The Journal of Supercomputing, 32, 197–226, 2005

Hyperplane Grouping and Pipelined Schedules: How to Execute Tiled Loops Fast on Clusters of SMPs

MARIA ATHANASAKI maria@cslab.ntua.gr
ARISTIDIS SOTIROPOULOS sotirop@cslab.ntua.gr
GEORGIOS TSOUKALAS gtsouk@cslab.ntua.gr
NECTARIOS KOZIRIS nkoziris@cslab.ntua.gr
PANAYIOTIS TSANAKAS panag@cslab.ntua.gr
National Technical University of Athens, School of Electrical and Computer Engineering, Computing Systems Laboratory, Zografou Campus, Zografou 15773, Greece

Abstract. This paper proposes a novel approach for the parallel execution of tiled Iteration Spaces onto a cluster of SMP PC nodes. Each SMP node has multiple CPUs and a single memory mapped PCI-SCI Network Interface Card. We apply a hyperplane-based grouping transformation to the tiled space, so as to group together independent neighboring tiles and assign them to the same SMP node. In this way, intranode (intragroup) communication is annihilated. Groups are atomically executed inside each node. Nodes exchange data between successive group computations. We schedule groups much more efficiently by exploiting the inherent overlapping between communication and computation phases among successive atomic group executions. The applied non-blocking schedule resembles a pipelined datapath, where group computation phases are overlapped with communication ones, instead of being interleaved with them. Our experimental results illustrate that the proposed method outperforms previous approaches involving blocking communication or conventional grouping schemes.

Keywords: supernodes, loop tiling, tile grouping, pipelined schedules, hyperplanes

1. Introduction

Modern high performance communication architectures allow new, low latency messaging protocols [15, 18–20] to provide the vehicle of very efficient communication in clusters. Available bandwidth is constantly increasing, while there is a trend towards offloading host CPU from the burden of communication [15] 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 [31], 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 [13,44], DMA support and User-Level [10] characteristics. With the advent of programmable NICs, many aspects of protocol processing can be off loaded from user space to the NIC, leaving the host processor to dedicate more cycles to the application [11,42]. In [4,9] various communication systems (i.e. Generic Active Messages, Virtual Memory Mapped Communication over a Myrinet architecture) are tested, evaluated and compared to each-other, while in [14, 35, 46] the effects of communication latency and bandwidth are explored.

Not only these novel network interfaces are reducing the message startup latency, but they can also alleviate the communication burden from the 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. Of course, in order to further reduce the communication overhead, one should take into consideration the message latencies imposed by the network. The utilized network topology remains always another critical issue, apart from efficient scheduling. As shown in [3, 8], using a "smart" topology, like a reconfigurable multi-ring network, with low diameter, low degree of connectivity and at the same time reasonable algorithmic and hardware complexity, we can further reduce the latency of transmitted messages.

In the past, many researchers presented methods for controlling the computation to communication grain for parallel loop execution. In order to reduce the communication overhead, as far as fine grain parallelism is concerned, several methods have been proposed to group together neighboring chains of iterations [32, 40], while preserving the optimal hyperplane schedule [17, 41, 45]. As far as coarse grain parallelism is concerned, Irigoin and Triolet proposed supernode partitioning [29] 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. Later, Ramanujam and Sadayappan in [38] 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 problem of determining the optimal shape was surveyed, and more accurate conditions were also given by others, as in [12, 16, 25–28, 47].

Hodzic and Shang [24] 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 [22] an alternative method for the problem of scheduling the tiles to single CPU nodes was proposed. Each atomic tile execution involves a communication and a computation phase and this is repeatedly done for all time planes. This sequence of communication and computation phases is compacted, by overlapping them. The proposed method acts like enhancing the performance of a processor's datapath with pipelining [37], 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, the computation grain is adjusted to

make room for this overhead and try to overlap with all communication, which can be done in parallel.

As far as tile execution to symmetric multiprocessors (SMPs) is concerned, in [34] Manjikian and Abdelrahman have presented a method of allocating tiles to the CPUs of a multiprocessor so as to minimize the required communication among them. In order to achieve this minimization, they assign to a processor neighboring tiles and they modify the wavefront in order to correspond just one tile to each processor during each time step.

Although, in [22], we first proposed the notion of pipelined schedule for clusters of single CPU nodes, the merit of using advanced communication architectures to achieve true overlapping was accentuated in [43]. In this paper we present a complete framework which extends the overlapping (pipelined) schedule, for multiple-CPU SMP nodes. We group together neighboring tiles along a hyperplane. Hyperplane-grouped tiles are concurrently executed by the CPUs of the same SMP node. In this way, we eliminate the need for tile synchronization and communication between intranode CPUs. As far as scheduling of groups is concerned, we take advantage of the overlapping schedule of [22] in order to "hide" each group communication volume within the respective computation volume.

In order to evaluate the proposed method, we use a Linux cluster of dual SMP nodes, interconnected using the same networking technology (PCI-SCI Dolphin D330 NICs), which supports shared memory programming either through PIO messaging or through DMA. We are using the NICs' kernel-level DMA support for messaging. We compare our method versus blocking schedules and vertical grouping of neighboring tiles along a specific dimension. Vertically grouped tiles are assigned to the same node, and an optimal hyperplane time schedule is applied. However, this imposes additional intranode synchronization delays. All experimental results show that when the hyperplane grouping of tiles together with the overlapping schedule are applied, the overall completion time is considerably reduced, under the condition of controlling the computation to communication grain.

The rest of this paper is organized as follows: In Section 2 we describe some hardware concepts used in our experiments, as well as basic terminology used throughout the paper and definitions of loop tiling. A short description of the non-overlapping and the overlapping schemes is given in Section 3. In Section 4 we supply an algorithm for the application of the overlapping scheme on clusters of SMP nodes and we investigate the resulting time schedule. In Section 5 we describe the experiments executed on a cluster of SMPs using PCI-SCI Network Interface cards. Finally, in Section 6 we summarize our results.

2. Background concepts

2.1. Hardware 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.

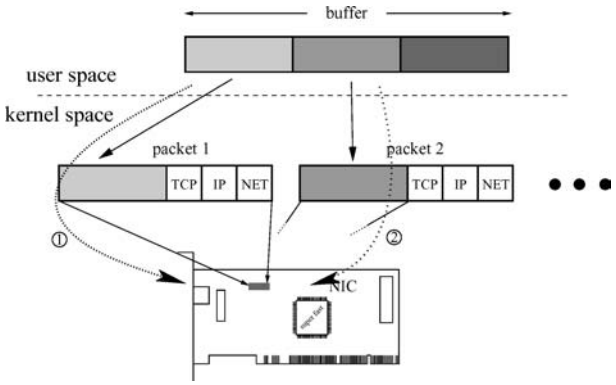

Figure 1. Single-Copy Protocol and packetization process.

The de-facto 100 Mbps networking of commodity clusters can be a bottleneck for many applications, when scaling beyond a small number of nodes. The last years, new networking technologies such as SCI [23], 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.

2.1.1. 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 Figure 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.

The previous sequence of actions is unavoidable when using legacy 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.

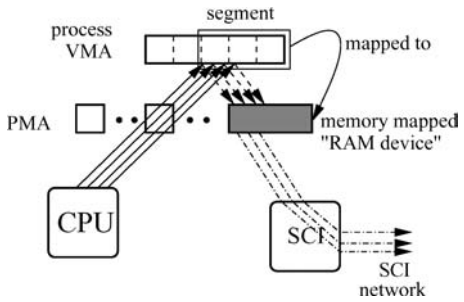

Figure 2. Locked and memory mapped "RAM device" for SCI communications.

2.1.2. 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 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. CPU is not used (or blocked from a program's perspective) during the transfer and can perform other (useful) tasks.

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. The allocated memory is first "pinned down" and then mapped to user's virtual memory (Figure 2). 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 Figure 1 was DMA enabled, a new DMA invocation should take place for each {data,TCP,IP,NET} packet, which would be time consuming.

