Parallel Computing 18 (1992) 625-637
North-Holland

PARCO 681

A hybrid scheme for efficiently executing nested loops on multiprocessors

Chien-Min Wang [a] and Sheng-De Wang [b]

[a] *Institute of Information Science, Academia Sinica, Nankang, Taipei 11529, Taiwan*
[b] *Department of Electrical Engineering, National Taiwan University, Taipei 10764, Taiwan*

Received 22 January 1991
Revised 5 November 1991

Abstract

Wang, C.-M. and S.-D. Wang, A hybrid scheme for efficiently executing nested loops on multiprocessors, Parallel Computing 18 (1992) 625-637.

In this paper, we address the problem of scheduling parallel processors for efficiently executing nested loops. The goal is to achieve optimal load-balancing by using a few scheduling and communication operations as possible. For this purpose, we propose a new scheduling scheme called *multilevel interleaved guided self-scheduling* (MIGSS). It is a hybrid scheduling scheme blending with run-time scheduling techniques and compile-time loop restructuring. Its run-time scheduling is based on guided self-scheduling. A compile-time loop transformation method is proposed to enhance the parallel execution performance. It is proved that, by this scheme, we can achieve optimal load-balancing and avoid significant communication overheads. Experimental results clearly show the relative performance improvement of MIGSS over guided self-scheduling.

Keywords. Guided self-scheduling; hybrid scheduling scheme; load-balancing; loop transformation; multiprocessor; nested loop; run-time overhead.

1. Introduction

Recently, it has become more and more difficult to achieve order of magnitude speedups by uniprocessor systems because device technology places an upper bound on the speed of any single processor. Multiprocessor systems offer a promising and powerful alternative for high-speed computing. Because of the flexibility, scalability, and high performance provided by multiprocessor systems, it is widely believed that this architecture will be a dominant architecture in the development of the future general purpose supercomputers.

In principle, the speedup obtained by multiprocessor systems depends only on the parallelism of thc application at hand. The more processors in a multiprocessor system the more speedup can be obtained. However, in order to correctly execute a parallel program on a multiprocessor system, processors must be appropriately coordinated. The overheads of communication, synchronization and scheduling are then incurred and may undo the benefit

Correspondence to: Sheng-De Wang, Department of Electrical Engineering, National Taiwan University, Taipei 10764, Taiwan of multiprocessor systems. In order to efficiently execute a parallel program on a multiprocessor system, the problem of scheduling parallel processors must be solved.

Most scheduling schemes proposed in the past [5,9,18,21,22] consider only an idealized form of the problem where task execution times are fixed and known in advance. This approach is called *static scheduling*. In reality, branching statements in programs, memory access interference, random processor latencies, and other 'random events' make task execution times impossible to predict accurately. For these cases, scheduling should be performed at run-time in order to execute parallel programs efficiently. This approach is called *dynamic scheduling*.

In this paper, we focus our attention on dynamic scheduling schemes to generate very efficient schedules of nested parallel loops for execution on multiprocessor systems. Nested parallel loops, whose iterations can be executed in parallel on different processors, provide the greatest potential of parallelism to be exploited for multiprocessor systems [14]. In addition, scheduling of nested parallel loops incurs less overhead at run-time.

Dynamic scheduling will incur scheduling overhead at run-time. Scheduling overhead can be significant if scheduling is done by system calls to the operating system. One existing technique to reduce scheduling overhead is *processor self-scheduling* [8,13,23]. In this scheme, an iteration of a parallel loop is assigned to a processor whenever it becomes available. Rather than issue a system call to the operating system for scheduling, processors can schedule themselves by performing a fetch-and-add operation on a shared variable. If the multiprocessor system has efficient hardware-implemented synchronization primitives for this operation, the scheduling overhead can be reduced significantly.

However, processor self-scheduling suffers a serious disadvantage. The total number of scheduling operations performed by processor self-scheduling is equal to the total number of iterations in a nested loop. Although the overhead of scheduling an iteration by processor self-scheduling is small, there can be so many scheduling operations such that the total scheduling overhead is still significant. To reduce the number of scheduling operations, *guided self-scheduling* (GSS) was proposed [20]. For a simple parallel loop, if iterations have constant execution time, GSS not only obtains an optimal schedule under any initial processor configuration but also uses the minimum possible number of synchronization points necessary to achieve optimal load-balancing.

A major disadvantage of GSS is that it does not always perform well for a nested loop. For a nested loop, the schedule generated by GSS is often not an optimal one. Further, the number of scheduling operations used by GSS may be very large, and hence GSS may incur significant communication overhead. To efficiently execute nested loops on multiprocessor systems, we propose a new scheduling scheme called *multilevel interleaved guided self-scheduling* (MIGSS). It is a hybrid scheduling scheme based on GSS and several loop transformations. It can be proved that MIGSS not only can achieve optimal load-balancing for a nested loop but also can avoid significant communication overhead.

The rest of this paper is organized as follows. Section 2 gives background information and necessary definitions. A brief introduction to guided self-scheduling is also given in this section. Section 3 and Section 4 present interleaved guided self-scheduling and multilevel interleaved guided self-scheduling, respectively. Section 4 also gives the experimental results. Finally, conclusions are given in Section 5.

