A Microbenchmark Suite for OpenMP 2.0

J. Mark Bull and Darragh O'Neill
EPCC, The King's Buildings, The University of Edinburgh,
Mayfield Road, Edinburgh EH9 3JZ, Scotland, U.K.
email: `m.bull@epcc.ed.ac.uk`

Abstract— **In this paper we present a set of extensions to an existing microbenchmark suite for OpenMP. The new benchmarks are targeted at directives introduced in the OpenMP 2.0 standard, as well as at the handling of thread-private data structures. Results are presented for a Sun HPC 6500 system, with an early access release of an OpenMP 2.0 compliant compiler, and for an SGI Origin 3000 system.**

I. Introduction

Synchronisation, loop scheduling and handling of thread-private data can all 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 [2], we described a microbenchmark suite for measuring the overheads associated with synchronisation and scheduling directives in OpenMP 1.0 [4].

In this paper, we present an extended version of this microbenchmark suite. The extended suite includes measurements of OpenMP 2.0 [5] features, as well as additional measurements of OpenMP 1.0 features where the overhead depends on the size of arrays which are specified as arguments to directive clauses. These new benchmarks are described in Section II.

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.

A similar set of microbenchmarks for OpenMP 1.0 was reported in [1], but the code is not publicly available. Other OpenMP benchmark suites include a version of the NAS Parallel Benchmarks [3] and the SPEComp suite [6]: these are kernel and application benchmarks respectively, rather than microbenchmarks.

In Section III, we present results from a Sun HPC 6500 system, where we use the suite to assess an early release version of an OpenMP 2.0 compliant Fortran compiler from Sun Microsystems, and from a SGI Origin 3000 system. In Section IV we draw some conclusions and outline future work.

II. New Benchmarks

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.

The new benchmarks consist of additions to the synchronisation benchmarks and an additional set of benchmarks related to clauses which take array arguments.

The new synchronisation benchmarks measure overheads of the `WORKSHARE` and `PARALLEL WORKSHARE` directives. These are benchmarked in a similar fashion to the `DO` and `PARALLEL DO` directives in the original suite. For example, to compute the overhead of the `WORKSHARE` directive, we measure the time taken to execute

```
!$OMP PARALLEL
      do j=1,innerreps
!$OMP WORKSHARE
         a = a + cos(a) - sin(a)
!$OMP END WORKSHARE
      end do
!$OMP END PARALLEL
```

where `a` is a one dimensional array with number of elements equal to the number of threads, and subtracting the reference time, that is the time taken to execute

```
      do j=1,innerreps
         a = a + cos(a) - sin(a)
      end do
```

where `a` is a one dimensional array with one element. The reason for using intrinsic functions `sin()` and `cos()` here is that the `WORKSHARE` directive cannot contain external subprogram calls without using the ELEMENTAL qualifier, which is only supported in Fortran 95. Since we do not which to restrict the benchmarks to Fortran 95 compliant compilers, we use intrinsic functions instead. Note that it is not possible to guarantee the elimination of the contribution from false sharing of elements of `a` to the measured overhead. Increasing the size of `a` would reduce the contribution from false sharing, but it also reduces the proportion of the execution time spent in the directive itself, and hence the sensitivity of the experiment. However, false sharing overhead is likely to be small in comparison to the contribution from synchronisation.

The array benchmarks measure the overheads of the PARALLEL directive with the PRIVATE, FIRSTPRIVATE, COPYIN, and REDUCTION clauses, and of the SINGLE directive with the COPYPRIVATE clause. These are measured in the same way as the PARALLEL and SINGLE directive overheads are measured in the synchronisation benchmark. For example, the PRIVATE overhead is measured by comparing the execution time of

```
      do j=1,innerreps
!$OMP PARALLEL PRIVATE(a)
         call delay(delaylength,a)
!$OMP END PARALLEL
      enddo
```

to the execution time of

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

In each case the argument of the clause is a single array, and the array size is varied in powers of three, from one to $3^{11} = 177147$.

