A Pipelined Execution of Tiled Nested Loops onto a Cluster of PCs using PCI-SCI NICs

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

Abstract— **This paper proposes a new method for the problem of executing tiled nested loop based programs onto clusters of PCs using PCI-SCI Network Interface Cards (NICs). Loop iterations are grouped together to create large tiles which are atomically executed by a processor. In order to minimize overall execution time, the objective is to avoid frequent data exchanges, thus synchronization points between successive tile executions which incur extra time overhead. Tiles should necessarily exchange data but also have large computational grain, so that their parallel execution becomes beneficial. We schedule tiles much more efficiently by exploiting the inherent overlapping between communication and computation phases among successive, atomic tile executions. Thus, our proposed nonblocking schedule resembles a pipelined datapath where computation phases are overlapped with communication ones, instead of being interleaved with them. In order to implement such schedule, we need low latency for messages and the capability of concurrently using CPU and NIC. We use DMA to remote write (send) data to other nodes, while the host CPU is computing all iterations within each tile. We achieve zero-copy messaging through pinned-down physical memory regions for DMA (PCI exported segments to SCI global space), ideal for fast data exchanges between neighboring tiles. Overall completion time is considerably minimized when applying our nonblocking schedule instead of the blocking one. The experimental results are performed on a Linux cluster of Pentium PCs interconnected with Dolphin's D330 PCI-SCI cards.**

Keywords— **Loop tiling, communication overlapping, zero-copy, send-receive primitives, DMA, user-level networking, parallel.**

I. 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. Even in tightly coupled shared memory multiprocessors, there exists non-negligible synchronization latencies due to the coherency protocols.

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 [12]. They introduced the initial model of loop tiling and gave conditions for a tile transformation to be valid. Tiles are required to be atomic, identical, bounded and their union spans the initial iteration space. However, it was early enough to give any thorough framework of determining the optimal tile characteristics (size, shape) based on communication reduction or overall makespan minimization.

In their paper Ramanujam and Sadayappan [15] gave a linear programming formulation for the problem of finding optimal shape tiles (thus determining optimal tile transformation H) that reduces communication by adjusting the tile shape accordingly. The use of a communication function that has to be minimized by linear programming approaches was also used by Boulet et al. in [3]. 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. Thus they gave a linear programming approach to determine optimal tile shape for any given volume. The problem of determining the optimal shape was surveyed, and more accurate conditions were also given by others as Xue [21].

Nevertheless, all above approaches have a minor disadvantage. They ignore the actual iteration space boundaries. Although tile shape is a determinant of communication reduction, the ultimate objective should be the overall tiled space completion time. In other words, what is of great importance, is an efficient way to schedule tiles to processors, trying to minimize overall execution time. Since modern microprocessors are very fast and capable of executing several computations in very few microseconds, tile sizes are assumed very large, sometimes grouping together thousands of iterations. Even in cache coherent machines, the remote cache references, due to the coherency protocol, impose tens of microseconds of latencies. So, for a computation to communication grain to be meaningful, tiles are large enough to encapsulate all dependence vectors. Thus every tile needs to exchange data only from its neighbors, one in each dimension. This is a very reasonable assumption, i.e. very large tiles, taking into consideration the high computational power of modern processors in comparison with network latencies.

Hodzic and Shang [11] proposed a method to correlate optimal tile size and shape, based on overall completion time reduction. They applied the hyperplane transformation to the loop tiles and generated a schedule where the objective is to reduce the overall time by adjusting the tile size and shape appropriately. However their approach considers a straightforward time schedule. 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 [9] we proposed 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 [14], 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. Implementation was done using MPI send-receive blocking and non-blocking primitives. Common MPI_send, MPI_receive primitives are usually implemented as non-blocking ones. In fact, to overcome this, we used synchronous primitives to emulate the blocking (non-overlapping case) and non-blocking asynchronous ones for the overlapping case. Results have shown that it the non-overlapping schedule is much better, however, in real word, hardware should provide support for it.

