Reducing the Communication Cost via Chain Pattern Scheduling*

Florina M. Ciorba Theodore Andronikos Ioannis Drositis
George Papakonstantinou and Panayotis Tsanakas
National Technical University of Athens
Computing Systems Laboratory
9th Heroon Polytechniou Zografou Campus
15780 Athens, Greece
{cflorina,tedandro,jdros,papakon}@cslab.ece.ntua.gr, panag@cs.ntua.gr

Abstract

This paper deals with general nested loops and proposes a novel scheduling methodology for reducing the communication cost of parallel programs. General loops contain complex loop bodies (consisting of arbitrary program statements, such as assignments, conditions and repetitions) that exhibit uniform loop-carried dependencies. Therefore it is now possible to achieve efficient parallelization for a vast class of loops, mostly found in DSP, PDEs, signal and video coding. We use computational geometry methods, that exploit efficiently the regularity of nested loops index spaces, in order to significantly reduce the communication cost, which in most cases is the main drawback of parallel programs' performance. Through extensive testing, we show that the proposed method outperforms in all cases the classic cyclic mapping, succeeding to reduce the communication by 15%-35%. This significant reduction of the communication volume makes our method a promising candidate to be incorporated into existing automatic parallel code generation tools.

Keywords*: patterns, data locality, communication cost, uniform dependence loop, nested loop scheduling, compute-intensive applications.*

*This work of F. Ciorba was supported by the Greek State Scholarships Foundation (I.K.Y.). This work of Dr. Tsanakas was supported in part by the PROTAGORAS project of NTUA.

1. Introduction

It is well known that most time consuming applications consist of nested DO(FOR) loops. The iterations within a loop nest are either *independent iterations* or *precedence constrained iterations*. Furthermore, the precedence constraints can be uniform (constant) or non-uniform throughout the execution of the program. The index space of a uniform dependence loop, due to the existence of dependence vectors, is partitioned into subspaces of points that can be executed at a certain time instance. The geometric representations of these sets form polygonal shapes called *wavefronts* or *patterns*, with special attributes and characteristics. In this paper we tackle the uniform precedence constraints case, and we present a novel scheduling technique for perfectly nested loops. The presented method exploits the geometric properties of the index space in order to reach an efficient geometric decomposition. As the problem of finding the optimal schedule with precedence constrains was proved to be NP-complete [6], even when the number of available processors is unbounded, many heuristic algorithms have been proposed, such as [1, 2, 13, 7] for task duplication-based scheduling, and [3, 14, 11] for bounded number of processors. However, we take a much simpler approach to the above problem, as the index space is characterized by regularity due to the presence of

uniform dependence vectors. In many cases where certain conditions between the dependence vectors hold, one can predict efficiently the exact geometric outline of the wavefront at each moment [5, 12]. One such a case is when the initial wavefront is "repeated" k times at moment k (see Fig. 2(b)). Using wavefronts one can find the *optimal* solution with respect to the total makespan, in contrast to the classic hyperplane approach [4] where the solution provided is nearly optimal, achieving a makespan of constant delay from the earliest possible.

To achieve performance on parallel systems, programs must take into account both parallelism and locality. There are two main classes of memory architectures for parallel machines: *distributed-memory* machines and *shared-memory* machines. The distributed-memory machines are built from a group of processors connected via a communication network. The shared-memory machines present the programmer with a single memory space that all processors can access. Shared-memory machines can either have a single shared memory that can be uniformly accessed by all the processors (*centralized* shared-memory machines), or the memories may be physically distributed across the processors (*distributed* shared-memory machines). An example of distributed-shared-memory systems are the Silicon Graphics Origin commercial machines.

The scheduling for distributed-memory systems is more challenging than for shared-memory systems due to the fact that the communication cost must be taken into account. Two things make the communication in multiprocessor systems more inefficient than in uniprocessors: long latencies due to inter-processor communication and multiprocessor-specific cache misses on machines with coherent caches. Long memory latencies mean that the amount of inter-processor communication in the program is a critical factor for performance. Thus it is important for computations to have good temporal locality. A computation has good data locality if it re-uses much of the data it has been accessing; programs with high temporal locality tend to require less communication. It is therefore important to take communication and temporal locality into consideration when deciding how to parallelize a loop nest and how to assign the iterations to processors.