III. Results

A. Sun HPC 6500

The results presented in this section were obtained from a Sun HPC 6500 system, with 18 400 MHz processors and 18Gb of main memory, running Solaris 2.7. Each measurement is the average of 10 runs, and each run itself repeats the measurement 50 times for the synchronisation benchmarks and 20 times for the array benchmarks.

Figures 1 and 2 show synchronisation overhead using Sun WorkShop f90 compiler, versions 6.1 and 6.2 (Early Access) respectively. Only the latter supports OpenMP 2.0 directives. We observe that the 6.2 version has substantially lower overheads than the 6.1 version (note different scales on vertical axes). This appears to be largely due to a much more efficient barrier synchronisation algorithm. However, the overhead of parallel regions is substantially larger than that for the BARRIER directive, and appear to scale linearly, not logarithmically, with number of processors. This suggests that either a less efficient barrier is being used at the end of the parallel region, or there is some other source of overhead present.

The WORKSHARE and PARALLEL WORKSHARE directives exhibit only slightly higher overhead than the DO and PARALLEL DO directives respectively. Thus is now seems possible to exploit Fortran 90 array syntax with OpenMP at no significant performance penalty.

Figure 3 shows the overhead of the various clauses with array arguments, where the dimension of the array is 729 elements, using the 6.2 version of the compiler. Also shown is the PARALLEL directive overhead for comparison. The most striking aspect of these results is that with the exception of the COPYPRIVATE clause, the overheads are large and scale poorly with number of processors. This is especially surprising in the case of the PRIVATE clause, as the private arrays must simply be allocated, and not assigned. This suggests scope for optimisations, such as parallelising array reductions over array elements.

The worse-than-linear scaling of the PRIVATE and FIRSTPRIVATE clauses suggests contention for some resource. To explore this further, we re-ran the benchmarks, linking with the thread aware malloc library libmtmalloc. This resulted in non-deterministic failures for larger thread numbers, but Figures 4 and 5 show the overheads for 8 threads and varying array sizes, without and with the thread aware malloc library. The results with libmtmalloc show a dramatic reduction in overhead of the PRIVATE and FIRSTPRIVATE clauses for smaller array sizes. This suggests that the private copies of the arrays are being allocated on the heap and not on the stack, resulting in contention for access to the free list. For larger array sizes, it appears that the memory allocation policy reverts to a single heap.

B. SGI Origin 3000

We have also run the benchmark suite on a SGI Origin 3000 with 128 400 MHz MIPS R12000 processors (of which we had access to 32) and 128 Gb of main memory, running IRIX 6.5. The compiler used was MIPSpro 7.3.1.1m f90, which does not support the OpenMP 2.0 extensions, so we were not able to run all of the new benchmarks. Figure 6 shows the synchronisation overheads on this system. The PARALLEL and PARALLEL DO directives have much lower overhead than BARRIER, DO and PARALLEL with REDUCTION. This suggests that, like the Sun WorkShop 6.2 compiler, two different barriers are being used in different directives. However, on the SGI, the faster barrier is used at the end of a parallel region, whereas on the Sun the slower barrier is used at the end of a parallel region.

A question commonly asked by OpenMP programmers is whether to parallelise a sequence of loops with multiple PARALLEL DO directives or with a single PARALLEL directive containing multiple DO directives. These benchmarks illustrate that the answer is entirely system dependent. In principle, there should be little difference, but under current compilers the overheads can be different by up to an order of magnitude in either direction.

Note also the high overhead of the SINGLE directive. The obvious implementation of SINGLE (mutually exclusive access to a flag which is set by the first thread to arrive) can be very expensive, especially in the case where all threads reach the directive at more-or-less the same time, as happens in our benchmark. Unless different thread arrival times are expected, using MASTER with an explicit barrier will give better performance.