2.2. Algorithmic model-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 j1 = l1 TO u1 DO
  ...
  FOR jn = ln TO un DO
    Loop Body
  ENDFOR
  ...
ENDFOR
```

where: l_i and u_i are affine functions of the outer loop indices.

Throughout the paper the following notation is used: N is the set of natural numbers and n is the number of nested FOR-loops of the algorithm. $J^n \subset Z^n$ is the set of indices: $J^n = \{j(j_1, \ldots, j_n) | j_i \in Z \wedge l_i \leq j_i \leq u_i, 1 \leq i \leq n\}$. Each point in this n-dimensional integer space is a distinct instantiation of the loop body. A dependence vector is denoted $d_i = (d_{i1}, \ldots, d_{in})^T, 1 \leq i \leq q$. The Dependence Matrix D of an algorithm is the concatenation of all dependence vectors of this algorithm: $D = [d_1 | d_2 | \ldots | d_q]$.

In a supernode or tiling transformation, the Iteration Space J^n is partitioned into identical n-dimensional parallelepiped areas (tiles or supernodes) formed by n independent families of parallel hyperplanes. Tiling 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, tiling transformation can be defined by n linearly independent vectors, which are the sides of the tiles. Similar to matrix H, matrix P contains the side-vectors of a tile as column vectors. It holds $P = H^{-1}$.

Formally, tiling transformation is defined as follows:

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

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. The Tile Space J^S and the Tile Dependence matrix D^S are defined as follows: $J^S = \{j^S \mid j^S = \lfloor Hj \rfloor, j \in J^n\}$, $D^S = \{d^S \mid d^S = \lfloor H(j_0 + d) \rfloor, d \in D, j_0 \in J^n | \lfloor Hj_0 \rfloor = 0\}$ where j_0 denotes the index points belonging to the first complete tile starting from the origin of the Iteration Space J^n. The Tile Space can be also written as $J^S = \{j^S(j_1^S, \ldots, j_n^S) \mid j_i^S \in Z \wedge l_i^S \leq j_i^S \leq u_i^S, 1 \leq i \leq n\}$, where l_i^S, u_i^S can be directly computed from the functions $l_1, \ldots, l_n, u_1, \ldots, u_n$ and the matrix H, as described in [1, 21]. 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 $HD \geq 0$ (see [29,38]). This ensures that tiles are atomic and that the initial execution order is preserved. In the opposite case, any execution order of tiles would result in a deadlock. In this paper, as in [22], we assume that all dependence vectors are smaller than the tile size, thus they are entirely contained in each tile's area, which means that $|HD| < 1$ [48], or, alternatively, that the tile 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.

3. Non-overlapping vs. overlapping schedule

In [24], 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: Firstly, the optimal tiling matrix H is determined and then it is applied to the original Iteration Space. The resulting Tile 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 $T = \wp(T_{\text{comp}} + T_{\text{comm}})$, where $\wp$ is the number of time hyperplanes needed to execute the algorithm, T_{comp} the execution time of a tile and T_{comm} the communication time. T_{comm} can be expressed as the communication startup latency (T_{startup}), and a factor expressing the transmission time (T_{transmit}). That is $T_{\text{comm}} = T_{\text{startup}} + T_{\text{transmit}}$.

Therefore, the overall parallel loop execution consists of atomic computations of tiles interleaved with communication for the transmission of the results to neighboring processors. Since Tile Space J^S has only the unitary dependence vectors (see Section 2.2), the optimal linear time schedule can be easily proved to be: $\Pi = [1 \;\; 1 \ldots 1]$. In Figure 3, the **non-overlapping schedule** is shown for a Tile Space using six processors. We see that the overall schedule has computation subphases interleaved with communication ones.

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. It would be ideal if a node was able to receive,

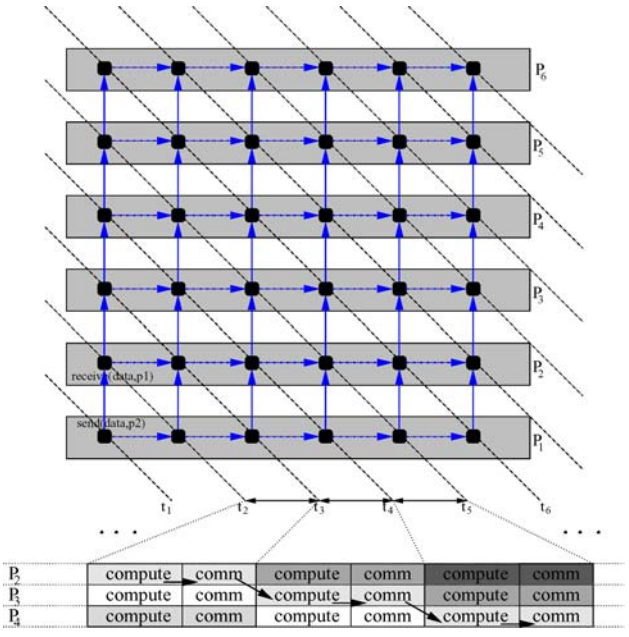

Figure 3. Non-overlapping Time Schedule.

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. In fact, even some part of the non-blocking communication needs the CPU, i.e. DMA initialization. Nevertheless, all subsequent data transferring actions 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 receive data, then compute and finally send the results to be used at the next time step by its neighbor. 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 would be for a processor to receive data from all neighbors to use them at $k+1$ time step, send data produced at previous time step $(k-1)$ and compute its results (Figure 4). In this case, every processor computes a tile and receives+sends data produced in the previous step or needed in next one, respectively.

In Figure 4 the **overlapping schedule** is shown. Note the arcs shown in Figure 4. They depict the actual flow of data between successive time steps (computes-sends-receives) in a

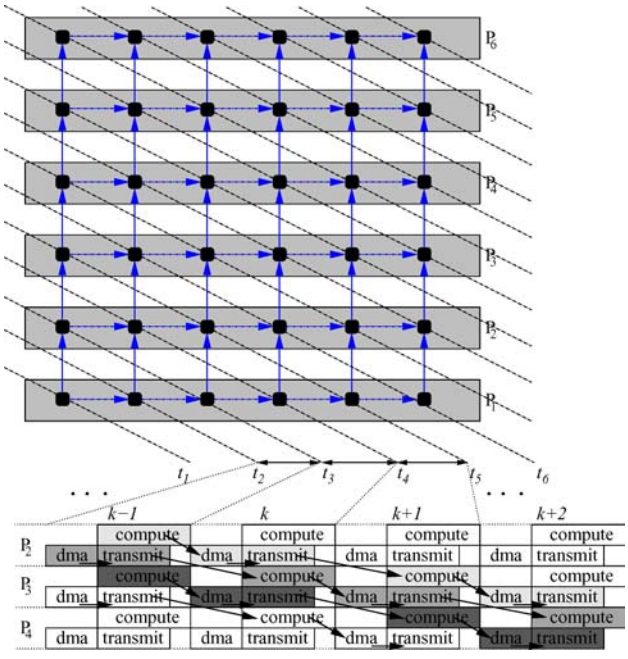

Figure 4. Overlapping Time Schedule.

pipelined way. The outcome of this schedule is to have successive computations overlapped with communication phases, thus 100% processor utilization. A more detailed description of this schedule can be found in [22] and [43]

If we implement the overlapping of computation and communication, then we will have the following scheme: A processor first initiates all the non-blocking send operations and then performs the actual atomic tile computations. While the processor performs computations, the NIC is receiving data from neighbors and sending previously computed data to others as well. When communication work is finished, the processor receives an interrupt.

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.

According to the previous properties, the total execution time for the overlapping schedule, as deduced from Figure 4, is given by:

$$T_{\text{overlap}} = \wp(t_{\text{start\_dma}} + \max(t_{\text{comp}}, t_{\text{comm\_dma}}) + t_{\text{synchro}}), \tag{1}$$

where $\wp$ is the number of execution steps of the resulting algorithm. The time needed to initiate the DMA engine is $t_{\text{start\_dma}}$, t_{comp} is the tile execution time, $t_{\text{comm\_dma}}$ is the communication time which can be overlapped with computation and t_{synchro} is the required synchronization time between successive time steps.

Since the concept of overlapping of actions is crucial, it should be noted that the actions initiated by a non-blocking call are overlapped with the actions initiated by calls following the non-blocking 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.

4. Application of the overlapping schedule to SMP nodes

Let us consider the following scenario: A 2-dimensional nested loop to be executed onto a cluster of 3 single CPU nodes. We tile the Iteration Space of the algorithm and assign each row of tiles to a CPU node. We should select the size and shape of tiles so that the Iteration Space is partitioned into 3 rows of tiles (since 3 CPUs are available). Then, the tiles can be computed using either the overlapping, or the non-overlapping schedule presented in Section 3.

If, instead of 3 single CPU nodes, we have 3 SMP nodes, with 2 CPUs each, then we can split each tile into two subtiles and assign each subtile to one of the CPUs of the corresponding SMP node, as indicated in Figure 5. Equivalently, we may tile the initial Iteration Space, selecting the size of tiles so as to get six rows of tiles. Then, we assign a row of tiles to each CPU of the SMP nodes and group together neighboring tiles assigned to the same SMP node, as in Figure 5. It is obvious that the tiles grouped together by this scheme cannot be simultaneously executed, unless they are split into subtiles.

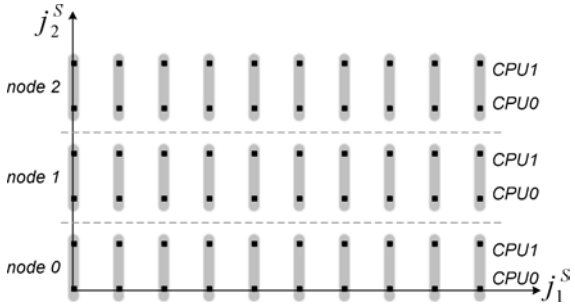

Figure 5. Vertical grouping.

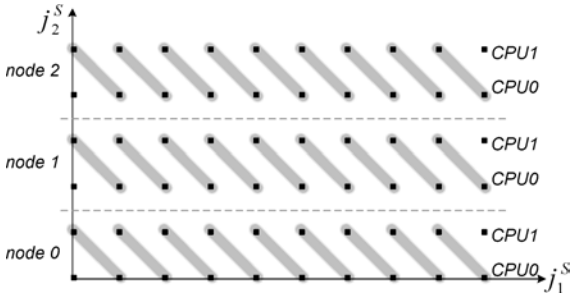

Figure 6. Hyperplane grouping.

A more efficient scheme can be obtained if we group the tiles assigned to the same SMP nodes as indicated in Figure 6. Then, both tiles belonging to the same group can be simultaneously executed by the CPUs of an SMP node.

4.1. Grouping transformation

In order to generate an appropriate time schedule, we need to group together the tiles of J^S that will be concurrently executed by the CPUs of the same SMP node. So, we further apply an additional supernode transformation to the Tile Space J^S. Thus, from the Tile Space J^S we produce the *Group Space* $J^G = \{j^G | j^G = \lfloor H^G j^S \rfloor, j^S \in J^S\}$. This *grouping transformation* is defined by the $n \times n$ non-singular matrix H^G. In correspondence to the tiling matrix H, we call the $n \times n$ matrix H^G as *grouping matrix*. The $n \times n$ matrix $P^G = (H^G)^{-1}$ is called *inverse grouping matrix*. The matrix P^G should consist only of integer elements and its column-vectors are parallel and equal in size to the edges of a group-hyperparallelepiped in J^S.

In order to be valid, a grouping transformation should preserve the constraint of atomicity of groups ($H^G D^S \geq 0$ in correspondence to $H D \geq 0$ for tiling). In addition, since within a group all tiles are concurrently executed by the CPUs of an SMP node, in order to preserve data consistency, there should be no direct or indirect dependence among them.

