Efficient Utilization of Memory Mapped NICs onto Clusters using Pipelined Schedules

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 describes the performance benefits attained using enhanced network interfaces to achieve low latency communication. We make use of DMA communication mode, to send data to other nodes, while the CPU performs useful calculations. Zero-copy communication is achieved through pinned-down physical memory regions, provided by NIC's driver modules. Our testbed concerns the parallel execution of tiled nested loops onto a Linux PC cluster with PCI-SCI NICs (Dolphin D330). Tiles are essentially exchanging data and should 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. The applied nonblocking schedule resembles a pipelined datapath where computation phases are overlapped with communication ones, instead of being interleaved with them. Experimental evaluation illustrates that when using enhanced communication features such as DMA transfers, memory-mapped interfaces and zero-copy mechanisms, overall performance is considerably improved compared to using conventional, CPU and kernel bounded, communication primitives.

1 Introduction

Modern high performance communication architectures allow new, low latency messaging protocols [7, 8, 9, 19] to provide the means for very efficient communication in clusters. Available bandwidth is constantly increasing, while there is a trend towards offloading host CPU from the burden of communication [19] through the use of bus mastering, DMA enabled NICs. In this way, CPU has more time to spend on useful application calculations.

When a (user level) process needs to access a conventional network interface, overall communication is delayed [14], since, through a system call, the OS switches to kernel level and assumes the copying of data from user areas to kernel areas for protection. Nevertheless, modern network technologies (i.e. SCI, Myrinet, etc.) are mitigating this startup latency with optimized communication protocols (i.e. VIA) with Zero-Copy [5], DMA support and User-Level [3] characteristics.

Not only these novel network interfaces are reducing the message startup latency, but they can also alleviate the communication burden from CPU. Current parallel applications should be rescheduled to exploit these enhanced features. The parallel execution of any computationally intensive code, containing nested loops, is a very good testbed for such enhanced communication architectures for clusters. Parallel loop execution requires for frequent synchronization points and extensive exchange of data between different nodes. Thus, loops are most suitable for being rescheduled, if we adopt zero-copy, DMA enabled, messaging features. The key issue is to mitigate communication overhead by efficiently controlling the computation to communication grain. When using enhanced network interfaces, the objective should also be to hide as much as possible this communication overhead, gaining extra cycles for useful computation, since the CPU is now disengaged.

In the past, many researchers presented methods for controlling the computation to communication grain for parallel loop execution. In order to alleviate the communication overhead, Irigoin and Triolet proposed supernode partitioning [13] of the iteration space, where neighboring iteration points are grouped together to build a larger computation node (tile) that can be atomically executed without any intervention. Data exchanges are also grouped and performed within a single message for each neighboring processor, at

the end of each atomic supernode execution.

In their paper Ramanujam and Sadayappan [16] gave a linear programming formulation for the problem of finding optimal tile shapes (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 [4]. 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 [20].

Nevertheless, all above approaches ignored 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. Hodzic and Shang [12] proposed a method to correlate optimal tile size and shape, based on overall completion time reduction. Their approach considers a straightforward time schedule, where 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 [10] 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 [15], 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. Experiments were conducted 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 the overlapping schedule is much better, however, in real world, hardware should provide support for it.

In this paper, we extend our work of [10], taking into consideration the new features (zero-copy and DMA) of the aforementioned novel network interfaces. We now use 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.

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 condition of controlling the computation to communication grain.

The rest of the paper is organized as follows: In Section 2, we present the modern communication architecture features used in clusters and elaborate on SCI approach. 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 a comparative experimental evaluation of both scheduling approaches using SCI primitives. Finally, in Section 6 we propose future work.

2 Clusters of Workstations – High Performance Features

Recent advances in high speed networks and improved microprocessor performance are making clusters of workstations an appealing vehicle for cost effective parallel computing. The trend in parallel computing is to move away from custom-designed platforms of the established HPC industry to general purpose systems consisting of loosely coupled components built up from single or multi-processor workstations or PCs.