Figure 6 shows the synchronisation overheads for mutual exclusion directives on the Origin 3000. Immediately evident are the high costs of CRITICAL and the lock routines, which although non-monotonic with number of processors, are consistent between different runs (recall that the figures presented here are the average of 10 runs, with 50 repeated measurements in each run). This is most likely due to the underlying implementation of mutual exclusion, which would bear re-examination.

Figure 8 shows the overhead of the various clauses with

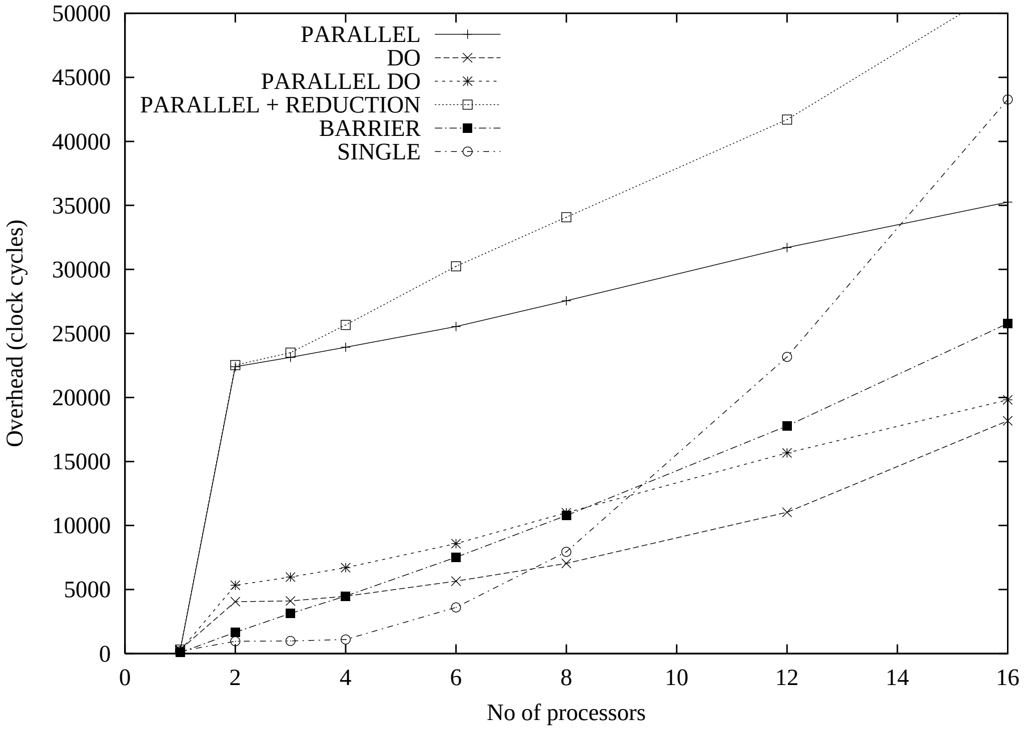

Fig. 1. Overhead of synchronisation directives on a Sun HPC 6500 with WorkShop 6.1 compiler

