt_c$), where t_c is one J^n iteration's computation 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($t_{transmit}$).

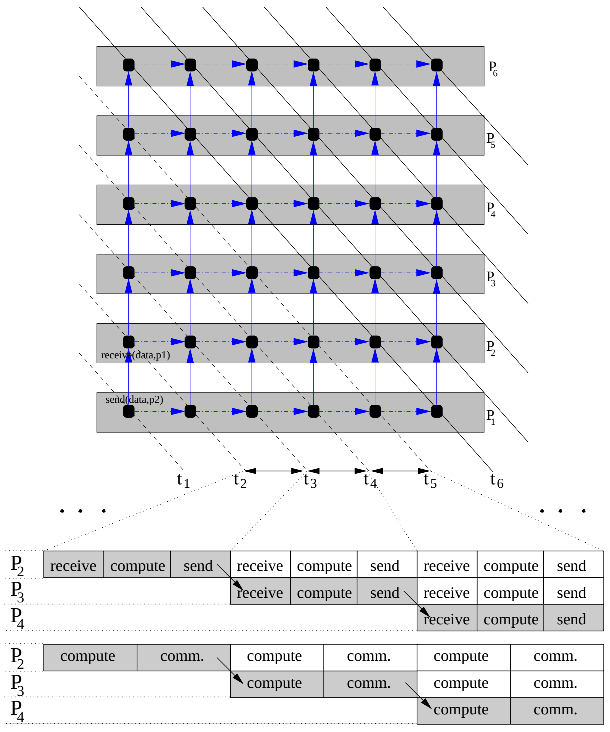

Figure 1. Non-overlapping time schedule

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: $\Pi = [1 \;\; 1 \ldots 1]$. For example, tile $j^S(j_1^S, j_2^S, \ldots, j_n^S)$ is executed at $\Pi j^S - \Pi j_{init}^S$ time step, where j_{init}^S is the first executed tile. Given that $j_{init}^S = \overrightarrow{0} \Rightarrow \Pi j_{init}^S = 0$, then j^S is executed at $j_1^S + j_2^S + \ldots + j_n^S$ time step. The number of time hyperplanes $P(g)$ in (3) is equal to $\Pi j_{final}^S - \Pi j_{init}^S + 1$, where j_{final}^S is the last executed tile.

In Fig. 1, the non overlapping schedule is shown for a tiled space using six processors. Each time step between successive hyperplanes contains a triplet of receive-compute-send non-overlapped subphases for each tile. All tiles along the same dimension are mapped to the same processor.

This quite straightforward model of execution results in very good execution times, since it exploits all inherent parallelism at the tile level. However, an important drawback of this execution model is that each processor has to wait for essential data before starting the computation of a certain tile, and wait for the transmission of the results to its neighbors, thus resulting in significant idle processor time.

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 successive 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.

It would be ideal if a node was able to receive, compute and send data at the same time. Modern network interfaces (NICs) are equipped with DMA engines 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, nonblocking message passing primitives mitigate processor waits for the completion of the respective messaging operations. In fact, even some nonblocking work needs the CPU initially, but most of it, including the transmission phase, can be ideally overlapped with other useful computation. A much more thorough look at the correct data flow in the non-overlapping 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 comprises the following actions during k time step: A processor receives neighboring data which will be used at $k+1$ time step, sends data produced at $k-1$ time step and computes data received at $k-1$ time step (Fig. 2).

In [9] 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 [9], 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. In this case, the optimal time schedule is $\Pi = [2 \dots 2 \underbrace{1}_{\text{pos } i} 2 \dots 2]$, where i denotes the maximum dimension. For example, tile $j^S(j_1^S, j_2^S, \dots, j_n^S)$ is executed at $\Pi' j^S - \Pi' j_{init}^S$ time step, where j_{init}^S is the first executed tile. Given that $j_{init}^S = \overrightarrow{0} \Rightarrow \Pi' j_{init}^S = 0$, then j^S is executed at $2j_1^S + 2j_2^S + \dots + 2j_{i-1}^S + 2j_{i+1}^S + \dots + 2j_n^S + j_i^S$ time step.

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

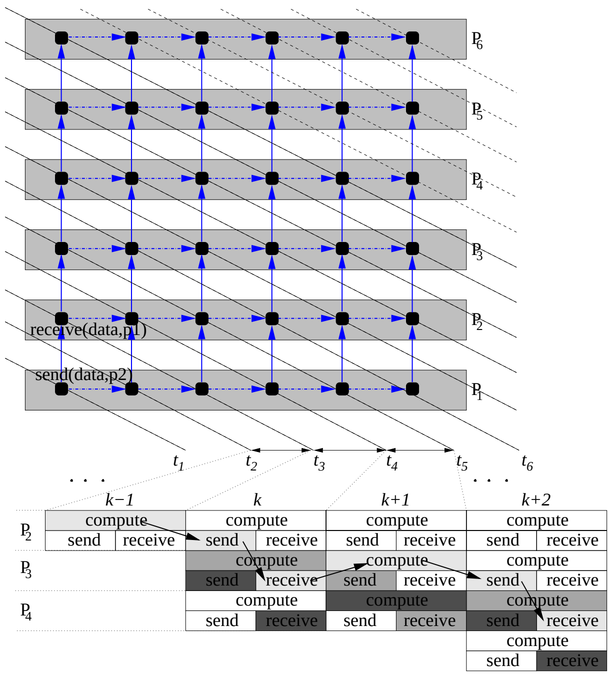

Figure 2. Overlapping time schedule

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.

