Measuring Synchronisation and Scheduling Overheads in OpenMP

J. M. Bull

EPCC, James Clerk Maxwell Building, The King's Buildings,
The University of Edinburgh, Mayfield Road, Edinburgh EH9 3JZ,
Scotland, U.K.

email: m.bull@epcc.ed.ac.uk

Abstract— **Overheads due to synchronisation and loop scheduling are an important factor in determining the performance of shared memory parallel programs. We present set of benchmarks to measure these classes of overhead for language constructs in OpenMP. Results are presented for three different hardware platforms, each with its own implementation of OpenMP. Significant differences are observed, which suggest possible means of improving performance.**

Keywords— **Benchmarking, OpenMP, synchronisation, scheduling, performance.**

I. Introduction

OpenMP is a relatively new industry standard for shared memory parallel programming. The standard defines a set of directives and library routines for both Fortran [6] and C/C++ [7]. For the first time, shared memory parallel programs can be made portable across a wide range of platforms.

Synchronisation and loop scheduling can both be significant sources of overhead in shared memory parallel programs. In OpenMP, the cost of these operations is dependent on their implementation in the OpenMP run-time library. In this paper we describe techniques for measuring the overheads associated with synchronisation and scheduling directives, and present results for OpenMP Fortran 90 implementations on an SGI Origin 2000, a Sun HPC 3500 and a Compaq Alpha server.

The basic technique is to compare the time taken for a section of code executed sequentially to the time taken for the same code executed in parallel enclosed in a given directive. Particular attention is paid to deriving statistically stable and reproducible results.

These overhead measurements can serve a number of purposes:

1. comparing the efficiency of the run-time libraries of different implementations of OpenMP and highlighting inefficiencies,
2. giving guidance on the performance implications of choosing between semantically equivalent directives (e.g. CRITICAL vs. ATOMIC vs. lock routines), and
3. allowing applications developers to estimate the synchronisation and scheduling overheads in their code by counting the number of directives executed and multiplying by the overhead time for each directive.

Whereas a number of low level benchmarks exist for distributed memory parallel programming models (see, for example, [1], [3]) the historical lack of standardisation in shared memory programming has made meant that little work has been done in this area. Barrier synchronisation is an important feature of this programming model, and previous studies of barrier performance include [2] and [5]. Loop scheduling methods have an extensive literature, with most authors reporting performance studies, though the emphasis is on comparing algorithms rather than implementations (see, for example, [4], [8], [9])

The remainder of this paper is organised as follows: Section II describes the techniques used to perform the overhead measurements, and Section III presents the results of the measurements on the three different systems. These results are analysed in Section IV and conclusions drawn in Section V.

II. Methodology

For a given parallel program, let T_p be the execution time of the program on p processors and T_s the execution time of the sequential version of the same program. We define the *overhead* of a parallel program to be $T_p - T_s/p$, the difference between the parallel execution time and the ideal time given perfect scaling of the sequential program.

To measure the overhead of OpenMP directives, the technique used is to compare the time taken for a section of code executed sequentially, to the time taken for the same code executed in parallel enclosed in a given directive. For example, we measure the overhead of the DO directive by measuring the time taken to execute

```
!$OMP PARALLEL
      do j=1,innerreps
!$OMP DO
         do i=1,omp_get_num_threads()
            call delay(delaylength)
         enddo
!$OMP END DO
      end do
!$OMP END PARALLEL
```

subtracting the reference time, that is the time taken to execute

```
      do j=1,innerreps
         call delay(delaylength)
      end do
```

on a single thread, and dividing by the number of directives executed (`innerreps`). The routine `delay` contains a dummy loop of length `delaylength`; this length is chosen so that the overhead of the directives and the time taken for by this routine are the same order of magnitude. The value of `innerreps` is chosen so that the execution time is significantly larger that the clock resolution, and also large enough that the cost of the enclosing PARALLEL directive can be ignored. Similar comparisons are used to measure the overheads of other directives with implied barriers: PARALLEL (with and without REDUCTION clause), PARALLEL DO, BARRIER and SINGLE.