2. Background

The multiprocessor system considered in this paper is a shared memory multiprocessor system that contains p identical processors. It is assumed that each processor has its own local memory and processors communicate through shared memory modules. Neither processors nor shared memory modules are distinguishable from their neighbors. Each shared memory module is capable of being accessed by any processor through the interconnection network. Systems in this class include the Alliant FX series, the Denelcor HEP, the University of Illinois Cedar system, the New York University Ultracomputer, and the IBM RP3 multiprocessor.

In a parallel program, we may observe the following three types of loops: *serial* or DOSER loops, *do all* or DOALL loops, and *do across* or DOACR loops. Iterations of a DOSER loop must be executed serially while iterations of a DOALL loop can be executed in parallel and in any order. In a DOACR loop, the execution of successive iterations can be partially overlapped. To simplify our notations, each loop is assumed to be normalized (i.e. its iteration space is of the form $[1, \ldots, N]$, $N \in Z^+$). We may assume that any (*serial* or *do all* or *do across*) DO loop has the following form:

```
DO I=1, N
  {B}
ENDDO
```

where I is the loop index, N is the loop bound, and B is the loop body. The loop body may contain a set of statements with constant (independent of I) execution time, or other loops. If no other loop is contained in the loop body, the DO loop is called an *innermost* loop. Otherwise, the DO loop is called an *outer* loop. In a nested loop, an individual loop can be enclosed by many outer loops. The *nest level* of an individual loop is equal to one plus the number of the enclosing outer loops. The *nest depth* of a nested loop is the maximum nest level of loops in the nested loop. In a perfectly (or one-way) nested loop of nest depth m, there exists exactly one loop at each nest level, i, $i = 1, 2, \ldots, m$. Therefore, a perfectly (or one-way) nested loop of nest depth m is a loop of the form:

```
DO I_1=1, N_1
  Do I_2=1, N_2
    ...
        DO I_m=1, N_m
          {B}
        ENDDO
    ...
  ENDDO
ENDDO
```

A loop is k-way nested if there exist k disjoint loops at the same level. For convenience, it is assumed that individual loops in an arbitrarily nested loop are numbered increasingly, in lexicographic order. Furthermore, the number of iterations of loop i is denoted by N_i. Nested loops that contain combinations of DOALL, DOACR, and DOSER loops are called *hybrid* loops. In this paper, we concern with the dynamic scheduling of hybrid nested loops composed of DOSER and DOALL loops only.

Most self-scheduling schemes assign a single iteration for execution at a time. *Guided self-scheduling* (GSS) [20] follows another approach by assigning several consecutive iterations, called an *iteration block*, to each idle processor. Each idle processor will receive a block of $\lceil R/p \rceil$ iterations when there are R iterations unassigned. Suppose that a DOALL loop with N iterations is to be executed on P processors. The number of synchronizations points used by this scheme can be proved to be p in the best case, and $O(p \ln\lceil N/p \rceil)$ in the worst case.

For perfectly nested DOALL loops, GSS can become more efficient by using *loop coalescing* [19]. Suppose that we have a perfectly nested DOALL loop L of nest depth m.

Loop coalescing coalesces all m individual DOALL loops into a single DOALL loop L' with $N = \prod_{i=1}^{m} N_i$ iterations through a set of transformations f_i, $i = 1, 2, \ldots, m$, that map the index I of the coalesced loop L' to the indexes I_i, of the original loop L such that $I_i = f_i(I)$. Each processor can compute locally f_i for a given index I. The global index I is kept in shared memory as a shared variable. The number of synchronization points used by GSS is $O(p \ln\lceil N/p \rceil)$. Comparing it with $O(mN)$ synchronization points used by self-scheduling

```
DOALL I = 1, 10
  DOSER J = 1, 5
    DOALL K = 1, 4
      {B}
    ENDDO
  ENDDO
ENDDO
```

(a)

```
DOSER J = 1, 5
  DOALL L = 1, 40
    I = (L - 1) / 4 + 1
    K = (L - 1) mod 4 + 1
    {B}
  ENDDO
ENDDO
```

(b)

```
PARBEGIN
  DOSER J = 1, 5
    DOALL L = 1, 20
      I = (L - 1) / 4 + 1
      K = (L - 1) mod 4 + 1
      {B}
    ENDDO
  ENDDO
  DOSER J = 1, 5
    DOALL L = 21, 40
      I = (L - 1) / 4 + 1
      K = (L - 1) mod 4 + 1
      {B}
    ENDDO
  ENDDO
PAREND
```

(c)

Fig. 1. An example of GSS and IGSS.

without loop coalescing, we observe that the number of synchronization points can be greatly reduced.