The de-facto 100Mbps networking of commodity clusters can be a bottleneck for many applications, when scaling beyond a small number of nodes. In the last years, new networking technologies such as SCI [11], Myrinet and Gigabit Ethernet offer increased bandwidth and low startup latencies, which however, are never efficiently utilized by user applications. Therefore, high-performance clusters are introduced, which provide the computationally intensive applications with increased performance using special communication primitives, such as Zero-Copy Protocols and DMA transfers.

2.1 Zero-Copy Protocols

Network protocol stacks, such as TCP/IP, aggravate the communication procedure with the extra copying of data sent or received, to and from kernel space, respectively. As Fig. 1 depicts, when sending data from an application (user space) buffer to the network, data must be initially copied from the application buffer to kernel buffers. TCP, IP and

network headers must be added and then, as a packet, transferred to NIC's buffer for transmission. A respective procedure takes place when data reach the receiving node.

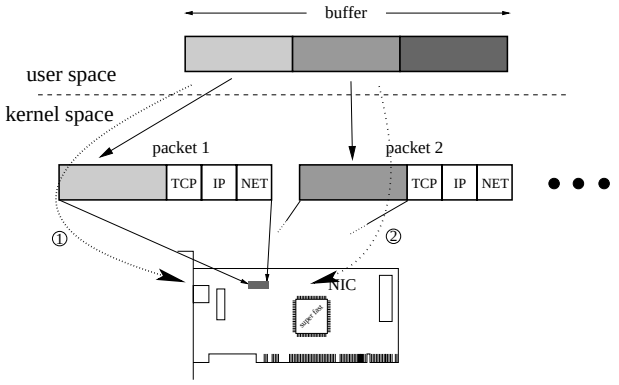

Figure 1. Single-Copy Protocol and packetization process

SCI Zero-Copy: The previous procedure is unavoidable when using customary network technologies, but could be avoided when novel communication technologies are used. SCI achieves Zero-Copy Communication, since it supports a Distributed Shared Memory approach, which is implemented using kernel area memory mapped regions for communication. 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 them directly to the imported memory segment (thus, no kernel copies). Data are transferred to the exporter's memory and communication is performed, without any kernel intervention. No other data processing is needed within each send, since SCI packetization and flow control is completely in hardware.

2.2 DMA transfers

Message data can be usually transferred to the NIC in two ways; Programmed I/O (PIO) mode and DMA mode. In PIO mode, CPU handles data transferring completely, word by word. For example, data transferring of 1Kwords involves the initial copying of these words from main memory to the NIC's buffers with the aid of 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 just programs the NIC's DMA engine with the information of which data to transfer from main memory and where to send it (Fig. 2). CPU is not used (or blocked from a program's perspective) during the transfer and can perform other (useful) tasks.

SCI DMA approach: The DSM feature of SCI allows the efficient use of its DMA capabilities. 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. 3). User is able to read/write that memory region like the ordinary memory regions returned by LIBC `malloc()`. Despite the fact that 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. On the contrary, even if the NIC in Fig. 1 was DMA enabled, a new DMA invocation should take place for each {data,TCP,IP,NET} packet, which would be time consuming.

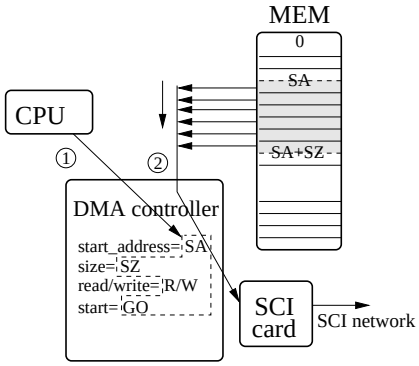

Figure 2. DMA or nonblocking send

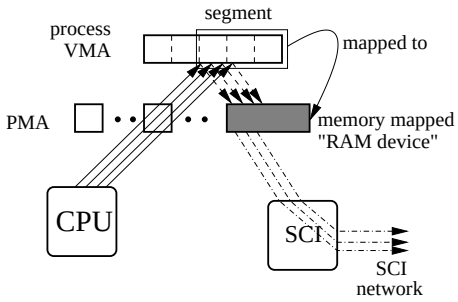

Figure 3. Locked and memory mapped "RAM device" for SCI communications

2.3 Combined approach

In this paper, we extend our work of [10] by proposing the use of a cluster of Linux PCs with SCI Network