It is obvious that, in order to achieve true overlapping of computations with communication phases, we have to alleviate the CPU from the burden of carrying the data transfers between neighboring nodes. In fact, when using common messaging libraries with regular send-receive primitives (i.e. sockets, or PVM or MPI etc.), then CPU is mainly responsible for the copying of the message from user space to kernel space and then to NIC buffers. How could it be possible under this scheme to achieve actual overlapping? Solution is given from the DMA. Although DMA incurs extra initial latency, it assumes all transfer without annoying the CPU, which in its turn can perform actual computations. DMA startup latency is due to copying from user (virtual) memory area, to specific (physical memory) pinned down regions, especially reserved for DMA transfers.

Since DMA is initiated through calls from kernel level, we thus introduce extra overhead, which could increase overall execution time. User Level Networking architectures, such as U-Net [6] and the ensuing VIA standard [20], allow for direct access of the NIC from virtual memory areas and without any kernel intervention (see [2]). These architectures seem to propose a fast messaging layer capable of achieving 2-10 microseconds of message startup latency. However these architectures are message passing oriented.

In this paper, we extend our work of [9] by proposing the use of a cluster of Linux PCs with SCI Network Interface Cards (NIC) connected to the I/O PCI bus. 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 [13]. 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 messaging is considerably reducing the initial latency for data transfers (for the effects of a zero-copy implementation of a common communication layer see [4]). 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 [8], [5], [17] 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 present the experimental results by implementing both scheduling approaches using SCI primitives. Finally, in Section 6 we propose future work.

II. Models – Loop Tiling

In this paper we consider algorithms with perfectly nested FOR-loops and uniform 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, (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.

A. 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, i.e. independent of the indices of computations.

B. Supernode Transformation

In a supernode transformation the index J^n space is partitioned into identical n-dimensional parallelpiped 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. Similar to matrix H, 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: $J^S = \{j^S | j^S = \lfloor Hj \rfloor, j \in J^n\}$, $D^S = \{d^S | d^S = \lfloor H(j_0 + d) \rfloor, d \in D, j_0 \in J^n | 0 \leq \lfloor Hj_0 \rfloor \leq 1\}$ 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 [12][15]. 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 $|HD| < 1$ 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. In this case every tile needs to exchange data only with its nearest neighbors, one in each dimension of J^n. 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. An analytical formula to calculate the exact communication cost was given in [21],[3] thus enabling the calculation of matrix H that imposes the minimum amount of communication for a given supernode size.

Finally, 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, thus 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 $t_{j^S} = \lfloor \frac{\Pi j^S + t_0}{disp\Pi} \rfloor$

III. Non-overlapping Schedule

In [11], Hodzic and Shang have presented a scheme for scheduling loops that have been transformed through a supernode transformation. Their approach is to minimize total execution time, as follows: The optimal tile size g 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:

$$T = P(g)(t_{comp} + t_{comm}), \tag{1}$$

where $P(g)$ is the number of time hyperplanes needed to execute the algorithm, t_{comp} the execution time of a tile, t_{comm} is the communication time and consists of a startup latency and transmission time $t_{transmit}$, thus $t_{comm} = t_{startup} + t_{transmit}$. Clearly, the total execution time depends on the tile size g, since it affects the number of time planes, the computation cost ($t_{comp} = gt_{comp_1}$, where t_{comp_1} is computation cost of a single iteration) and the communication volume (V_{comm}).

Let us now consider the implementation of the above schedule in a cluster of workstations, interconnected with a fast local area network. In this context, the execution time of a computation and communication phase consists of: the computation time t_{comp}, the startup communication time $t_{startup}$ and the send transmission time $t_{transmit}$.

The overall parallel loop execution consists of atomic computations of tiles interleaved with communication for the transmission of the results to neighboring processors. Since tiled space J^S has the unitary dependence vectors, the optimal linear time schedule can be easily proved to be: $\Pi = [1\ 1 \ldots 1]$. Analytically, each time step between successive hyperplanes contains a triplet of compute-startup-transmit non-overlapped subphases for each tile. There is no separate receive phase, since receive is performed automatically by the recipient's NIC, without any intervention of the respective CPU. All tiles along a specific dimension are mapped to the same processor. If we cluster together the startup and transmit subphases and call it "communication subphase" (t_{comm}), then we see that the overall schedule has computation subphases interleaved with communication ones(Fig. 1).