For hybrid perfectly nested loops, loop coalescing can not be directly applied. Consider the hybrid perfectly nested loop L in *Fig. 1(a)*, which was given in [20] as an example. By loop coalescing, we have a total of $N = 200$ iterations. However, assigning consecutive iterations of the coalesced loop to each idle processor will fail because of the presence of the DOSER loop in L. Only the first four consecutive iterations can be assigned at once. In other words, at most four processors can be used in parallel. This problem can be eliminated by applying *loop interchange* [2,24]. It was proved [20] that, in a hybrid perfectly nested loop, any DOALL loop can be interchanged with any DOSER loop in a deeper nest level. Loop interchange can be applied repeatedly and independently for any pair of (DOALL, DOSER) loops. Hence, the first loop and the second loop can be interchanged and then loop coalescing is applied as shown in *Fig. 1(b)*. Now the first 40 successive iterations can be assigned at once.

The last case that remains to be discussed is multiway nested loops. When there do not exist bi-directional dependences across loops at the same nest level, a loop transformation called *loop distribution* [15] can be used to transform a multiway nested loop into several one-way nested loops. For a multiway nested loop with bi-directional dependences across loops at the same nest level, barrier synchronizations can be used and the scheduling of multiway nested loops is identical to the scheduling of perfectly nested loops [20].

3. Interleaved Guided Self-Scheduling

For a hybrid perfectly nested loop, the GSS algorithm tries to permute the indexes through loop interchange so that the largest possible set of parallel iterations corresponds to successive iterations of the coalesced loop. This strategy aims at reducing the number of synchronization points. However, the issue of load-balancing is ignored at all. As a result, it does not always obtain an optimal schedule. For example, consider executing the loop L in *Fig. 1(a)* on 12 processors scheduled under GSS. As mentioned in the previous section, the loop L will be transformed into the loop in *Fig. 1(b)* through loop interchange and loop coalescing. Note that barrier synchronizations are needed for the DOSER loop in *Fig. 1(b)*. *Figure 2(a)* and *Fig. 2(b)* show the barrier synchronizations needed for the loop L and the loop in *Fig. 1(b)*, respectively. Each node represents four parallel iterations of the innermost DOALL loop and each horizontal line represents a barrier. Now 40 parallel iterations can be assigned at once. However, due to barrier synchronizations, the next 40 iterations cannot start execution until the previous 40 iterations all finish execution. Therefore, the completion time will become $20b$ if each iteration takes b units of execution time and the startup time of each processor is the same. However, the optimal completion time is only $17b$ for the case of utilizing 12 processors. Clearly, the schedule generated by GSS is not optimal.

For a hybrid multiway nested loop, it is possible to eliminate this problem by applying *high-level spreading* to independent loops at the same nest level. Spreading is the act of allocating different program fragments to different processors. It is called high-level spreading if spreading is done at the loop or subroutine level. By applying high-level spreading, independent loops at the same nest level can be executed in parallel. This provides another type of parallelism and can prevent processors from waiting for barriers to be opened.

For hybrid perfectly nested loops, high-level spreading can not be applied directly because there is only one loop at each nest level. To generate an optimal schedule for a hybrid perfectly nested loop, we propose a scheduling scheme called *interleaved guided self-scheduling* (IGSS). The basic idea of IGSS is to split a hybrid perfectly nested loop into several independent loops and then apply high-level spreading to the generated loops. Since the resulting loops are independent, loop splitting must be applied to outer DOALL loops only. As in the GSS algorithm, loop interchange and loop coalescing can be applied to the resulting loops for reducing the number of synchronization points.

For example, the loop L in *Fig. 1(a)* can be transformed into the loop in *Fig. 1(c)*. The corresponding barrier synchronizations are shown in *Fig. 2(c)*. High-level spreading is achieved by executing instances of the two DOALL loops in an interleaved order. In other words, the ith instance of the first DOALL loop is to be assigned immediately after all iterations of the $(i-1)$th instance of the second DOALL loop were assigned and the ith instance of the second DOALL loop is to be assigned immediately after all iterations of the ith instance of the first DOALL loop were assigned. Instances are executed on processors scheduled under GSS. Hence, it is called interleaved GSS. Some important properties of IGSS are stated in the following lemmas. Their proofs are directly from GSS and omitted here.

Lemma 1. *For any instance, each of the last $p-1$ iteration blocks assigned for execution contains exactly one iteration.*

Lemma 2. *If iterations have constant execution time b, all processors finish executing an instance within b units of time different from each other.*

In general, several independent loops can be generated by loop splitting. It is called s-way interleaved GSS or IGSS(s) if the number of independent loops generated by loop splitting is s. In the following theorem, we give sufficient conditions that optimal load-balancing can be achieved by IGSS(s). Consider the perfectly nested loop in *Fig. 3(a)*. Suppose that it is transformed into the loop in *Fig. 3(b)*, where $\Sigma_{i=1}^{s} N_1^i = N_1$. Let L_i^j denote the jth instance of the ith DOALL loop and T_i^j denote the time instant that all iterations of L_i^j are completed. We have the following theorem.

Theorem 1. *If iterations have constant execution time b and, for each DOALL loop i, $(N_1 - N_1^i)N_3 \geq p-1$, then no processor will stay idle unless all instances have been assigned.*

Proof. We shall prove this theorem by contradiction. According to IGSS, a processor will stay idle only when it try to execute an instance of some DOALL loop and the previous instance of the same DOALL loop had not been finished. Without loss of generality, we may assume that the first time a processor stayed idle was at the time instant t when processor r tried to execute the $(j+1)$th instance of the ith DOALL loop and the jth instance of the ith DOALL loop had not been finished. Obviously, $t < T_i^j$.