Interface Cards (NIC) connected to the I/O PCI bus. We map process pages to physical pre-reserved pinned down memory regions, especially for DMA communications. The mapping procedure assists in implementing a zero copy mechanism, since accesses to mapped pages corresponds to accesses in physically contiguous kernel memory. In this way, we also satisfy DMA's prerequisite that data are located in contiguous physical memory, since most DMA engines can only access physical memory addresses.

We propose the use of DMA to remote write data to neighboring nodes, while the CPU is performing calculations. Every node reserves the aforementioned pinned down regions of memory as message buffers. Buffers serving as destinations need to be exported to the SCI global address space, in contradiction to those serving as source buffers. According to the SCI communication protocol, senders import exported destination segments to their virtual address space. DMA is then instructed to transfer data from a source buffer to an imported destination buffer. We use the SISCI API [6], [9], for all system calls related to SCI. Synchronization between communicating nodes is achieved through the SCI interrupt mechanism.

2.4 Preliminaries - Supernode Transformation

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

Dually, supernode transformation can be defined by n linearly independent vectors, which are the sides of the supernodes. Similar to matrix H, matrix P contains the side-vectors of a supernode as column vectors. It holds $P = H^{-1}$. The reader is referred to [10] for a thorough analysis.

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 (details can be found in [10]).

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 $\lfloor HD \rfloor < 1$ 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 define $V_{comp} = det(P) = g$, where g is called the *tile grain* or *size*.

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 [20],[4] 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 tile $j^S \in J^S$ will be executed at $t_{j^S} = \Pi j^S + t_0$ where $t_0 = -min \Pi i^S : i^S \in J^S$.

3 Non-overlapping Schedule

In [12], 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 V_{comp} (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:

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

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. 4).

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 computationally idle processor time.

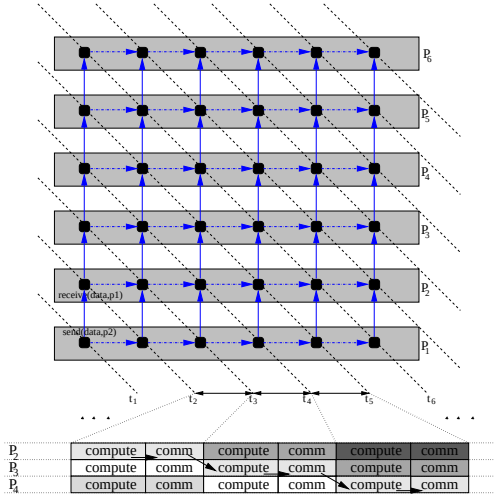

Figure 4. Non-overlapping Time Schedule

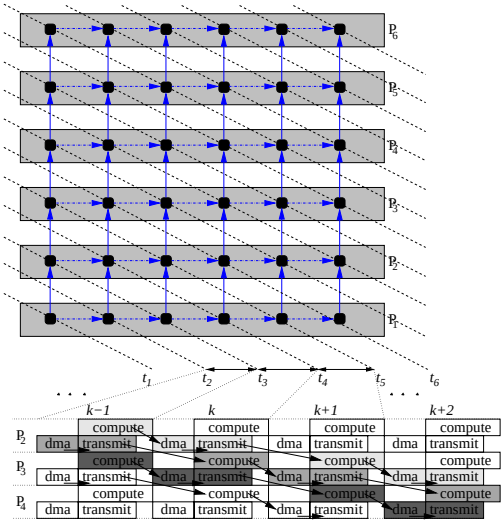

Figure 5. Overlapping Time Schedule

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. 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. 4).