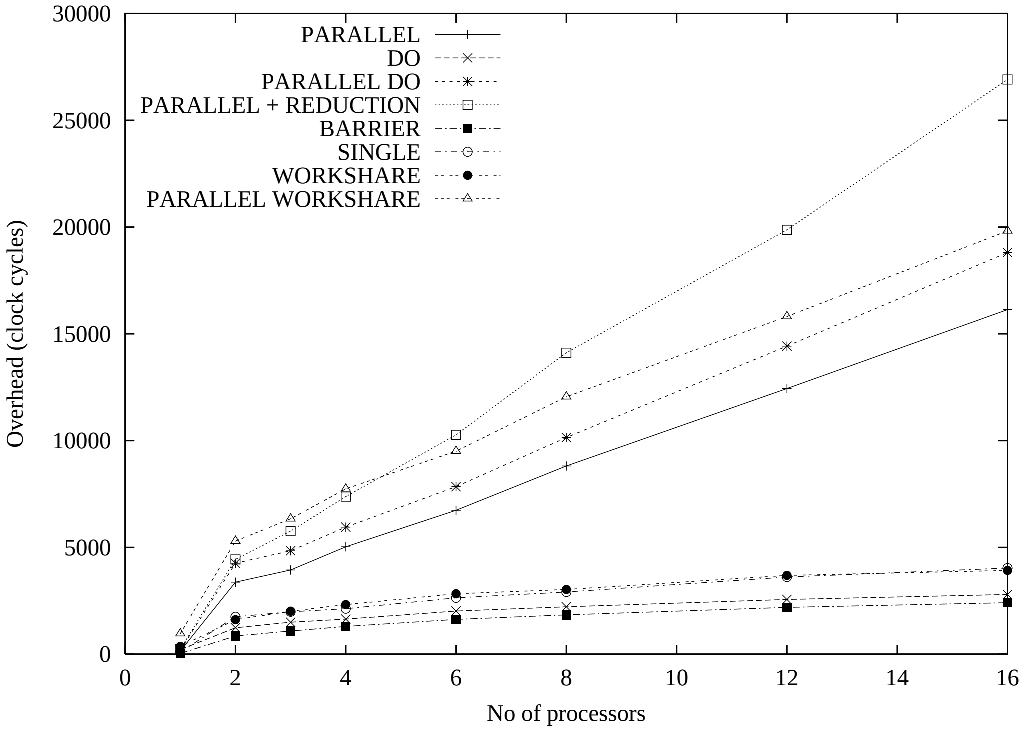

Fig. 2. Overhead of synchronisation directives on a Sun HPC 6500 with WorkShop 6.2EA compiler

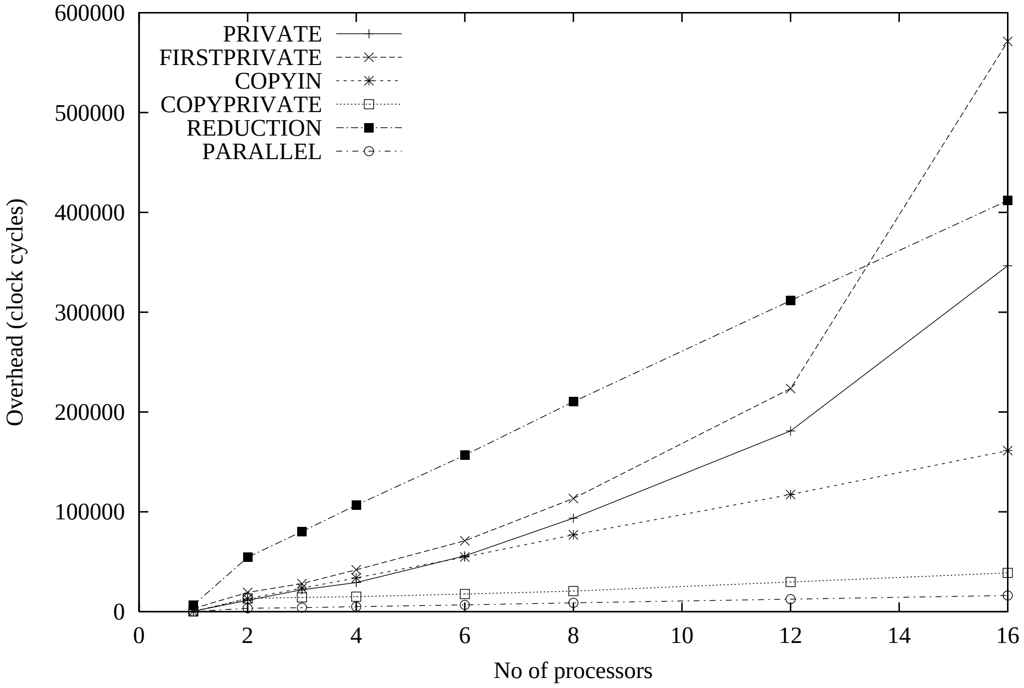

Fig. 3. Overhead of array clauses (array size = 729) on a Sun HPC 6500 with WorkShop 6.2 compiler