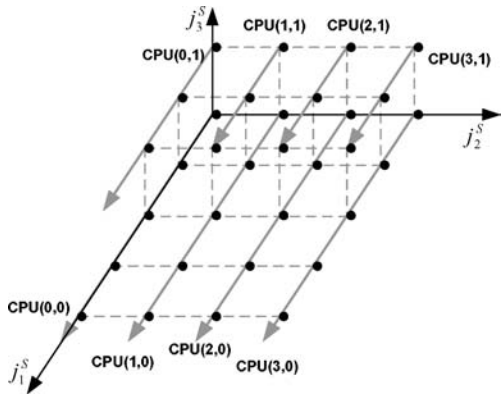

Figure 7. Set of tiles assigned to an SMP node.

4.2. Intuition of our algorithm

Let us consider a 3-dimensional Tile Space J^S. We want to assign all tiles along dimension j_1^S to the same CPU of an SMP node. Since all CPUs within a node have access to the shared memory, we are assigning neighboring rows of tiles, which exchange data, to the CPUs of the same node. In this way, the part of the Tile Space assigned to a node will have the rectangular form depicted in Figure 7.

We seek for an appropriate transformation matrix that will group together the tiles of Figure 7 which can be executed simultaneously by different CPUs. So we shall group together the tiles that belong to the same plane which is perpendicular to the vector (1, 1, 1), as indicated in Figure 8. In the sequel, we shall call this grouping scheme as "**hyperplane grouping**". On the contrary, any other grouping scheme along a specific dimension, such as the one presented in Figure 5, will be called "**vertical grouping**". It is obvious that vertical grouping imposes additional synchronization overhead due to intragroup tile dependencies.

The column-vectors of the inverse grouping matrix P^G define a hyper-parallelepiped (in general) that contains the group tiles, similar to the way the columns of P define a tile. So, as shown in Figure 9, the appropriate vectors are $p_1^G = (1, 0, 0)$, $p_2^G = \lambda(-1, 1, 0)$ and $p_3^G = \mu(-1, 0, 1)$. (In Figures 7–9 it holds $\lambda = 4$, $\mu = 2$.) Thus, the appropriate inverse

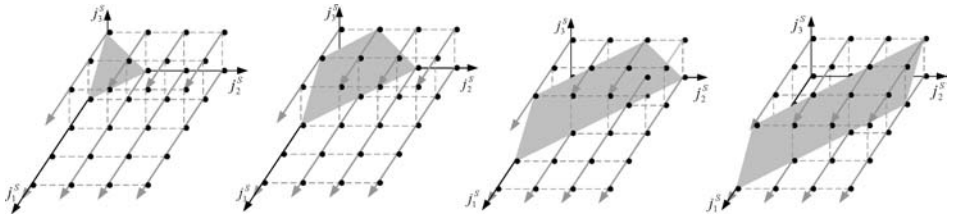

Figure 8. Groups of tiles executed simultaneously in an SMP node.

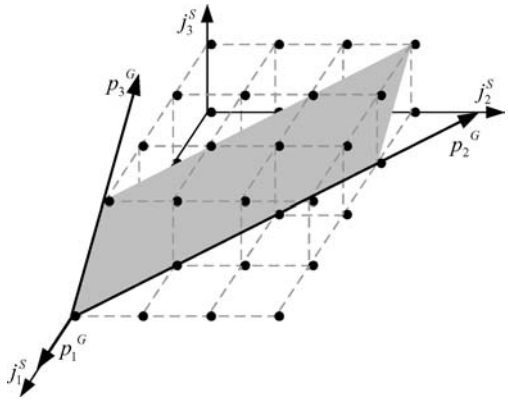

Figure 9. Constructing inverse grouping matrix.

grouping matrix is

$$P^G = \begin{bmatrix} 1 & -\lambda & -\mu \\ 0 & \lambda & 0 \\ 0 & 0 & \mu \end{bmatrix},$$

where $\lambda, \mu \in N$. Thus, the maximum number of tiles grouped together will be $\lambda \times \mu$ and this product must be equal to the number of CPUs inside a node, so as to assign one tile to each CPU during each time step.

4.3. Determining P^G according to the number of CPUs within a node

Consider now the general case, where we have an n-dimensional tiled Iteration Space and a cluster of SMP nodes, each with m processors inside. Our objective is to assign the tiles of J^S along the 1-st dimension to the same CPU of an SMP node. Let us assume that the natural number m can be written as $m = m_2 \times m_3 \times \cdots \times m_n$, where $m_2, m_3, \ldots, m_n \in N$. Then, we select the grouping matrices to be

$$P^G = \begin{bmatrix} 1 & -m_2 & \ldots & -m_n \\ 0 & m_2 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & m_n \end{bmatrix}, \quad H^G = (P^G)^{-1} = \begin{bmatrix} 1 & 1 & \ldots & 1 \\ 0 & \frac{1}{m_2} & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & \frac{1}{m_n} \end{bmatrix}. \quad (2)$$

The maximum number of tiles contained inside a group is $det(P^G) = m$, exactly equal to the number of CPUs inside each SMP node.

Theorem 1 *Matrix H^G, defined by formula* (2)*, defines a legal grouping transformation, according to our algorithmic model.*

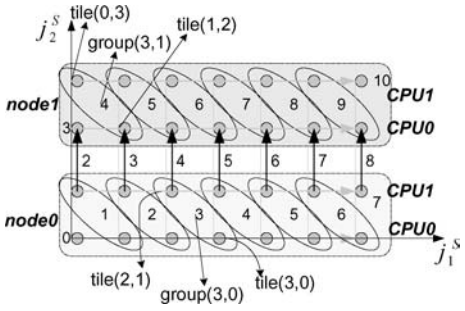

Figure 10. 2D example.

Proof: In order to prove that H^G defines a legal grouping transformation, it suffices to prove that $H^G D^S \geq 0$, where D^S is the dependence matrix of the Tile Space J^S and that any two tiles ($j^S, j^{S'} \in j^G$) within the same group are independent. We have assumed (see Section 2.2) that the dependence matrix D^S contains only 0's and 1's. Consequently, the first condition is apparently valid. In order to prove the second condition, we assume that the dependence matrix D^S is equal to the unitary matrix. Even if there is a dependence vector with more than one 1's, it is the sum of more than one unitary dependence vectors. So it will be included in the following proof as an indirect dependence:

If tiles $j^S, j^{S'} \in J^S$ belong to the same group j^G, then it holds that: $\lfloor H^G j^S \rfloor = \lfloor H^G j^{S'} \rfloor \Rightarrow j_1^S + j_2^S + \cdots + j_{n-1}^S + j_n^S = j_1^{S'} + j_2^{S'} + \cdots + j_{n-1}^{S'} + j_n^{S'}$. In addition, if there is a direct or an indirect dependence from j^S to $j^{S'}$, it holds that $j^{S'} = j^S + \sum_{i=1}^{n} \lambda_i d_i$, where $\lambda_i \in N$ and d_i is a unitary dependence vector. Thus, $j_i^{S'} = j_i^S + \lambda_i, i = 1, \ldots, n$. Therefore, the equality $j_1^S + j_2^S + \cdots + j_{n-1}^S + j_n^S = j_1^{S'} + j_2^{S'} + \cdots + j_{n-1}^{S'} + j_n^{S'}$ can be rewritten as follows: $\lambda_1 + \lambda_2 + \cdots + \lambda_n = 0$. As $\lambda_1, \ldots, \lambda_n \in N$, it holds that $\lambda_1 = \cdots = \lambda_n = 0$. Consequently, there is no direct or indirect dependence between two tiles belonging to the same group $j^G \in J^G$ and all tiles of a group in J^G can be computed simultaneously by the CPUs of an SMP node. Thus, the above grouping transformation is valid according to our algorithmic model. □

Example 1 We have a cluster of SMP nodes with 2 CPUs and a NIC each. We assume a 2-dimensional rectangular Tile Space J^S. Let us assign the tiles along dimension j_1^S to the same CPU, as indicated in Figure 10 by the grey arrows. The CPUs of the same SMP node will process two neighboring rows of tiles.

Then, during the time step $t = 0$, the CPU-0 of the SMP node0 computes tile $(0, 0)$. During the time step $t = 1$, the CPU-0 of node0 computes tile $(1, 0)$, while the CPU-1 of the same SMP node computes tile $(0, 1)$. Similarly, during the time step $t = 2$, the CPU-0 computes tile $(2, 0)$, while the CPU-1 computes tile $(1, 1)$. At the same time, the data computed in tile $(0, 1)$, which are necessary for the computation of tile $(0, 2)$, can be sent to node1. During the time step $t = 3$, the CPUs of node0 can continue the execution as above, while the CPUs of node1 start executing the same routine with the rows of tiles $(\bullet, 2)$ and $(\bullet, 3)$.

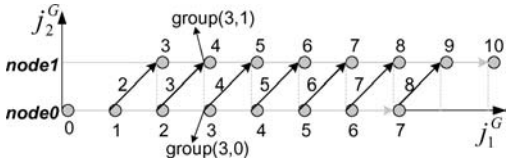

Figure 11. Group Space for the 2D example.

In order to construct a time schedule for this example, we group together the tiles that should be concurrently executed by the same SMP node. In particular, we perform *grouping* to the Tile Space J^S, as indicated in Figure 10 and derive the Group Space J^G (Figure 11). The appropriate grouping matrices, according to the formula (2), for this case are

$$P^G = \begin{bmatrix} 1 & -2 \\ 0 & 2 \end{bmatrix}$$

and

$$H^G = (P^G)^{-1} = \begin{bmatrix} 1 & 1 \\ 0 & \frac{1}{2} \end{bmatrix}.$$

In this way, tiles $(1, 0)$ and $(0, 1)$ which, as we have already mentioned, are simultaneously executed by the same SMP node, are grouped together in $j^G = \lfloor H^G(1, 0)^T \rfloor = \lfloor H^G(0, 1)^T \rfloor = (1, 0)^T$. Similarly, tiles $(2, 0)$ and $(1, 1)$ are grouped together in $j^G = (2, 0)^T$. In Figures 10–11, the time step, when each group will be computed, is shown, together with the time step, where each data transfer will take place. It can be easily deduced that a group $j^G = (j_1^G, j_2^G) \in J^G$ will be executed during the time step $t(j^G) = j_1^G + j_2^G$ in the SMP node j_2^G. Therefore, the linear time scheduling vector for this example is $\Pi^G = (1, 1)$.

4.4. Linear time schedule

Theorem 2 *The appropriate linear time scheduling vector for the Group Space derived by grouping, as defined in formula* (2), *is* $\Pi^G = (1, 1, \ldots, 1)$.

Proof: Applying the grouping transformation defined by formula (2), the 1-st column-vector of the dependence matrix $D^S = I$ is transformed to the vector $d_n^{G'} = H^G d_n^S = (1, 0, \ldots, 0)^T$. In addition, the j-th column-vector of the dependence matrix $D^S = I$, $j = 2, \ldots, n$, is transformed to the vector $H^G d_j^S = (1, 0, \ldots, 0, \frac{1}{m_j}, 0, \ldots, 0)^T$. It imposes the dependencies $(1, 0, \ldots, 0, \lfloor \frac{1}{m_i} \rfloor, 0, \ldots, 0)^T = (1, 0, \ldots, 0, 0, 0, \ldots, 0)^T$ and $(1, 0, \ldots, 0, \lceil \frac{1}{m_j} \rceil, 0, \ldots, 0)^T = (1, 0, \ldots, 0, 1, 0, \ldots, 0)^T$ in the Group Space. Thus, the

dependence matrix of the Group Space can be written as:

$$D^G = \begin{pmatrix} 1 & 1 & \dots & 1 & 1 \\ 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & \dots & 0 & 1 \end{pmatrix}.$$

We are searching for an appropriate linear time scheduling vector $\Pi^G = (\pi_1^G, \dots, \pi_n^G)$ such that each group $j^G \in J^G$ is computed during the time step $t = \Pi^G j^G$. Consider the last $(n-1)$ coordinates of a group indicating which SMP node of the cluster will execute this group. Then, the groups $j^G = (j_1^G, \dots, j_n^G)$ and $j^{G'} = (j_1^G + 1, j_2^G, \dots, j_n^G)$ will be successively computed within the same SMP node. There is a dependence between them, as indicated by the first column of D^G, but there is no need for a communication step between their successive computation steps, because the necessary data are already located in the local shared memory of the SMP node. Consequently, their time distance $\Pi^G j^{G'} - \Pi^G j^G = \pi_1^G$ may be equal to 1. Thus $\pi_1^G = 1$. In addition, the i-th column of D^G $(i = 2, \dots n)$ imposes a dependence between the groups $j^G = (j_1^G, \dots, j_n^G)$ and $j^{G'} = (j_1^G + 1, j_2^G, \dots, j_{i-1}^G, j_i^G + 1, j_{i+1}^G, \dots, j_n^G)$. These groups are executed in neighboring SMP nodes, thus a communication step is required between their computation steps. It means that their time distance $\Pi^G j^{G'} - \Pi^G j^G = \pi_1^G + \pi_i^G$ must be equal to 2. Consequently, $\pi_i^G = 1$, $i = 2, \dots, n$. So, the vector $\Pi^G = (1, 1, \dots, 1)$ is selected for the linear time scheduling of our Group Space J^G. □

