Feedback Guided Dynamic Scheduling of Nested Loops

D. J. Hancock[1], J. M. Bull[2], R. W. Ford[1], and T. L. Freeman[1]

[1] Centre for Novel Computing, University of Manchester, Manchester, M13 9PL, U.K.
[2] Edinburgh Parallel Computing Centre, University of Edinburgh, Edinburgh, EH9 3JZ, U.K.

Abstract. In previous papers ([2], [3], [6]) feedback guided loop scheduling algorithms have been shown to be very effective for certain loop scheduling problems. In particular they perform well for problems that involve a sequential outer loop and a parallel inner loop, and timing information gathered during one execution of the parallel inner loop can be used to inform the scheduling of the subsequent execution of this loop. In this paper we consider the extension of these feedback-guided scheduling algorithms to the more important case of nested parallel loops, again within a sequential outer loop. We describe three alternative ways of scheduling nested loops; two are based on reducing the nested loops to a single loop and applying one-dimensional techniques; the third addresses the multidimensionality of the nested loops directly.

Keywords: Parallelism, Loop Scheduling, Load Balancing, Nested Loops.

1 Introduction

Loops, and particularly nested loops, are a very rich source of parallelism for many applications. Consequently a variety of algorithms that aim to schedule loop iterations to processors of a shared-memory machine in an almost optimal way (so-called loop scheduling algorithms) have been suggested. In a number of recent papers, Bull, Ford and co-workers ([2], [3], [6]) have shown that a Feedback Guided Dynamic Loop Scheduling (FGDLS) algorithm performs well for certain one-dimensional loops. In this paper we introduce a new multi-dimensional FGDLS algorithm which extends these ideas to the more general case of nested loops. The relative performance of the algorithm is compared with the original FGDLS algorithm using a number of synthetic benchmarks.

2 Loop Scheduling

In recent papers, Bull [2] and Bull et al. [3] (see also Ford et al. [6]) have proposed a loop scheduling algorithm, termed Feedback Guided Dynamic Loop Scheduling (FGDLS). The major difference between FGDLS and other scheduling algorithms, such as guided self-scheduling (see [9]) and affinity scheduling (see [7]), results from the assumption that the workload is changing only slowly from one execution of a loop to the next, so that observed timing information from the current execution of the loop can, and *should*, be used to guide the scheduling of the next execution of the same loop. In this way it is possible to limit the number of chunks into which the loop iterations are

divided to be equal to the number of processors. By careful design of the algorithm, it is possible thereby to limit the loss of performance (caused by overheads such as additional synchronisation, loss of data locality, and reductions in the efficiency of loop unrolling and pipelining) associated with guided self-scheduling algorithms and the synchronisation costs of affinity scheduling algorithms.

Note that almost all scheduling algorithms are designed to deal with the case of a single parallel loop. The application of the algorithms to nested parallel loops proceeds by either (a) treating only the outermost loop as parallel, or (b) coalescing the loops into a single parallel loop (see Section 4.2). The results of Section 5 show that there are benefits to be gained by treating nested parallel loops directly, rather than through transformation to a single loop.

3 Feedback Guided Dynamic Loop Scheduling

The feedback-guided algorithms described in [2] and [3] are designed to deal with the case of the scheduling, across p processors, of a single loop:

```
DO SEQUENTIAL J = 1, NSTEPS
   DO PARALLEL K = 1, NPOINTS
      CALL LOOP_BODY(K)
   END DO
END DO
```

Assume that the scheduling algorithm has defined the following lower and upper loop iteration bounds (for the p processors) on outer iteration step t:

$$l_j^t, h_j^t \in \mathbb{N}, j = 1, 2, \ldots, p,$$

where $\mathbb{N}$ are the natural numbers, and $t \in \mathbb{N}$ satisfies $1 \leq t <$ NSTEPS. (Note that $l_1^t = 1$, $h_p^t =$ NPOINTS, and $l_{k+1}^t = h_k^t + 1$.) Further assume that the corresponding measured execution times on iteration t are available and are given by $T_j^t, j = 1, 2, \ldots, p$. Given this execution time data, the objective is to define a new (hopefully improved) loop schedule for iteration $t+1$. A piecewise constant approximation to the actual workload at iteration t is given by

$$\hat{w}_i^t = \frac{T_j^t}{h_j^t - l_j^t + 1}, \;\; l_j^t \leq i \leq h_j^t, \;\; j = 1, 2, \ldots, p.$$