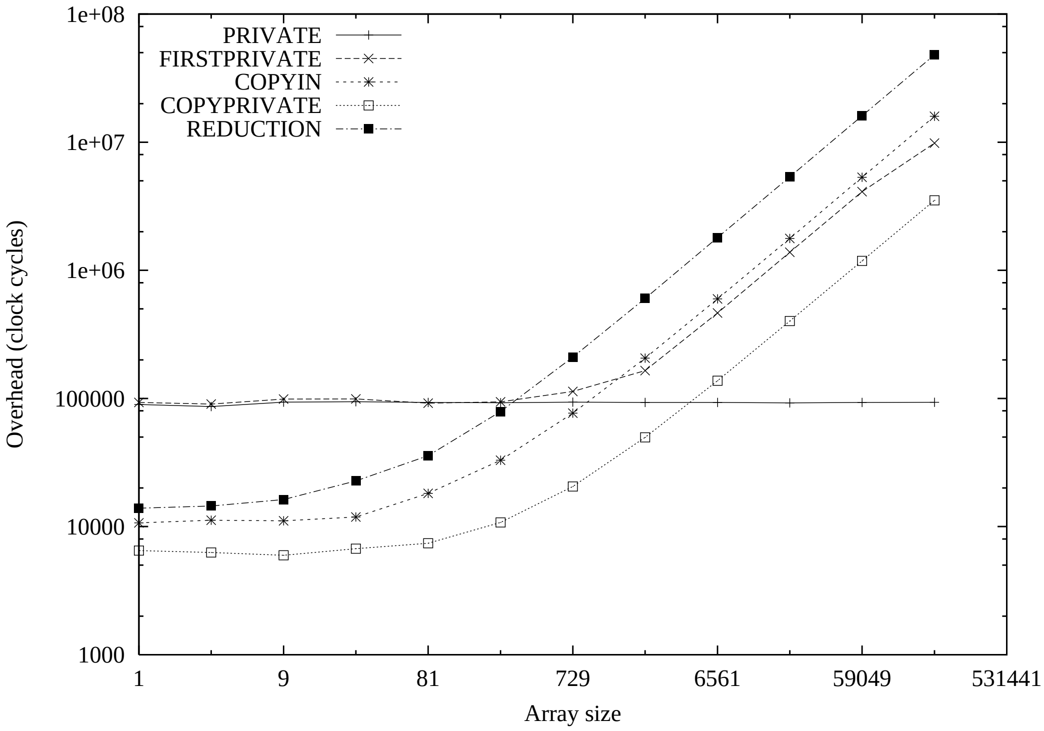

Fig. 4. Overhead of array clauses on 8 processors of a Sun HPC 6500 with WorkShop 6.2 compiler