Similar to expression (3) the total execution time in the overlapping schedule is given by:

$$T_{overlap} = P'(g)\max(t_{comp}, t_{comm}), \tag{4}$$

where again here $t_{comm} = t_{startup} + t_{transmit}$.

The number of execution time steps denoted by P' in (4), which depends on tile volume g, is equal to the execution time $\Pi' j^S_{final} + 1$ of the final executed tile j^S_{final}, given that the initial tile is executed at time step 0. So, from (4) we get:

$$T_{overlap} = (\Pi' j^S_{final} + 1)\max(t_{comp}, t_{comm}), \tag{5}$$

This theoretical result will be evaluated in the following sections by comparison with experimental results of kernels using our scheduling theory. In the next sections, we will describe the properties of two experimental platforms concerning the evaluation of our scheduling theory.

Since the concept of overlapping of actions is crucial, it should be noted that the actions initiated by a nonblocking call are overlapped with the actions initiated by calls following the nonblocking call. On the contrary, a blocking call implies no overlapping of actions, since a following call can be initiated only after the blocking call has completed.

5. MPI - TCP/IP - FastEthernet

Message Passing Interface (MPI) library on top of TCP/IP networking layers over FastEthernet network interface is the most common configuration for implementing and executing parallel applications on computer systems. MPI (and the ongoing MPI-2) is the de facto standard library specification for message-passing. TCP/IP is the network protocol layering mostly used even in LANs, despite its wide area networking nature. The descendant of 100Mbps Ethernet (FastEthernet) is 1 Gbps Ethernet (Gigabit Ethernet - GbE) and is the standard interconnection technology used to build Beowulf clusters serving as affordable parallel machines.

We used this configuration in order to evaluate our scheduling method. It was already known that MPI standard included the notion of nonblocking communication implemented in the `MPI_I*` commands (e.g. `MPI_Isend` initiates a nonblocking send). Our scheduling theory would be easily tested using pseudocode segments similar to the following:

```
MPI_Isend(sbuf, neighboring_nodes);
MPI_Irecv(rbuf, neighboring_nodes);
compute();
MPI_Wait(MPI_Isend);
MPI_Wait(MPI_Irecv);
```

According to the MPI standard, in the above pseudocode segment the computation (`compute()`) is performed concurrently with transmission and reception. However, after the first experimental results, we noticed that the nonblocking method, compared to the blocking one, had equal or even worse performance in terms of execution times, which was against our intuition. This oddity was due to several reasons, which are described below.

5.1. Platform dependent inefficiencies

All MPI implementations essentially follow the MPI standard, which was evolved considering portability as its first priority. Due to the fact that not every aspect of this wide standard is implementable on every possible platform, many MPI implementations adopt a more conservative approach, implementing only a part of all functions and routines, exactly as specified by the standard. According to the MPI standard, `MPI_Isend` and `MPI_Irecv` commands are the ones to be used for nonblocking communication. However, since not every networking platform is capable to carry out nonblocking communication calls (e.g. lack of DMA engine, protocol inneficiencies, etc.), in those cases both MPI calls underneath use blocking primitives, resulting in a loss of their expected nonblocking functionality.

Given the fact that MPI nonblocking commands could not deliver the nonblocking performance needed, we considered bypassing MPI library routines and conduct our experiments using Unix sockets as communication primitives. We decided to program on a lower level, in order to acquire more accurate control over the communication process and achieve the desired nonblocking features. In this way, we would also avoid the additional buffer copying imposed by the use of MPI library. However, current networking protocols and hardware commodity were obstacles to true overlapping communication with computation.

5.2. Protocol inefficiencies

Network programming in Unix environment is most times synonymous to socket interface, which is the most common way of communication among Unix hosts. After initializing the two endpoints of a communication, each node holds a socket descriptor which can be used in collaboration with `write` and `read` system calls for sending and receiving data, respectively. During the initialization phase, the communication type is determined as either reliable or unreliable. MPI libraries always depend on reliable communication (TCP/IP) and never on unreliable (UDP/IP), even though reliable communication is slower, since they would have to implement the reliability procedure in application level.

In order to comprehend the mechanism of such communication, the following system call is described in a reliable connection context:

```
(void) write(sd, buffer, len);
```

where `sd` is a socket descriptor, `buffer` is a pointer to the buffer where data are kept, and `len` is the length of data measured in bytes. After issuing this system call, the following actions take place: Firstly, the CPU execution mode turns from user mode to kernel mode (Fig. 3). The data buffer, if larger than a certain threshold, is not sent as a whole but is broken into smaller pieces. Each of these pieces is copied to kernel space by the CPU, which also adds TCP, IP and Ethernet headers to the packet. CPU then programs NIC's DMA engine to transfer the packet to NIC's buffers and can immediately perform other tasks, including data computations. As soon as DMA finishes, CPU is interrupted to be informed of that. The actions' chain is shown in Fig. 3.