The *feedback-guided block scheduling algorithm* defines new iteration bound limits for iteration $t+1$, $l_j^{t+1}, h_j^{t+1} \in \mathbb{N}, j = 1, 2, \ldots, p$, so that this piecewise constant function is approximately equipartitioned amongst the p processors:

$$\sum_{i=l_j^{t+1}}^{h_j^{t+1}} \hat{w}_i^t \approx \frac{1}{p} \sum_{i=1}^{NPOINTS} \hat{w}_i^t = \frac{1}{p} \sum_{k=1}^{p} T_k^t, j = 1, 2, \ldots, p. \tag{1}$$

Thus the observed workload at iteration t is approximately equally distributed amongst the processors at iteration $t+1$ (see Figure 1).

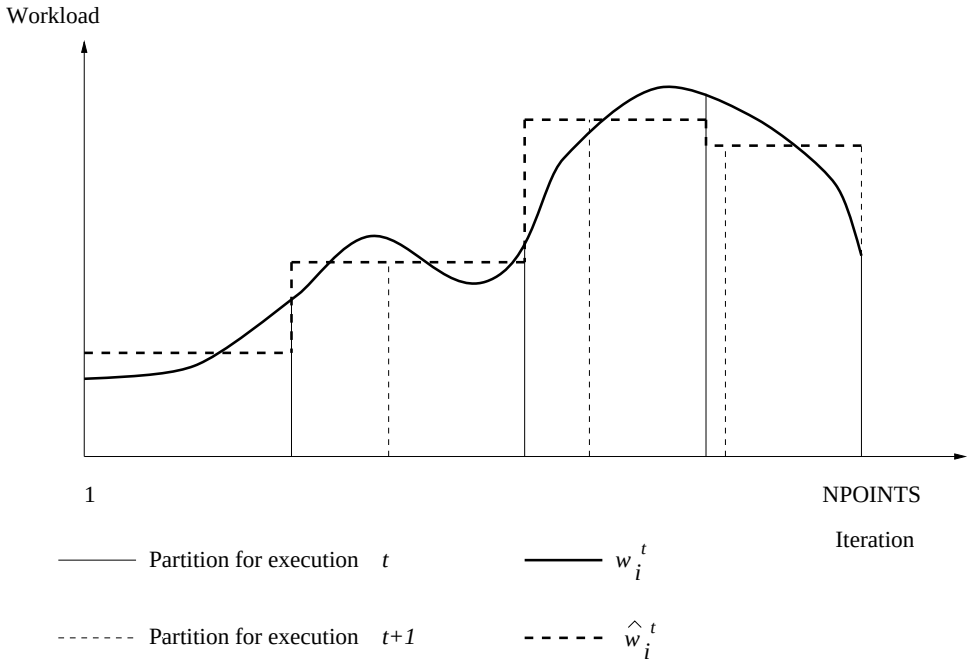

Fig. 1. Feedback-guided Block Scheduling

Bull [2] also describes a second algorithm (*feedback guided affinity scheduling*) that incorporates feedback-guidance in affinity scheduling [7]. Both these feedback-guided algorithms have been shown to be effective for problems where the workload of the parallel loop changes slowly from one execution to the next (see [2], [3]) — this is a situation that occurs in many applications. In situations where the workload is rapidly changing, guided self-scheduling or affinity scheduling algorithms are likely to be more efficient.