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, within each time step, the node should send and receive 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 previ-

ous 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 the 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. 5 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. 5. They depict the actual flow of data between successive time steps (computes–dma setups –transmits) in a 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 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.

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

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

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 total 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_{sync} is required synchronization time between successive time steps.

5 Experiments

5.1 Execution Environment

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 high performance cluster features, we ran two type of experiments. The first one implements the overlapping algorithm and is compared to the second one which implements the non-overlapping algorithm.

The test application was implemented using C and the SISCI API [9]. Execution times were measured using **gettimeofday()** Linux system call.

5.2 Experimental Application

We experimented using the following 3-D loop:

```
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]);
```

The 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, $i = (0, \ldots, 8)$. During each time step, every processor in the ij plane with coordinates (i, j) receives from neighboring processors $(i-1, j)$ and $(i, j-1)$, computes and sends to processors $(i+1, j)$ and $(i, j+1)$.

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.

When evolving from a multicycle non-pipelined datapath to a pipelined one, we introduce pipeline registers among consecutive stages [15]. Similarly, when evolving from the non-overlapping schedule to the overlapping one, we added extra buffers for receiving and sending data, while transforming the data on the tile's cube (Fig. 6).

The above test application was executed using various `DIMX`×`DIMY`×`DIMZ` initial J^3 index spaces. Typical experimental values for `DIMX=DIMY` were 12 or 24 and for `DIMZ` were $256K$, $512K$, or $2048K$. We measured execution

Table 1. Internal Part of Program's Main Loop.

sequence of functions	respective SCI calls	Action performed
`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

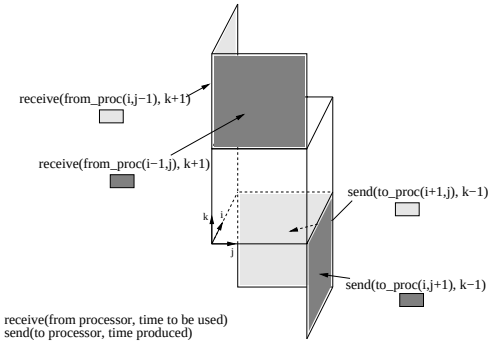

Figure 6. Extra Buffering for the Overlapping Case

times for the following overlapping and non-overlapping cases $12 \times 12 \times 512K$ (also in [17]), $24 \times 24 \times 256K$ and $24 \times 24 \times 2048K$.

From (2), the total (theoretical) execution time for the overlapping case is:

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

where in our case, because there are 3 processors in each dimension i and j, we have $\sum_{i \neq k} x_i = 2 \times (3-1)$. Since the initial space height is `DIMZ` and tile height z is the problem's variable, there are `DIMZ`$/z$ tiles in k dimension, so x_k is equal to `DIMZ`$/z - 1$.

The communication phase of a node with each neighboring node involves the receiving or sending of $x_i \times z$ floats or $4 \times x_i \times z$ bytes.

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

The total computation time for each application execution, either overlapping or non-overlapping, is constant and can be seen in Fig. 7 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. This is measured using the following code:

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

The computation time for the overlapping case, when including the SCI communication functions is shown in Fig. 7. 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.

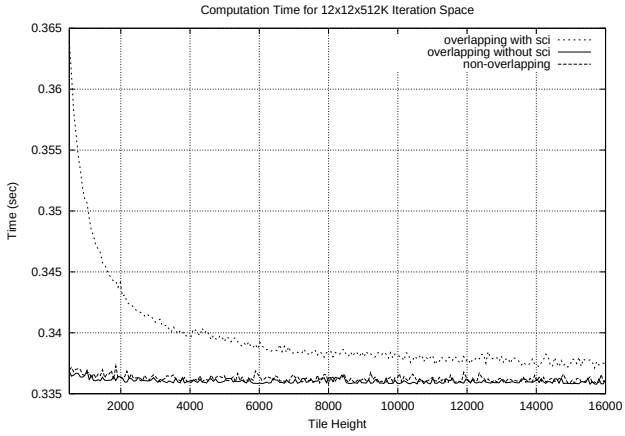

Figure 7. Comparison of Experimental Computation Times for $12 \times 12 \times 512K$

Overlapping and non-overlapping overall execution times for each problem are plotted in Figs. 8, 9 and 10. 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.