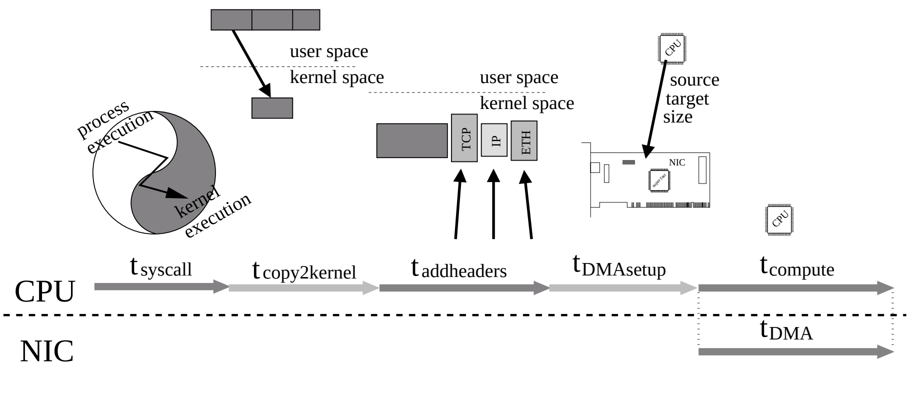

Figure 3. Successive sending phases for a packet, including CPU and NIC. (a) CPU execution switches from user to kernel mode, (b) CPU copies data from user space to kernel space, (c) CPU adds protocol headers, (d) CPU sets up DMA engine, (e) NIC DMA transfers data to its buffers, while CPU is free for computations

The use of MPI, on top of this layering, decreases the overlapping time percentage, since all MPI communication actions, which are not included in the aforementioned hierarchy, are buffering and bookkeeping operations and occupy the CPU exclusively.

5.3. Implementation

From the aforementioned facts, it is inferred that the only part of communication from which CPU is alleviated and can perform useful computations, is during the DMA data transfer to NIC's buffers (Fig. 3e). Since such overlapping is not adequate for implementing our scheduling theory, we decided to **simulate** our schedule using MPI on top of TCP/IP over FastEthernet.

We ran our experiments on a cluster with 16 identical 500MHz Pentium III nodes. Each node has 128MB of RAM and 10GB hard drive and runs Linux with 2.2.14 kernel version. Each node runs its own OS kernel, installed in local drive to avoid NFS traffic during the experiment's time. The cluster nodes are interconnected using 100Mbps FastEthernet. There exists a frontend machine serving as router, to isolate the cluster from the other LAN, so that non-local network traffic is banned. The communication library used is the MPICH implementation on MPI.