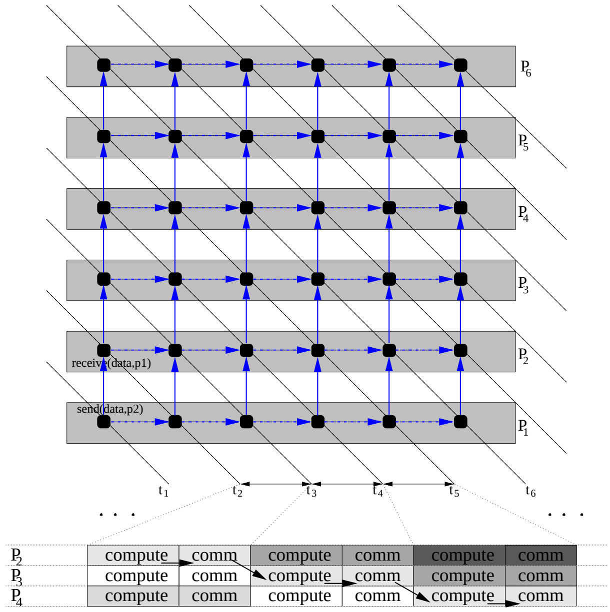

Fig. 1. Non-overlapping Time Schedule

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 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 a significant idle processor time.

IV. 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. It would be ideal if a node was able to compute and send data at the same time. Modern network interfaces have DMA engines that enable them to work in parallel with the CPU. This means that some communication work can be overlapped with actual CPU cycles. When communication work is finished, processor receives an interrupt. In fact, even some part of the nonblocking communication needs the CPU, but the transmission phase can be ideally overlapped with useful computation.

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 compute data, then initiate the communication and finally send the results to be used at the next step by its neighbor (Fig. 1).

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 schedule, then we could overlap some of actual communication time with computations. This means that, in each time step, the node should send and receive[1] data that is not directly dependent to the data computed at this step. A valid execution scheme would be for a processor to compute data received the previous time step, receive data that will be used in computations the next time step, send data that were computed the previous time step. In this case, every processor computes a tile, and receives+sends data needed in next step or produced in the previous step, respectively.

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 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 $2j_1^S + 2j_2^S + \ldots + 2j_{i_1}^S + 2j_{i+1}^S + \ldots + 2j_n^S + j_i^S$, (starting from $t = 0$) where i is the dimension along which all tiles are mapped to the same processor.

In Fig. 2 the overlapping schedule is shown. Consider, for example, processor P3 at k time step: While it makes the computation for a tile, he concurrently sends the results produced during $k-1$ time step and receives data from neighbors, to be used during the computation of next tile at $k+1$ time step. Note the arrows show in Fig. 2. They depict the actual flow of data between successive time steps (computes–dma setups –transmits) in pipelined way. The outcome of this schedule is to have successive computations overlapped with communication phases, thus a 100% theoretical processor utilization.

If we consider the possibility to overlap computation with

[1]This operation does not steal computation cycles.

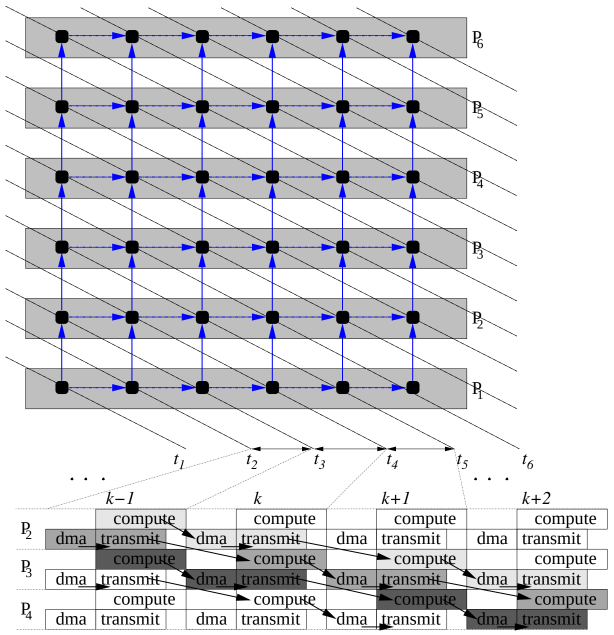

Fig. 2. Overlapping Time Schedule

communication, then we could have the following scheme: A processor first initiates all the nonblocking send operations and then performs the actual atomic tile computations. While the processor performs computations, the NIC is receiving data from neighbors and sends previously computed data to others as well.