According to the assumption and Lemma 2, we know that no processor will stay idle before the time instant $T_i^j - b$. In other words, we have $T_i^j - b \leq t$. Furthermore, iterations in instances L_{i+1}^j through L_{i-1}^{j+1} is to be assigned after the time instant $T_i^j - b$. By definition, the number of iterations in instances L_{i+1}^j through L_{i-1}^{j+1} is $(N_1 - N_1^i)N_3 \geq p-1$. According to Lemma 1, at least $p-1$ iteration blocks were assigned after the time instant $T_i^j - b$. Since iterations have constant execution time b, at time instant t processor r will receive an iteration block for execution. However, this contradicts the assumption. Therefore, this theorem must be true. □

Note that the total number of synchronization points used by IGSS(s) is at most s times the total number of synchronization points used by GSS. Fewer generated loops are preferred for reducing the number of synchronization points. The optimal number of independent loops

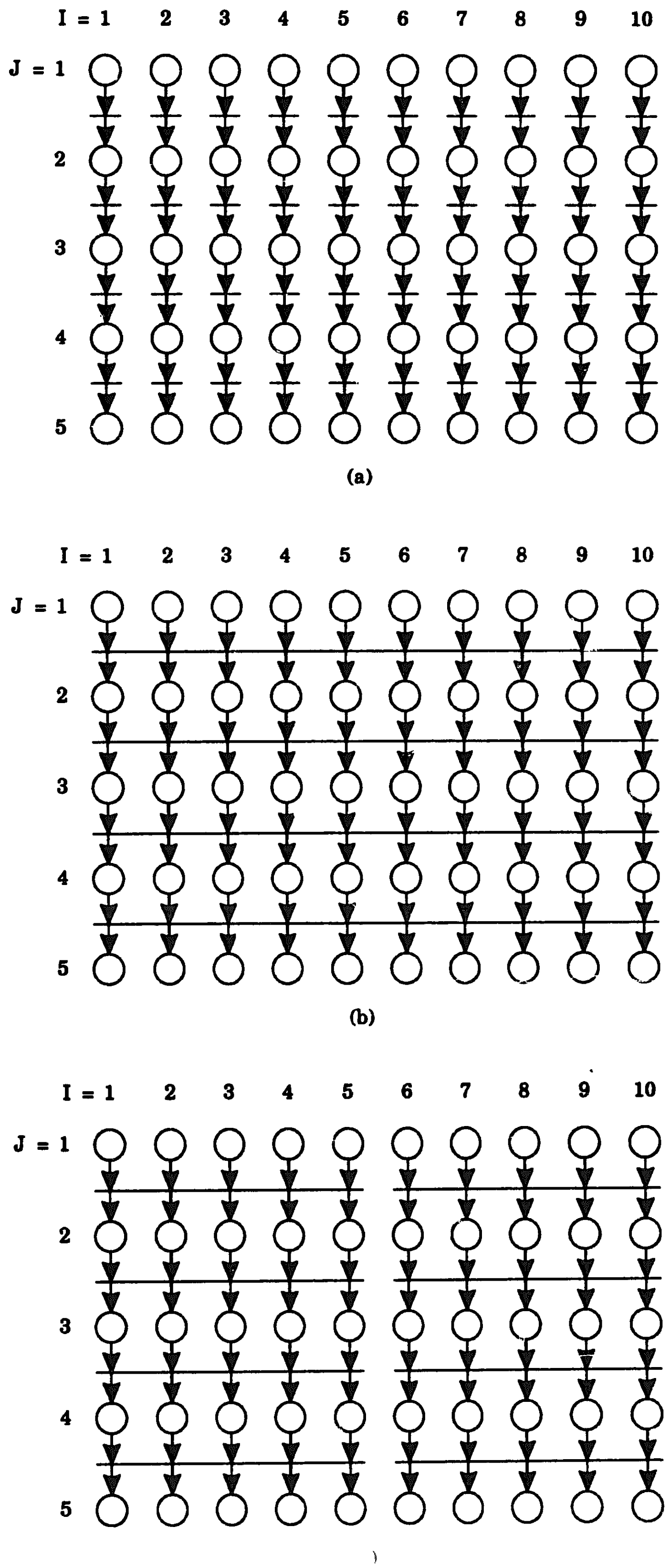

Fig. 2. Barrier synchronizations for loops in *Fig. 1*.

```
DOALL $I_1$ = 1, $N_1$
  DOSER $I_2$ = 1, $N_2$
    DOALL $I_3$ = 1, $N_3$
      {$B$}
    ENDDO
  ENDDO
ENDDO
```

(a)

```
PARBEGIN
  DOSER $I_2$ = 1, $N_2$
    DOALL $I$ = 1, $N_1^1 N_3$
      {$B$}
    ENDDO
  ENDDO
  DOSER $I_2$ = 1, $N_2$
    DOALL $I$ = 1, $N_1^2 N_3$
      {$B$}
    ENDDO
  ENDDO
  ...
  DOSER $I_2$ = 1, $N_2$
    DOALL $I$ = 1, $N_1^s N_3$
      {$B$}
    ENDDO
  ENDDO
PAREND
```

