Feedback Guided Dynamic Loop Scheduling: Algorithms and Experiments

J. Mark Bull

Centre for Novel Computing, Dept. of Computer Science,
University of Manchester, M13 9PL, U.K.
markb@cs.man.ac.uk

Abstract. Dynamic loop scheduling algorithms can suffer from overheads due to synchronisation, loss of locality and small iteration counts. We observe that timing information from previous executions of the loop can be utilised to reduce these overheads. We introduce two new algorithms for dynamic loop scheduling which implement this type of feedback guidance, and report experimental results on a distributed shared memory architecture. Under appropriate circumstances, these algorithms are observed to give significant performance gains over existing loop scheduling techniques.

1 Introduction

Minimising load imbalance is a key activity in producing efficient implementations of applications on parallel architectures. Since loops are the most significant source of parallelism in many applications, the scheduling of loop iterations to processors can be an important factor in determining performance. Most of the existing techniques for dynamic loop scheduling on shared memory machines are variants of, or are derived from, *guided self-scheduling* (GSS) [6]. These techniques share some common characteristics—they assume no prior knowledge of the workload associated with the iterations of the loop, and they all divide the loop iterations into a set of *chunks*, where there are more chunks than processors. The key observations that motivate the new algorithms presented here are the following:

- Some prior knowledge of the workload associated with the iterations of the loop may be available, particularly if we can assume that the workload has not changed too much since the previous execution of the loop. This is often the case in simulation of physical systems where the parallel loop is over a spatial domain and is executed at every time-step.
- Dividing the loop iterations into more chunks than processors can hurt performance, as it may incur overheads associated with additional synchronisation, loss of both inter- and intra-processor locality, and reduced efficiency of loop unrolling or pipelining.

Our algorithms are designed to utilise knowledge about the workload derived from the measured execution time of previous occurrences of the loop with the aim of reducing the incurred overheads.

2 Feedback-Guided Scheduling Algorithms

2.1 Self-scheduling Algorithms

The simplest scheduling algorithm of all is *block* or *static* scheduling, which divides the number of the loop iterations by the number of processors as equally as possible. No attempt is made to dynamically balance the load, but overheads are kept to a minimum: no synchronisation is required and the chunk size is as large as possible. Of dynamic algorithms, the simplest is *self-scheduling* [8] in which a central queue of iterations is maintained. As soon as a processor finishes an iteration, it removes the next available iteration from the queue and executes it. *Chunk self-scheduling* [3] allows each processor to remove a fixed number k of iterations from the queue instead of one. This reduces the overheads at the risk of poorer load balance.

Guided self-scheduling (GSS) [6] attempts to overcome the difficulty of choosing a chunk size by dynamically varying the chunk size as the execution of the loop progresses, starting with large chunks and making them progressively smaller. There is also the possibility that the load may not be well balanced—for example if the loop has N iterations and the first N/p iterations contain more than $1/p$th of the workload. A number of algorithms have been proposed which are essentially similar to GSS, and differ mainly in the manner in which the chunk size is decreased as the algorithm progresses. These include *adaptive guided self-scheduling* [1], *factoring* [2], *tapering* [4] and *trapezoid self-scheduling* [9].

Affinity scheduling [5] uses per-processor work queues in order to reduce contention between processors. Each processor initially has on its queue the iterations it would have been assigned under block scheduling. Another benefit of affinity scheduling is that it allows temporal locality of data across multiple executions of the parallel loop to be exploited. Variants of affinity scheduling to reduce synchronisation costs have been proposed in [7] and [10].

2.2 New Algorithms

In order to exploit timing information, we require access to a fast, accurate hardware clock. Since our objective is to keep the chunk size a large as possible, we only permit measurement of the execution time of whole chunks, and not of iterations within a chunk.

The first algorithm we describe is a feedback guided version of block scheduling. We divide the loop iterations into p chunks, not necessarily of equal size, where p is the number of processors. On each execution of the parallel loop we measure the total execution time for the chunk of iterations assigned to each processor. Dividing this time by the number of iterations on each processor gives us a mean load per iteration for that chunk, and hence a piecewise constant approximation to the true workload. We then choose new bounds for each processor to use for the next execution of the loop, based on an equipartition of the area under this piecewise constant function. By keeping a running total of the

areas under each constant piece, this can be achieved in $O(p)$ time. This could be reduced to $O(\log_2(p))$, using parallel prefix operations, but this would only be worthwhile for large values of p.

The second algorithm is a feedback guided version of affinity scheduling. On each execution of the parallel loop we measure the total execution time for the chunk of iterations initially assigned to *and subsequently executed by* each processor. We derive a mean load per iteration, and hence a piecewise constant approximation to the true workload in the same way as for the feedback guided block algorithm. Equipartitioning this approximation to the workload gives us the initial assignment of iterations for the next execution of the loop. This is similar in spirit to the *dynamic affinity* algorithm of [7], which initialises each processor with the same number of iterations as it executed on the previous execution of the loop.