In order to achieve actual overlapping of computation and communication, hardware should assist. The CPU and the NIC must be able to work simultaneously on different tasks. The most important issue is support from DMA, which should exist and be enabled to the NIC. Another aspect is that the invocation of DMA communication should be done in user level (User-Level DMA), without kernel intervention. Furthermore, zero-copy communications should be used and finally, the software packetization process involved in every communication must be avoided. Below we describe the DMA communication mode and analyze each aspect of the proposed solution.

DMA communication mode. Message data can be usually transferred in two ways. Programmed I/O (PIO) mode and DMA mode. In PIO mode, CPU handles the transferring completely, word by word. For example, to send 1Kword from one node to another, the first task is the transferring of 1K words from the main memory to NIC's buffers. Using PIO mode for the NIC, CPU has to read all data from main memory and store the to NIC's registers. These are lost CPU cycles, from the view of a parallel application, since CPU could be executing other useful calculations. On the contrary, using DMA mode for the NIC, CPU just programs the NIC's DMA engine with which data to get from main memory and where to send them (Fig. 3). CPU is not blocked during the transferring and can perform other (useful) tasks.

Zero-Copy communication. Common network trans-

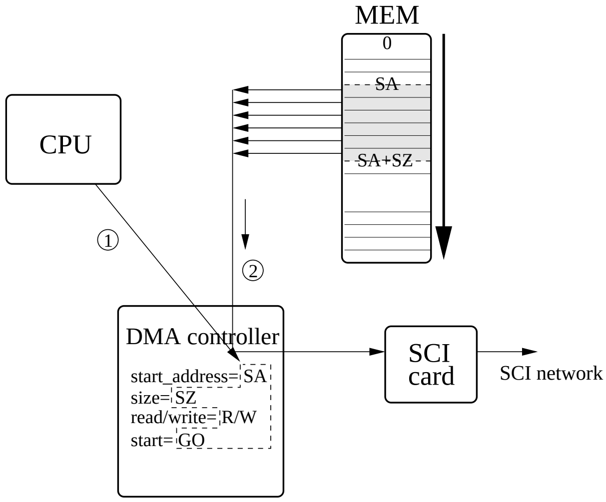

Fig. 3. DMA or nonblocking send

fers usually require for copying of data between user space and kernel space, as before. In network technologies like TCP/IP over FastEthernet this is essential, since data must be processed before transmitted. However, there are other network technologies like Scalable Coherent Interface (SCI) that avoid such copying procedures by performing zero-copy communication.

An SCI communication scenario has the following stages: One process in an SCI node exports a memory segment which is imported by a process that resides in another SCI node. The imported memory is part of the process' virtual memory. When the importing node needs to send data, it just writes it to the imported memory segment. Without any kernel intervention, data is transferred to the exporter's memory and communication is performed (Fig. 4). No other data processing is needed for each send.

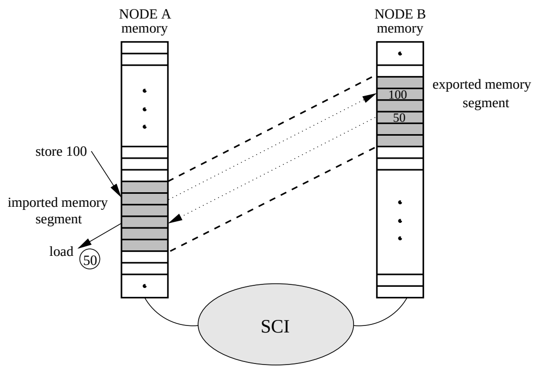

Fig. 4. SCI load/store instruction over exported segment

Moreover, when a memory segment is needed, the kernel is called to allocate physically contiguous memory. This is achieved by allocating kernel space and creating the corresponding entries in process page table. When user process writes to the allocated memory segment, writes on physically contiguous memory are performed. This segment is also "pinned down", so it cannot be swapped out to secondary memory. As a result, data on this memory segment can be transferred using DMA without any processing. SCI NICs have DMA capabilities, which in combination with the zero-copy communication properties, provide a very powerful platform to experiment with.