(b)

Fig. 3. An example of IGSS(s).

```
DOALL I = 1, $N_1$
  DOSER J = 1, $N_2$
    DOALL K = 1, $N_3$
      A(I,J,K) = A(I,J-1,K-1) + A(I,J-1,K)
    ENDDO
  ENDDO
ENDDO
```

(a)

```
PARBEGIN
  DOALL I = 1, $N_1-M$
    DOSER L = 1, $N_2N_3$
      A(I,J,K) = A(I,J-1,K-1) + A(I,J-1,K)
    ENDDO
  ENDDO
  DOSER J = 1, $N_2$
    DOALL L = 1, $N'N_3$
      A(I,J,K) = A(I,J-1,K-1) + A(I,J-1,K)
    ENDDO
  ENDDO
  DOSER J = 1, $N_2$
    DOALL L = $N'N_3+1$, $MN_3$
      A(I,J,K) = A(I,J-1,K-1) + A(I,J-1,K)
    ENDDO
  ENDDO
PAREND
```

(b)

Fig. 4. A example of MIGSS(2).

to be generated should be changed depending on the target program and on the target multiprocessor. In our experience, two-way interleaved GSS performs well for most cases.

As a numerical example, consider executing the loop in *Fig. 1(a)* on 12 processors scheduled under IGSS(2). As shown in *Fig. 1(c)*, either DOALL loop contains 20 iterations. Hence, 16 synchronization points are needed for every instances. Therefore, a total of 160 synchronization points is needed. Compared with GSS, the total number of synchronization points is increased from 115 to 160. The completion time is reduced from 20b to 17b.

4. Multilevel Interleaved Guided Self-Scheduling

A disadvantage of IGSS is that the number of synchronization points used by IGSS is not minimum for achieving optimal load-balancing. To make matters worse, both GSS and IGSS ignore communication overhead. For example, consider the hybrid nested loop in *Fig. 4(a)*.

Suppose that $N_1 = 10$, $N_2 = 5$ and $N_3 = 4$. At most 400 global read operations and 200 global write operations are needed. Compared with 160 synchronization operations used by IGSS(2), communication overhead is a more serious problem in this example.

According to the semantic of DOSER loops, there are data communications between different iterations of a DOSER loop. In order to reduce the number of communication operations, a DOSER loop should be executed serially on a single processor. However, loop interchange and loop coalescing are necessary to achieve optimal load-balancing. Since loop interchange and loop coalescing make iterations of a DOSER loop be spread to different processors, many communication operations are needed. In order to achieve optimal load-balancing by using as less communication operations as possible, we propose a scheduling scheme called *multilevel interleaved guided self-scheduling* (MIGSS). The basic idea of MIGSS is to perform loop interchange and loop coalescing only when it is necessary.

For example, consider executing the perfectly nested loop in *Fig. 4(a)* on p processor. The number of communication operations is minimum if each instance of the DOSER loop at nest level 2 is executed serially on a single processor. Only N_1 iterations of the DOALL loop at nest level 1 are scheduled for parallel execution. However, optimal load-balancing cannot be achieved in this way. Our solution is to split the DOALL loop at level 1 into two loops. The first loop is intended to reduce the number of communication operations. For this purpose, each instance of the DOSER loop at level 2 is executed serially on a single processor. The DOALL loop at level 1 is scheduled under GSS. The second loop is intended to achieve optimal load-balancing and is executed by IGSS. *Figure 4(b)* shows the result of MIGSS(2).

Note that the more iterations in the first loop the less communication operations are used. On the other hand, too many iterations in the first loop may make optimal load-balancing impossible to be achieved. An important problem is to determine the minimum number of iterations in the second loop such that optimal load-balancing still can be achieved. Suppose that the minimum number of iterations in the second loop to ensure optimal load-balancing is M. If iterations have constant execution time b, according to Lemma 2, all processors finish executing the first loop within N_2N_3b units of time different from each other. For optimal load-balancing, all processors finish executing these two loops within b units of time different from each other. Therefore, it must satisfy $MN_2N_3b \geq (p-1)N_2N_3b$. Hence, we have $M \geq p-1$. For convenience, we choose $M = p$ in the MIGSS algorithm.

The above scheme can be applied to any perfectly nested (DOALL, DOSER, DOALL) loop. To reduce as many communication operations as possible, this transformation should be performed whenever there is a perfectly nested (DOALL, DOSER, DOALL) loop and the outer DOALL loop contains more than p iterations. In general, any perfectly nested loop can be expressed as the loop in *Fig. 5* because consecutive DOALL or DOSER loops can be coalesced into a single DOALL or DOSER loop. The complete MIGSS algorithm is shown in *Fig. 6*. The process starts from the DOALL loop at level 1. After loop splitting is applied to DOALL loops at level $i-1$, there are two DOALL loops at level $i-1$. The latter DOALL loop is further transformed into two DOSER loops, each containing a DOALL loop at level i as its loop body. The above process can then be applied to DOALL loops at level i. As a result, IGSS is applied to DOALL loops at each level. Hence, it is called multilevel IGSS.