3 Experimental Evaluation

3.1 Test Problems

To test our algorithms we use a workload consisting of a Gaussian distribution whose midpoint oscillates sinusoidally, defined at the ith iteration and the tth loop execution by

$$w_i^t = \exp\left\{-\left(\frac{i-(N/2+N\sin(2\pi t/\tau)/4}{N/8}\right)^2\right\}$$

where N is the number of iterations, and τ is the period of the oscillation. By varying τ we can control how rapidly the load distribution changes from one execution of the loop to the next—a large value gives a nearly static load, a small value gives a rapidly varying one. We execute the parallel loop 1000 times with values of the period τ varying between 10 and 1000.

We use three different loop bodies. Loop Body 1 simply causes the processor to spin for a time Dw_i^t where D is a fixed (wall clock) time period:

```
for (i=0; i<N; i++){
   interval = D * load(i);
   spin(interval);
}
```

Loop Body 2 increments the ith element of an array a number of times proportional to w_i^t:

```
for (i=0; i<N; i++){
   khi = (N/2) * load(i);
   for (k=0; k<khi; k++) a[i]++;
}
```

For small chunk sizes, more than one processor at a time may be updating the same cache line of a, resulting in a significant number of coherency misses in a cache coherent system. In Loop Body 3 each iteration updates elements in a column of a two-dimensional array using the corresponding element of a second array and its four nearest neighbours, where the number of elements updated is proportional to w_i^t.

```
for (i=1; i<=N; i++){
   khi = N * load(i);
   for (k=1; k=<khi; k++){
       a[i][k] = 0.125 * (4.0 * b[i][k] + b[i+1][k] + b[i-1][k]
                                        + b[i][k+1] + b[i][k-1] );
   }
}
```

On alternate executions of the loop the roles of the two arrays are reversed: the elements of a are used to update b. This loop body is designed to test the algorithms' ability to exploit data affinity between subsequent executions. We can now define the four test problems used:

Problem 1 Loop body 1, $N = 1000$, $D = 10^{-4}$ seconds.
Problem 2 Loop body 1, $N = 1000$, $D = 10^{-3}$ seconds.
Problem 3 Loop body 2, $N = 10000$.
Problem 4 Loop body 3, $N = 1000$.

3.2 Results and Discussion

We ran each of the four test problems on 16 processors of a Silicon Graphics Origin 2000 system (with 195 MHz R10000 processors), using the following scheduling algorithms: block, feedback guided (FG) block, affinity, feedback guided affinity, simple self-scheduling (SS), guided self-scheduling (GSS) and trapezoid self-scheduling (TSS), recording the execution time T_p. We also ran the block scheduling algorithm on one processor to give a sequential reference time T_s. The computational cost of feedback guidance is low: each call to the hardware clock requires approximately $0.35\mu s$ and the computation of the new partition for 16 processors approximately $45\mu s$. Figures 1 to 4 show the efficiency (computed as T_s/pT_p) as a function of the period τ of the oscillating workload.

For Problem 1 the workload on each iteration is sufficiently small for the overheads of synchronised access to work queues to dominate the efficiency of the self-scheduling algorithms. The result of this is that simple self-scheduling (which requires $O(N)$ accesses to the central queue), and affinity scheduling (which requires $O(p \log(N/p^2))$ accesses to each of the p local queues) perform no better than block scheduling. GSS with $O(p \log N)$ accesses and TSS with $4p$ accesses to the central queue respectively, are more efficient. Adding feedback guidance to affinity scheduling does nothing to reduce the total number of queue accesses, even though there are fewer accesses to queues of other processors, and the additional overhead of timing each chunk of iterations is incurred. FG block