When parallelizing nested loops, the following tasks need to be performed: (1) *detection of the inherent parallelism*, (2) *computation scheduling*, (3) *computation mapping*, (4) *explicitly managing the memory and communication* and (5) *generating the code* to be executed on each processor. Here we focus on the first three tasks: parallelism detection, computation scheduling and computation mapping (while taking into account the communication and temporal locality), or in other words the *time-scheduling* and *space-mapping*. We accomplish these via the use of a novel and efficient communication scheme we call *chain pattern* for data exchange between processors. Current work is undergoing for generating code based on this scheme for various high performance architectures.

This work is an extension of [5][12], in which a geometric method of pattern-outline prediction was presented for uniform dependence loops, but no space mapping method was presented. In [5] and [12] no communication cost was taken into account, and the Unit-Execution-Time (UET) model was assumed. In contrast, in this paper the cost for communication is accounted for and a trade-off is given between different space-mapping schemes. Several wavefront scheduling methods are presented and compared in [8]. In this paper we combine the method in [5] with the static cyclic scheduling of [8], thus presenting the chain pattern scheduling, which is an improvement with respect to [5]&[8] in that it enhances the data locality utilizing methods from computational geometry, to take advantage of the regularity of the index space. In particular, we identify groups of iteration points that we call *chains*, defining points connected via a specific dependence vector, called communication vector $\mathbf{d}_c$ (see [12]). Specific chains are mapped to the same processor to enhance the data locality. It is a well-known fact that the communication overhead in most cases determines the quality and efficiency of the parallel code. The fundamental idea behind our algorithm is that regardless of the underlying interconnection network (FastEthernet,

GigabitEthernet, SCI, Myrinet), or of the number of processors within a node (for SMP systems), or of the system's homogeneity or heterogeneity, reducing the communication cost always yields enhanced performance. Given two scheduling policies that use the same number of processors, the one requiring less data exchange between the processors will almost certainly outperform the other. In this paper we propose that the chunks assigned to each processors be chosen so as to ensure data locality and reduced communication. Actually, using the geometric notion of chains and taking advantage of the regularity of the index space, this can be achieved by simply following the flow of dependence vectors. Extensive experimental comparisons of our scheme with the well-known cyclic mapping [8] show that for the same number of processors the volume of communication is reduced by $15\% - 35\%$.

The contributions of this work are the following: (1) we present a simple method for scheduling nested loops to enhance the data locality, without the complicated transformations of other approaches (e.g., tiling); (2) we introduce a new scheme for the efficient space-time mapping that trades off execution vs. communication time in order to achieve overall minimum completion time.

2. Definitions and Notation

The index space J of an n-dimensional uniform dependence loop is a n-dimensional subspace of $\mathbb{N}^n$. Due to the existence of dependence vectors, only a certain set of points can be executed at every moment [12]. The geometric border of this set, forms a polygonal shape called *pattern*. Our scheduling algorithm executes every index point at its *earliest computation time* (ECT), imposed by the existing dependence vectors. This policy guarantees the *optimal* execution time of the entire loop (assuming an unbounded number of processors). The index space is partitioned into disjoint *time subsets* denoted Pat_k, $k \geq 0$, such that Pat_k contains the set of points of J, whose earliest computation time is k. We call Pat_k the *pattern* corresponding to moment $\mathbf{k}$. By definition, Pat_0 denotes the boundary (pre-computed*) points. The geometric shape of the subset of index points that can be computed initially is called initial pattern (see Fig. 1) and is denoted as Pat_1. The *pattern-outline* is the upper boundary of each Pat_k and is denoted as $pat_k, k \geq 1$. The *pattern points* are those points that are necessary in order to define the polygon shape of the pattern. The rest of the points just complete the polygon area. The *pattern vectors* are those dependence vectors $\mathbf{d}_i$ whose end-points are the pattern points of the initial pattern. Fig. 1 depicts these definitions.