By this scheme, the number of communication operations can be greatly reduced. As an illustration, we conduct an experiment to show the performance improvement of MIGSS over GSS. A simulator is implemented to study the performance of variant scheduling schemes. It accepts a nested loop as input. Communications and synchronizations must be explicitly specified. The hybrid perfectly nested loop in *Fig. 4(a)* with $N_1 = 16$ and $N_2 = N_3 = 10$ is chosen in the experiment. Each iteration is assumed to take 30 cycles of computation time. It is also assumed that no global read operations are needed when the DOSER loop at level 2 is executed serially. Synchronizations operations are always needed for scheduling and barriers.

```
DOALL I1 = 1, N1
  DOSER I2 = 1, N2
    DOALL I3 = 1, N3
      DOSER I4 = 1, N4
        ...
          DOALL I2m+1 = 1, N2m+1
            {B}
          ENDDO
        ...
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

Fig. 5. A general hybrid perfectly nested loop.

Input:

1. L_i : a DOALL loop that has N_i and contains B_i as its loop body.

Output:

1. A parallel program that is semantically equivalent to L_i and suitable for MIGSS.

Algorithm:

```
1. If $B_i$ is not a DOSER loop then return $L_i$.
2. Let $L_j$ be the DOSER loop in $B_i$. If $B_j$ is not a DOALL loop then return $L_i$.
3. Let $L_k$ be the DOSER loop in $B_j$.
4. Loop Splitting: Generate DOALL loops $L_i^1$ and $L_i^2$.
    (a) $B_i^1 \leftarrow$ serialized code of $B_i$ and $N_i^1 \leftarrow N_i - p$.
    (b) $B_i^2 \leftarrow B_i$ and $N_i^2 \leftarrow p$.
5. IGSS: Generate DOSER loops $L_j^1$ and $L_j^2$ and DOALL loops $L_k^1$ and $L_k^2$.
    (a) $B_k^1 \leftarrow B_k$ and $N_k^1 \leftarrow \lceil N_i^2/2 \rceil N_k$.
    (b) $B_k^2 \leftarrow B_k$ and $N_k^2 \leftarrow \lfloor N_i^2/2 \rfloor N_k$.
    (c) $B_j^1 \leftarrow L_k^1$ and $N_j^1 \leftarrow N_j$.
    (d) $B_j^2 \leftarrow L_k^2$ and $N_j^2 \leftarrow N_j$.
6. Recursion: $B_j^1 \leftarrow$ MIGSS($L_k^1$) and $B_j^2 \leftarrow$ MIGSS($L_k^2$).
7. return $(L_i^1, L_j^1, L_j^2)$.
```

Fig. 6. The MIGSS algorithm.

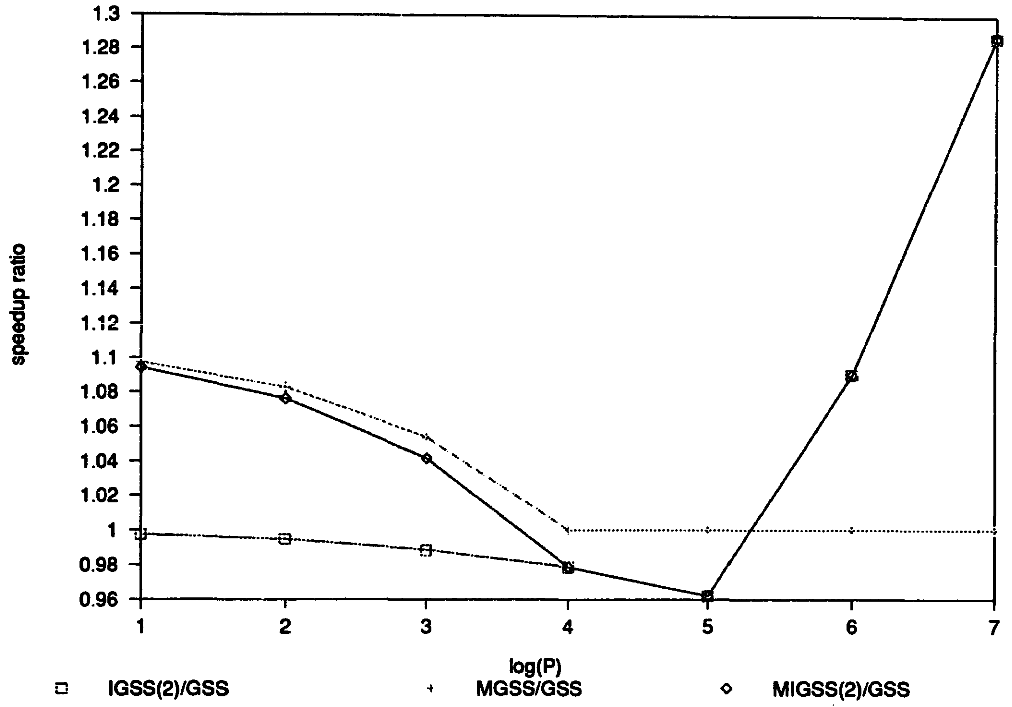

Fig. 7. Speedup ratios for $t_{ov} = 2$.

The overheads of communication and synchronization operations have a severe impact on the speedup obtained. In general, it is not a constant and dependent on many factors. These factors in turns depend on the actual multiprocessor system. Since we are concerned with the relative performance improvement of MIGSS over GSS, in our experiment, the time to perform a communication or synchronization operation is assumed to be a constant. Let t_{ov} be the overhead constant. To show the influence of different overheads, we conduct the experiment for $t_{ov} = 2$ and $t_{ov} = 10$, respectively.

Four scheduling schemes are simulated. They are GSS, IGSS(2), MGSS, and MIGSS(2). *Figures 7* and *8* show the speedup ratios of the other three schemes over GSS. First, observe that the speedup ratio of MGSS over GSS is equal to 1 when $P \geq 16$. Recall that the multilevel scheduling technique is effective only when $N_1 > P$ and $N_1 = 16$ in the experiment. Therefore, for $P \geq 16$, MGSS is identical to GSS. For the same reason, MIGSS(2) is identical to IGSS(2) when $P \geq 16$.

