Chain Pattern Scheduling for nested loops*

Florina Ciorba, Theodore Andronikos and George Papakonstantinou

Computing Systems Laboratory,
Computer Science Division,
Department of Electrical and Computer Engineering,
National Technical University of Athens,
Zographou Campus, 15773, Athens, Greece

Email: {cflorina, tedandro}@cslab.ntua.gr

Abstract

It is well known that most time consuming applications consist of nested DO(FOR) loops. The iterations within a loop nest can 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 exploiting the geometric properties of the index space in order to arrive at an efficient geometric decomposition. General multiprocessor scheduling with precedence constrains was proved to be NP-complete [1], even when the number of available processors is unbounded [2]. However, in our case things are simpler, 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 [3][4]. One such a case is when the initial

*Partially supported by the Greek State Scholarships Foundation.
wavefront is “repeated” $k$ times at moment $k$ (see Figure 2(b)). Using wavefronts one can find the optimal solution regarding the total makespan, in contrast to the classic hyperplane approach [5] where the solution provided is nearly optimal, achieving a makespan of constant delay from the earliest possible.

When parallelizing nested loops, the following tasks need to be performed: (1) detection of the inherent parallelism and application of program transformations that enable it; (2) computation scheduling (specifying when the different computations are performed); (3) computation mapping (specifying where the different computations are performed); (4) explicitly managing the memory and communication (this is an additional task for distributed memory systems); and (5) generating the code so that each processor will execute its apportioned computation and communication explicitly. In this paper we focus on the first three tasks: parallelism detection, computation scheduling and computation mapping, or in other words the time-scheduling and space-mapping. We achieve these via the use of a novel and efficient communication scheme we call chain pattern for data exchange between processors. We are currently working on the code generation based on the this scheme for various high performance architectures.

This work is an extension of [3][4], in which a geometric method of pattern-outline prediction was presented for uniform dependence loops, but no space mapping method was presented. In [3] and [4] 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 [6]. In this paper we combine the method in [3] with the static cyclic scheduling of [6], thus presenting the chain pattern scheduling, which is an improvement with respect to [3][6] in that it enhances the data locality utilizing methods from computational geometry, in order to take advantage of the regularity of the index space. In particular, we identify groups of iteration points that we call chains, which are points connected via a specific dependence vector, called communication vector $d_c$ [4]. Subsequently, we map such chains to the same processor so as to enhance the data locality.

The contributions of this work are the following: (1) we present a simple method for scheduling nested loops while enhancing 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. It can also be noted that our method is general enough to be applied to various high performance computing architectures, such as distributed systems, SMP systems, heterogeneous systems.
Keywords: patterns, data locality, uniform dependence loop, nested loop scheduling.

1 Definitions and Notation

The index space $J$ of an $n$-dimensional uniform dependence loop is a $n$-dimensional subspace of $N^n$. Due to the existence of dependence vectors, only a certain set of points can be executed at every moment [4]. The geometric border of this set, forms a polygonal shape called pattern. Our scheduling policy executes every index point at its earliest computation time (ECT), forced by the existing dependence vectors. This policy guarantees the optimal execution time of the entire loop (under the assumption of 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 $k$. 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 Figure 1) and is denoted as $Pat_1$. The pattern-outline is the upper boundary of each $Pat_k$ and is denoted as $pat_k$, $k \geq 1$. In this work, the points of the pattern outlines are given and computed in clockwise order that reflects the lexicographic ordering of the initial loop, unless noted. Therefore, $pat_k$ is a list of points, not a simple set. 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 $d_i$ whose end-points are the pattern points of the initial pattern. Their number is usually denoted as $m$.

Figure 1 depicts these definitions.

Partitioning the index space into time patterns 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 [7] [8]. 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 Figure 1. This dependence vector is called communication vector and will be referred to as $d_c$, and is usually chosen to be the dependence (pattern) vector that incurs the largest amount of communication (in most case this is the vector with the “smallest” coordinates). Therefore, $J$ is partitioned into such chains, denoted $C_j = \{ j + k d_c | k \in \mathbb{N}, j \in Pat_0 \}$. $C_j$ is called the chain with origin in $j$. The points of any $C_j$

*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.
Figure 1: The index space of an example algorithm with 4 dependence vectors $d_1 = (1, 3), d_2 = (2, 2), d_3 = (4, 1)$, and $d_4 = (4, 3)$. Each shaded polygon depicts the subset of $J$ that can be executed at a certain moment. This is what we call pattern; the first 3 patterns are showed. The dashed lines mark the border of each pattern, called pattern outline. Except $d_4$, all other vectors contribute to the formation of the pattern outline, therefore are called pattern vectors. On the axes lie the pre-computed at moment $t = 0$ (boundary) points. The communication vector in this example is $d_c = d_2$. The chain with origin $(0, 0)$ $(C_{(0,0)})$ is shown, as well as the chains it sends data to due to $d_1$, $d_4$ and $d_3$, i.e. $C_{(0,2)}$, $C_{(1,0)}$ and $C_{(3,0)}$ respectively.

communicate via $d_i$ ($i$ designating every dependence vector except $d_c$) with the points of chain $C_{j1}$. It is appropriate now to define the **pattern width** as the number of iteration points along chain $C_{j1}$ which belong to $Pat_k$, $k \geq 1$. In Figure 1, for all points between the lines formed by $d_1$ and $d_3$ the pattern width is 2 (points) for all patterns, whereas for all other points it is 1 (point) for all patterns. Hence, it can be noted that the pattern width depends on the dependence vectors of the problem.

