Simple Code Generation for Special UDLs

Florina M. Ciorba[1], Theodore Andronikos[1,2], Dimitris Kamenopoulos[1], Panagiotis Theodoropoulos[1], and George Papakonstantinou[1]

[1] Computing Systems Laboratory
[2] School of Applied Mathematics and Physics
National Technical University of Athens,
15780 Zographou, Athens, Greece

Abstract. This paper focuses on transforming sequential perfectly nested loops into their equivalent parallel form. A special category of FOR nested loops is the uniform dependence loops (UDLs), which yield efficient parallelization techniques. An automatic code generation tool for shared and distributed memory machines, has been developed in order to automatically parallelize these perfectly nested loop structures. This work is based on parallelizing a special category of UDLs, called Scaled GRIDs. Given a sequential loop structure the tool produces its asymptotically optimal equivalent parallel program assuming that a fixed number of processors is available. The new methodology introduced in this paper, for transforming any nest of Scaled GRID UDLs into their equivalent parallel code, always produces the equivalent parallel code in polynomial time. Furthermore, if the targeted architecture is a shared memory system, the produced parallel code is the optimal time scheduling for the available number of processors. However, if the targeted architecture is a distributed memory system, the produced parallel code is an efficient scheduling for this number of processors, and the experimental results are very close to the ideal speedup.

1 Introduction

Loop structures are a potentially rich source of parallelism in programs written in a high-level programming language, such as C or FORTRAN. Therefore, parallelization of loop structures, by assigning and executing different loop iterations to and on each processor of a parallel computer may lead to dramatic improvements in performance. The main goal of parallelizing compilers is exploiting the potential parallelism that resides in these loops structures, by transforming a sequential program into a practically equivalent parallel form. This paper focuses on transforming sequential perfectly nested loop structures into their equivalent parallel form. A special category of FOR nested loops is the uniform dependence loops, which yield efficient parallelization techniques. Many real-life algorithms from digital signal, image and speech processing (e.g. convolution, LU decomposition, dynamic time warping) fall in this category.

When parallelizing loop structures for parallel computers, there are three initial tasks that need to be done. The first deals with the *detection* of the inherent parallelism and applying any program transformation that may enable this. The second task deals with specifying when the different computations are carried out (*computation scheduling*) and the third, where the different computations are carried out

(*computation mapping*). However, if the parallel computer has a distributed memory system, an additional task would be to manage the memory and communication explicitly. Nevertheless, upon completion of all above tasks, the final task in parallelizing is to *generate the code* so that each processor will execute its apportioned computation and communication explicitly. An automatic code generation tool for shared and distributed memory machines, has been developed in order to automatically parallelize these perfectly nested loop structures.

Related work. The work in [1] focused on *computation scheduling*, i.e., finding a schedule of computations in time, while preserving the data dependencies of the initial loop nest. Many methods have focused on accomplishing the optimal time scheduling. The first of these is the hyperplane method introduced by Lamport in [10]. Other methods use free scheduling [8], tiling transformation [6, 7], earliest computation times combined with hyperplane execution [1, 3], and they differ on the method used for partitioning the index space of the loop nests. Another method [14] uses also the data spaces of the loop nest besides the iteration index space, and explicitly formulates array references as transformations from the statement iteration space to data spaces. Other researchers [9] have identified different communication patterns in distributed computations in order to reduce the overall communication cost. In [2] an algorithm was presented for computing an affine scheduling compatible with a given affine computation mapping, if such a schedule exists.

This paper focuses on *computation scheduling*, *computation mapping* and *code generation* for perfectly nested uniform dependence loops. The general scheduling problem of uniform dependence loops is a very special case of scheduling directed acyclic graphs (DAGs). The general DAG multiprocessor scheduling with precedence constraints is known to be NP-complete [5, 15] even when the number of processors is unbounded [12]. Many researchers have tackled the special cases of DAG scheduling [4, 8] hoping to come up with efficient polynomial algorithms. This work is based on parallelizing the special category of uniform dependence loops (UDLs), called Scaled GRIDs (SGRIDs). Every UDL in this category has the general structure of a GRID, i.e., all dependence vectors reside on the axes and the value of the non-zero coordinate is an arbitrary positive integer. GRIDs are a particular case of SGRIDs, where the dependence vectors are the unitary vectors along the axes.