According to the previous properties, the total execution time for the overlapping schedule is given by:

$$T_{overlap} = P'(g)(t_{start\_dma} + \max(t_{comp}, t_{comm\_dma}) + t_{synchro}), \quad (2)$$

where $P'(g) = 2(x_1(g) + x_2(g) + \ldots + x_{j-1}(g) + x_{j+1}(g) + \ldots + x_n(g)) + x_j(g) + 1 = 2\sum i \neq j x_i(g) + x_j(g) + 1$ is the number of hyperplanes and j is the maximal coordinate. The time needed to initiate the DMA engine is t_{start_dma}, t_{comp} is the tile execution time, t_{comm_dma} is the communication time which can be overlapped with computation and $t_{synchro}$ is required synchronization time between successive time steps.

V. Experimental Results

We assembled a 9-node Pentium III-800MHz cluster with 2 alternative interconnecting networks, FastEthernet and SCI ring at 100MBps using Dolphin's D330 cards. Our test application is the following code:

```
for(i=1; i<DIMX; i++)
  for(j=1; j<DIMY; j++)
    for(k=1; k<DIMZ; k++)
      A[i][j][k]  =
 func(A[i-1][j][k], A[i][j-1][k], A[i][j][k-1]);
```

where A is an array of $\mathtt{DIMX}\times\mathtt{DIMY}\times\mathtt{DIMZ}$ floats and $\mathtt{DIMX} = \mathtt{DIMY} \ll \mathtt{DIMZ}$. Each tile is a rectangle with ij, ik and kj sides. The dimension k is the largest one, so all tiles along k-axis are mapped on the same processor. For simplicity we denote: V as tile's volume, $X \equiv \mathtt{DIMX}$ and $Z \equiv \mathtt{DIMZ}$, so $V = X^2 Z$. Each tile has i,j dimensions equal to x and the tile's "height" along k=axis equal to z. There are $\frac{X}{x}$ tiles on dimension i and j and $\frac{Z}{z}$ tiles on dimension k. Tile's volume is equal to $g = x^2 z$, and since the number of available processors is initially known, the only unknown parameter is z.

We applied both blocking and nonblocking scheduling. We calculated the optimal volume $g^{opt}_{blo}(z)$ and $g^{opt}_{nonblo}(z)$ for which the overall completion time of the problem is minimum for both the overlapping and non-overlapping case, respectively.

Our first implementation code used Linux POSIX threads. However, SISCI API [8] in combination with the SCI driver is not a thread-safe environment, even if we kept communication part in one thread and computation part in another. To overcome this problem, we decided for a single thread implementation, which would, however, perform computation and communication in parallel. This concurrency is shown in Table I, which depicts the internal part of the implemented program's main loop.

Similarly to the case of evolution from a multiple-cycles non-pipelined processor to a pipelined one, where registers are introduced among pipeline stages ([14]), evolving

TABLE I
INTERNAL PART OF PROGRAM'S MAIN LOOP.

commands	SCI calls	meaning
`trigger_interrupt(n-1)`	SCITriggerInterrupt()	Inform "previous" node(s) (n-1) "I am ready to accept data"
`wait_for_interrupt(n+1)`	SCIWaitForInterrupt()	Wait till "next" node(s) (n+1) is ready to receive data
`send_dma(n+1, data)`	SCIPostDMAQueue()	Initiation of DMA transferring to neighboring nodes
`compute()`	compute()	Computation
`wait_for_dma()`	SCIWaitForDMAQueue()	Wait for DMA to complete
`trigger_interrupt(n+1)`	SCITriggerInterrupt()	Inform "next" node(s) (n+1) "Your data have arrived"
`wait_for_interrupt(n-1)`	SCIWaitForInterrupt()	Wait till "previous" node(s) (n-1) has finished sending data