In this paper we consider the extension of feedback-guided loop scheduling algorithms to the case of nested loops:

```
DO SEQUENTIAL J = 1, NSTEPS
   DO PARALLEL K1 = 1, NPOINTS1
        .
      DO PARALLEL KM = 1, NPOINTSM
         CALL LOOP_BODY(K1,...,KM)
      END DO
        .
   END DO
END DO
```

4 Algorithms

Of the approaches described in this section, the first approach addresses the multi-dimensional problem directly by explicitly partitioning the M-dimensional iteration space.

The latter two approaches are based on the transformation of the nested loops into an equivalent single loop, followed by the application of a one-dimensional scheduling algorithm to this single loop; the first is based on coalescing the nested loops into a single loop, while the second uses a space-filling curve technique to traverse the M-dimensional iteration space.

4.1 Recursive Partitioning

This algorithm generalises the algorithm described in Section 3 to multiple dimensions. We assume that the scheduling algorithm has defined a partitioning into p disjoint subregions of the M-dimensional iteration space on the outer iteration t:

$$S_j^t = [l_{1j}^t, u_{1j}^t] \times [l_{2j}^t, u_{2j}^t] \times \cdots \times [l_{Mj}^t, u_{Mj}^t], \quad j = 1, 2, \ldots, p,$$

so that $\bigcup_{j=1}^{p} S_j^t$ is the complete iteration space. Based on the corresponding measured execution times, T_j^t, an M-dimensional piecewise constant approximation W_t to the actual workload at iteration t can be formed. The recursive partitioning algorithm then defines new disjoint subregions $S_j^{t+1}, j = 1, 2, \ldots, p,$ for scheduling the next outer iteration $t+1$, so that $\bigcup_{j=1}^{p} S_j^{t+1}$ is the complete iteration space and so that the piecewise constant workload function W_t is approximately equi-partitioned across these subregions. This is achieved by recursively partitioning the dimensions of the iteration space — the dimensions are treated from outermost loop to innermost loop, although it would be easy to develop other, more sophisticated, strategies to take account of loop length for example. To give a specific example, if p is a power of 2, then the first splitting (partitioning) point is such that the piecewise constant workload function W_t is approximately bisected in its first dimension; subsequent splitting points further approximately bisect W_t in the other dimensions.

4.2 Loop Coalescing

This algorithm proceeds by coalescing the M parallel loops into a single loop of length `NPOINTS1*NPOINTS2*`$\cdots$`*NPOINTSM` (see [1] or [9] for details of loop coalescing). For example, for the case $M = 4$ the coalesced loop is given by:

```
DO SEQUENTIAL J = 1, NSTEPS
   DO PARALLEL K = 1, NPOINTS1*NPOINTS2*NPOINTS3*NPOINTS4
      K1 = ((K-1)/(NPOINTS4*NPOINTS3*NPOINTS2)) + 1
      K2 = MOD((K-1)/(NPOINTS4*NPOINTS3),NPOINTS2) + 1
```

```
        K3 = MOD((K-1)/(NPOINTS4),NPOINTS3) + 1
        K4 = MOD(K-1,NPOINTS4) + 1
        CALL LOOP_BODY(K1,K2,K3,K4)
      END DO
    END DO
```

Having coalesced the loops we can use any of the one-dimensional scheduling algorithms described in Section 2 to schedule the resulting single loop. The numerical results of [2], [3], show that the feedback guided scheduling algorithms are very effective for problems where the workload of the (one-dimensional) parallel loop changes slowly from one execution to the next. This is the situation in our case and thus we use the one-dimensional Feedback-guided Block Scheduling algorithm to schedule the coalesced loop.

4.3 Space-filling Traversal

This algorithm is based on the use of space-filling curves for traversal of the iteration space. Such curves visit each point in the space exactly once, and have the property that any point on the curve is spatially adjacent to its neighbouring points. Any one-dimensional scheduling algorithm can be used to partition the resulting one-parameter curve into p regions (again for the results of the next section the Feedback-guided Block Scheduling algorithm is used). This algorithm readily extends to M-dimensions, although our current implementation is restricted to $M = 2$.