Our test application was a 3-D loop to be executed on the 16 nodes. We used a nested loop with only one assignment statement i.e.:

```
for (i=1; i<DIMX; i++)
 for (j=1; j<DIMY; j++)
  for (k=1; k<DIMZ; k++)
   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 computation time at a reasonable value, so as to efficiently control computation to communication grain. In this case, 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 $P^{(i,j)}$, $i \in [0,3]$, $j \in [0,3]$. During each time step, every processor in the ij plane with coordincates (i,j) receives from neighboring processors $(i-1,j)$ and $(i,j-1)$, computes and sends to processors $(i+1,j)$,$(i,j+1)$.

For the overlapping case to achieve overlapping of computation and communication, we need additional space compared to the non-overlapping case on each node to buffer the surfaces that are received or being sent to every neighboring node, while altering the data during the computation of (i,j,k) tile. The overall time schedule for k time step and the extra buffer to achieve overlapping is shown in Fig. 4. The additional buffering resembles the pipeline registers among consecutive pipeline stages, during the evolution from a multicycle non-pipelined datapath to a pipelined one [18].

In order to compare both theories, we implemented both the overlapping (nonblocking) and the non-overlapping (blocking) cases. 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

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)

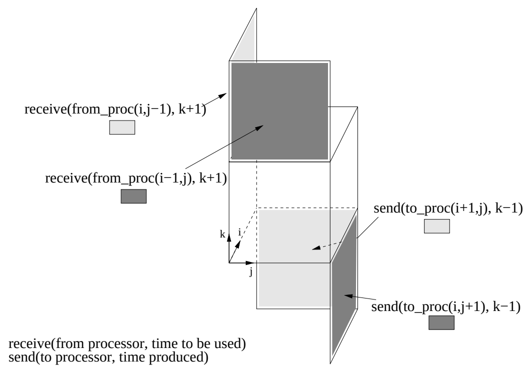
receive(from_proc(i,j−1), k+1)
receive(from_proc(i−1,j), k+1)
send(to_proc(i+1,j), k−1)
send(to_proc(i,j+1), k−1)
k
i
j
receive(from processor, time to be used)
send(to processor, time produced)

Figure 4. Timing and extra buffering for the overlapping case

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

In our experiments, `MPI_Ssend`[1] is used instead of `MPI_Send` in the blocking case, in order to simulate the absence of parallelism and overlapping in the blocking case. `MPI_Ssend` call returns when the message sent has been received from the receiving process.

[1]MPI Synchronous send

5.4. Results

In Figures 5, 6 and 7 the improvement in the execution of the blocking and nonblocking version of the algorithm is shown. The nonblocking version is much faster and it would be even faster if there were a special subsystem that would handle the communication part, so that it would be completely in parallel with the computation part.

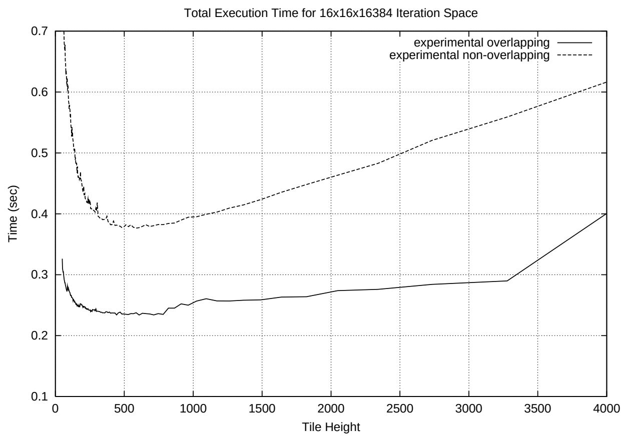

Figure 5. MPI over TCP/IP over FastEthernet: Results for 16x16x16384 space

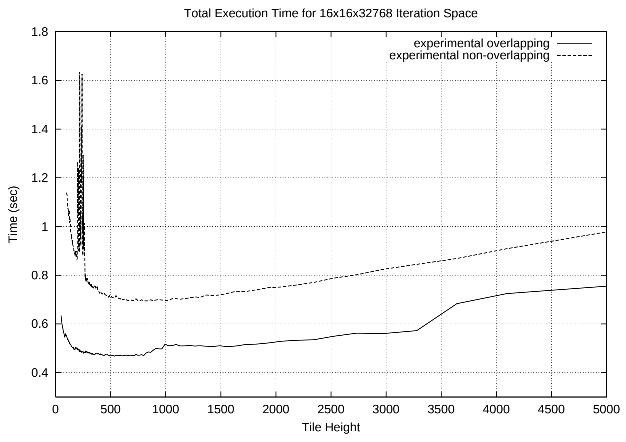

Figure 6. MPI over TCP/IP over FastEthernet: Results for 16x16x32768 space

The experiments were concerning three cases, depending on the size of initial J^n iteration space. We selected a $16 \times 16 \times 16384$ space, a $16 \times 16 \times 32768$ and a $32 \times 32 \times 4096$ space, where $\mathtt{DIMX} \times \mathtt{DIMY} \times \mathtt{DIMZ}$ represent the boundaries of i, j, k axes respectively. This means that, for example, in the first case, $i = 1 \ldots 16$, $j = 1 \ldots 16$ and $k = 1 \ldots 16384$.

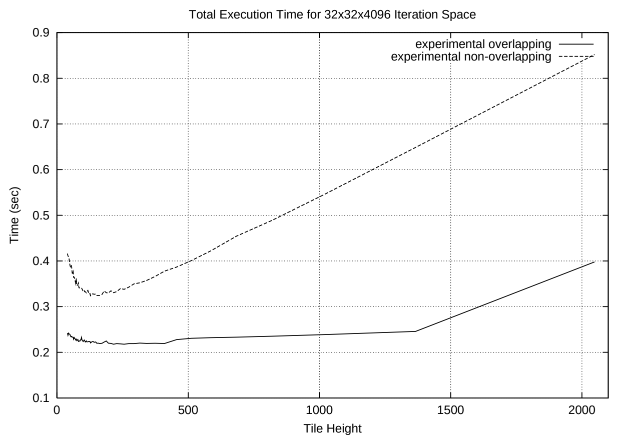

Figure 7. MPI over TCP/IP over FastEthernet: Results for 32x32x4096 space