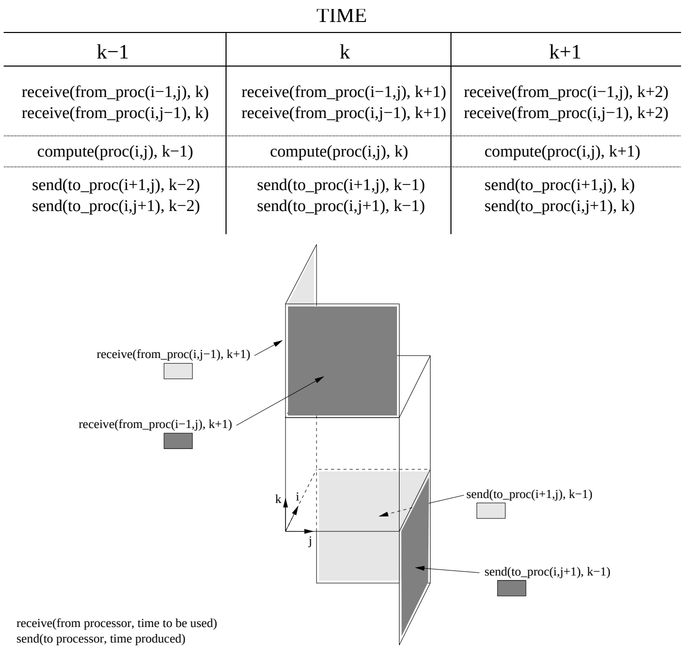

Fig. 5. Timing and Extra Buffering for the Overlapping Case

from non-overlapping schedule to the pipelined one, we added extra buffers for receiving and sending data which are shown with light and dark grey shading in Fig. 5. The compute is performed on the data in the middle cube, while receive is performed on the additional top buffering surfaces and send from the additional bottom buffering surfaces. During each time step, the actions performed by each node are: "receive" from its neighboring nodes data (without executing a distinct command), "compute" on these data and "send" results to its neighboring nodes. The overall time schedule for k time step is also shown in Fig. 5.

In table I, 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 non-overlapping case is implemented by swapping the `compute()` and `send_dma(n+1, data)` calls.

The problem is solved using the following parameters: `DIMX = DIMY = 12` and `DIMZ = 512K`. Since there are 3 processors in each dimension i and j, it is obvious that $x = 3$. Experimental results, drawn in Fig. 6, show that overlapping schedule properties result in faster total execution times.

Using our scheduling theory, the total execution time for the overlapping case is:

$$T_{overlap}(z) = (2\sum_{i\neq k} x_i + x_k + 1)(t_{start\_dma} + t_{comp} + t_{synchro}), \tag{3}$$

where in our case $\sum_{i\neq k} x_i$ is equal to $2(\frac{X}{x} - 1)$ and x_k is equal to $\frac{Z}{z} - 1$. The DMA initiation time t_{start_dma} is $(20\frac{z}{4096} + 1) \times t_{sci\_start\_dma}$, because of SCI's DMA limit of 64KB or 16Kfloats per DMA operation. For a computation to communication grain to be meaningful (to achieve roughly equal computation to communication times, thus complete overlapping), we assumed a larger amount of data to be sent, so each DMA send involves the transmission of 20 times the actual data to be sent. Due to synchronization purposes between two successive time steps, nodes have to signal each other using SCI interrupts that impose constant delay $t_{synchro} = 4 \times t_{sci\_interrupt}$. The computation time t_{comp} of each tile is equal to $zx^2 \times t_{comp_1}$, where t_{comp_1} is the time needed to compute one iteration point. We use two approximations for this value. One is constant and the other depends on tile's height and total number of tiles executed by one node (Fig. 7). We ran several tests to derive the following values: $t_{sci\_start\_dma} = 49.2usec$, $t_{sci\_interrupt} = 18.8usec$.

In Fig. 8, the experimental results are compared to our theoretical functions derived from (3). Each theoretical function uses one of the two different approximations of t_{comp_1}. Both of them are very close to the experimental times measured. This is also proved from the fact that, the tile heights, for which total execution time is minimized in each theoretical case, are close to the experimental minimum (Fig. 9).