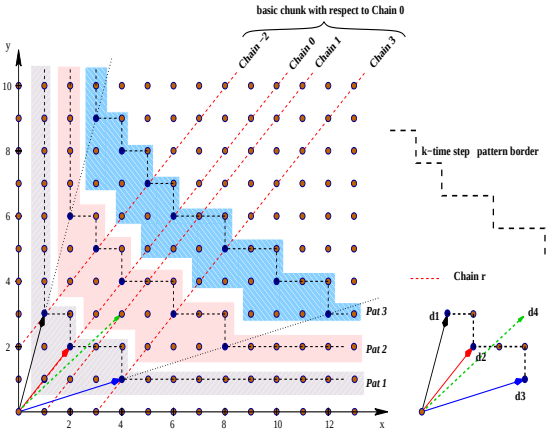


Figure 1. This is the index space of a loop with 4 dependence vectors: $\mathbf{d}_1 = (1, 3)$, $\mathbf{d}_2 = (2, 2)$, $\mathbf{d}_3 = (4, 1)$, and $\mathbf{d}_4 = (4, 3)$.

Each shaded polygon depicts what we call *pattern*, i.e., the subset of J that can be executed at a certain moment. The first 3 patterns are showed. The dashed lines mark the border of each pattern, called *pattern outline*. Except $\mathbf{d}_4$, all other vectors contribute to the formation of the pattern outline, therefore are called *pattern vectors*. On the axes lie the pre-computed boundary points. The communication vector in this example is $\mathbf{d}_c = \mathbf{d}_2$. The chain with origin $(0, 0)$ $(C_{(0,0)})$ is shown, as well as the chains it sends data to due to $\mathbf{d}_1$, $\mathbf{d}_4$ and $\mathbf{d}_3$, i.e., $C_{(0,2)}$, $C_{(1,0)}$ and $C_{(3,0)}$ respectively.

Partitioning the index space into *time patterns*

*The *pre-computed* points designate iteration points that do not belong to the first quadrant of the index space, but represent initial values for specific problems.

is beneficial for the time scheduling. However it is not sufficient for an efficient *space scheduling*, because it does not address the data locality issue. Therefore, an additional decomposition of the index space is necessary in order to enhance the data locality [9] [10]. Usually, there are certain sequences of points that are computed by the same processor. Each such sequence can be viewed as a chain of computations, that is created by a certain dependence vector, as shown in Fig. 1. This dependence vector is called *communication vector* and will be referred to as $\mathbf{d}_c$. It is usually chosen to be the dependence (pattern) vector that incurs the largest amount of communication (in most cases this is the vector with the "smallest" coordinates). The goal of this scheduling scheme is eliminating the communication cost incurred by $\mathbf{d}_c$, hence, attributing the communication cost of the loop nest to the remaining $m^{\dagger}$ dependence vectors: $\mathbf{d}_1, \ldots, \mathbf{d}_m$.

The communication vector $\mathbf{d}_c$ defines the following family of lines in n-dimensional space: $\mathbf{j} = \mathbf{p} + \lambda \mathbf{d}_c$, where $\mathbf{p} \in \mathbb{N}^n$ and $\lambda \in \mathbb{R}$. This way, every index point belongs to one such line. Thus, by defining the chain $C_{\mathbf{r}}$ with *offset* $\mathbf{r}$ to be $\{\mathbf{j} \in J \mid \mathbf{j} = \mathbf{r} + \lambda \mathbf{d}_c$, for some $\lambda \in \mathbb{R}\}$, we partition the index space J into a set $\mathcal{C}$ of such chains. The offset $\mathbf{r}$ is chosen so as to have *at least one* of its coordinates equal to 0, in other words $\mathbf{r}$ is a precomputed point. $|C_{\mathbf{r}}|$ and $|\mathcal{C}|$ are the cardinalities of $C_{\mathbf{r}}$ and $\mathcal{C}$, respectively, and C_M is the cardinality of a maximal chain[‡] of $\mathcal{C}$. The points of $C_{\mathbf{r}}$ communicate via $\mathbf{d}_i$ (i designating any dependence vector except $\mathbf{d}_c$) with the points of $C_{\mathbf{r}_i{}'}$.

Let $D^{in}_{\mathbf{r}}$ be the volume of the "incoming" data for $C_{\mathbf{r}}$, i.e., the number of index points on which the points of $C_{\mathbf{r}}$ depend on. Similarly, $D^{out}_{\mathbf{r}}$ is the volume of the "outgoing" data for $C_{\mathbf{r}}$, i.e., the number of index points which depend on the points of $C_{\mathbf{r}}$. The total communication associated with $C_{\mathbf{r}}$ is $D^{in}_{\mathbf{r}} + D^{out}_{\mathbf{r}}$. In the rest, NP denotes the number of available processors.