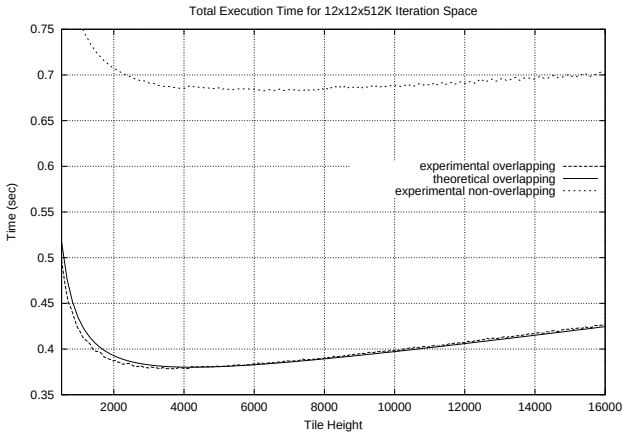

Figure 8. Experimental Total Execution Times for $12 \times 12 \times 512K$

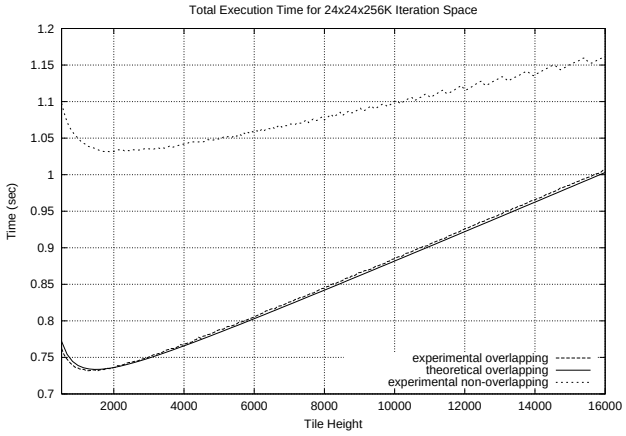

Figure 9. Experimental Total Execution Times for $24 \times 24 \times 256K$

In Fig. 11, the experimental result is compared to our analytical result derived from (3). The plot for the experimental time measured, is very close to the theoretical function. This is due to the fact that (3) includes a thorough and detailed analysis of actual possible time delay parameters. For example, from the minimum of each function in Fig. 11, it can be easily calculated that the difference between experimental minimum and theoretical minimum is nearly 0.2%, achieved in very close values of tile heights.