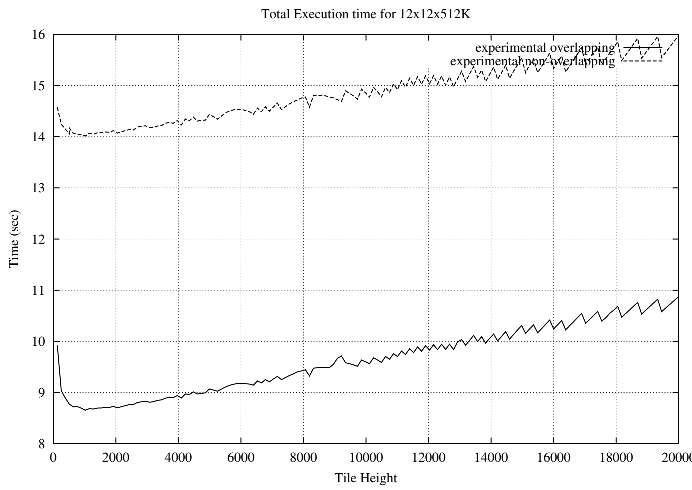

Fig. 6. Experimental Total Execution Times

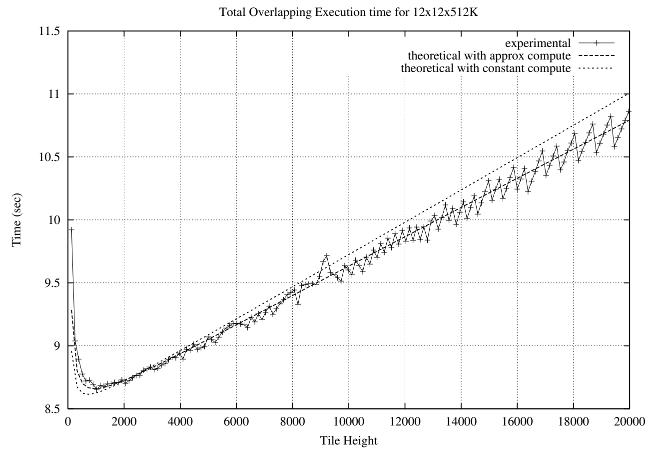

Fig. 8. Experimental Overlapping Total Execution Time compared to two Theoretical Times