Problem description. Given a sequential loop of SGRID UDLs structure, assuming that a fixed number of processors is available, produce its asymptotically optimal equivalent parallel program. The method proposed in this paper always produces the equivalent parallel code in polynomial time. If the targeted architecture is a shared memory system, the produced parallel code is the optimal time scheduling for the available number of processors. However, if the targeted architecture is a distributed memory system, the produced parallel code is an efficient scheduling for the specified number of processors, and the experimental results are very close to the ideal speedup.

Our approach. Starting with the sequential program, containing as input the loop nest, information about the program is extracted, namely: depth of the loop nest (n), size of the iteration space ($|J|$), and the dependence vectors set (DS). After the dependence analysis, in the iteration index space, a special pattern is obtained, which has the property the points it contains do not depend on each other. Considering the num-

ber of independent points within the pattern is p, the pattern is replicated throughout the whole iteration space, therefore obtaining collections of p independent iteration points, that can be concurrently executed. If each pattern is considered to be a cartesian point in an auxiliary space (J_{aux}), the pattern execution order is traversing this space in a zig-zag manner. Next, the loop nest is scheduled for execution according to the available number of processors, which preferably must be a multiple of p. Finally, a file containing the equivalent parallel version of the sequential program is produced. A tool has been developed implementing the above approach. The file produced by this tool uses MPI library calls for communication between processors that need to exchange data. The developed tool targets the MPI platform (i.e., a distributed memory system) because it provides a flexible environment for developing high performance parallel applications. Message passing occurs whenever a processor needs data located in the memory of another processor. MPI provides an efficient standard to implement message passing applications on different platforms. Our tool uses point to point communication between processors, which is the fundamental communication mode of any MPI multicomputer.

Contribution. The main contribution of this paper is the new scheduling methodology, which is polynomial in the size of the input, in this case, the sequential program. Also, in contrast with other iteration index space partitioning methods, such as tiling where the points within a tile are dependent on each other, the iteration points that constitute the special pattern are completely non-dependent on each other, and can therefore, be executed concurrently. The optimality of the proposed method comes from using a fixed number of hardware resources, and scheduling every eligible for execution iteration point such that most/completely all of the available hardware resources are in use, while preserving the precedence constraints at any time step.

The tool has been tested on a cluster with 16 identical 500MHz Pentium nodes. Each node has 256 MB of RAM and 10GB hard drive and runs Linux with 2.4.20 kernel version. The reported experimental results proved to be very close to the ideal speedup. The paper is structured as follows: the terminology used throughout the paper is given next in Section 2. In Section 3 is given the definition of the special category SGRIDs of UDLs along with an illustrative example, whereas in Section 4 the scheduling policy that the tool uses is presented. Section 5 explains the way the parallel code is generated from the sequential one. Section 6 shows the experimental results along with illustrative charts. Finally, the experimental results are summarized in Section 7 along with proposed further work.

2 Terminology

This paper focuses on perfectly nested FOR-loop structures with uniform data dependencies. The algorithmic model used is the one described in reference [11, p.58], where $\mathtt{l}_i, \mathtt{u}_i \in \mathbb{Z}$, $1 \leq i \leq n$ are integer valued constants representing the lower and upper limits for loop variables, and $S_1, \ldots, S_k$ are assignment statements. The depth of the loop nest is n and $J = \{(i_1, \ldots, i_n) \in \mathbb{N}^n \mid l_r \leq i_r \leq u_r, 1 \leq r \leq n\}$ is the n-dimensional iteration index space. Each point in this discrete cartesian n-dimensional space is a distinct iteration of the loop body and is represented as coordinates in

the index space. $\mathbf{L} = (l_1, \ldots, l_n)$ and $\mathbf{U} = (u_1, \ldots, u_n)$ are the **initial** and **terminal points**, respectively, of the index space, $\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n$ are the unitary axes vectors and $DS = \{\mathbf{d}_1, \ldots, \mathbf{d}_m\}$ is the set containing the m dependence vectors. By definition, dependence vectors are always $>$ than $\mathbf{0}$, where $\mathbf{0} = (0, \ldots, 0)$ and $>$ is the *lexicographic* ordering. The dependence vectors are "uniform", i.e., constant all over the index space. The cardinality of J, denoted $|J|$, is $\prod_{i=1}^{n}(u_i - l_i + 1)$. The following section gives the definition of the special category of UDLs approached herein, *Scaled* GRIDs.