[†]In this paper n is the dimension of the index space and m the number of dependence vectors, excluding $\mathbf{d}_c$.

[‡]Generally, in an index space there are many maximal chains.

3. Chain Pattern Scheduling

The *chain pattern scheduling* algorithm is devised to enhance the data locality of programs with uniform nested loops using the concept of chains, and taking advantage of the optimality of the scheduling method based on patterns. By mapping all points of a chain $C_{\mathbf{r}}$ to a single processor, communication incurred by $\mathbf{d}_c$ for those points is completely eliminated. Furthermore, assuming that $C_{\mathbf{r}}$ sends data to $C_{\mathbf{r}_1}, \ldots, C_{\mathbf{r}_m}$ due to dependence vectors $\mathbf{d}_1, \ldots, \mathbf{d}_m$, by mapping $C_{\mathbf{r}}$ and $C_{\mathbf{r}_1}, \ldots, C_{\mathbf{r}_m}$ to the *same* processor the communication incurred by $\mathbf{d}_1, \ldots, \mathbf{d}_m$ is also eliminated. Similarly, the chains from which $C_{\mathbf{r}}$ receives data, $C_{\mathbf{r}'_1}, \ldots, C_{\mathbf{r}'_m}$, can also be mapped to the same processor, thus eliminating even more communication. Generally, it will not be possible to map all these chains to the same processor; yet the important thing is the following: when assigning a new chain to the processor that executed $C_{\mathbf{r}}$, systematically pick one of $C_{\mathbf{r}_1}, \ldots, C_{\mathbf{r}_m}, C_{\mathbf{r}'_1}, \ldots, C_{\mathbf{r}'_m}$ instead of choosing in a random or arbitrary manner. This strategy is guaranteed to lead to significant communication reduction.

To better illustrate our point we consider two different scenarios: (1) unbounded NP – high communication $\Rightarrow$ moderate performance, and (2) fixed NP – moderate communication $\Rightarrow$ good performance. A common feature of both scenarios is that chains are mapped to processors starting with chain $C_{(0,0)}$ and proceeding with the chains to its left (or above) and right (or below), like a fan spreading out. It is straightforward that by doing this more points become available for execution (are "released" from their dependencies) than in any other way.

3.1. Scenario 1: Unbounded NP – High Communication

This is the case where there are enough available processors so that each chain is mapped to a different processor (see Fig. 2). Two similar examples are given: (a) A loop with three dependence vectors, $\mathbf{d}_1 = (1, 3)$, $\mathbf{d}_2 = \mathbf{d}_c = (2, 2)$ and $\mathbf{d}_3 = (4, 1)$ with chains formed along $\mathbf{d}_2$. (b) A loop with two dependence vectors, $\mathbf{d}_1 = (4, 1)$ and

$\mathbf{d}_2 = \mathbf{d}_c = (2, 2)$, with chains also formed along $\mathbf{d}_2$. Nota that in both cases, 24 chains are created. However this scenario is somewhat unrealistic because for larger index spaces the number of chains would be much greater and, therefore, the number of processors required for assigning one chain to one processor would be prohibitive. On the other hand, this scenario does not support any kind of data locality (except for $\mathbf{d}_2$), requiring a large volume of communication between the processors. A chain $C_{\mathbf{r}}$ has, in general, to receive data from m different chains and to send data to m different chains. This implies that both $D^{in}_{\mathbf{r}}$ and $D^{out}_{\mathbf{r}}$ are bounded above by mC_M (recall that C_M is the cardinality of a maximal chain). The total volume of communication induced by this scheme, denoted $\mathcal{V}$, is then: $\mathcal{V} \approx 2mC_M|\mathcal{C}|$. It is obvious that such a high volume of communication diminishes the overall performance of the parallel program.