Notice that, in [22, 43], for the single CPU pipelined schedule, Π was $(1, 2, \dots, 2)$ according to the UET-UCT theory [2]. In other words, the optimal overlapping schedule could be achieved when we had equal computation to communication times, so that all communication could be hidden (overlapped) with the computation phase. Nevertheless, in the SMP case presented here, the labeling of coordinates of groups, that is the grouping transformation P^G slightly skews the space (see Figure 10 and the resulting Group Space in Figure 11, the relative positions of groups (3, 0) and (3, 1)). So the optimal overlapping schedule is achieved by $(1, 1, \dots, 1)$. Notice, also, that this scheduling vector is not the same with Hodzic's [24] scheduling vector, since we are now referring to groups, while Hodzic was scheduling tiles.

4.5. Assigning tiles to CPUs

For node labeling reasons, consider that the available SMP nodes form a virtual $(n-1)$-dimensional mesh. Thus, each node is identified by a $(n-1)$-dimensional vector. Note, however, that it is not a physical layout restriction, but a convention to give each node a unique tag. Then, the last $(n-1)$ coordinates of a group indicate the SMP into which it will be executed. The first coordinate affects only the time of its execution. Thus, a tile

$j^S = (j_1^S, \ldots, j_n^S)$, belonging to group $j^G = (j_1^G, \ldots, j_n^G)$, will be executed in node $(j_2^G, \ldots, j_n^G) = (\lfloor \frac{j_2^S}{m_2} \rfloor, \ldots, \lfloor \frac{j_n^S}{m_n} \rfloor)$.

Similarly, inside each SMP we consider a $(n-1)$-dimensional CPU virtual mesh containing labels $\{\vec{cpu} \in Z^{n-1} \mid 0 \leq cpu_x < m_{x+1}, 1 \leq x \leq n-1\}$. Then, a tile $j^S = (j_1^S, \ldots, j_n^S)$ will be executed by CPU $(j_2^S \% m_2, \ldots, j_n^S \% m_n)$ of SMP node $(\lfloor \frac{j_2^S}{m_2} \rfloor, \ldots, \lfloor \frac{j_n^S}{m_n} \rfloor)$. So, apparently, only tiles with the same coordinate j_1^S will be assigned to the same CPU of the same node.

4.6. *Generalization: Grouping along an arbitrary dimension of J^S*

If we want to assign the iterations along the i-th dimension of J^S to the same CPU of an SMP node, then it can be similarly proven that the appropriate grouping matrices are

$$P^G = \begin{bmatrix} m_1 & \ldots & 0 & 0 & 0 & \ldots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & & \vdots \\ 0 & \ldots & m_{i-1} & 0 & 0 & \ldots & 0 \\ -m_1 & \ldots & -m_{i-1} & 1 & -m_{i+1} & \ldots & -m_n \\ 0 & \ldots & 0 & 0 & m_{i+1} & \ldots & 0 \\ \vdots & & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \ldots & 0 & 0 & 0 & \ldots & m_n \end{bmatrix},$$

$$H^G = \begin{bmatrix} \frac{1}{m_1} & \ldots & 0 & 0 & 0 & \ldots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & & \vdots \\ 0 & \ldots & \frac{1}{m_{i-1}} & 0 & 0 & \ldots & 0 \\ 1 & \ldots & 1 & 1 & 1 & \ldots & 1 \\ 0 & \ldots & 0 & 0 & \frac{1}{m_{i+1}} & \ldots & 0 \\ \vdots & & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \ldots & 0 & 0 & 0 & \ldots & \frac{1}{m_n} \end{bmatrix}, \tag{3}$$

where $m_1 \times \cdots \times m_{i-1} \times m_{i+1} \times \cdots \times m_n = m$. As previously, the time scheduling vector is $\Pi^G = (1, \ldots, 1)$. In addition, a tile $j^S = (j_1^S, \ldots, j_n^S)$ belonging to group $j^G = (j_1^G, \ldots, j_n^G)$, will be executed within node $(j_1^G, \ldots, j_{i-1}^G, j_{i+1}^G, \ldots, j_n^G)$ by CPU $(j_1^S \% m_1, \ldots, j_{i-1}^S \% m_{i-1}, j_{i+1}^S \% m_{i+1}, \ldots, j_n^S \% m_n)$.

Example 2 We have a cluster of SMP nodes with 2 CPUs and a NIC each. We assume a 3-dimensional rectangular Tile Space J^S. Let us assign the tiles along dimension j_3^S to the same CPU, as indicated in Figure 12 by the grey arrows. The CPUs of the same SMP node will execute two neighboring rows of tiles which belong to the same $j_1^S - j_3^S$ plane.

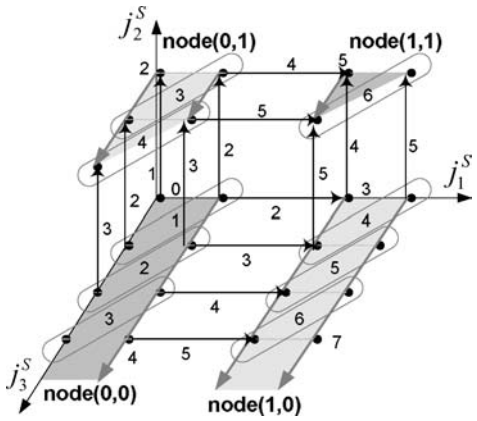

Figure 12. 3D example.

In respect to the formula (4), we choose the grouping matrices to be:

$$P^G = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ -2 & -1 & 1 \end{bmatrix} \quad \text{and} \quad H^G = (P^G)^{-1} = \begin{bmatrix} \frac{1}{2} & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}.$$

In Figure 12 we show the grouping of tiles and when each computation step and each communication step will be executed. It can be easily deduced that a group $(j_1^G, j_2^G, j_3^G) \in J^G$ will be executed in node (j_1^G, j_2^G) during the time step $t(j^G) = j_1^G + j_2^G + j_3^G$. Therefore, the linear time scheduling vector for this example is $\Pi^G = (1, 1, 1)$.

4.7. Optimal selection of m_k's

Lemma 1 *The function $f(x_1, \ldots, x_n) = x_1 + \cdots + x_n$, where $x_1 \times \cdots \times x_n = c$ and $x_1, \ldots, x_n > 0$, is minimized when $x_1 = \cdots = x_n = c^{\frac{1}{n}}$.*

Lemma 2 *The function $f(x_1, \ldots, x_n) = \frac{a_1}{x_1} + \cdots + \frac{a_n}{x_n}$, where $x_1 \times \cdots \times x_n = c$, $a_1, \ldots, a_n$ are positive constants and $x_1, \ldots, x_n$ are positive, is minimized when $x_i = a_i(\frac{c}{a_1 \times \cdots \times a_n})^{\frac{1}{n}}$, $i = 1, \ldots, n$.*

Proof: It holds that $\frac{a_1}{x_1} \times \cdots \times \frac{a_n}{x_n} = \frac{a_1 \times \cdots \times a_n}{c} = \text{constant}$. Thus, according to Lemma 1 the function $f(x_1, \ldots, x_n)$ is minimized when $\frac{a_1}{x_1} = \cdots = \frac{a_n}{x_n} = (\frac{a_1 \times \cdots \times a_n}{c})^{\frac{1}{n}} \Rightarrow x_i = a_i(\frac{c}{a_1 \times \cdots \times a_n})^{\frac{1}{n}}$, $i = 1, \ldots, n$. □

Let us consider a rectangular Tile Space J^S: $\forall j^S \in J^S$ it holds $0 \leq j_i^S < u_i^S$, $0 \leq i \leq n$. A tile $j^S \in J^S$ is assigned to the group $j^G = \lfloor H^G j^S \rfloor = (\lfloor \frac{j_1^S}{m_1} \rfloor, \ldots, \lfloor \frac{j_{i-1}^S}{m_{i-1}} \rfloor, j_1^S + \cdots + j_n^S, \lfloor \frac{j_{i+1}^S}{m_{i+1}} \rfloor, \ldots, \lfloor \frac{j_n^S}{m_n} \rfloor)$. According to the time scheduling vector

$\Pi^G = (1, \ldots, 1)$, it will be computed during the time step $t(j^G) = \lfloor \frac{j_1^S}{m_1} \rfloor + \cdots + \lfloor \frac{j_{i-1}^S}{m_{i-1}} \rfloor + \lfloor \frac{j_{i+1}^S}{m_{i+1}} \rfloor + \cdots + \lfloor \frac{j_n^S}{m_n} \rfloor + j_1^S + \cdots + j_n^S$.

The group $(0, 0, 0)$ will be computed during the first time step $t_{\min} = 0$. The group $(\lfloor \frac{u_1^S - 1}{m_1} \rfloor, \ldots, \lfloor \frac{u_{i-1}^S - 1}{m_{i-1}} \rfloor, \sum_{k=1}^{n} (u_k^S - 1), \lfloor \frac{u_{i+1}^S - 1}{m_{i+1}} \rfloor, \ldots, \lfloor \frac{u_n^S - 1}{m_n} \rfloor)$ will be computed during the last time step $t_{\max} = \lfloor \frac{u_1^S - 1}{m_1} \rfloor + \cdots + \lfloor \frac{u_{i-1}^S - 1}{m_{i-1}} \rfloor + \lfloor \frac{u_{i+1}^S - 1}{m_{i+1}} \rfloor + \cdots + \lfloor \frac{u_n^S - 1}{m_n} \rfloor + (u_1^S - 1) + \cdots + (u_n^S - 1)$. Thus, the number of execution steps required for the completion of the algorithm will be $\wp = t_{\max} - t_{\min} + 1 \Rightarrow$

$$\wp = \sum_{k \neq i} \left\lceil \frac{u_k^S}{m_k} \right\rceil + \sum_{k=1}^{n} u_k^S - 2n + 2 \tag{4}$$

In order to minimize the total completion time, we should apparently choose the i-th dimension, along of which we allocate the tiles to the same CPU, so that it holds $u_i^S \geq u_k^S, \forall k = 1, \ldots, n$, as u_i^S is the only upper bound of J^S which is involved in (4) only once.

After the selection of the i-th dimension, we can eliminate the ceiling functions involved in the expression (4) as follows: $\sum_{k \neq i} \frac{u_k^S}{m_k} + \sum_{k=1}^{n} u_k^S - 2n + 2 \leq \wp < \sum_{k \neq i} \frac{u_k^S}{m_k} + \sum_{k=1}^{n} u_k^S - n + 2$. Thus, we can assert that the completion time of the algorithm is approximately minimum when the expression $\sum_{k=1}^{n} u_k^S$ is minimized. According to Lemma 2 this condition is valid, if

$$m_k = u_k^S \left(\frac{m}{u_1^S \ldots u_{i-1}^S u_{i+1}^S \ldots u_n^S} \right)^{\frac{1}{n-1}}, \quad k = 1, \ldots, n, k \neq i. \tag{5}$$

Of course, it is not always feasible because the numbers m_i should be natural. But it always applies an approximate criterion for the selection of m_k's. Intuitively, it means that m_k's should be chosen so that $\frac{u_k^S}{m_k}$'s are as close to each other as possible.

Example 3 Let us consider a cluster of SMP nodes with $m = 4$ CPUs each and a 3-dimensional space J^S with size $20 \times 100 \times 20$. It means that $u_1^S = 20, u_2^S = 100, u_3^S = 20$. Then, according to our previous analysis the best choice will be: $i = 2, m_1 = 20(\frac{4}{20 \times 20})^{\frac{1}{2}} = 2, m_3 = \frac{m}{m_1} = 2$. If we apply these values in the expression (4) we get that the number of steps required for the completion of the algorithm will be $\wp = 156$. In contrast, if we chose $m_1 = 4, m_3 = 1$, then the expression (4) would get the value $\wp = 161 > 156$.