For mutual exclusion directives, we take a similar approach. For example, to measure the overhead of the CRITICAL directive, we measure the execution time for

```
!$OMP PARALLEL
         do j=1,innerreps/omp_get_num_threads()
!$OMP CRITICAL
            call delay(delaylength)
!$OMP END CRITICAL
         end do
!$OMP END PARALLEL
```

subtract the same reference time as above, and divide by `innerreps`. This method is also applied to `omp_set_lock` and `omp_unset_lock` pairs. For the ATOMIC directive, we replace the subroutine call with an increment to a shared variable. (In general it is preferable to use a subroutine call as the loop body, since we can be certain that the compiler is generating the same code for both the parallel version and the sequential reference version of the loop body).

To measure scheduling overheads, we compare the execution time of

```
!$OMP PARALLEL
      do j=1,innerreps
!$OMP DO SCHEDULE(schedtype,chunksize)
         do i=1,itersperthr*omp_get_num_threads()
            call delay(delaylength)
         end do
      end do
!$OMP END PARALLEL
```

to the execution time for

```
         do i=1,itersperthr
            call delay(delaylength)
         end do
```

on a single thread. There is a large parameter space which could be explored, since the overhead depends not only on the number of threads but also on the number of loop iterations per thread, the time taken to execute the loop body, and on the chunksize. For simplicity, however, we fix both the number of iterations per thread (equal to 1024) and we chose the value of `delaylength` so the the loop body executes for approximately the same number of cycles (equal to 100) on each system. We then make overhead measurements for STATIC, DYNAMIC and GUIDED schedules with various different chunk sizes.

To obtain accurate results, we have to be careful in our choice of clock routines. Second differences of raw clock values are used to compute overheads, and the run time for each measurement cannot be too large, since we need to repeat the measurements many times for statistical stability. We therefore have to ensure that the values returned by the clock routine are not only sufficiently accurate (typically to the nearest microsecond), but also sufficiently precise. In particular, clock routines returning 32-bit floating point clock values in seconds (e.g. `etime`) will result in significant loss of precision. The Fortran 90 `system_clock` routine is only accurate to the nearest 100 microseconds on some systems.

To obtain statistically meaningful results, we repeat each overhead measurement 50 times within each run, and for 20 different runs. The reason for this is that we observe a much larger variability between different runs that between different measurements within the same run. The cause of this is not clear—it may be due to the physical memory locations of synchronisation variables altering from run to to run.

Within each run we compute the mean and standard deviation σ of the 50 measurements. Since we do not have exclusive access to the hardware platforms, we test for "clean" runs (i.e. those which have not been affected by other processes in the system) by checking for large values of σ, and for cases where there are many outliers (values more than 3σ above the mean). Runs which fail this test are rejected.

III. Results

We have run the synchronisation and scheduling benchmarks on the following systems: a Sun HPC 3500 with eight 400Mhz UltraSparc II processors using the KAI guidef90 preprocessor (version 3.7), an SGI Origin 2000 with 40 195MHz R10000 processors, using the MIPSpro f90 compiler (version 7.2.1.3m) and a Compaq Alpha server with four 525 MHz Alpha EV5/6 processors, using the Digital f90 compiler (version 5.2-705). On the Origin 2000 we were only able to obtain access to eight processors at a time.

Figures 1 to 3 show the measured overheads against number of processors for the implied barrier directives on the three systems. Note the change of scale on both axes in Figure 3.

Figures 4 to 6 show the measured overheads against number of processors for the mutual exclusion constructs on the three systems. Again note the changes of scale in Figure 6. Measurements for the lock/unlock pair on the Compaq Alpha server could not be obtained, as the code often deadlocked.

Figures 7 to 9 show the measured overheads against chunk size for the different schedules on the three systems. In each case the results are for four processors. Since we keep the number of loop iterations per processor constant, the results for other numbers of processors are very similar. Measurements for the DYNAMIC schedule on the Compaq Alpha server could not be obtained, as the code

often deadlocked.

In all cases the overheads are reported in processor clock cycles, to best allow comparison between the different systems.

IV. Analysis

A. Barrier type synchronisation

Figure 1 shows that both BARRIER and DO directives have similar performance on the Sun HPC 3500, suggesting that nearly all the cost of the DO directive is in the implied barrier. Both appear to scale well with increasing numbers of processors. The PARALLEL and PARALLEL DO directives take approximately twice as long as the BARRIER directive. Adding a REDUCTION clause to the PARALLEL directive significantly increases its cost and makes it less scalable. The SINGLE directive performance is curious, in that it cost less than the BARRIER, even though the SINGLE directive has an implied BARRIER. Either a different mechanism is being used, or some of the cost of the implied barrier is hidden, as the barrier check-in takes place in parallel with the code contained in the directive.