## 2 Chain Pattern Scheduling

The *chain pattern scheduling* is devised to enhance the data locality of programs containing uniform nested loops using the concept of chains, while taking advantage of the optimality of the scheduling method based on patterns. When enhancing data locality, there are three possible scenarios with respect to the volume of communication and overall performance of the parallel program: (1) little or no communication at all – high performance, (2) little communication
– mediocre performance, and (3) moderate communication – good performance. By mapping all points of a chain $C_j$ to a single processor, communication incurred by $d_c$ is completely eliminated. Furthermore, mapping all points of a chain $C_j$ and all other chains it communicates with, to the same processor, the communication incurred by the other dependence vectors is also eliminated, increasing the performance of the parallel program using this time-space mapping.

A common feature of all three scenarios is that chains are mapped to processors starting with the 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 this way more points are released (from their dependencies) for execution than in any other way.

For the first scenario, consider the examples shown in Figure 2. This is the case where there are enough available processors so that each chain is mapped to a different processor and executed concurrently. Two similar examples are given: (a) An index space with three dependence vectors, $d_1 = (1, 3)$, $d_2 = d_c = (2, 2)$ and $d_3 = (4, 1)$ where chains are formed along $d_2$. (b) An index space with two dependence vectors, $d_1 = (4, 1)$ and $d_2 = d_c = (2, 2)$, where chains are formed also along $d_2$. One can see that in both cases, 24 chains would be needed. This usually is an unrealistic assumption, since the number of processors must depend on the problem size, which in most cases is pretty large.

![Figure 2: Scenario (1) little or no communication – high performance: every chain is mapped to a different processor (unrealistic assumption with respect to the number of available processors).](image-url)
For the second scenario, consider the examples given in Figure 3. In order to reduce the amount of communication, the chains “above”\(^{†}\) and “below”\(^{‡}\) \(d_c\) are mapped to as many different processors as needed for this scenario, so as to minimize the number of required processors. In particular, in (a) chains below \(d_c\) are mapped to 3 different processors so as to eliminate the communication due to \(d_3\), while above \(d_c\) chains are mapped to 2 different processors so as to eliminate the communication due to \(d_1\). Similarly in (b) chains below \(d_c\) are mapped to 3 different processors, thus eliminating the communication due to \(d_1\). However, there is no other dependence vector creating communication above the chosen \(d_c\), therefore, the mapping scheme applied below \(d_c\) is also applied to the chains above \(d_c\), hence yielding no communication at all. Therefore, the case depicted in (b) can be considered a particular case of scenario (1), where there are sufficiently few dependence vectors so that all communication is eliminated using a reasonable number of processors. This scenario is more realistic than (1), however it does not yield very good performance, since the number of processors required is in most cases small, therefore limiting the parallelism to a certain extent.

Figure 3: Scenario (2) little communication – mediocre performance: chains above and below \(d_c\) are mapped to processors so as to minimize the communication due to the dependence vectors above and below \(d_c\).

\(^{†}\)The chains “above” designate chains to the left of chain \(C_{(0,0)}\).

\(^{‡}\)The chains “below” designate chains to the right of chain \(C_{(0,0)}\).
For the third scenario, consider the cases shown in Figure 4. In order to have moderate communication, but better performance than in scenario (2), an arbitrary number of processors was considered (herein 5 processors). In particular, in (a) the first three processors are mapped to chains below $d_c$, whereas the other two processors were mapped to chains above $d_c$. The difference from Figure 3(a) is that in this case, no all chains span the entire index space, but only a part of it (i.e., below or above $d_c$). The benefits of doing so are that a more realistic number of processors is assumed while still having acceptable communication, and performance is increased as such. In (b) due to the fact that there is no dependence vector above $d_c$, 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 Figure 3(b), with the difference that 5 processors were used instead of 3. This scenario has the advantage that it does not limit the degree of parallelism because it uses all the available processors, hence being the most realistic of all three presented scenarios.

Figure 4: Scenario (3) moderate communication – good performance: chains are mapped to the available processors so as to minimize communication (the most realistic of all scenarios regarding the number of processors).

For the sake of simplicity, for all the above scenarios, regardless of the pattern width, every chain was assigned to a single processor. One should note, however, that depending on the pattern width, each chain may be assigned to more than one processors in a way that no
communication among them is required. This is best suited for the case where the target architecture is a distributed memory system, that usually contains single processor nodes. On the other hand, if the width of the pattern is one, a chain may still consist of more than one processor. However, the processors will have to communicate during execution. This is best suited to the case where the target architecture is a symmetric multiprocessor system (SMP), where processors within 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 of points. It is obvious that the ratio \( \frac{\text{communication time}}{\text{processing time}} \) is critical for deciding which scenario and which architecture suits best for a specific application. Theoretical formulas can be derived for obtaining the optimal makespan as a function of the number of processors and the ratio \( \frac{\text{communication time}}{\text{processing time}} \).

3 Conclusion

The pattern chain scheduling described above has some similarities with the static cyclic and static block scheduling methods described in [6]. The similarities are: the assignment of iterations to a processor is determined a priori and remains fixed – yielding reduced scheduling overhead; they require explicit synchronization between dependent chains or tiles. However, the three scheduling methods are in the same time different, and the main differences are: iterations within a chain are independent, while within a tile they are not (this promotes the adaptability of our scheduling to different architectures, as mentioned above); our method significantly enhances the data locality, hence performs best when compared to the static cyclic and block methods. Our methodology can be easily augmented to automatically generate the appropriate SPMD parallel code, and we are currently working towards this goal. Preliminary experimental results show that the presented method outperforms other scheduling techniques and corroborate the efficiency of the adaptive scheduling.

References