3.2. Scenario 2: Fixed NP – Moderate Communication

This scenario is designed to minimize the communication and enhance the data locality, thus increasing the overall performance of the parallel program. For this scenario, two alternative mappings are considered, as shown in Fig. 3 and 4. In order to have moderate communication, and thus better performance than in scenario (1), an arbitrary NP was considered (herein 5 processors). The first mapping (see Fig. 3(a)(b)) is an implementation of the well known cyclic mapping [8], where each chain from the pool of unassigned chains is mapped in a cyclic fashion, starting with $C_{(0,0)}$. This means, that the same processor will execute chains $C_{(0,0)}$, $C_{(5,0)}$, $C_{(10,0)}$ and so on. The communication volume for this mapping depends on NP and on the chain-offsets $\mathbf{r}_1, \ldots, \mathbf{r}_m$ corresponding to dependence vectors $\mathbf{d}_1, \ldots, \mathbf{d}_m$. In particular, if NP is *greater than the maximum coordinate* appearing in one of the offsets $\mathbf{r}_1, \ldots, \mathbf{r}_m$, then the volume of communication is prohibitive, basically being the same with the one in scenario (1), i.e., $\mathcal{V} \approx 2mC_M|\mathcal{C}|$. Otherwise, if NP is equal to a coordinate of an offset $\mathbf{r}_i$, the communication incurred by the corresponding dependence

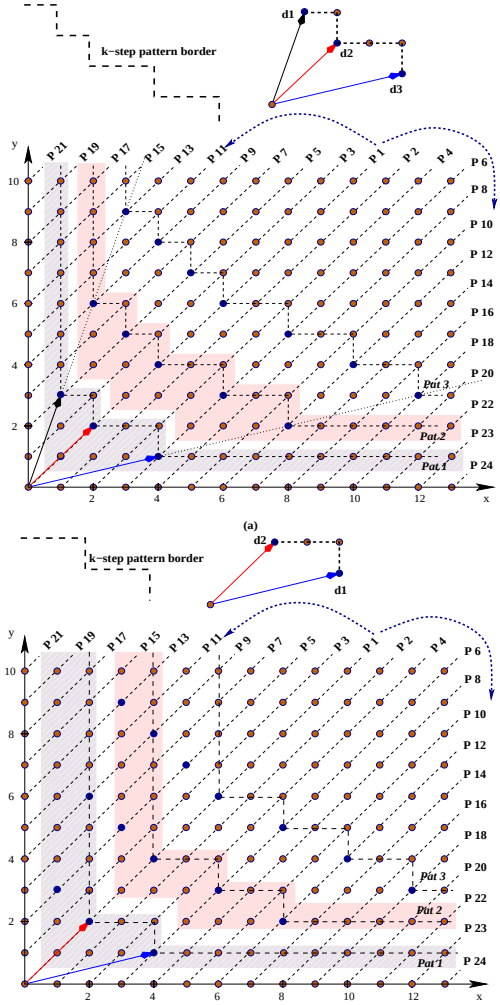


Figure 2. Scenario (a) *Unbounded NP – high communication ⇒ moderate performance*: every chain is mapped to a different processor (unrealistic assumption with respect to the number of available processors).

vector $\mathbf{d}_i$ is eliminated. Hence, let q be the number of offsets that have one of their coordinates equal to NP; then the volume of communication is $\mathcal{V} \approx 2(m-q)C_M|\mathcal{C}|$.

The second mapping, differs from the previous in that it intentionally zeroes the communication cost imposed by as many dependence vectors as possible. In particular, in Fig. 4(a) the first three processors execute the chains "below" $\mathbf{d}_c$ in a round-robin fashion, whereas the other two processors execute the chains above $\mathbf{d}_c$, again in a round-