A first order Hilbert curve (see [5]) is used to fill the 2-dimensional iteration space, and the mapping of two-dimensional coordinates to curve distances is performed using a series of bitwise operations. One drawback of such a curve is that the space filled must have sides of equal lengths, and the length must be a power of 2.

5 Numerical Results and Conclusions

This section summarises the results of a preliminary set of experiments that simulate the performance of the three algorithms described in Section 4 under synthetic load conditions (full results will be presented in the final paper). Simulation has been used for these preliminary experiments for three reasons;

- for accuracy (reproducible timing results are notoriously hard to achieve on large HPC systems (see Appendix A of Mukhopadhyay [8])),
- so that the instrumenting of the code does not affect performance, and
- to allow large values of p to be tested.

Statistics of both load imbalance and data reuse (measured as the percentage of iteration indices that remain assigned to the same processor) were recorded as each of algorithms attempted to load-balance a number of computations with different load distributions.

In terms of load balance, there is little to choose between the three algorithms, although *slice* (loop coalescing algorithm) tends to outperform the other algorithms for

small numbers of processors, and, for higher numbers of processors, where the total amount of work available per processor becomes quite small, *sft* (space-filling traversal algorithm) tends to perform best.

We would expect the relatively course-grain structure of *rp* to be advantageous when we consider the data reuse achieved by the algorithms and this is borne out by our experiments. In terms of intra-iteration data reuse, *rp* clearly outperforms both *slice* and *sft* for all load types and for nearly all numbers of processors. The superiority of *rp* is greatest for larger numbers of processors. The explanation lies in the structure of the algorithms; in particular the fact that *rp* treats the multi-dimensional iteration space directly is advantageous for data reuse.

Overall we observe that the load balance properties of the three algorithms are very similar and it is not possible to identify one of the algorithms as better than the others. In contrast *rp* is substantially better than the other algorithms in terms of intra-iteration data reuse, particularly for larger numbers of processors.

Further experiments to evaluate the performances of the algorithms on a parallel machine are now required.

References

1. D. F. Bacon, S. L. Graham and O. J. Sharp, (1994) *Compiler Transformations for High-Performance Computing*, ACM Computing Surveys, vol. 26, no. 4, pp. 345–420.
2. J. M. Bull, (1998) *Feedback Guided Loop Scheduling: Algorithms and Experiments*, Proceedings of Euro-Par'98, Lecture Notes in Computer Science, Springer-Verlag.
3. J. M. Bull, R. W. Ford and A. Dickinson, (1996) *A Feedback Based Load Balance Algorithm for Physics Routines in NWP*, Proceedings of Seventh ECMWF Workshop on the Use of Parallel Processors in Meteorology, World Scientific.
4. J. M. Bull, R. W. Ford, T. L. Freeman and D. J. Hancock, (1999) *Convergence of Feedback Guided Dynamic Loop Scheduling*, CNC Technical Report, Department of Computer Science, University of Manchester.
5. A. R. Butz, (1971) *Alternative Algorithm for Hilbert's Space-Filling Curve*, IEEE Trans. on Computers, vol. ??, no. ??, pp. 424–426.
6. R. W. Ford, D. F. Snelling and A. Dickinson, (1994) *Controlling Load Balance, Cache Use and Vector Length in the UM*, Proceedings of Sixth ECMWF Workshop on the Use of Parallel Processors in Meteorology, World Scientific.
7. E. P. Markatos and T. J. LeBlanc, (1994) *Using Processor Affinity in in Loop Scheduling on Shared Memory Multiprocessors*, IEEE Transactions on Parallel and Distributed Systems, vol. 5, no. 4, pp. 379–400.
8. N. Mukhopadhyay, (1999) *On the Effectiveness of Feedback-guided Parallelisation*, PhD Thesis, Department of Computer Science, University of Manchester.
9. C. D. Polychronopoulos and D. J. Kuck, (1987) *Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers*, IEEE Transactions on Computers, C-36(12), pp. 1425–1439.