Next, observe that, when the number of available processors is small, the speedup ratio of IGSS(2) over GSS is less than 1 while the performance improvement of MGSS and MIGSS(2) over GSS is significant. This can be explained by the fact that the interleaved scheduling technique aims at load-balancing while the multilevel scheduling technique aims at reducing the communication overhead. When the number of available processors is small, reducing the communication overhead is more important than load-balancing. Other evidence comes from the observation that the performance improvement of MGSS and MIGSS(2) over GSS becomes more significant as the overhead constant increases.

On the contrary, when the number of available processors grows, load-balancing becomes more and more important. It can be observed that, as the number of available processors approaches the total number of parallel iterations (160 in the experiment), the performance improvement of IGSS(2) and MIGSS(2) over GSS becomes more significant. It can also be

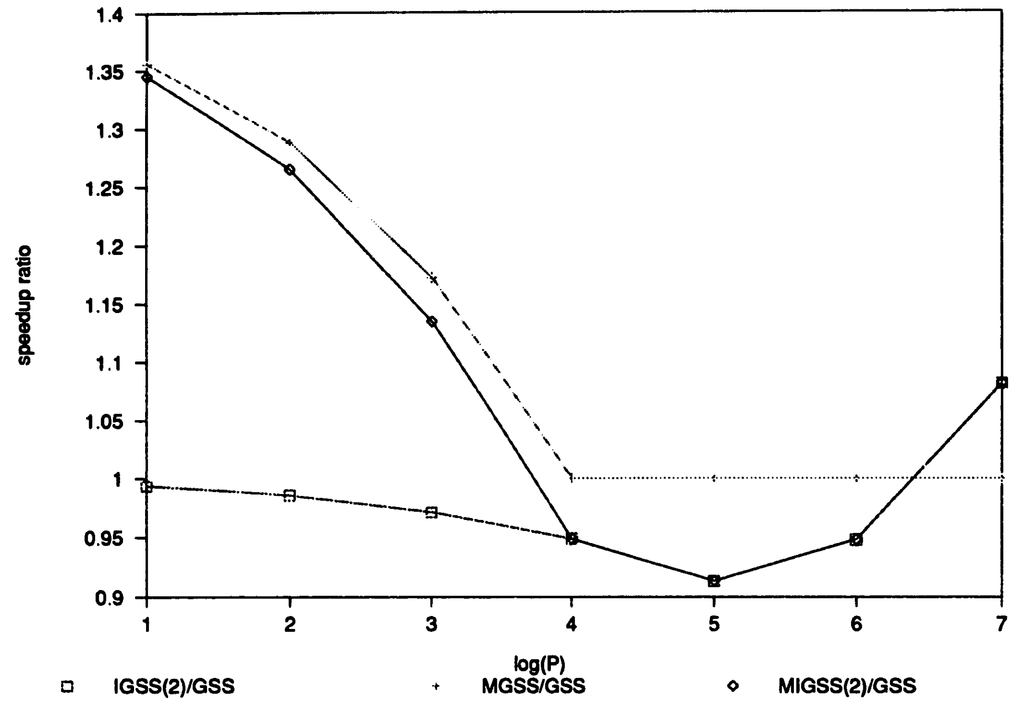

Fig. 8. Speedup ratios for $t_{ov} = 10$.

observed that the speedup ratios of IGSS(2) and MIGSS(2) over GSS decrease as the overhead constant increases.

The common characteristics of *Figs. 7* and *8* is that the speedup ratio of MIGSS(2) over GSS is approximately equal to the speedup ratio of MGSS over GSS for a small number of processors and to the speedup ratio of IGSS(2) over GSS for a large number of processors. In other words, the performance improvement of MIGSS(2) over GSS for a small number of processors is primarily due to the proposed multilevel scheduling technique and the performance improvement of MIGSS(2) over GSS for a large number of processors is primarily due to the proposed interleaved scheduling technique. Therefore, MIGSS performs well in either case.

5. Conclusions

The optimal load-balancing and the minimization of overheads are two most important objectives in scheduling parallel processors. In this paper, two self-scheduling schemes were reviewed. Processor self-scheduling can achieve optimal load-balancing but fails to minimize overheads. Guided self-scheduling performs well for a simple parallel loop but does not perform well for a hybrid nested loop.

To achieve both objectives for hybrid nested loops, we propose a new scheduling scheme. It is a hybrid scheme of the run-time scheduling and the compile-time loop transformations. Its run-time scheduling is based on the guided self-scheduling algorithm. The difference to the guided self-scheduling is in that it performs at compile-time appropriate loop transformations to achieve both objectives. Optimal load-balancing can be achieved through loop splitting and high-level spreading. This technique can prevent processors from waiting for barriers to be opened. The minimization of communication overheads can be achieved through multilevel spreading. Tasks with coarser granularity are preferred unless optimal load-balancing cannot be achieved. We have derived a sufficient condition to achieve optimal load-balancing. An experiment has been conducted and the results clearly show the performance improvement of the proposed scheme over guided self-scheduling.

References

[1] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers: Principles, Techniques, and Tool* (Addison-Wesley, Reading, MA, 1986).

[2] J.R. Allen and K. Kennedy, Automatic loop interchange, *ACM SIGPLAN Notices* 19 (6) (Jun. 1984) 233–246.