3 Scaled GRIDs

Definition 31 *The Scaled GRIDs are a subclass of the general class of UDLs in which the set DS is partitioned into two disjoint subsets: AXV and $DS - AXV$. The AXV (Axes Vectors) subset contains all the dependence vectors that lie along the axes, i.e., each dependence vector has the form $\mathbf{d} = \lambda \cdot \mathbf{e}_i$, where $\mathbf{e}_i$ is the unitary vector along dimension i and $\lambda \geq 1$ is a positive integer. Further, we require that for every i, $1 \leq i \leq n$, there exists at least one $\mathbf{d} = \lambda \cdot \mathbf{e}_i$.*

Definition 32 *The special pattern (SP) is a collection of iteration points that are non-dependent on each other throughout the execution of the whole nest, obtained by considering the vectors with the smaller values from the AXV set, such that all other dependence vectors of $DS - AXV$ do not create any dependency between iteration points within the pattern.*

The well known GRIDs are a special case of SGRIDs in which $\lambda = 1$ for all the dependence vectors, and any resulting SP contains only one iteration point, due to the unitary dependence vectors that characterize GRIDs.

Considering the number of completely independent points within the pattern is p, by replicating the SP throughout the whole iteration space, collections of p independent points are obtained, that obviously can be concurrently executed, provided there are available hardware resources. Therefore, the loop nest can be scheduled according to the available number of processors, that should preferably be a multiple of p. In case the number of processors at hand is not a multiple of p then at some moment, several of the processors will stay idle because of no available iteration points eligible for execution. Also, if the number of available resources is less than p the results produced by our tool are not satisfactory. Therefore, for the sake of simplicity, the case of having multiple of p processors at disposal is tackled.

Example 31 *As an example of an SGRID, consider a two-dimensional index space representing the following two-level loop nest:*

```
for (i=1; i<=10; i++)
    for (j=1; j<=10; j++) {
        A[i,j]=(A[i,j-2]*A[i-2,j])^20;
        B[i,j]=2*B[i,j-3]^10+B[i-4,j]^10-1;
    }
```

For this example, the initial point of the index space is $\mathbf{L} = (1,1)$, the terminal point is $\mathbf{U} = (10,10)$ and the cardinality of $|J| = 100$ points. The dependence vectors are $\mathbf{d}_1 = (0,2)$, $\mathbf{d}_2 = (2,0)$, $\mathbf{d}_3 = (0,3)$ and $\mathbf{d}_4 = (4,0)$. Due to the size of the smallest vectors on each axe, the SP is a collection of 4 independent iteration points. The iteration index space with the replicated patten is shown in Figure 1(a), while the auxiliary space is depicted in Figure 1(b). Notice the initial point in the auxiliary space is $\mathbf{L_{aux}} = (0,0)$ and contains $(1,1)$ (the initial iteration point), $(2,1)$, $(1,2)$ and $(2,2)$ iteration points. The terminal point in the auxiliary space $\mathbf{U_{aux}} = (4,4)$ contains the iteration points: $(9,9)$, $(9,10)$, $(10,9)$ and $(10,10)$ (the terminal iteration point). The cardinality of the auxiliary space is $|J_{aux}| = 25$ patterns. ◀