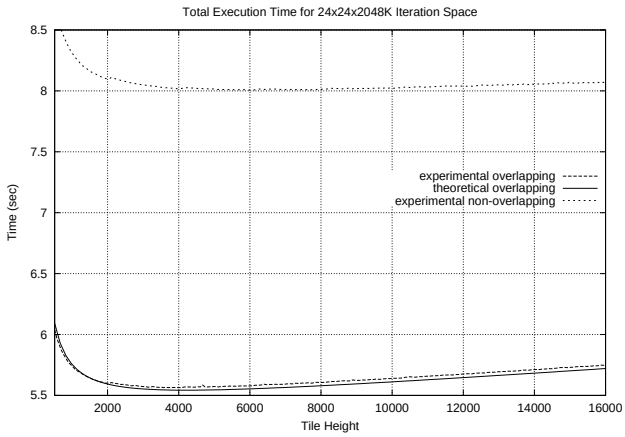

Figure 10. Experimental Total Execution Times for $24 \times 24 \times 2048K$

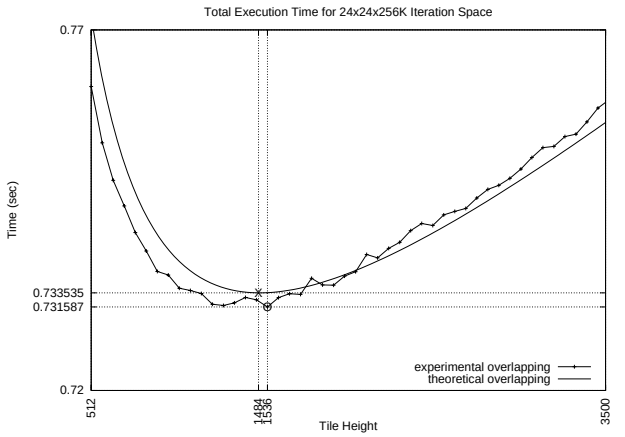

Figure 11. Comparison of Experimental and Theoretical Minima (Fig. 9 zoomed in)

6 Conclusions – Future Work

In this paper we described the performance benefits attained when using memory mapped network interfaces with zero-copy features and DMA engines in parallel loop execution. We reduced overall execution time by overlapping computation and communication for each tile execution. We have shown that the theoretically calculated overall time, following the optimal hyperplane transformation and the pipelined schedule, 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. Since DMA is initiated through calls from kernel level, we thus introduce extra overhead, which could increase overall execution time. User Level Network-

ing architectures, such as U-Net [7] and the ensuing VIA standard [19], allow for direct access of the NIC from virtual memory areas and without any kernel intervention (see [2], [3]).

At the moment there is no public available hardware VIA implementation for PCI-SCI cards, that uses DMA as communication mode. In fact, in [8], a VIA solution for SCI was presented, using PIO as the only available communication mode. It is obvious that we do need overlapping, so even avoiding kernel syscall overheads is not enough. In [18] 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.

References

[1] T. Andronikos, N. Koziris, G. Papakonstantinou, and P. Tsanakas. Optimal Scheduling for UET/UET-UCT Generalized N-Dimensional Grid Task Graphs. *Journal of Parallel and Distributed Computing*, 57(2):140–165, May 1999.

[2] M. Banikazemi, B. Abali, L. Herger, and 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*, to appear.

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

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

[5] 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. In *Proceedings of the International Conference on Supercomputing*, pages 243–249, Melbourne, Australia, 1998.

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

[7] T. Eicken, A. Basu, V. Buch, and 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*, pages 40–53, Copper Mountain, Colorado, Dec 1995.

[8] 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, Apr 2001.

[9] F. Giacomini, T. Amundsen, A. Bogaerts, R. Hauser, B. Johnsen, H. Kohmann, R. Nordstrom, and 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.

[10] 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, Apr 2001 (best paper award).

[11] H. Hellwagner. The SCI Standard and Applications of SCI. In H. Hellwagner and A. Reinefield, editors, *Scalable Coherent Interface (SCI): Architecture and Software for High-Performance Computer Clusters*, pages 3–34. Springer-Verlag, Sep 1999.

[12] E. Hodzic and W. Shang. On Supernode Transformation with Minimized Total Running Time. *IEEE Trans. on Parallel and Distributed Systems*, 9(5):417–428, May 1998.

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

[14] V. Karamcheti and 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*, pages 51–60, Oct 1994.

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

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

[17] A. Sotiropoulos, G. Tsoukalas, and 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, Oct 2001.

[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), help with IPDPS 2000*, pages 4–15, Cancun, Mexico, May 2000.

[19] The Virtual Interface Specification Version 1.0. http://www.viarch.org.

[20] J. Xue. Communication-Minimal Tiling of Uniform Dependence Loops. *Journal of Parallel and Distributed Computing*, 42(1):42–59, 1997.