We deliberately selected small i, j maximum values, so that k is always larger than the other two. The tiled space will have k dimension as its larger one, so mapping all tiles to the same processor is performed along the k-axis. This gave us the opportunity to use all 16 processors for the different tiles projected in ij plane. Without lack of generality, we selected unitary initial dependencies, thus the optimal tiling is a rectangular one. For every one of the above three problems, we were using all 16 processors, that is, 4 processors along each i and j dimensions. This means that all tiles, for example in the first case, had sizes of $4 \times 4 \times z$ where z was a variable (z is denoted as tile height, since it is the size of tile along axis k). For all possible values of z, ranging from 4 to $\frac{32768}{4}$, we ran both, complete, non overlapping and overlapping MPI programs, and calculated the size of $z_{optimal}$ for which the minimum completion time is achieved. Figures 5, 6 and 7 summarize our results for all three cases.

6. Implementation using Scalable Coherent Interface

In the previous section, there were several obstacles that hinted us from implementing the proposed scheduling theory due to the customary network technologies used. We could not allow for concurrent use of CPU and NIC due to limited capabilities of the networking subsystem. This could be avoided if novel communication technologies such as Scalable Coherent Interface (SCI) were used.

SCI supports a Distributed Shared Memory approach for communication among hosts. An SCI communication scenario involves the following stages: A process in an SCI node exports a memory segment which is imported by a process that resides in another SCI node. Every imported memory segment is directly mapped to the PCI I/O space of the PCI-SCI NIC. It is part of the importer's (process) virtual memory through the prior invocation of an `SCIConnectSegment()` driver call. When the importing node needs to send data, it just writes it directly to the imported memory segment, while when it needs to read data, it just reads it directly from the imported segment (Fig. 8).

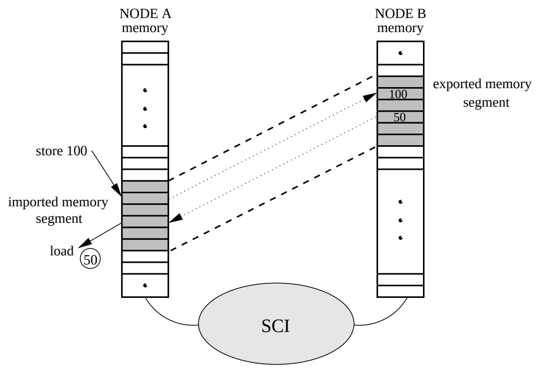

Figure 8. SCI load/store instruction over exported segment: (a) process on node A writes value 100 to a location in the imported segment and the value is transferred to node B exported segment, (b) process on node A reads a location in the imported segment and the value 50 is transferred from node B exported segment

6.1. DMA transfers

Message data can be usually transferred in two ways; Programmed I/O (PIO) mode and DMA mode. In PIO mode, CPU handles data transfer completely, which was the case in the previous paragraph. The data transfer of 1Kwords, involves the initial copying of these words from main memory to the NIC's buffers; the task would be performed by the host's CPU. From a parallel application's point of view, these are considered "lost" CPU cycles, since useful calculations could have been executed instead.

On the contrary, using DMA mode, CPU only programs the NIC's DMA engine with the information of which data to transfer from main memory and where to send it (Fig. 9). CPU is not blocked during the transfer and can perform other (useful) tasks.

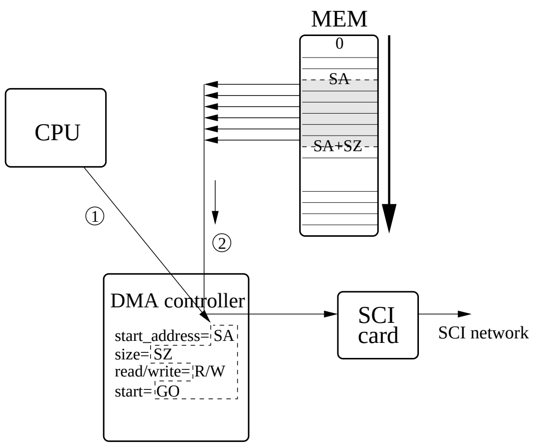

Figure 9. DMA or nonblocking send

In order to send data using DMA, it is essential for the data to be stored contiguously in physical memory. Being able to handle DSM communication using PIO mode, SCI has developed mechanisms that enable efficient memory allocation and usage. Using special SCI driver calls, the system returns physically contiguous allocated memory. This is performed using the `__get_free_pages()` kernel routine. The allocated memory is first "pinned down" and then mapped to user's virtual memory (Fig. 10). User is able to read/write that memory region like the ordinary memory regions returned by LIBC `malloc()`. Despite the fact that SCI DMA transfer is only invoked as a kernel system call, the complete transfer of the specific memory area will be performed with only one DMA invocation.