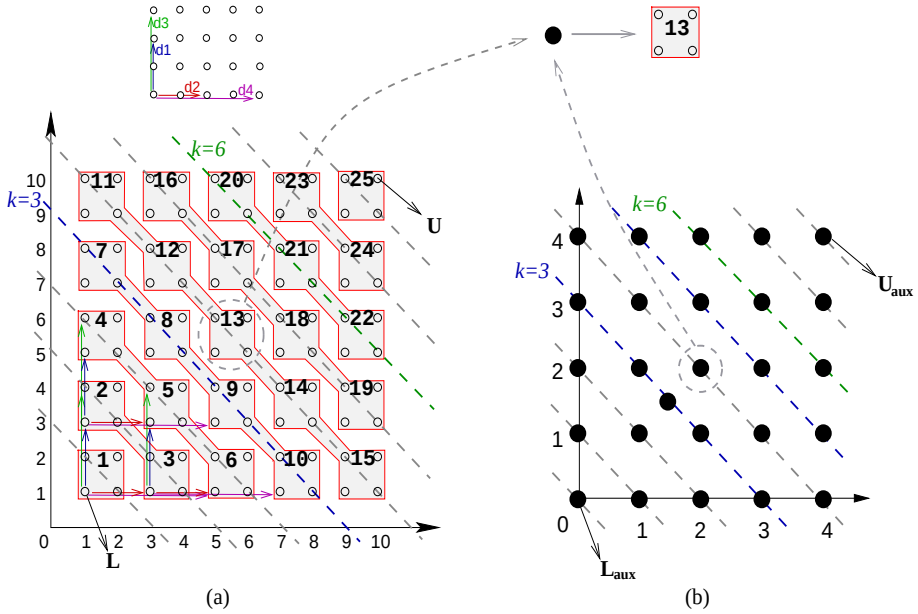

Fig. 1. *(a) the iteration index space J and (b) the auxiliary space J_{aux} of Example 31*

The iteration index space J is partitioned into hyperplanes (or hypersurfaces) using the concept of *earliest computation time* [1,13], that is, all the points that can be executed the earliest at moment k are said to belong to hyperplane k. As it can be seen in Figure 1, all the points that belong to hyperplane 3 in J, belong to the same hyperplane 3 in the auxiliary space J_{aux} too. Of course, this holds for every hyperplane of J.

Next section describes the scheduling policy adopted in the tool.

4 Scheduling Policy

The main contribution of this paper is the scheduling policy used to parallelize the loop nest. The tool schedules concurrently all the eligible iteration points gathered in a pattern, because there is no need to exchange data between iteration points of the same pattern. Given an iteration space replicated with the special pattern, it is traversed pattern by pattern according to the lexicographic ordering of the coordinates resulting if each pattern is taken as a cartesian point in an *auxiliary space*. Therefore, the order of patterns execution is the lexicographically traversal of the points on every hyperplane (zig-zag ordering) of the auxiliary space (noted J_{aux}), as shown in Figure 1(b). In other words, all the patterns in the iteration space J are executed in a lexicographic order, . At every moment, all the available points for execution are executed onto the existing hardware resources, in groups of p processors. The processors within each such group work concurrently, due to being assigned iteration points that do not depend on each other. It is obvious though, that two distinct groups of processors may exchange data (i.e., communicate), if there is any dependency between the iteration points of one pattern and iteration points of the other.

Consider the available resources are x processors. They are gathered so as to form groups of p processors. Initially, the first group executes the first pattern (i.e., $\mathbf{L_{aux}}$), the second executes the second pattern, and so on, until the last group executes the x/p-th pattern. Once the initial pattern assignment is done, every other pattern for each group is determined by skipping x/p positions from the currently assigned pattern. In other words, the first group gets the $\mathbf{L_{aux}} + x/p$-th pattern, the second group gets the $\mathbf{L_{aux}} + x/p + 1$-th pattern, and so on, until the last group gets the $\mathbf{L_{aux}} + 2 \times x/p$-th pattern. One can notice that, when all groups are assigned patterns belonging to the same hyperplane, they can all compute concurrently. This happens only on hyperplanes that have x/p patterns. The scheduling continues this way, until reaching and executing the pattern containing the terminal point $\mathbf{U}$ of J.

Considering that the loop nest of Example 31 is written in a high level programming language (such as C), a classical parser is used to extract necessary information about the program. In other words, the depth of the loop nest ($n = 2$), the size of the iteration index space (i.e., $|J| = 100$), and the $m = 4$ dependence vectors are determined.