Figure 2 shows that the BARRIER, DO, PARALLEL and PARALLEL DO behave in much the same way on the SGI Origin 2000 as on the Sun HPC 3500, except that the number of cycles required is around four times fewer. Even given the difference in clock speeds, this is a little surprising, as the Origin 2000 has physically distributed memory, whereas the Sun is a true shared memory architecture. On the Origin 2000, the PARALLEL directive with a REDUCTION clause scales particularly poorly. On eight processors it six times more expensive than a simple PARALLEL. Again the SINGLE directive shows different scaling behaviour to BARRIER, suggesting a different implementation. However, on more than four processors it is less efficient than PARALLEL, so a simple implementation based on testing the thread number and using the standard barrier would improve performance.

On the Compaq Alpha server (Figure 3) we see behaviour and costs very similar to the Origin 2000 for most of the directives. The differences are that the REDUCTION clause is less expensive (though its scaling behaviour is hard to deduce from just four processors) and the SINGLE directive performs just like the BARRIER and DO directives.

On all three systems, it would appear that a barrier is used at the beginning and end of the parallel region. This seems unnecessary, since a barrier normally consist of two phases: the master waits for all the slaves to check in, then signals the slaves to check out. The first phase should suffice for the end of a parallel region, and the second phase for the beginning. Note that on all three systems threads appear to busy-wait between parallel regions.

The treatment of reduction variables could also potentially be optimised. The scaling behaviour is consistent with an implementation using atomic updates to a shared variable, which at best scales linearly with the number of processors p, and may be worse than this due to memory contention effects. A tree-based scheme should scale as $\log p$, and could be incorporated into the barrier. Another potential advantage of this method is that it can be designed to give deterministic results for a given number of processors, a property useful for debugging purposes which is lacking in an atomic update scheme.

B. Mutual exclusion synchronisation

Figure 4 shows that all three mutual exclusion constructs have similar performance on the HPC 3500 and that scaling is very good, with little increase in cost above two processors. The reason why ATOMIC should be cheaper on two and three processors is not clear.

On the Origin 2000 (Figure 5) the scaling behaviour roughly linear in p, though the costs are lower for small numbers of processors than on the HPC 3500. The ATOMIC directive is noticeably cheaper than the other two constructs. Figure 6 shows that on the Alpha server the CRITICAL directive is very expensive (an order of magnitude greater than on the other two systems) whilst the ATOMIC directive has very low overhead.

These results show that on the whole these directives are well implemented, with the obvious exception of the CRITICAL directive on the Alpha server. The poor scaling on the Origin 2000 is probably a feature of the physically distributed memory system.

C. Loop Scheduling

The results in Figure 7 show that on the HPC 3500, that block cyclic scheduling (STATIC,n) costs the same as a block schedule for large chunk sizes, but increases rapidly as the chunk size decreases. For small chunk sizes the overhead is roughly linear in the number of chunks per thread. Dynamic scheduling has a similar pattern of behaviour, but is around five times more expensive than block cyclic. GUIDED has a similar cost to DYNAMIC with a large chunk size, and this only increases slowly with decreasing chunksize. This is to be expected, as the chunk size here is the minimum size of a chunk—most of the iterations are executed in large chunks. Note that results are not shown for large chunk sizes with the GUIDED schedule, as this would result in load imbalance.

On the Origin 2000 (Figure 8) the overhead of the block cyclic schedule does not converge to that of the block schedule as the number of chunks per thread decreases. The cost of the dynamic schedule increases very rapidly with increasing numbers of chunks, so that with a chunk size of one, the cost is an order of magnitude more than on the HPC 3500. The cost of GUIDED also increases noticeably as the chunk size is reduced.

On the Alpha server, the block cyclic schedule is around 30 times more expensive than the block schedule with large chunk sizes, and slightly more expensive that the GUIDED scheme, though for small chunk sizes it is comparable with the other two systems.