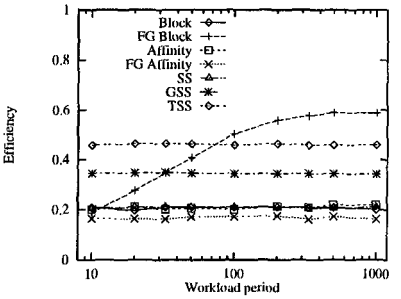

Fig. 1. Problem 1

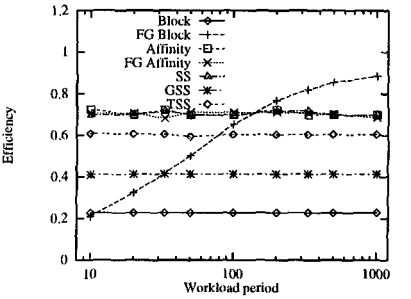

Fig. 2. Problem 2

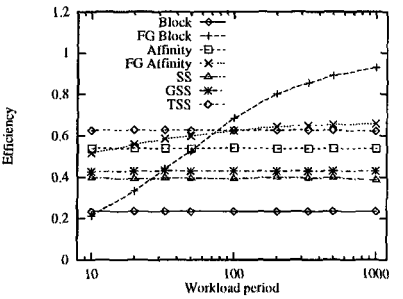

Fig. 3. Problem 3

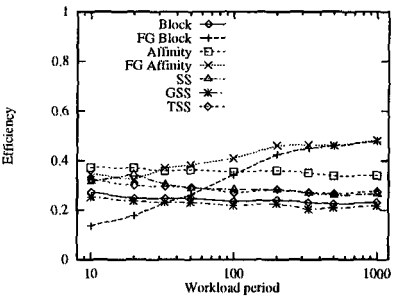

Fig. 4. Problem 4

scheduling avoids all synchronisation costs, and for sufficiently slowly changing workload ($\tau \geq 100$) it is the most efficient of the algorithms tested.

For Problem 2 the workload on each iteration is larger, and synchronisation overhead is less important. SS and affinity scheduling successfully balance the load, whereas GSS and TSS (with larger chunks) do not. Feedback guidance does not improve affinity scheduling, but again for sufficiently slowly changing workload ($\tau \geq 200$), FG block scheduling shows the best performance.

For Problem 3, false-sharing induced communication is the most significant overhead. Both SS and GSS produce many small chunks, with consecutive chunks likely to be executed on different processors. Affinity scheduling also produces many small chunks, but there is a higher likelihood of consecutive chunks being executed on the same processor. TSS produces only $4p$ chunks, and thus it performs well even though it may not successfully balance the load. Apart from the fastest moving workload ($\tau = 10$), FG affinity scheduling is more efficient than affinity scheduling as it further increases the likelihood of consecutive chunks being executed on the same processor. FG block scheduling is again poor for rapidly evolving workload, as it does not balance the load well, but as τ increases, the advantage of having only p chunks becomes significant. For these cases, it is by far the most efficient algorithm.

For Problem 4, both synchronisation and communication overheads are important. None of the central queue algorithms perform well, as they do not ex-

ploit data affinity between executions of the parallel loop. For rapidly changing workload ($\tau \leq 20$), affinity scheduling is the most efficient, but as the workload becomes more predictable, FG affinity scheduling gives the best performance. For small τ, FG block scheduling is very poor, but as τ increases, its performance approaches that of feedback guided affinity scheduling.

4 Conclusions

We have described two algorithms which make use of feedback information to reduce the overheads associated with scheduling a parallel loop in a shared variable programming model. Our experiments show that under certain conditions, when there is sufficient correlation between the workload on successive executions of the parallel loop, these new algorithms will outperform traditional loop self-scheduling methods, sometimes by a considerable margin.

References

1. Eager, D.L. and Zahorjan, J. (1992) *Adaptive Guided Self-Scheduling*, Technical Report 92-01-01, Department of Computer Science and Engineering, University of Washington.
2. Hummel, S.F., Schonberg, E. and Flynn, L.E. (1992) *Factoring: A Practical and Robust Method for Scheduling Parallel Loops*, Communications of the ACM, vol. 35, no. 8, pp. 90–101.
3. Kruskal, C.P. and Weiss, A. (1985) *Allocating Independent Subtasks on Parallel Processors*, IEEE Trans. on Software Engineering, vol. 11, no. 10, pp. 1001–1016.
4. Lucco, S. (1992) *A Dynamic Scheduling Method for Irregular Parallel Programs*, in Proceedings of ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, San Francisco, CA, June 1992, pp. 200–211.
5. Markatos, E.P. and LeBlanc, T.J. (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.
6. Polychronopoulos, C. D. and Kuck, D. J. (1987) *Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers*, IEEE Transactions on Computers, C-36(12), pp. 1425–1439.
7. Subramaniam, S. and D.L. Eager (1994) *Affinity Scheduling of Unbalanced Workloads*, in Proceedings of Supercomputing '94, IEEE Comp. Soc. Press, pp. 214–226.
8. Tang, P. and Tew, P-C. (1986) *Processor Self-Scheduling for Multiple Nested Parallel Loops*, in Proceedings of 1986 Int. Conf. on Parallel Processing, pp. 528–535, St. Charles, IL.
9. Tzen, T.H. and Ni, L.M., (1993) *Trapezoid Self-Scheduling Scheme for Parallel Computers*, IEEE Trans. on Parallel and Distributed Systems, vol. 4, no. 1, pp. 87–98.
10. Yan, Y., C. Jin and X. Zhang (1997) *Adaptively Scheduling Parallel Loops in Distributed Shared-Memory Systems* IEEE Trans. on Par. and Dist. Systems, vol. 8, no. 1, pp. 70–81.