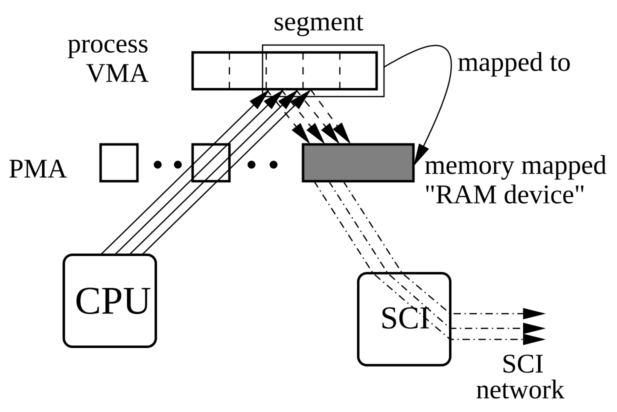

Figure 10. Locked (pinned down) and memory mapped "RAM device" for SCI communications. While CPU computes data, results are stored to physically contiguous memory, ready to be sent as a whole using DMA

6.2. Implementation

We used 9 800MHz Pentium-III nodes interconnected with an SCI network based on Dolphin's D330 SCI NICs. Each node has 128MB of main memory. The OS is Linux with kernel from the 2.4.x series. In order to assess the benefits of the proposed scheduling theory, we ran two type of experiments. The first one implements the overlapping (nonblocking) algorithm and is compared to the second one which implements the non-overlapping (blocking) algorithm. The test application was implemented using C and the SISCI API [12]. We also compare our overlapping experimental results to the results calculated using the theoretical formula given by equation 5.

We conducted our SCI experiments using the same 3-D loop used in the MPI over TCP/IP over FastEthernet case. Our 9 cluster nodes were organized as a 3×3 array of processors. The optimal tiling is in rectangular tile shapes. Each tile is a cube with ij, ik and kj sides. Without lack of generality, we selected k dimension to be the largest one, so all tiles along k-axis are mapped to the same processor $P^{(i,j)}$, $i \in [0, 2]$, $j \in [0, 2]$. The data exchanges, timing and buffering characteristics of the MPI case (Fig. 4) are also valid in this case.

According to expression (5), the theoretical total execution time for the SCI overlapping case is:

$$T_{overlap}(z) = (2(x_i + x_j) + x_k + 1)\max(t_{comp}, t_{comm}), \tag{6}$$

where in our case, $j^S_{final}(x_i, x_j, x_k)$ is the last executed tile according to the overlapping scheduling theory.

We assume that $t_{comp} = t_{comm}$ so $\max(t_{comp}, t_{comm}) = t_{comp}$. Expression (6) does not include the communication time that cannot be overlapped with computation time. So (6) must be enriched with terms t_{start_dma} and t_{sync}, that represent time to initiate a DMA procedure and synchronization between nodes, respectively. So (6) becomes:

$$T_{overlap}(z) = (2(x_i + x_j) + x_k + 1)(t_{start\_dma} + t_{comp} + t_{sync}). \tag{7}$$

In the above expression, x_i is equal to $\texttt{DIMX}/3-1$, similarly x_j is equal to $\texttt{DIMY}/3-1$, and x_k is equal to $\texttt{DIMZ}/z - 1$. Tile's height is denoted by z and since a tile contains $g = xyz$ iteration points and x, y are already known as problem variables, in our experiments only z is unknown.

Due to the need for synchronization between any two successive time steps, nodes have to signal each other using SCI interrupts, which impose a constant delay, $t_{sync} = 4 \times t_{sci\_interrupt}$. We ran several ping-pong tests and derived the values $t_{start\_dma} = 49.2usec$ and $t_{sci\_interrupt} = 18.8usec$.

The total computation time for the execution of each application, either overlapping or non-overlapping, is constant and can be seen in Fig. 11 for the "non-overlapping case" and the "overlapping case without SCI". The latter concerns the execution of the overlapping case, having commented out all the SCI communication functions. In this way we only measure the pure computation time t_{comp}, which is calculated using the following code:

```
gettimeofday(start, NULL);
compute();
gettimeofday(end, NULL);
```

The computation time for the overlapping case, considering also the SCI communication functions, is shown in Fig. 11. The decreasing plot is due to the frequent kernel invocations which are servicing interrupts for SCI communication: local CPU, apart from `compute()`, also handles both `SCITriggerInterrupt` executed on a neighboring node and `SCIPostDMAQueue` executed on the current node. In the beginning of each experiment, the tile size is small, so there is a substantial number of exchanged interrupt signals (`SCITriggerInterrupt`) and data transmissions (`SCIPostDMAQueue`) routines existing in main loop body. When the number of iterations is reduced due to increased tile size, the CPU time consumed on handling interrupts is decreased, and finally converges to the non-overlapping case. Thus, the pure compute time used to calculate the theoretical plots should come from the non-overlapping case.

The internal part of the nonblocking program's main loop can be seen in Table 1. Since `send_dma()` is not blocking, the `compute()` call is concurrently executed. After the execution of `wait_for_dma()`, it is assured that both computation and communication are already completed. The blocking program is implemented by swapping the `compute()` and `send_dma(n+1,data)` calls.

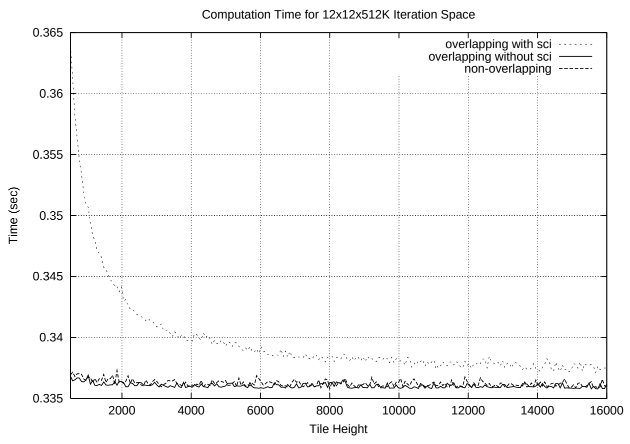

Figure 11. SCI: Comparison of Experimental Computation Times for $12 \times 12 \times 512K$

Table 1
Internal Part of Program's Main Loop.

sequence of functions	respective SCI calls	Action performed
`trigger_interrupt(n-1)`	SCITriggerInterrupt()	Inform "prev" node(s) "Ready to accept data"
`wait_for_interrupt(n+1)`	SCIWaitForInterrupt()	Wait till "next" node(s) ready to receive data
`send_dma(n+1, data)`	SCIPostDMAQueue()	Init of DMA transfer to neighboring nodes
`compute()`	compute()	Computation
`wait_for_dma()`	SCIWaitForDMAQueue()	Wait for DMA to complete
`trigger_interrupt(n+1)`	SCITriggerInterrupt()	Inform "next" node(s) "Data have arrived"
`wait_for_interrupt(n-1)`	SCIWaitForInterrupt()	Wait till "prev" node(s) has finished sending

The above test application was executed using initial J^3 index spaces with various DIMX$\times$DIMY$\times$DIMZ sizes. Typical experimental values for DIMX=DIMY were 12 or 24 and for DIMZ were $256K$, $512K$, or $2048K$. We measured execution times for the following overlapping and non-overlapping cases $12\times12\times512K$ (also in [19]), $24\times24\times256K$ and $24\times24\times2048K$.

Overlapping and non-overlapping overall execution times for each problem are plotted in Figs. 12, 13 and 14. It can be seen that, in all cases, overlapping (pipelined) executions, which take advantage of the cluster's high performance communication features, are considerably faster than the non-overlapping (blocked) ones.

In Fig. 15, the experimental result is compared to our analytical calculations derived from (7). The plot for the experimental time measured, is very close to the theoretical function. This is due to the fact that (7) includes a thorough and detailed analysis of actual possible time delay parameters. For example, from the minimum of each function in Fig. 15, it can be easily calculated that the difference between experimental minimum and theoretical minimum is nearly 0.2%, achieved for values of tile heights, which are very close to each other.

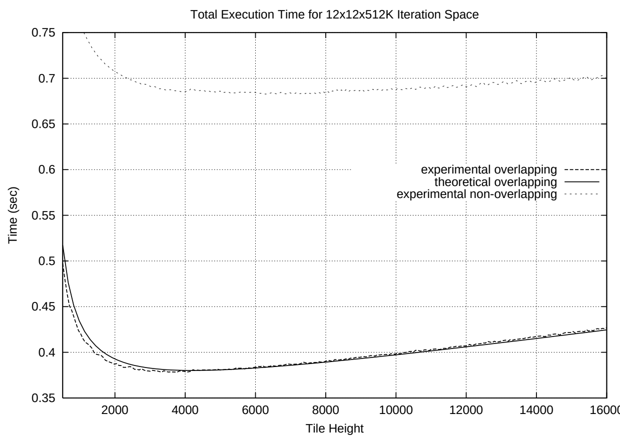

Figure 12. Experimental Total Execution Times for $12 \times 12 \times 512K$

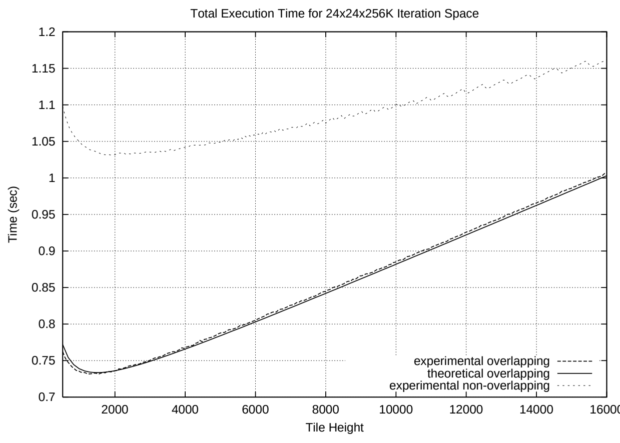

Figure 13. Experimental Total Execution Times for $24 \times 24 \times 256K$

7. Conclusions

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. Both experimental results using SCI and simulation results using MPI show that our scheduling theory achieves better execution times than a corresponding non overlapping one. Experimental results have also shown that the theoretically calculated overall time, following the optimal hyperplane transformation, is very similar to the experimental results.

The only available hardware to conduct experiments was PCI-SCI NICs which use kernel DMA initialization. However, if we could avoid all kernel initialization of DMA, then the initial DMA startup time could have been considerably reduced. Since DMA is initiated through calls from kernel level, we thus introduce extra overhead, which could

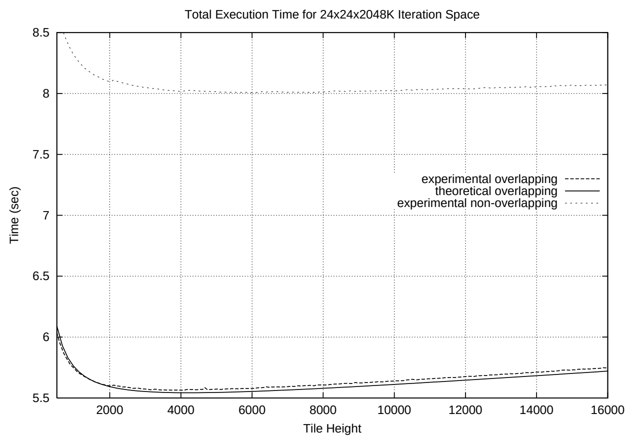

Figure 14. Experimental Total Execution Times for $24 \times 24 \times 2048K$

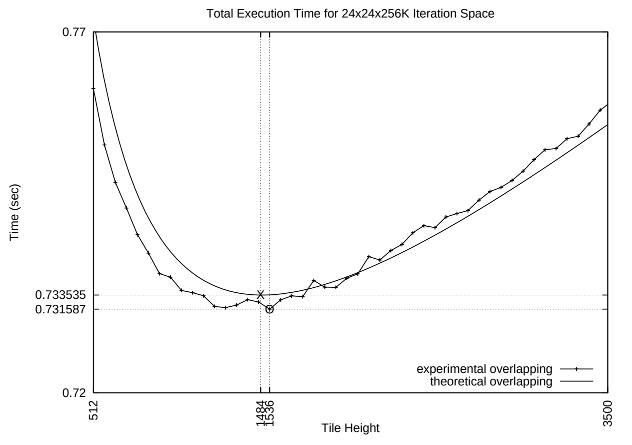

Figure 15. Comparison of Experimental and Theoretical Minima (Fig. 13 zoomed in)

increase overall execution time. User Level Networking architectures, such as U-Net [20] and the ensuing VIA standard [21], allow for direct access of the NIC from virtual memory areas and without any kernel intervention (see [22], [23]).