[3] Alliant Computer System Corp., FX/Series Architecture Manual, Acton, MA, 1985.

[4] U. Banerjee, Speedup of ordinary programs, Ph.D. dissertation, Univ. of Illinois at Urbana-Champaign, DCS Rep. UIUCDCS-R-79-989, Oct. 1979.

[5] E.G. Coffman, Jr., Ed., *Computer and Job-Shop Scheduling Theory* (Wiley, New York, 1976).

[6] R.G. Cytron, Doacross: Beyond vectorization for multiprocessors, *Proc. 1986 Internat. Conf. on Parallel Processing* (Aug. 1986) 836–844.

[7] R.G. Cytron, Limited processor scheduling of Doacross loops, *Proc. 1987 Internat. Conf. on Parallel Processing* (1987) 226–234.

[8] Z. Fang, P.C. Yew, P. Tang and C.Q. Zhu, Dynamic processor self-scheduling for general parallel nested loops, *Proc. 1987 Internat. Conf. on Parallel Processing* (1987) 1–10.

[9] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theorey of NP-Completeness* (Freeman, San Francisco, CA, 1979).

[10] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph and M. Snir, The NYU Ultracomputer-Designing and MIMD shared-memory parallel machine, *IEEE Trans. Comput.* C-32 (Feb. 1983).

[11] R.L. Graham, Bounds on multiprocessing timing anomalies, *SIAM J. Appl. Math.* 17 (2) (Mar. 1969).

[12] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms* (Computer Science Press, Rockville, MD, 1978).

[13] C. Kruskal and A. Weiss, Allocating independent subtasks on parallel processors, *IEEE Trans. Software Engrg.* SE-11 (Oct. 1985).

[14] D.J. Kuck et al., The effects of program restructuring, algorithm change and architecture choice on program performance, *Proc. 1984 Internat. Conf. on Parallel Processing* (Aug. 1984).

[15] D.A. Padua, Multiprocessors: Discussion of some theoretical and practical problems, Ph.D. dissertation, Univ. of Illinois at Urbana-Champaign, DCS Rep. UIUCDCS-R-79-990, Nov. 1979.

[16] C.D. Polychronopoulos, D.J. Kuck and D.A. Padua, Execution of parallel loops on parallel processor systems, *Proc. 1986 Internat. Conf. on Parallel Processing* (1986) 519–535.

[17] C.D. Polychronopoulos, On program restructuring, scheduling, and communication for parallel processor systems, Ph.D. dissertation, CSRD 595, Center of Supercomput. Res. Develop., University of Illinois, Aug. 1986.

[18] C.D. Polychronopoulos and U. Banerjee, Processor allocation for horizontal and vertical parallelism and related speedup bounds, *IEEE Trans. Comput.* C-36 (4) (Apr. 1987) 410–420.

[19] C.D. Polychronopoulos, Loop coalescing: A compiler transformation for parallel machines, *Proc. 1987 Internat. Conf. Parallel Processing*, St. Charles, IL (Aug. 1987).

[20] C.D. Polychronopoulos and D.J. Kuck, Guided self-scheduling, A practical scheduling scheme for parallel supercomputers, *IEEE Trans. Comput.* C-36 (12) (Dec. 1987) 1425–1439.

[21] S. Sahni, Scheduling multipipline and multiprocessor computers, *IEEE Trans. Comput.* C-33 (Jul. 1984).

[22] H.S. Stone, Multiprocessor scheduling with the aid of network flow algorithms, *IEEE Trans. Software Engrg* SE-3 (1) (Jan. 1977) 85–93.

[23] P. Tang and P.C. Yew, Processor self-scheduling for multiple-nested parallel loops, *Proc. 1986 Internat. Conf. on Parallel Processing* (Aug. 1986) 528–535.

[24] M.J. Wolfe, Optimizing supercompilers for supercomputers, Ph.D. dissertation, Center for Supercomput. Res. Develop. Rep. 329, Univ. Illinois, Urbana, 1982.