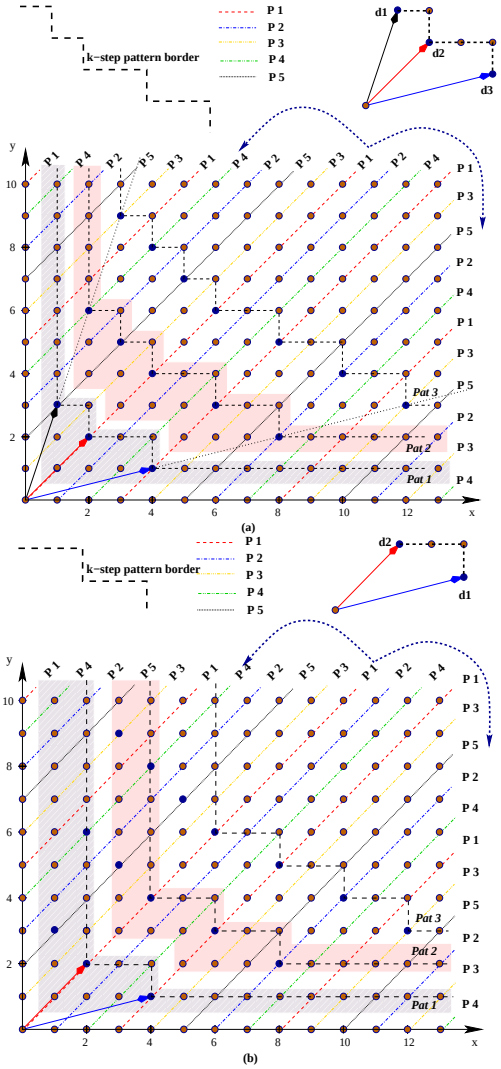


Figure 3. Scenario (2) *cyclic mapping*: chains are mapped to the available processors in a cyclic fashion starting with chain C_0.

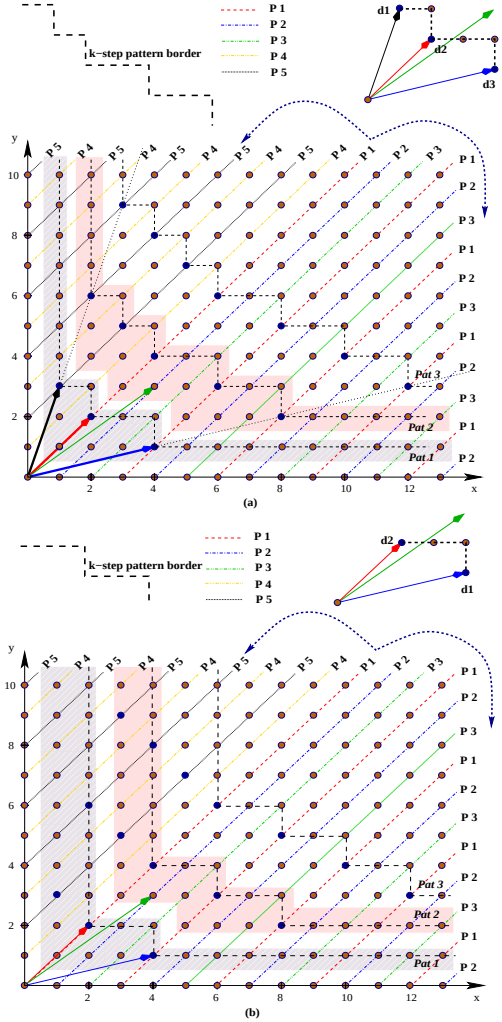


Figure 4. Scenario (2) *mapping with offset 3 along the x axis, and offset 2 along the y axis*: chains are mapped to the available processors so as to minimize communication imposed by $\mathbf{d}_c$, $\mathbf{d}_1$ and $\mathbf{d}_3$.

robin fashion. This way, the communication cost attributed to $\mathbf{d}_3$ is eliminated for the chains $C_{(0,0)}$ to $C_{(13,0)}$ and the communication cost attributed to $\mathbf{d}_1$ is eliminated for the chains $C_{(0,1)}$ to $C_{(0,10)}$. The difference from Fig. 3(a) is that in this case, the processors do not span the entire index space, but only a part of it (i.e., below or above $\mathbf{d}_c$). The benefits of doing so are that a more realistic NP is assumed while still having acceptable communication needs, and performance is increased as a result. In Fig. 4(b) due to the fact that there is no dependence vector above $\mathbf{d}_c$, a *cyclic* mapping of chains is possible, starting with the chain $C_{(0,0)}$ and moving above and below, so as to incrementally release points for execution. This is similar to Fig. 3(b), with the difference that 5 processors were used instead of 3. The advantage of this scenario is that it does not limit the degree of parallelism because it uses *all the available* processors, hence being the most realistic of the two scenarios.

To estimate the volume of communication in this case we reason as follows: consider NP as the sum of $q_1, \ldots, q_l$ (i.e., NP $= q_1 + \ldots + q_l$ and, ideally, $l = m$) such that each q_i is a coordinate

of offset $\mathbf{r}_i$. By proper assignment of groups of q_i chains to processors, the communication cost incurred by $\mathbf{d}_i$ is greatly diminished throughout the index space. Even if such a decomposition of NP in groups of q_i processors is not possible, it is always possible to choose the appropriate integers $q_1, \ldots, q_l$ such that each q_i is a coordinate of $\mathbf{r}_i$ and NP$> q_1+\ldots+q_l$. Thus, by letting $q = q_1+\ldots+q_l$, we conclude that the volume of communication is $\mathcal{V} \approx 2(m-q)C_M|\mathcal{C}|$.