At the moment there is no public available hardware VIA implementation for PCI-SCI cards, that uses DMA as communication mode. In fact, in [24], a VIA solution for SCI was presented, using PIO as the only available communication mode. It is obvious that we essentially need overlapping, so even avoiding kernel system calls' overhead is not enough. In [25] a novel hardware implementation of a PCI-SCI bridge is presented, supporting both downstream and upstream Address Translation Tables (ATTs), thus capable of exporting any arbitrary virtual memory page and access it directly by DMA, but this is also not available to us. Intuitively, we expect that if we achieve 100% overlapping of computation with communication, then overall execution time using our scheduling theory will reach its minimum.

REFERENCES

1. W. Shang, J. Fortes, Independent Partitioning of Algorithms with Uniform Dependencies, IEEE Trans. on Computers 41 (2) (1992) 190–206.
2. E. H. Hollander, Partitioning and Labeling Loops by Unimodular Transformations, IEEE Trans. on Parallel and Distributed Systems 3 (4) (1992) 465–476.
3. 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 57 (2) (2000) 941–955.
4. 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, Toronto, Canada, 2000, pp. 469–476.
5. F. Irigoin, R. Triolet, Supernode Partitioning, in: Proc. 15th Ann. ACM SIGACT-SIGPLAN Symp. Principles of Programming Languages, San Diego, California, 1988, pp. 319–329.
6. J. Ramanujam, P. Sadayappan, Tiling Multidimensional Iteration Spaces for Multicomputers, Journal of Parallel and Distributed Computing 16 (1992) 108–120.
7. P. Boulet, A. Darte, T. Risset, Y. Robert, (Pen)-ultimate tiling?, INTEGRATION, The VLSI Jounal 17 (1994) 33–51.
8. E. Hodzic, W. Shang, On Supernode Transformation with Minimized Total Running Time, IEEE Trans. on Parallel and Distributed Systems 9 (5) (1998) 417–428.
9. 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 57 (2) (1999) 140–165.
10. V. Karamcheti, A. Chien, Software Overhead in Messaging Layers: Where Does the Time Go?, in: Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems, 1994, pp. 51–60.
11. F. O. Carroll, H. Tezuka, A. Hori, Y. Ishikawa, The Design and Implementation of Zero Copy MPI Using Commodity Hardware with a High Performance Network, in: Proceedings of the International Conference on Supercomputing, Melbourne, Australia, 1998, pp. 243–249.
12. F. Giacomini, T. Amundsen, A. Bogaerts, R. Hauser, B. Johnsen, H. Kohmann, R. Nordstrom, P. Werner, Low Level SCI software functional specification-Software Infrastructure for SCI, ESPRIT Project 23174, http://www.dolphinics.com/downloads/nt/pdf_zip/ SISCI_API-2_1_1.pdf.
13. M. Eberl, H. Hellwagner, B. Herland, M. Schulz, SISCI - Implementing a Standard Software Infrastructure on an SCI Cluster, in: 1st Workshop Cluster Computing, TU-Chemnitz, 1997.
14. M. Schulz, SISCI Pthreads Implementation report, ESPRIT Project 23174 - SISCI, Deliverable 4.1.4, Technische Universität München (Nov 1999).
15. J. Xue, On Tiling as a Loop Transformation, Parallel Processing Letters 7 (4) (1997) 409–424.
16. J. Xue, Communication-Minimal Tiling of Uniform Dependence Loops, Journal of

Parallel and Distributed Computing 42 (1) (1997) 42–59.
17. W. Shang, J. Fortes, Time Optimal Linear Schedules for Algorithms with Uniform Dependencies, IEEE Trans. on Computers 40 (6) (1991) 723–742.
18. D. Patterson, J. Hennessy, Computer Organization & Design. The Hardware/Software Interface, Morgan Kaufmann Publishers, San Francisco, CA, 1994, pp. 364–367.
19. A. Sotiropoulos, G. Tsoukalas, N. Koziris, A Pipelined Execution of Tiled Nested Loops onto a Cluster of PCs using PCI-SCI NICs, in: Proceedings of the 2001 SCI-Europe Conference, Dublin, Ireland, 2001.
20. T. Eicken, A. Basu, V. Buch, W. Vogels, U-Net: A User-Level Network Interface for Parallel and Distributed Computing, in: Proceedings of the 15th ACM Symposium on Operating System Principles, Copper Mountain, Colorado, 1995, pp. 40–53.
21. The Virtual Interface Specification Version 1.0, http://www.viarch.org.
22. M. Banikazemi, B. Abali, L. Herger, D. K. Panda, Design Alternatives for VIA and an Implementation on IBM Netfinity NT Cluster, Special Issue of Journal of Parallel and Distributed Computing on Cluster and Network-Based Computing .
23. M. Blumrich, Network interface for protected, user-level communication, Ph.D. thesis, Princeton University (Apr 1996).
24. K. Ghouas, K. Omang, H. Bugge, VIA over SCI - Consequences of a Zero Copy Implementation and Comparison with VIA over Myrinet, in: Proceedings of the Workshop on Communication Architecture for Clusters (CAC' 2001) in conjunction with Int'l Parallel and Distributed Processing Symposium (IPDPS '01), San Francisco, 2001.
25. M. Trams, W. Rehm, D. Balkanski, S. Simeonov, Memory Management in a Combined VIA/SCI Hardware, in: Proceedings of Intl. Workshop on Personal Computer based Networks of Workstations (PC-NOW 2000), held with IPDPS 2000, Cancun, Mexico, 2000, pp. 4–15.