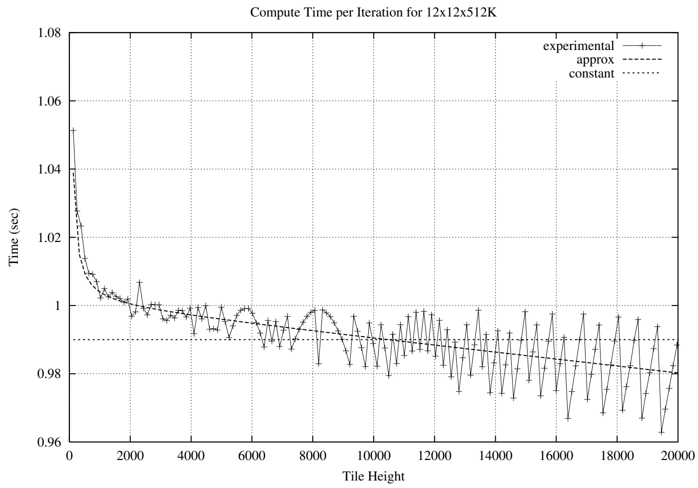

Fig. 7. Compute Time per Iteration

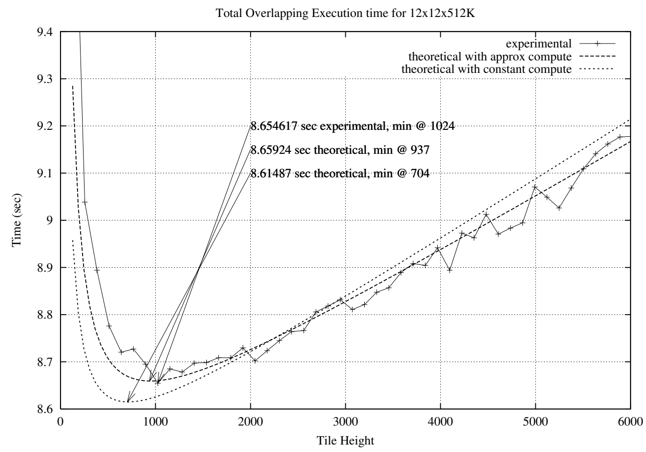

Fig. 9. Comparison of Experimental and Theoretical Minima of Overlapping Scheduling

VI. Conclusions – Future Work

In this paper we proposed a novel approach for the problem of minimizing the completion time for loop tiles. We reduced overall execution time 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.

However, if we could avoid all kernel initialization of DMA, then the initial DMA startup time could have been considerably reduced. User Level Networking together with SCI could provide with a very powerful communication architecture. At the moment there is no public available hardware VIA implementation for PCI-SCI cards, that uses DMA as communication mode. In fact, in [7], a VIA solution for SCI was presented, using PIO as the only available communication mode. Future work includes the combination of virtues provided by VIA and SCI to fully exploit the hardware of NICs for optimizing parallel applications with intensive nested loops.

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] M. Blumrich, *Network Interface for Protected, User-Level Communication*, PhD thesis, Princeton University, April 1996.

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

[4] F. O' Carroll, H. Tezuka, A. Hori and Y. Ishikawa, *The Design and Implementation of Zero Copy MPI Using Commodity Hardware with a High Performance Network,* Proceedings of the International Conference on Supercomputing, pp 243–249, Melbourne, Australia, 1998.

[5] M. Eberl, H. Hellwagner, B. Herland, M. Schulz, *SISCI - Implementing a Standard Software Infrastructure on an SCI Cluster,* 1st Workshop Cluster Computing, TU-Chemnitz, November 1997.

[6] Th. Eicken, A. Basu, V. Buch and W. Vogels, *U-Net: A User-Level Network Interface for Parallel and Distributed Computing,* Proceedings of the 15th ACM Symposium on Operating System Principles, Copper Mountain,Colorado, pp. 40–53, Dec 1995.

[7] K. Ghouas, K. Omang and 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, April 2001.

[8] 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.

[9] G. Goumas, A. Sotiropoulos and N. Koziris, *Minimizing Completion Time for Loop Tiling with Computation and Communication Overlapping,* in Proceedings of IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS'01), San Francisco, April 2001 (best paper award).

[10] H. Hellwagner, *The SCI Standard and Applications of SCI,* Scalable Coherent Interface-Architecture and Software for High-Performance Computer Clusters, edited by H. Hellwagner and A. Reinefield, Springer-Verlag, pp. 3–34, Sept. 1999.

[11] 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.

[12] 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.

[13] V. Karamcheti and A. Chien, *Software Overhead in Messaging Layers: Where Does the Time Go?,* Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 51–60, Ocotber 1994.

[14] D. Patterson, J.Hennessy, *Computer Organization & Design. The Hardware/Software Interface.* Morgan Kaufmann Publishers, pp.364–367, San Francisco, CA, 1994.

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

[16] F. Seifert, *Development of system software to integrate the Virtual Interface Architecture (VIA) into the Linux operating system kernel for optimized message passing,* Diploma Thesis, University of Technology Chemnitz, September 1999.

[17] M. Schulz, *SISCI Pthreads Implementation report* Technical Report, ESPRIT Project 23174 - SISCI, Deliverable 4.1.4, November 1999.

[18] M. Trams, W. Rehm, D. Balkanski and 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,pp. 4-15, Cancun/Mexico, May 2000.

[19] M. Trams, R. Schlosser, W. Rehm, *Design Choices and First Results of Our VIA-Capable PCI-SCI Bridge,* in proceedings of IEEE Intl. Conference on Cluster Computing (CLUSTER2000), Pages 349-350, Nov 28 - Dec 1, 2000, Chemnitz, Germany.

[20] *The Virtual Interface Specification. Version 1.0*, http://www.viarch.org.

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