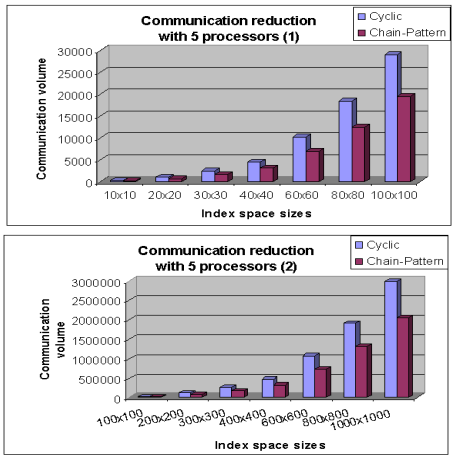


Figure 5. Experimental results for NP = 5

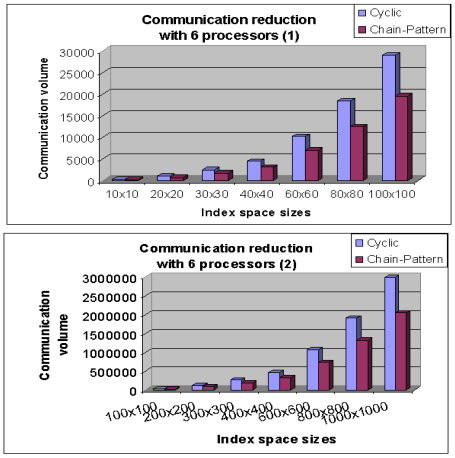


Figure 6. Experimental results for NP = 6

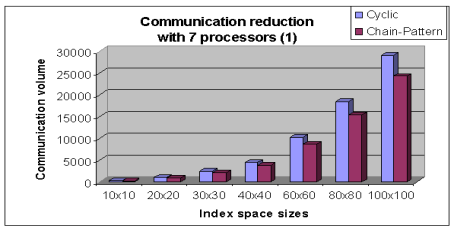
For the sake of simplicity, in both scenarios every chain was assigned to *a single* processor. This is best suited for distributed-memory systems, that consist usually of single processor nodes. Note however, that each chain may be assigned to more than one processors such that no communication among them is required. This is best suited for symmetric multiprocessor systems (SMPs), where processors of a node can communicate through the locally (intra-node) shared-memory. Our approach is also suitable for heterogeneous networks, in which case a processor with higher computational power would be given either longer or more chains, whereas one with lesser computational power would be given either fewer or shorter chains. It is obvious that the ratio $\frac{communication\ time}{processing\ time}$ is critical for deciding which scenario and which architecture suits best a specific application.

4. Performance Results

A program written in C++, which emulates the distribute-memory systems model, was used to validate the proposed methodology. This program implements both the cyclic scheduling method [8], and the chain pattern scheduling proposed here. We tested index spaces ranging from 10×10 to 1000×1000 index points. For all index spaces, the four dependence vectors of the loop nest given in Fig. 1 and the communication vector (2,2) were considered. Fig. 5–8 give the simulation results when NP ranges from 5 to 8 processors. Note that in every case, and for all index space sizes, the chain pattern mapping performs better than the classic cyclic mapping. In particular, the communication reduction achieved with the chain pattern mapping ranges from 15% - 35%.

5. Conclusion

The chain pattern scheduling presented here has some similarities with the static cyclic and static block scheduling methods described in [8]. The similarities are: the assignment of iterations to a processor is determined a priori and remains fixed – yielding reduced scheduling overhead; all three methods require explicit synchronization between dependent chains or tiles. However, the three methods are in the same time different, mainly because: iterations within a chain are independent, whereas