If the size of J^S is $20 \times 120 \times 150$ ($u_1^S = 20, u_2^S = 120, u_3^S = 150$, then, according to our previous analysis, the best choice will be: $i = 3, m_1 = 20(\frac{4}{20 \times 120})^{\frac{1}{2}} = 0.816$. The closest natural number which divides $m = 4$ is $m_1 = 1$. Thus $m_2 = \frac{m}{m_1} = 4$. If we apply these values in the expression (4), we get that the number of steps required for the completion of the algorithm will be $\wp = 336$. In contrast, if we chose $m_1 = m_2 = 2$, then the expression (4) would get the value $\wp = 356 > 336$.

As one can easily observe in the previous example, the significance of the selection of m_k's, as it has just been described, is less when the maximum dimension u_i^S is much longer than dimensions $u_1^S, \ldots, u_{i-1}^S, u_{i+1}^S, \ldots, u_n^S$. So, it may be preferable to choose the values of

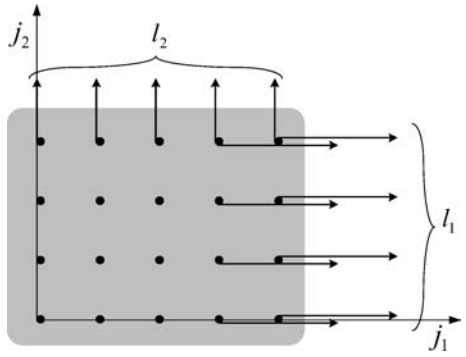

Figure 13. Tile communication load.

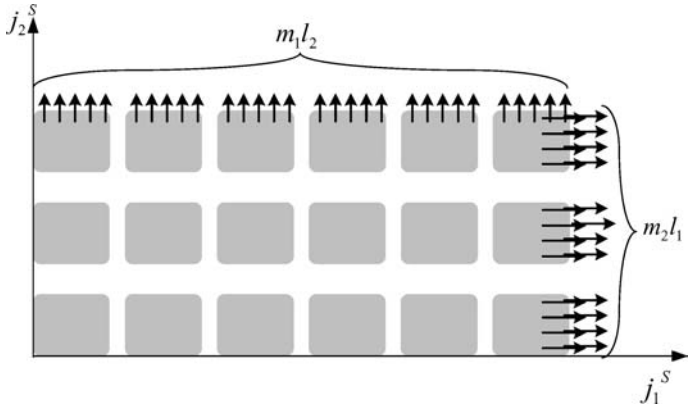

Figure 14. Communication load of a group.

m_k's taking into consideration the minimization of the communication requirements among the SMP nodes.

Let us represent with l_k the communication load of a tile along of the k-th dimension, as indicated in Figure 13. If we group together $m_1 m_2$ tiles, then the communication loads among the SMP nodes will be $l_1 m_2 = \frac{m}{m_1} l_1$ and $l_2 m_1 = \frac{m}{m_2} l_2$, as indicated in Figure 14. Similarly, if we group together $m_1 \ldots m_{i-1} m_{i+1} \ldots m_n$ tiles, then the communication loads among the nodes of the cluster will be $\frac{m}{m_k} l_k$. Thus the total communication load of a group will be $l_{\text{total}} = m(\frac{l_1}{m_1} + \cdots + \frac{l_{i-1}}{m_{i-1}} + \frac{l_{i+1}}{m_{i+1}} + \cdots + \frac{l_n}{m_n})$. According to Lemma 2 it is minimized when $m_k = l_k(\frac{m}{l_1 \cdots l_{i-1} l_{i+1} \cdots l_n})^{\frac{1}{n-1}}$, $k = 1, \ldots, n$, $k \neq i$. Of course, as the numbers m_k should be natural, this criterion is also approximative.

In the rest of this paper we shall theoretically and experimentally compare the proposed methods with each other. Although our above theoretical results can be applied to any convex tile space, as explained in Section 2.2, we shall go on using only rectangular tile spaces,

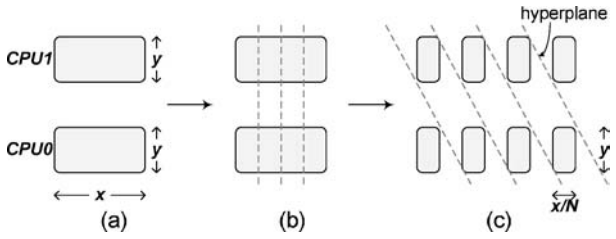

Figure 15. Splitting tiles in vertical scheme.

as in our previous examples. We consider that this simplification is convenient for clearly expressing some ideas and it does not constrain any of the advantages or disadvantages of the proposed methods.

4.8. Comparison

In this section we shall compare vertical grouping, which is indicated in Figure 5, with the proposed scheme of hyperplane grouping, which is shown in Figures 6 and 10 in the case of a 2-dimensional algorithm and a cluster of SMPs with 2 CPUs each.

As we have already mentioned, vertical grouping cannot exploit the computational power of both CPUs of our SMPs unless we split each tile into smaller subtiles and compute some of them in parallel, as shown in Figure 15. Let us assume that a CPU needs time α for the computation of a tile with dimensions x, y (Figure 15(a)). Consequently, it will need time $\frac{\alpha}{N}$ for the computation of a respective subtile with dimensions $\frac{x}{N}$, y (Figure 15(c)). The subtiles which are created can be computed by 2 CPUs in $N + 1$ computational steps, interleaved with N synchronization steps, following an optimal linear time schedule (1, 1) as in Figure 15(c). If the average time consumed for the synchronization of 2 CPUs of an SMP node is $t_{\text{synch\_in}}$, then the total time required for the computation of a pair of initial tiles is

$$\beta = \alpha \frac{N+1}{N} + N t_{\text{synch\_in}}. \tag{6}$$

β is minimized when

$$N = \sqrt{\frac{\alpha}{t_{\text{synch\_in}}}}. \tag{7}$$

Therefore, the minimum value of β is $\beta_{\min} = \alpha + 2\sqrt{\alpha t_{\text{synch\_in}}} > \alpha$.

If we consider an Iteration Space with size $X \times Y$, tiled with rectangular tiles with size $x \times y$, (for example in Figures 5 and 6 we have $\frac{X}{x} = 10$, $\frac{Y}{y} = 6$), then we have the following options:

1. Following the **non-overlapping** scheme (which can be implemented using blocking calls) in combination with **vertical grouping**, the number of time steps required for

the completion of the algorithm is $\wp = \frac{X}{x} + \frac{Y}{2y} - 1$. The minimum duration of a time step is $\beta_{\min} + t_{\text{comm}}$, where t_{comm} is the time required for the communication between two SMP nodes. Thus, the total time required is $T_{\text{blocking,vertical}} = \wp(\beta_{\min} + t_{\text{comm}}) \simeq (\frac{X}{x} + \frac{Y}{2y})(\beta_{\min} + t_{\text{comm}})$.

2. Following the **overlapping** scheme (which can be implemented using non-blocking calls) in combination with **vertical grouping**, the number of time steps required for the completion of the algorithm is $\wp = \frac{X}{x} + \frac{Y}{y} - 2$. According to the formula (1), if we set $t_{\text{comp}} = \beta_{\min}$, the minimum duration of a time step is $t_{\text{start\_dma}} + \max(\beta_{\min}, t_{\text{comm\_dma}}) + t_{\text{synchro}}$. Thus, the total time required is $T_{\text{non-blocking,vertical}} = \wp(t_{\text{start\_dma}} + \max(\beta_{\min}, t_{\text{comm\_dma}}) + t_{\text{synchro}}) \simeq (\frac{X}{x} + \frac{Y}{y})(t_{\text{start\_dma}} + \max(\beta_{\min}, t_{\text{comm\_dma}}) + t_{\text{synchro}})$. If $\beta_{\min} \geq t_{\text{comm\_dma}}$, then $T_{\text{non-blocking,vertical}} \simeq (\frac{X}{x} + \frac{Y}{y})(t_{\text{start\_dma}} + \beta_{\min} + t_{\text{synchro}})$.
3. Following the **overlapping** scheme in combination with **hyperplane grouping**, the number of time steps required for the completion of the algorithm is $\wp = \frac{X}{x} + \frac{3Y}{2y} - 2$. According to the formula (1), if we set $t_{\text{comp}} = \alpha$, the minimum duration of a time step is $t_{\text{start\_dma}} + \max(\alpha, t_{\text{comm\_dma}}) + t_{\text{synchro}}$. Thus, the total time required is $T_{\text{non-blocking,hyperplane}} = \wp(t_{\text{start\_dma}} + \max(\alpha, t_{\text{comm\_dma}}) + t_{\text{synchro}}) \simeq (\frac{X}{x} + \frac{3Y}{2y})(t_{\text{start\_dma}} + \max(\alpha, t_{\text{comm\_dma}}) + t_{\text{synchro}})$. If $\alpha \geq t_{\text{comm\_dma}}$, then $T_{\text{non-blocking,hyperplane}} \simeq (\frac{X}{x} + \frac{3Y}{2y})(t_{\text{start\_dma}} + \alpha + t_{\text{synchro}})$.

In most real problems it holds that $\frac{Y/y}{X/x} = \lambda \ll 1$. Therefore, the overlapping scheme in combination with vertical grouping is more efficient than the non-overlapping scheme, in case that $\beta_{\min} \geq t_{\text{comm}}$, when $t_{\text{comm\_dma}} > (t_{\text{start\_dma}} + \beta_{\min} + t_{\text{synchro}}) \frac{\frac{Y}{2y}}{\frac{X}{x} + \frac{Y}{2y}} \Leftrightarrow t_{\text{comm}} > \frac{\lambda}{2}(t_{\text{start\_dma}} + \beta_{\min} + t_{\text{synchro}})$. In addition, the overlapping scheme, in combination with hyperplane grouping, is more efficient than the overlapping scheme, in combination with vertical grouping, when $(\frac{X}{x} + \frac{3Y}{2y})(t_{\text{start\_dma}} + \alpha + t_{\text{synchro}}) < (\frac{X}{x} + \frac{Y}{y})(t_{\text{start\_dma}} + \alpha + 2\sqrt{\alpha t_{\text{synch\_in}}} + t_{\text{synchro}})$. If we consider $t_{\text{start\_dma}} + t_{\text{synchro}} \ll \alpha$, then, we get $2\sqrt{\frac{t_{\text{synch\_in}}}{\alpha}} > \frac{\lambda/2}{1+\lambda} \simeq \frac{\lambda}{2} \Rightarrow t_{\text{synch\_in}} > \alpha(\frac{\lambda}{4})^2$. This is due to the fact that, using vertical grouping, the pipeline filling is faster, while, using hyperplane grouping, the pipeline throughput is faster. So, hyperplane grouping is preferable when the mapping dimension of the Tile Space is long enough in comparison to its other dimensions. However, in any case, the hyperplane grouping has the advantage that it needs no extra tiling inside each tile in order to exploit the computational force of the CPUs.

Consequently, which communication and grouping policy is optimal, depends on the hardware characteristics. That is, one should estimate the time parameters involved in the model (computation, transfer initialization overhead, actual transfer overhead) and determine which scheme is going to give the peak performance. In general, the purpose of the overlapping scheme, in combination with hyperplane grouping, is to exploit all modern architectural characteristics of NICs, such as DMA, RDMA, Zero Copy, or even NICs with embedded processors. Thus, this scheme will be optimal when these characteristics are actually available.

5. Experimental verification

5.1. Experimental platform and algorithm

In [43] we applied the pipelined schedule proposed in [22], using a cluster of single CPU nodes with PCI-SCI NICs. In this paper, in order to evaluate the proposed methods, we ran our experiments on a Linux SMP cluster with 8 identical nodes. Each node had 128M of RAM and 2 Pentium III 800 MHz CPUs. The cluster nodes were interconnected with an SCI ring, using SCI Dolphin's PCI-SCI D330 cards. 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 physically contiguous pre-reserved memory regions, for DMA communications.

Our test application was the following code:

```
for(i=1; i<=X; i++)
   for(j=1; j<=Y; j++)
      for(k=1; k<=Z; 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 $X \times Y \times Z$ floats and $X = Y \ll Z$. Without lack of generality, we select as a tile a rectangle with ij, ik and jk sides. The dimension k is the largest one, so all tiles along k-axis are mapped onto the same processor, as proposed in Section 4.7. 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 along dimensions i and j and $\frac{Z}{z}$ tiles along 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 vertical and hyperplane grouping, using both blocking and non-blocking communication primitives. Since both vertical and hyperplane grouping can be combined with both overlapping and non-overlapping communication, we experimented with all four combinations. For each exemplary Iteration Space and each possible tile height, we calculated the total execution time for the above schemes. In order to implement these schemes, we used Linux POSIX threads with semaphores for the synchronization among the processors of an SMP node and the SISCI driver and libraries for the communication among the SMP nodes.

5.2. Tuning parameters

First of all, as far as the implementation of vertical grouping is concerned, we experimentally verified formula (7), in order to find the optimal execution time for a couple of tiles by an SMP node. We assigned the computation of two tiles to the two processors of an SMP node and measured their execution time in respect to the number of subtiles into which each tile was cut, in order not to violate the iteration dependencies. The experimental results, along

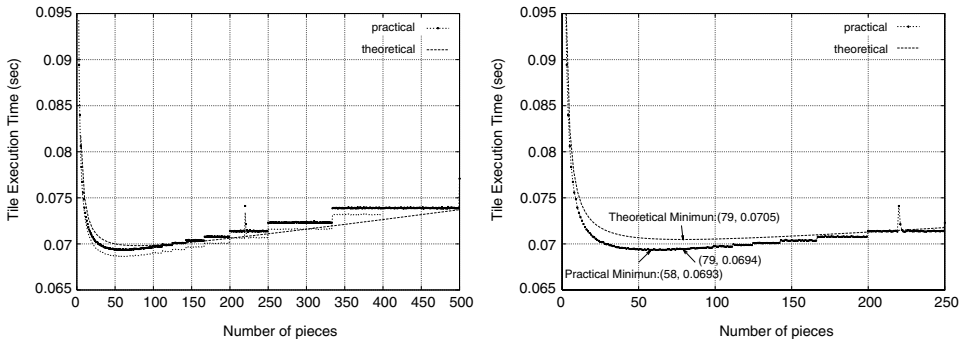

Figure 16. Vertical grouping—Tile execution time in respect to the number of slices a tile is cut.

with the theoretically expected curve, are plotted in Figure 16. The theoretical plot was calculated using the formula (6) with $\alpha \simeq 69$ msec and $t_{\text{synch\_in}} \simeq 11$ μsec. These values were experimentally measured by running a simple code fragment thousands of times and calculating the average execution time. If we find the $N_{\text{best,theoretical}}$, that is the point N where the theoretical minimum is achieved and for this N we find the corresponding experimental overall time, then the difference between this value and the experimental minimum is less than 0, 15%. So we can safely use $N_{\text{best,theoretical}}$ as N_{best}. This can be simply justified as follows: If we consider a shift δN of N, then the shift of β will be $\delta\beta = -\alpha \frac{\delta N}{N(N+\delta N)} + t_{\text{synch\_in}}\delta N$. If in this formula we set $N = N_{\text{best,theoretical}}$ we get that:

$$\frac{\delta\beta}{\beta_{\min}} = \frac{\left(\frac{\delta N}{N_{\text{best,theoretical}}}\right)^2}{1 + \frac{\delta N}{N_{\text{best,theoretical}}}} \frac{1}{2 + \sqrt{\frac{\alpha}{t_{\text{synch\_in}}}}}.$$

Therefore, the less the parameter $t_{\text{synch\_in}}$ is in comparison to α, the less important the exact selection of N is. Intuitively, in the extreme case, where $t_{\text{synch\_in}}$ is 0, we could always achieve the same results, no matter how fine grained the parallelism is (i.e. for very large N's). However, $t_{\text{synch\_in}}$ is always considerable and cannot be ignored for real life SMP architectures.

5.3. Experimental results

Once vertical grouping was implemented and approximated with a theoretical formula, we implemented both blocking and non-blocking communication schemes. As far as the blocking communication scheme is concerned, it was implemented using the pseudo-code of Table 1. On the other hand, the non-blocking scheme was implemented using the pseudo-code of Table 2. Notice that during each time step every SMP node in the ij plane with coordinates (i, j) receives from neighboring nodes $(i-1, j)$ and $(i, j-1)$, computes and sends to nodes $(i+1, j), (i, j+1)$ (Figure 17). Since the `send_dma()` call is not blocking, the computation of the tiles will be performed concurrently with the transferring of data among

Table 1. Non-overlapping scheme Implementation.

Thread 0	*Thread 1*
`foreach group assigned to node(i,j) do{`	`foreach group assigned to node(i,j) do{`
`  receive from node(i-1,j)`	`  receive from node(i,j-1)`
`  receive from node(i,j-1)`	`  compute_tile(i,j,k,CPU1)`
`  compute_tile(i,j,k,CPU0)`	`  send to node(i+1,j)`
`  send to node(i,j+1)`	`  send to node(i,j+1)`
`  semaphore_post(sem_s1)`	`  semaphore_post(sem_s2)`
`  semaphore_wait(sem_s2)`	`  semaphore_wait(sem_s1)`
`}`	`}`

Table 2. Overlapping scheme Implementation.

Thread 0	*Thread 1*	*Explanation*
`foreach group assigned to node(i,j) do{`	`foreach group assigned to node(i,j) do{`	
`  trigger_interrupt to node(i-1,j)`		Inform "previous" nodes:
`  trigger_interrupt to node(i,j-1)`	`  trigger_interrupt to node(i,j-1)`	"I am ready to accept data"
	`  wait_interrupt from node(i+1,j)`	Wait until "next" nodes
`  wait_interrupt from node(i,j+1)`	`  wait_interrupt from node(i,j+1)`	are ready to accept data
	`  send_dma(node(i+1,j),data)`	Initialization of DMA
`  send_dma(node(i,j+1),data)`	`  send_dma(node(i,j+1),data)`	transfer to neighboring nodes
`  compute_tile(i,j,k,CPU0)`	`  compute_tile(i,j,k,CPU1)`	
	`  wait_dma()`	Wait for DMA to complete
`  wait_dma()`	`  wait_dma()`	
	`  trigger_interrupt to node(i+1,j)`	Inform "next" nodes:
`  trigger_interrupt to node(i,j+1)`	`  trigger_interrupt to node(i,j+1)`	"Your data has arrived"
`  wait_interrupt from node(i-1,j)`		Wait until "previous" nodes
`  wait_interrupt from node(i,j-1)`	`  wait_interrupt from node(i,j-1)`	have finished sending data
`  semaphore_post(sem_s1)`	`  semaphore_post(sem_s2)`	Implementation of a barrier
`  semaphore_wait(sem_s2)`	`  semaphore_wait(sem_s1)`	
`}`	`}`	

the SMP nodes. After the execution of `wait_dma()`, it is assured that both computation and communication are already completed.