The tool generates from the dependence analysis the number of independent iteration points that form a pattern, i.e., $p = 4$. After determining this information, and before applying the scheduling policy described previously, the tool requires as user input the number of available hardware resources the sequential code is to be parallelized on. Supposing 12 processors are available, they are scattered so as to form three groups of p processors each, having determined previously that a pattern contains 4 independent points. According to the scheduling policy, the first such group (G_1) gets assigned the pattern of the first hyperplane (hyperplane 0), the second (G_2) gets assigned the next lexicographic pattern (hyperplane 1) and the third (G_3) gets the next lexicographic pattern (also on hyperplane 1). After the initial assignment of patterns to each group, the next pattern for each is the one found three positions further in the lexicographic ordering. In other words, the second pattern for G_1 is found by skipping three patterns from the initial pattern position (0,0), i.e. the first pattern on hyperplane 2, (0,2). The second pattern for G_2 is the pattern at (1,1), three position further

from the initial pattern (0,1), and G_3 gets (2,0), three position further from the initial pattern (1,0). The *skipping method* of traversing the index space is used in order to avoid doubly executing some pattern (or iteration points) and constitutes the main benefit of using multiple groups of processors.

The loop body of the sequential program is taken together with all assignment statements it contains and is passed on to each processor that will execute it for the appropriate iteration index point. It is important to recall that the only way to distinguish processors in MPI is referencing them by their rank. A rank is a positive integer assigned to each processor in the multicomputer that designates the identification number of the processors throughout the parallel execution. Therefore, in order to know from which processors to receive necessary data and/or to send computed data to, the available processors are assigned ranks as follows: processors from first group get ranks between 0 and 3, processors from second group get ranks between 4 and 7 and the ones of third group get ranks between 8 and 11. The parallel code is generated upon completion of above phases and is described in the next section.

5 Parallel Code Generation

This section gives the parallel code the tool produced for the Example 31. The generated pseudocode for the equivalent parallel version of the program is the following:

```
forall (available processors)
 /* get coordinates of initial pattern and processor rank */
 current_pattern = get_initial_pattern(rank);
 while (current_pattern!=out_of_auxiliary_space)
  {
  /* we know the current pattern and the rank, find the */
  /* iteration point to be executed by each processor with rank */
  current_iteration_position=find_position(current_pattern, rank);
  /* receive data from all points the current point depends on */
  MPI_Recv(from_all_dependencies);
  /* execute the loop body associated to the current iteration */
  execute(Loop_Body, current_iteration_position);
  /* send data to all points that depend on current point */
  MPI_Send(to_all_dependencies);
  /* move on to the next pattern */
  current_pattern=Skip_3_patterns(current_pattern);
  }
endforall
```

where the function `get_initial_pattern(rank)` is the following:

```
get_initial_pattern(rank){
    if (0 <= rank <=3) return L_aux
    else if (4 <= rank <=7) return Skip_1_pattern(L_aux)
    else if (8 <= rank <=11) return Skip_2_patterns(L_aux)
}
```

Herein `L_aux` represents the initial point of the auxiliary space, i.e., $\mathbf{L_{aux}}$. This function returns the initial pattern coordinates for every of the three groups of 4 processors using the skipping method and according to their rank values. The generated file containing MPI is ready to be executed onto the available processors.

6 Results

The MPI code the tool produced uses point-to-point communication and standard blocking send and receive operations, meaning that a send or receive operation blocks the current process until resources used for the operation can be reutilized. The experiments were ran on a cluster with 16 identical 500MHz Pentium nodes. Each node has 256MB of RAM and 10GB hard drive and runs Linux with 2.4.20 kernel version. The MPI (MPICH) was used to run the experiments over the FastEthernet interconnection network.

The performance of the tool was evaluated by experimenting with randomly generated UDLs. The speedup of the tool from the sequential program was measured. The tool can schedule to achieve speedups that scale with the number of available hardware resources. As the experimental results (Figure 2) validated, the introduced methodology obtained near optimal results for seven reasonably sized randomly generated applications, given below. The tool adapts well to shared memory systems by simply replacing the MPI communication calls with read and write calls from and to the shared memory of the multicomputer. The testing examples are the following:

1. $n = 2$, $|J| = 400$ points, $d_1 = (0, 2)$, $d_2 = (1, 0)$ and $d_3 = (1, 2)$
2. $n = 2$, $|J| = 600$ points, $d_1 = (0, 1)$, $d_2 = (1, 0)$ and $d_3 = (1, 1)$
3. $n = 2$, $|J| = 1024$ points, $d_1 = (0, 4)$, $d_2 = (2, 0)$ and $d_3 = (4, 2)$
4. $n = 3$, $|J| = 1000$ points, $d_1 = (0, 0, 2)$, $d_2 = (0, 2, 0)$, $d_3 = (2, 0, 0)$ and $d_4 = (2, 2, 2)$
5. $n = 3$, $|J| = 2744$ points, $d_1 = (0, 0, 2)$, $d_2 = (0, 2, 0)$, $d_3 = (1, 0, 0)$ and $d_4 = (1, 2, 2)$
6. $n = 3$, $|J| = 2744$ points, $d_1 = (0, 0, 2)$, $d_2 = (0, 1, 0)$, $d_3 = (1, 0, 0)$ and $d_4 = (1, 1, 2)$
7. $n = 3$, $|J| = 3375$ points, $d_1 = (0, 0, 1)$, $d_2 = (0, 1, 0)$, $d_3 = (1, 0, 0)$ and $d_4 = (1, 1, 1)$

7 Conclusions and Further Work

The experimental results proved to be very close to the ideal speedup. However, by using standard blocking send/receive communication primitives, the tool introduces a communication overhead which can slightly reduce the speedup when the number of utilized processors exceeds an optimal threshold, i.e., when the communication to computation ratio becomes higher than one. On the other hand, the choice of using blocking communication primitives brings the advantage of accuracy regarding the appropriate data exchanged between processors at execution, and the experimental results proved to be near optimal even in this case.

Further work will focus on optimizing the tool with non-blocking communication primitives, and on porting the tool onto different interconnection networks such as SCI and Myrinet, both intended for reducing the time each processors spends communicating, in other words, the communication costs.