The principal observation from these results is that schedules with small chunk sizes incur substantial overheads on all three systems. Much of this probably results from each chunk incurring a function call, since the usual implementation is to create a routine containing the loop body and pass this to the scheduling routine in the

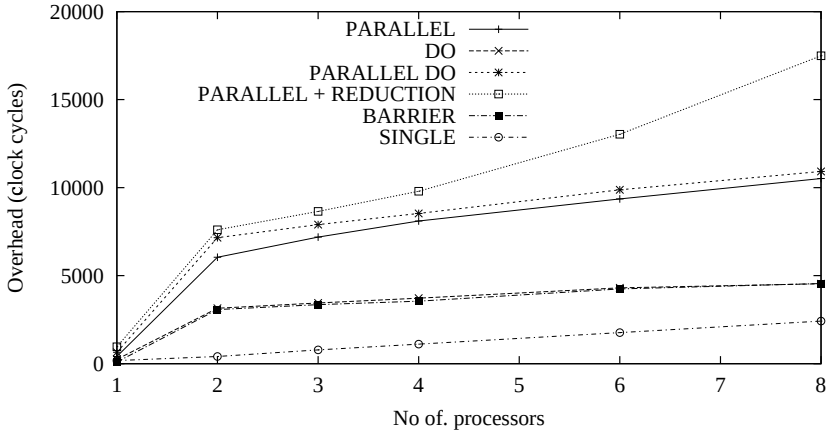

Fig. 1. Synchronisation overheads on Sun HPC 3500

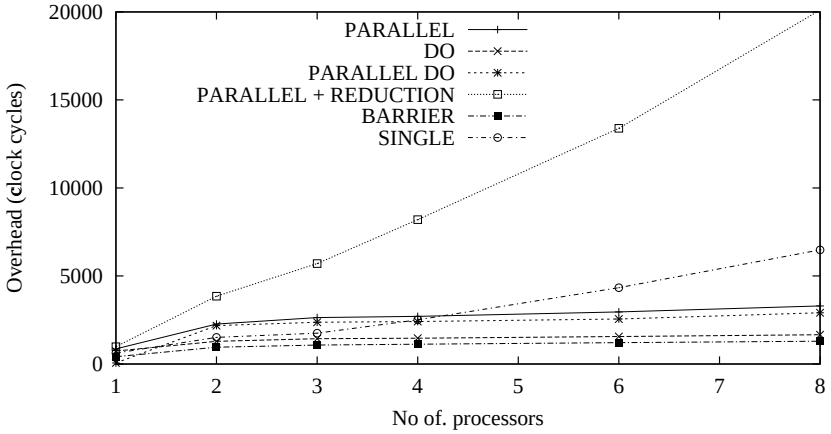

Fig. 2. Synchronisation overheads on SGI Origin 2000

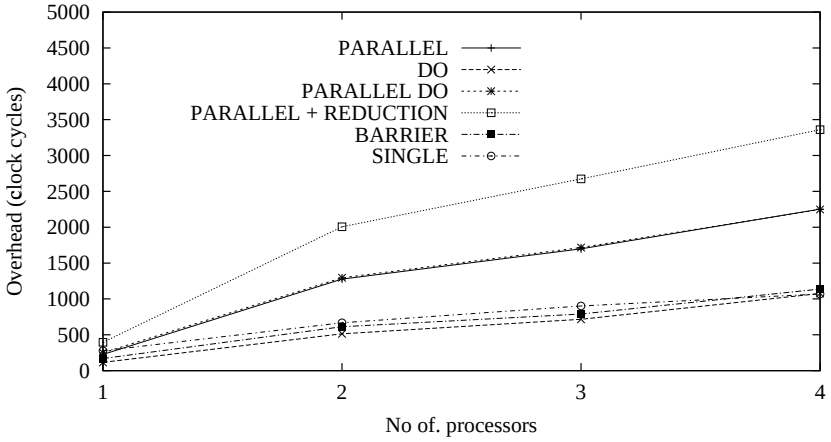

Fig. 3. Synchronisation overheads on Compaq Alpha server

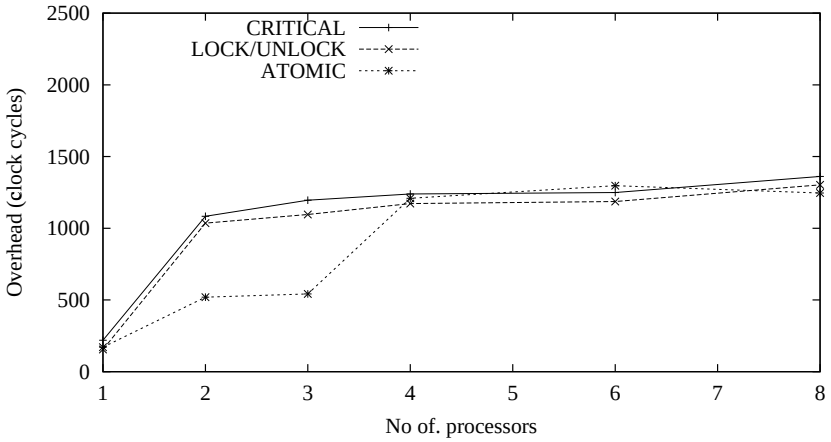

Fig. 4. Mutual exclusion overheads on Sun HPC 3500

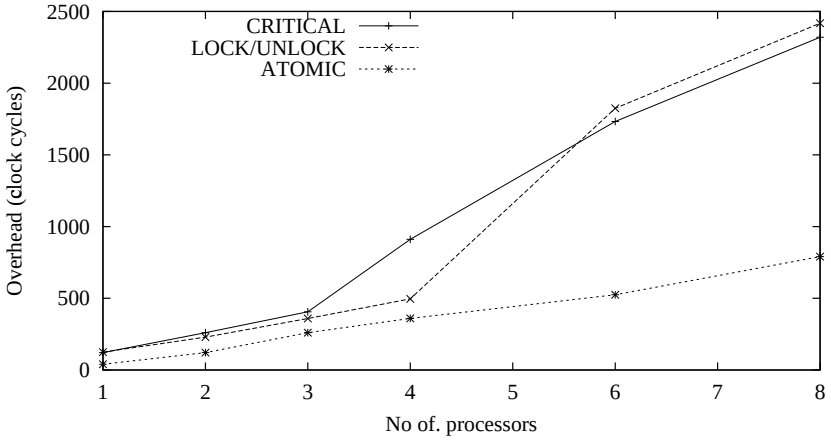

Fig. 5. Mutual exclusion overheads on SGI Origin 2000

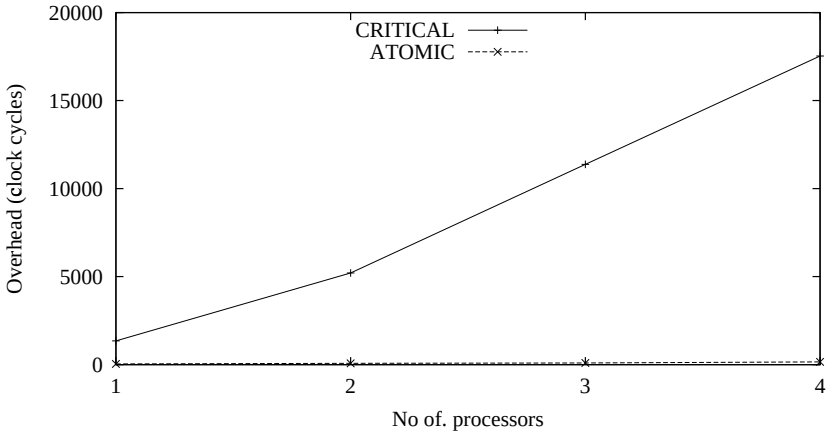

Fig. 6. Mutual exclusion overheads on Compaq Alpha server

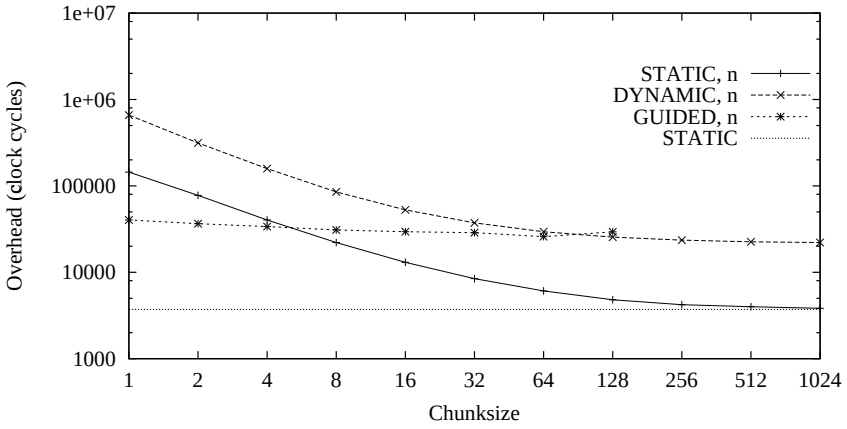

Fig. 7. Scheduling overheads on Sun HPC 3500