The implementation of vertical and hyperplane grouping was achieved by a proper `compute_tile(i,j,k,CPUx)` procedure. In order to implement vertical grouping, we used the pseudocode of Table 3. The number of subtiles inside a tile was selected according to formula (7). Notice that, the implementation of hyperplane grouping was much simpler, as it is shown in Table 3.

The problem was solved using various values of $X = Y$ and Z. For each schedule, we are interested in the overall minimum execution time achieved at an optimally selected tile height (see [22, 24, 43]). The experimental results, shown in Figures 18–19, illustrate that, in every case, non-blocking communication is preferable to blocking communication and hyperplane grouping is preferable to vertical grouping. The lowest minimum is clearly achieved when using hyperplane grouping, in combination with non-blocking communication, in all cases.

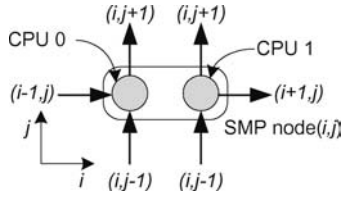

Figure 17. CPU communication directions.

As far as hyperplane grouping, in combination with non-blocking communication, is concerned, according to our scheduling theory (formula (4)), the number of time steps required for the completion of an experiment is $\wp(x, y, z) = \frac{3X}{2x} + \frac{2Y}{y} + \frac{Z}{z} - 4$. The minimum duration of a time step, as mentioned in Section 4.8, is $(t_{\text{start\_dma}} + t_{\text{comp}} + t_{\text{synchro}})$. Thus, $T_{\text{non-blocking,hyperplane}} = (\frac{3X}{2x} + \frac{2Y}{y} + \frac{Z}{z} - 4)(t_{\text{start\_dma}} + t_{\text{comp}} + t_{\text{synchro}})$. This formula was used to produce the theoretical curves of Figures 18–19 with values $t_{\text{start\_dma}} + t_{\text{synchro}} = 100\,\mu$ sec and $t_{\text{comp}} = x^2 z t_{\text{comp1}}$, where t_{comp1} is the execution time of a single iteration and it was measured equal to 39,6 nsec.

One can easily verify from Figures 18-19 that the graphs of the theoretical model are very close to the corresponding experimental graphs not only at the desired minimum, but along the whole graph. Thus, the theoretical model of scheduling is strongly verified by the experimental results.

5.4. Scalability issues

The theoretical model presented in Section 4 is general enough, so as not to be differentiated when scaling up the underlying hardware architecture. However, in this section, we shall examine some practical problems, which may rise.

For example, if we add more SMP nodes, the initial iteration space may be cut into smaller tiles. Thus, the computation to communication ratio of each tile $\frac{t_{\text{comp}}}{t_{\text{comm\_dma}}}$ may reduce for two reasons: First, less computations are assigned to each SMP node, while the amount of

Table 3. Vertical vs. Hyperplane Grouping.

compute_tile(i,j,k,CPU0)	*compute_tile(i,j,k,CPU1)*
Vertical grouping	
`foreach subtile of this tile do{`	`foreach subtile of this tile do{`
`  compute each iteration of this subtile`	
`  semaphore_post(sem1)`	`  semaphore_post(sem2)`
`  semaphore_wait(sem2)`	`  semaphore_wait(sem1)`
	`  compute each iteration of this subtile`
`}`	`}`
Hyperplane grouping	
compute_tile(i,j,k,CPU0)	*compute_tile(i,j,k,CPU1)*
`compute each iteration of this tile`	`compute each iteration of this tile`

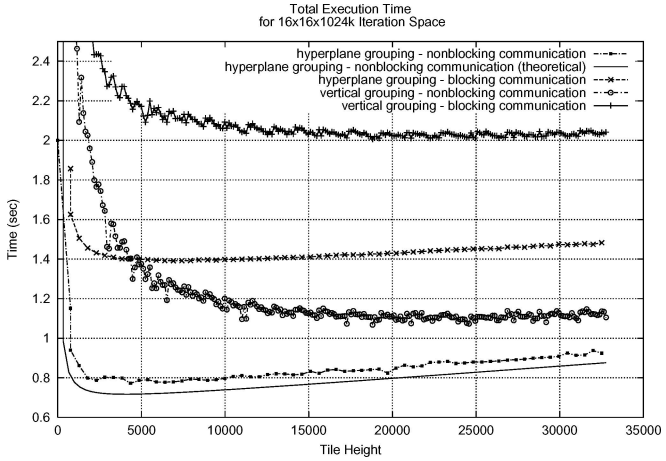

Figure 18. Experimental results.

data transfer required is not proportionally reduced. Second, if the network is saturated (by more SMP nodes trying to send more data in more messages to each other), the increase in $t_{\text{comm\_dma}}$ will be more than relative to the increase in the volume of data transmitted. However, considering an application with uniform dependences, as described in the algorithmic model in Section 2.2, and a torus interconnection topology, such as the one used for our experiments, the network will be never saturated due to the increase of SMP nodes. This is because each node need to communicate only with its neighbors, thus there are no shared resources among different communication channels. Thus, only the first reason mentioned above can potentially cause some trouble when adding more SMP nodes. But, if it still holds $t_{\text{comp}} \geq t_{\text{comm\_dma}}$, nothing will change in the implementation of our model. In the opposite case ($t_{\text{comp}} < t_{\text{comm\_dma}}$), the use of even more nodes will not be efficient. This problem will not concern our scheduling, but it will mean that the communication architecture is too slow to exploit all the computation power of the computing system. Then, it would be better not to use all the nodes available, as implied in [24].

If we add more CPUs inside each SMP node, we may again cut the initial iteration space into smaller tiles. The computation to communication ratio of each tile $\frac{t_{\text{comp}}}{t_{\text{comm\_dma}}}$ will be decreased again, but only for one reason: Less computations are assigned to each CPU. In particular, $\frac{t_{\text{comp}}}{t_{\text{comm\_dma}}}$ will be conversely proportional to the number of CPUs inside each SMP node. In this case, no more data need to be sent through the interconnection network, since the additional CPUs communicate with each other and with the preexisting CPUs through the SMP node's shared memory. However, t_{synchro} and $t_{\text{start\_dma}}$ will slightly increase, because, firstly, more CPUs need to initialize their DMA sends and receives and, secondly, these operations can not be executed at the same time by different threads of the same node (no thread-safe environment—see the implementation code of Table 2). This problem can be solved by assigning all communication overhead to one thread only and at the same time reducing the computation overhead of this thread. Following that technique, CPUs do not remain idle waiting to synchronize with each other, since the amount of

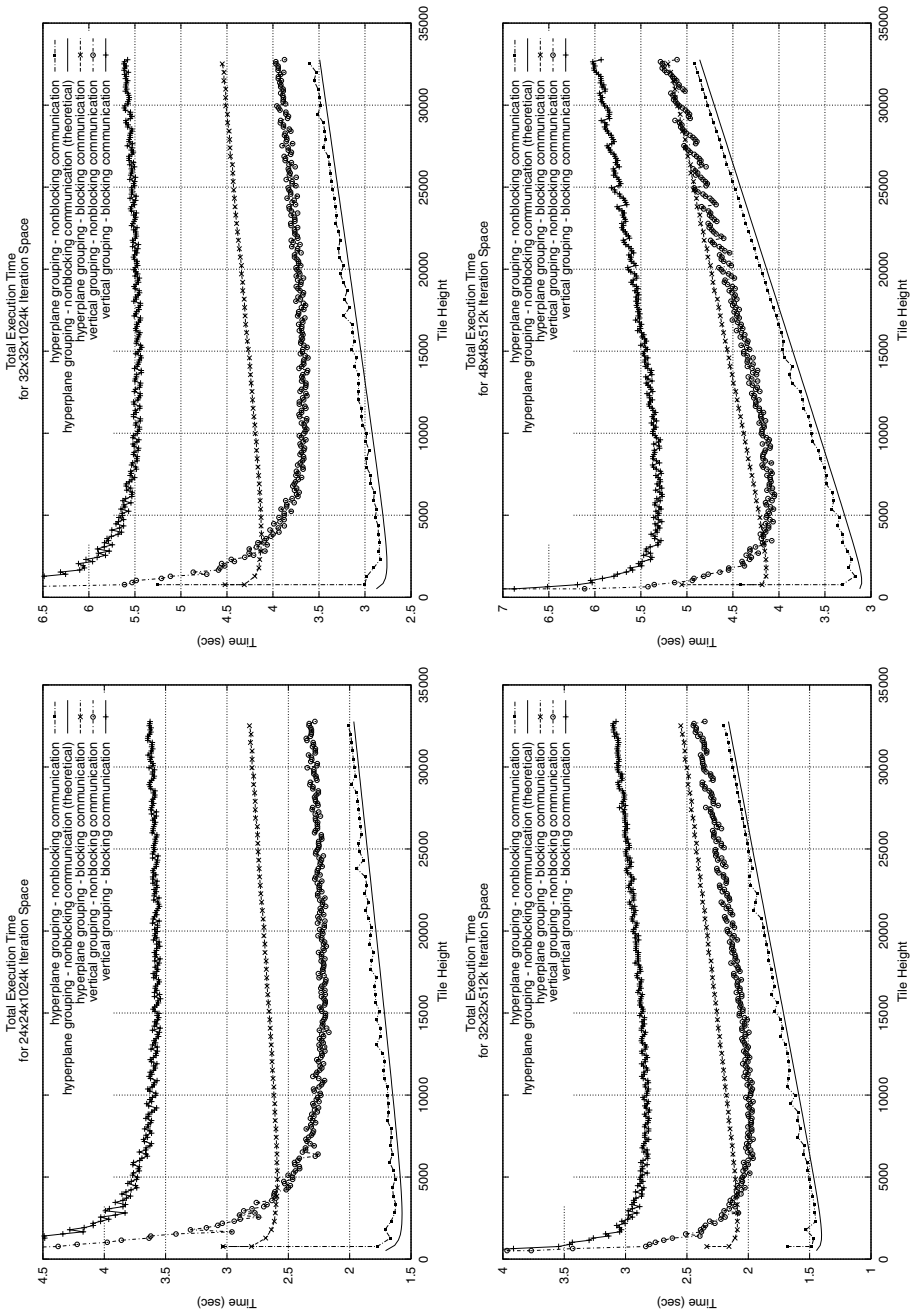

Figure 19. Experimental results.

computations assigned to the communicating thread may be properly calculated, so as the total communication + computation overhead to be evenly distributed among the CPUs of each node. The exact solution of this problem concerns our future work.