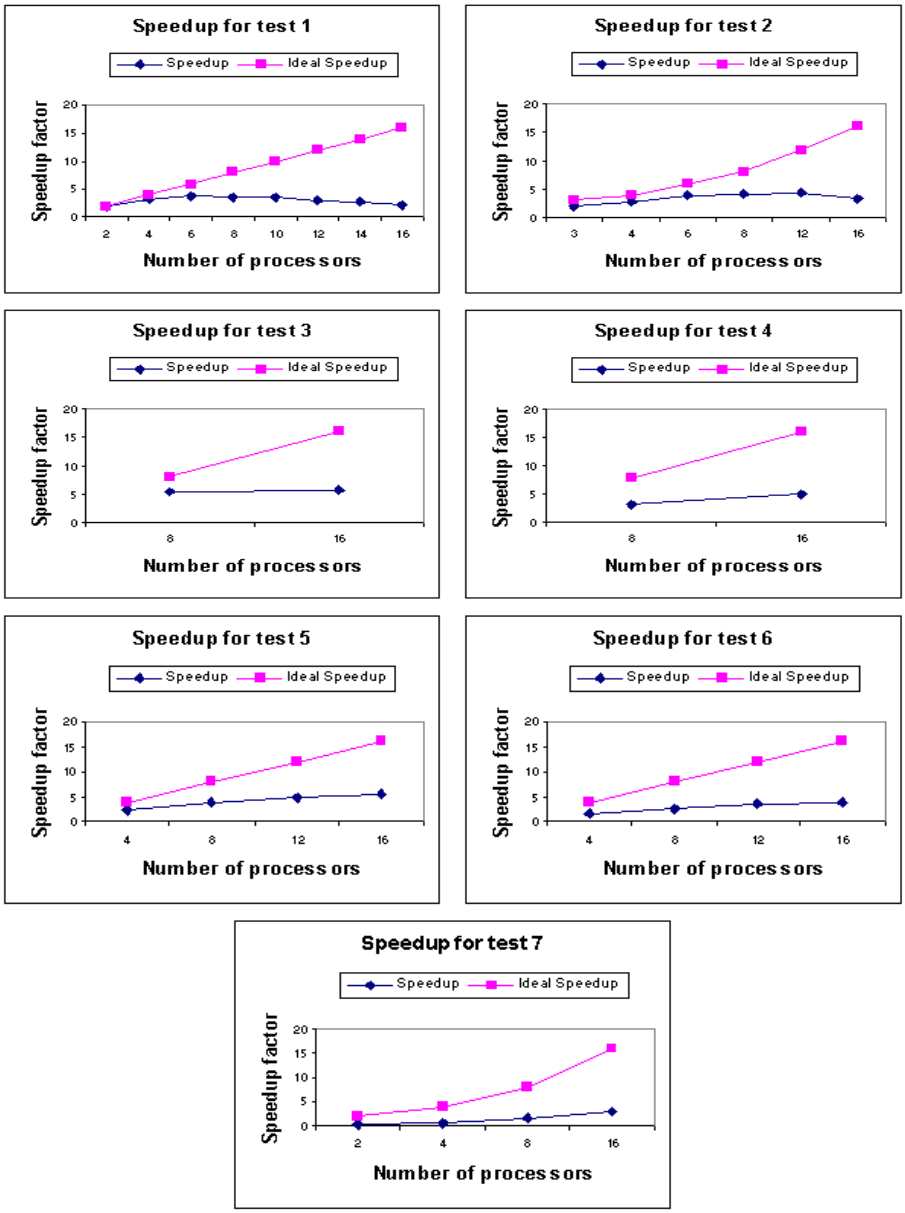

Fig. 2. *Speedup for seven testing examples*

References

1. T. Andronikos, M. Kalathas, F.M. Ciorba, P. Theodoropoulos, and G. Papakonstantinou. Scheduling nested loops with the least number of processors. In *Proceedings 21st IASTED*

International Conference on Applied Informatics, 2003.
2. A. Darte, C. Diderich, M. Gengler, and F. Vivien. Scheduling the computations of a loop nest with respect to a given mapping. In *Proceedings 8th International Workshop on Compilers for Parallel Computers*, Aussois, France, 2000.
3. I. Drositis, T. Andronikos, A. Kokorogiannis, G. Papakonstantinou, and N. Koziris. Geometric scheduling of 2-D uniform dependence loops. In *Proceedings International Conference on Parallel and Distributed Systems (ICPADS)*, pages 259–264, Korea, 2001.
4. D.W. Engels, J. Feldman, D.R. Karger, and M. Ruhl. Parallel processor scheduling with delay constraints. In *Proceedings 12th Annual Symposium on Discrete Algorithms (SODA)*, pages 577–585, New York, NY, 2001.
5. M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-completeness*. W.H. Freeman, NY, 1979.
6. G. Goumas, M. Athanasaki, and N. Koziris. Automatic code generation for executing tiled nested loops onto parallel architectures. In *Proceedings 17th ACM Symposium on Applied Computing (SAC)*, pages 876–881, 2002.
7. G. Goumas, A. Sotiropoulos, and N. Koziris. Minimizing completion time for loops tiling with computation and communication overlapping. In *Proceedings International Symposium on Parallel and Distributed (ICPADS)*, California, 2001.
8. N. Koziris, G. Papakonstantinou, and P. Tsanakas. Optimal time and efficient space free scheduling for nested loops. *The Computer Journal*, 39(5):439–448, 1996.
9. A.D. Kshemkalyani and M. Singhal. Communication patterns in distributed computations. *Journal of Parallel and Distributed Computing*, 62:1104–1119, 2002.
10. L. Lamport. The parallel execution of DO loops. *Communications of the ACM*, 37(2):83–93, 1974.
11. D. I. Moldovan. *Parallel Processing: From Applications to Systems*. Morgan Kaufmann, California, 1993.
12. C. Papadimitriou and M. Yannakakis. Toward an architecture-independent analysis of parallel algorithms. *SIAM Journal of Computers, (Extended Abstract in Proceedings STOC88)*, 19:322–328, 1990.
13. G. Papakonstantinou, T. Andronikos, and I. Drositis. On the parallelization of UET/UET-UCT loops. *NPSC Journal on Computing*, 2001.
14. K.-P. Shih and J.-P. Sheu. Statement-level communication-free partitioning techniques for parallelizing compilers. *The Journal of Supercomputing*, 15:243–269, 2000.
15. J. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10:384–393, 1975.