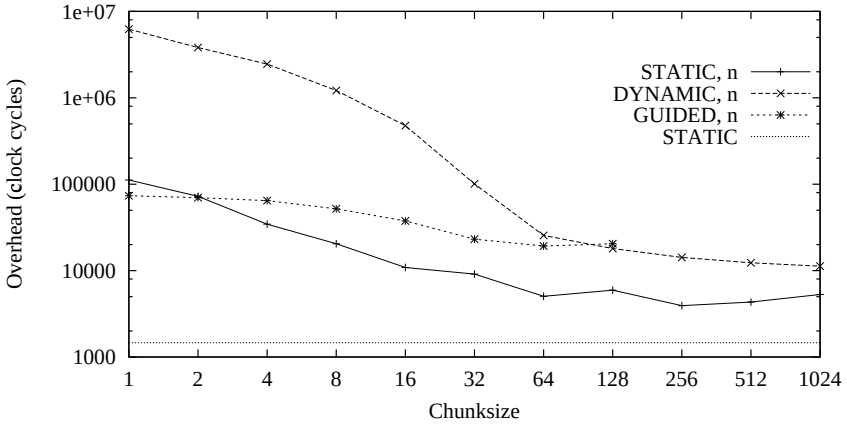

Fig. 8. Scheduling overheads on SGI Origin 2000

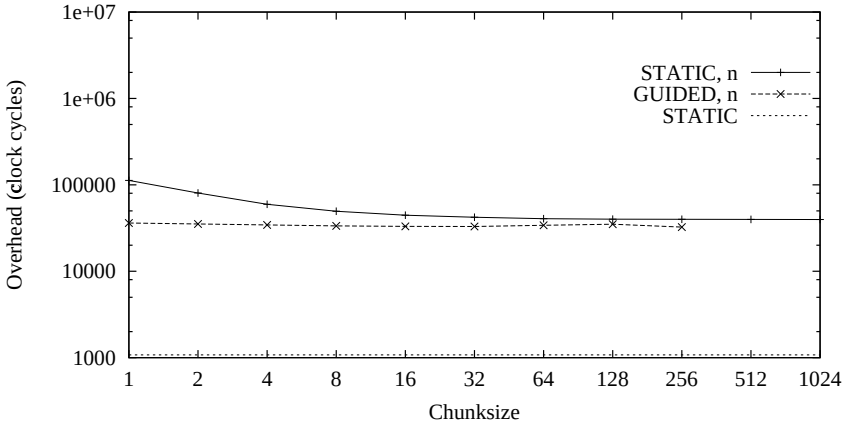

Fig. 9. Scheduling overheads on Compaq Alpha server

OpenMP library. For block cyclic schedules, this could potentially be avoided by having the compiler generate the required loop bounds, rather than relying on the run-time library. In general, there would seem to be scope for useful optimisations here.

V. Conclusions

We have presented a set of benchmarks for measuring the overheads of synchronisation and scheduling in OpenMP. Particular emphasis has been laid on obtaining statistically meaningful measurements. The benchmarks have been run on three common shared memory platforms and the results show some significant and important differences both between directives and between the different platforms. Some potential areas for optimisation have been highlighted.

References

[1] Addison, C.A., Getov, V.S., Hey, A.J.G., Hockney, R.W. and Walton, I.C. (1991) *The GENESIS Distributed-memory Benchmarks*, Computer Benchmarks, J.J. Dongarra and W. Gentzsch (Eds), Advances in Parallel Computing, Vol 8, pp 257–271.

[2] Grunwald, D. and S. Vajracharya (1994) *Efficient Barriers for Distributed Shared Memory Computers* in Proceedings of 8th International Parallel Processing Symposium, April 1994.

[3] Hockney, R.W and Berry, M., (eds) (1991) *Public International Benchmarks for Parallel Computers*, PARKBENCH Committee: Report—1.

[4] 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.

[5] Mellor-Crummey, J. and Scott, M. (1991) *Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors* ACM Transactions on Computer Systems, vol. 9, no. 1, pp. 21–65.

[6] OpenMP Architecture Review Board (1997) *OpenMP Fortran Application Program Interface.*

[7] OpenMP Architecture Review Board (1998) *OpenMP C and C++ Application Program Interface.*

[8] 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.

[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.