Another aspect of scalability (concerning the scheduling algorithm, not the hardware) is having so large iteration spaces that we cannot cut them into so few tiles. That is, applying a tile selection technique, such as the ones presented in [30, 33, 36, 39], we may get more rows of tiles than the CPUs available. Then we should apply a more complicated technique for assigning tiles to SMP nodes and CPUs as described in [5]. However, the techniques proposed in [5] are based on the conclusions of this paper.

6. Conclusion

In this paper we presented a novel approach for the time scheduling of tiled nested loops on a cluster of SMP nodes using the advanced features (DMA, Zero Copy) of modern communication architectures. We minimized the total execution time by overlapping the computation with communication. In addition, we achieved the maximum CPUs utilization with a proper grouping transformation. Hyperplane grouping and pipelined schedules can be efficiently combined with advanced communication architectures for SMP nodes to execute tiled loops much more efficiently, with the least possible synchronization overheads.

Acknowledgments

We wish to express our profound gratitude to the anonymous reviewers for their suggestions, which considerably increased the clarity and quality of the original manuscript.

References

1. C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)*, Williamsburg, VA, pp. 39–50, April 1991.
2. 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, 1999.
3. H. R. Arabnia and S. M. Bhandarkar. Parallel stereocorrelation on a reconfigurable multi-ring network. *Journal of Supercomputing (Kluwer Academin Publishers), Special Issue on Parallel and Distributed Processing*, 10(3):243–270, 1996.
4. S. Araki, A. Bilas, C. Dubnicki, J. Edler, K. Konishi, and J. Philbin. User-space communication: A quantitative study. In *Proceedings of the 1998 Supercomputing Conference on High Performance Networking and Computing (SC98)*. Orlando, Florida, Nov. 1998.
5. M. Athanasaki, E. Koukis, and N. Koziris. Scheduling of tiled nested loops onto a cluster with a fixed number of SMP nodes. In *Proceedings of the 12-th Euromicro Conference on Parallel, Distributed and Network based Processing (PDP04)*. IEEE Computer Society Press, A Coruna, Spain, pp. 424–433, 2004.
6. M. Athanasaki, A. Sotiropoulos, G. Tsoukalas, and N. Koziris. A pipelined execution of tiled nested loops on SMPs with computation and communication overlapping. In *Proceedings of the Workshop on Compile/Runtime Techniques for Parallel Computing, in conjunction with 2002 Int'l Conference on Parallel Processing (ICPP-2002)*. Vancouver, Canada, pp. 559–567, 2002.

7. M. Athanasaki, A. Sotiropoulos, G. Tsoukalas, and N. Koziris. Pipelined scheduling of tiled nested loops onto clusters of SMPs using memory mapped network interfaces. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing (SC2002)*. IEEE Computer Society Press, Baltimore, Maryland, Nov. 2002.
8. S. M. Bhandarkar and H. R. Arabnia. Parallel computer vision on a reconfigurable multiprocessor network. *IEEE Trans. on Parallel and Distributed Systems*, 8(3): 292–310, 1997.
9. A. Bilas, C. Liao, and J. P. Singh. Using network interface support to avoid asynchronous protocol processing in shared virtual memory systems. In *Proceedings of the 26th Int'l Symposium on Computer Architecture ISCA-26*. Atlanta, GA, pp. 282–293, 1999.
10. M. Blumrich. *Network Interface for Protected, User-Level Communication*. PhD thesis, Princeton University, April 1996.
11. N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet. A Gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, 1995.
12. P. Boulet, A. Darte, T. Risset, and Y. Robert. (Pen)-ultimate tiling? *INTEGRATION, The VLSI Jounal*, 17:33–51, 1994.
13. 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 Int'l Conference on Supercomputing*. Melbourne, Australia, pp. 243–249, 1998.
14. F. T. Chong, R. Barua, F. Dahlgren, J. Kubiatowicz, and A. Agarwal. The sensitivity of communication mechanisms to bandwidth and latency. In *Proceedings of the HPCA-4 High Performance Communication Architectures*, pp. 37–46, 1998.
15. Compaq, Intel, and Microsoft. *Virtual Interface Architecture Specification*, Dec. 1997.
16. F. Desprez, J. Dongarra, and Y. Robert. Determining the idle time of a tiling: New results. *Journal of Information Science and Engineering*, 14:167–190, 1997.
17. I. Drossitis, G. Goumas, N. Koziris, G. Papakonstantinou, and P. Tsanakas. Evaluation of loop grouping methods based on orthogonal projection spaces. In *Proceedings of the Int'l Conference on Parallel Processing*. Toronto, Canada, pp. 469–476, Aug. 2000.
18. 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*. Copper Mountain, Colorado, pp. 40–53, Dec. 1995.
19. 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.
20. 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.
21. G. Goumas, M. Athanasaki, and N. Koziris. An efficient code generation technique for tiled iteration spaces. *IEEE Trans. on Parallel and Distributed Systems*, 14(10):1021–1034, 2003.
22. 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.
23. H. Hellwagner. The SCI standard and applications of SCI. In H. Hellwagner and A. Reinefield, eds., *Scalable Coherent Interface (SCI): Architecture and Software for High-Performance Computer Clusters*. Springer-Verlag, pp. 3–34, Sept. 1999.
24. E. Hodzic and W. Shang. On supernode transformation with minimized total running time. *IEEE Trans. on Parallel and Distributed Systems*, 9(5):417–428, 1998.
25. E. Hodzic and W. Shang. On time optimal supernode shape. *IEEE Trans. on Parallel and Distributed Systems*, 13(12):1220–1233, 2002.
26. K. Hogstedt, L. Carter, and J. Ferrante. Determining the idle time of a tiling. In *Principles of Programming Languages (POPL)*, pp. 160–173, Jan. 1997.
27. K. Hogstedt, L. Carter, and J. Ferrante. Selecting tile shape for minimal execution time. In *ACM Symposium on Parallel Algorithms and Architectures*, pp. 201–211, 1999.
28. K. Hogstedt, L. Carter, and J. Ferrante. On the parallel execution time of tiled loops. *IEEE Trans. on Parallel and Distributed Systems*, 14(3):307–321, 2003.

29. F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th Ann. ACM SIGACT-SIGPLAN Symp. Principles of Programming Languages*. San Diego, California, pp. 319–329, 1988.
30. M. Kandemir, J. Ramanujam, and A. Choudary. Improving cache locality by a combination of loop and data transformations. *IEEE Trans. on Parallel and Distributed Systems*, 48(2):159–167, 1999.
31. V. Karamcheti and A. Chien. Software overhead in messaging layers: where does the time go? In *Proceedings of the 6th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 51–60, Oct. 1994.
32. C.-T. King, W.-H. Chou, and L. Ni. Pipelined data-parallel algorithms: Part II design. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):430–439, 1991.
33. M. Lam, E. Rothberg, and M. Wolf. The cache performance and optimizations of blocked algorithms. In *Second Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Santa Clara, California, pp. 63–74, April 1991.
34. N. Manjikian and T. S. Abdelrahman. Exploiting wavefront parallelism on large-scale shared-memory multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 12(3):259–271, 2001.
35. R. Martin, A. Vahdat, D. Culler, and T. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *Proceedings of Int'l Symposium on Computer Architecture*. Denver, CO, June 1997.
36. N. Park, B. Hong, and V. Prasanna. Tiling, block data layout and memory hierarchy performance. *IEEE Trans. on Parallel and Distributed Systems*, 14(7):640–654, 2003.
37. D. Patterson and J.Hennessy. *Computer Organization & Design. The Hardware/Software Interface*. Morgan Kaufmann Publishers, San Francisco, CA, pp. 364–367, 1994.
38. J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16:108–120, 1992.
39. L. Renganarayana and S. Rajopadhye. A geometric programming framework for optimal multi-level tiling. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing (SC2004)*, Pittsburgh, PA USA, Nov. 2004.
40. J.-P. Sheu and T.-S. Chen. Partitioning and mapping nested loops for linear array multicomputers. *Journal of Supercomputing*, 9:183–202, 1995.
41. J.-P. Sheu and T.-H. Tai. Partitioning and mapping nested loops on multiprocessor systems. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):430–439, 1991.
42. P. Shivam, P. Wyckoff, and D. Panda. EMP: Zero-copy OS-bypass NIC-driven gigabit ethernet message passing. In *Proceedings of the ACM Supercomputing 2001 (SC2001)*. Denver, CO, USA, Nov. 2001.
43. A. Sotiropoulos, G. Tsoukalas, and N. Koziris. Enhancing the performance of tiled loop execution onto clusters using memory mapped network interfaces and pipelined schedules. In *Proceedings of the 2002 Workshop on Communication Architecture for Clusters (CAC'02), Int'l Parallel and Distributed Processing Symposium (IPDPS'02)*. Fort Lauderdale, Florida, April 2002.
44. H. Tezuka, F. Carroll, A. Hori, and Y. Ishikawa. Pin-down cache: A virtual memory management technique for zero-copy communication. In *Proceedings of 12th Int'l Parallel Processing Symposium*. Orlando, FL, pp. 308–314, March 1998.
45. P. Tsanakas, N. Koziris, and G. Papakonstantinou. Chain grouping: A method for partitioning loops onto mesh-connected processor arrays. *IEEE Trans. on Parallel and Distributed Systems*, 11(9):941–955, 2000.
46. R. Wang, A. Krishnamurthy, R. Martin, T. Anderson, and D. Culler. Modeling communication pipeline latency. In *Proceedings of SIGMETRICS '98/PERFORMANCE '98 Conference on the Measurement and Modeling of Computer Systems*, June 1998.
47. J. Xue. Communication-minimal tiling of uniform dependence loops. *Journal of Parallel and Distributed Computing*, 42(1):42–59, 1997.
48. J. Xue. On tiling as a loop transformation. *Parallel Processing Letters*, 7(4):409–424, 1997.