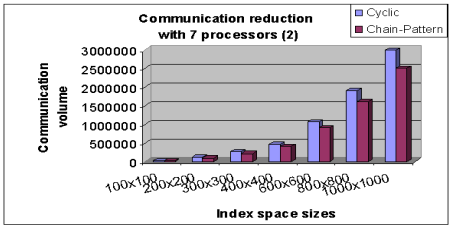
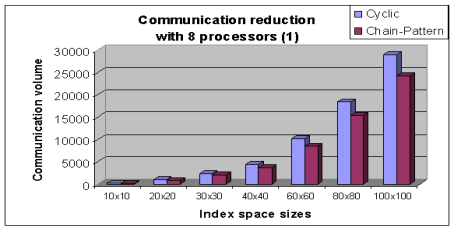


Figure 7. Experimental results for NP = 7



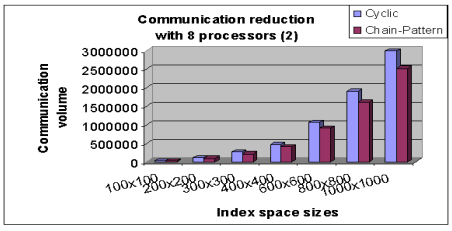


Figure 8. Experimental results for NP = 8

within a tile they are not (this promotes the adaptability of our scheduling to different architectures, as previously mentioned); our method significantly enhances the data locality, hence performs best when compared to the static cyclic or block methods. Our methodology can be easily augmented to automatically generate the appropriate SPMD parallel code, and we are currently working towards this goal. Extensive tests show that the proposed method outperforms in all cases the classic cyclic mapping, succeeding to reduce the communication by 15%-35%, which corroborate the efficiency of our approach.

References

[1] I. Ahmad and Y.-K. Kwok. On exploiting task duplication in parallel program scheduling. *IEEE Trans. on Par. and Dist. Sys.*, 9(9):872–892, 1998.

[2] S. Darbha and D. Agrawal. Optimal scheduling algorithm for distributed-memory machines. *IEEE Trans. on Par. and Dist. Systems*, 9(1):87–95, 1998.

[3] M. Dhodhi, I. Ahmad, and A. Yatama. An integrated technique for task matching and scheduling onto distributed hererogeneous computing system. *J. of Par. and Dist. Comp.*, 62(9):1338–1361, 2002.

[4] I. Drositis, T. Andronikos, M. Kalathas, G. Papakonstantinou, and N. Koziris. Optimal loop parallelization in n-dimensional index spaces. In *Proc. of the 2002 Int'l Conf. on Par. and Dist. Proc. Techn. and Appl. (PDPTA'02)*, 2002.

[5] I. Drositis, T. Andronikos, A. Kokorogiannis, G. Papakonstantinou, and N. Koziris. Geometric pattern prediction and scheduling of uniform dependence loops. In *5th Hellenic Eur. Conf. on Comp. Math. and its Appl. - HERCMA 2001*.

[6] M. Garey and D. Johnson. *Comp. and Intractability, a Guide to the Theory of NP-completeness*. W.H. Freeman & Co., New York, 1979.

[7] O. Kang and D. Agarwal. Scalable scheduling for symmetric multiprocessors (smp). *J. of Par. and Dist. Computing*, 63(3):273–285, 2003.

[8] N. Manjikian and T. Abdelrahman. Exploiting Wavefront Parallelism on Large-Scale Shared-Memory Multiprocessors. *IEEE Trans. on Par. and Dist. Systems*, 12(3):259–271, 2001.

[9] B. L. Massingill, T. G. Mattson, and B. A. Sanders. Patterns for parallel application programs. In *Proc. of the 6th Pattern Lang. of Prog. (PLoP '99)*, 1999.

[10] T. G. Mattson, B. A. Sanders, and B. L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 1st edition, 2004.

[11] C. Oguz, M. Ercan, T. Cheng, and Y. Fung. Heuristic algorithms for multiprocessor task scheduling in a two-stage hybrid flow-shop. *Eur. J. of Open Resources*, 149(2):390–403, 2003.

[12] G. Papakonstantinou, T. Andronikos, and I. Drositis. On the parallelization of UET/UET-UCT loops. *NPSC J. on Computing*, 2001.

[13] C.-I. Park and T.-Y. Choe. An optimal scheduling algorithm based on task duplication. *IEEE Trans. on Comp.*, 51(4):444–448, 2002.

[14] A. Radulescu and A. van Gemund. Low-cost task scheduling for distributed-memory systems. *IEEE Trans. on Par. and Dist. Sys.*, 13(6):648–658, 2002.