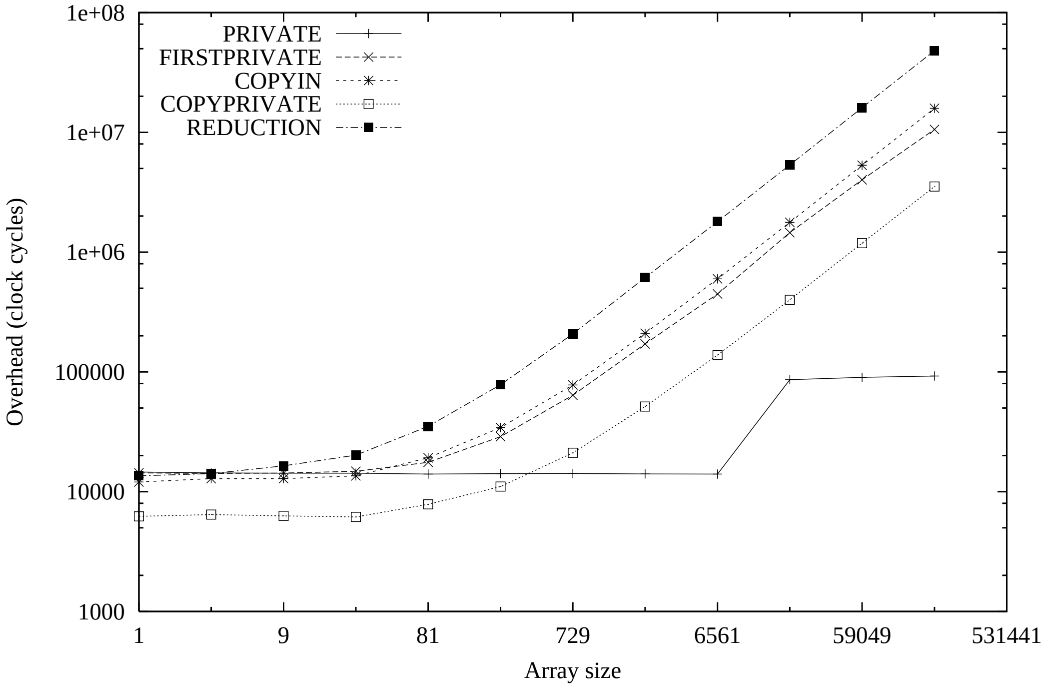

Fig. 5. Overhead of array clauses on 8 processors of a Sun HPC 6500 with WorkShop 6.2 compiler, using libmtmalloc

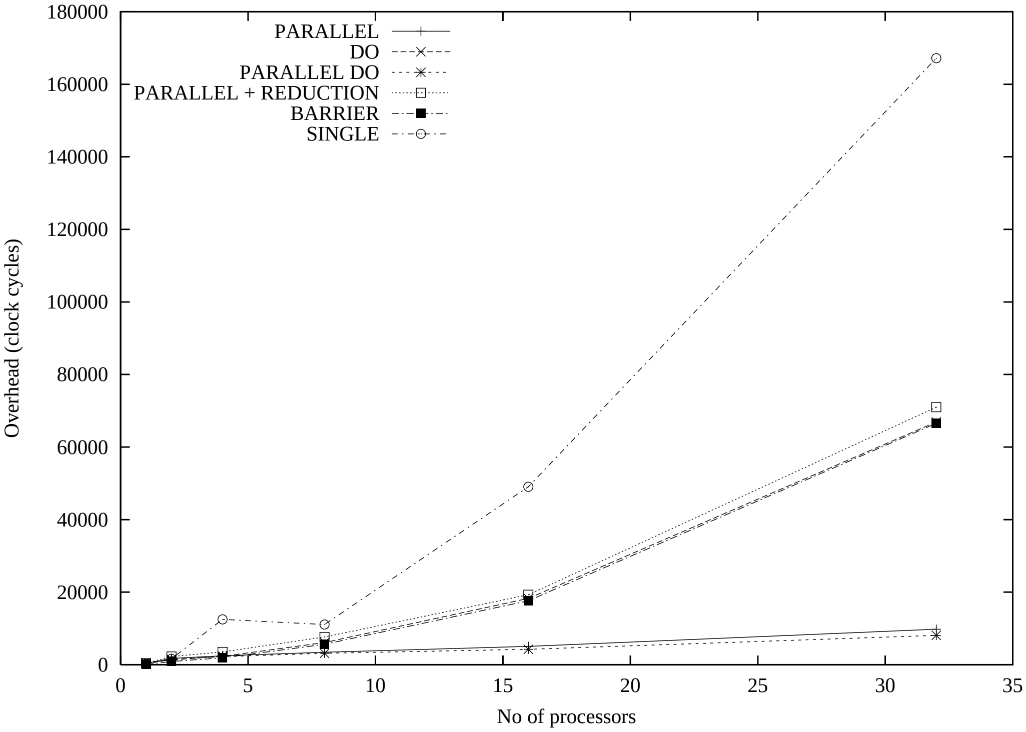

Fig. 6. Overhead of synchronisation directives on a SGI Origin 3000

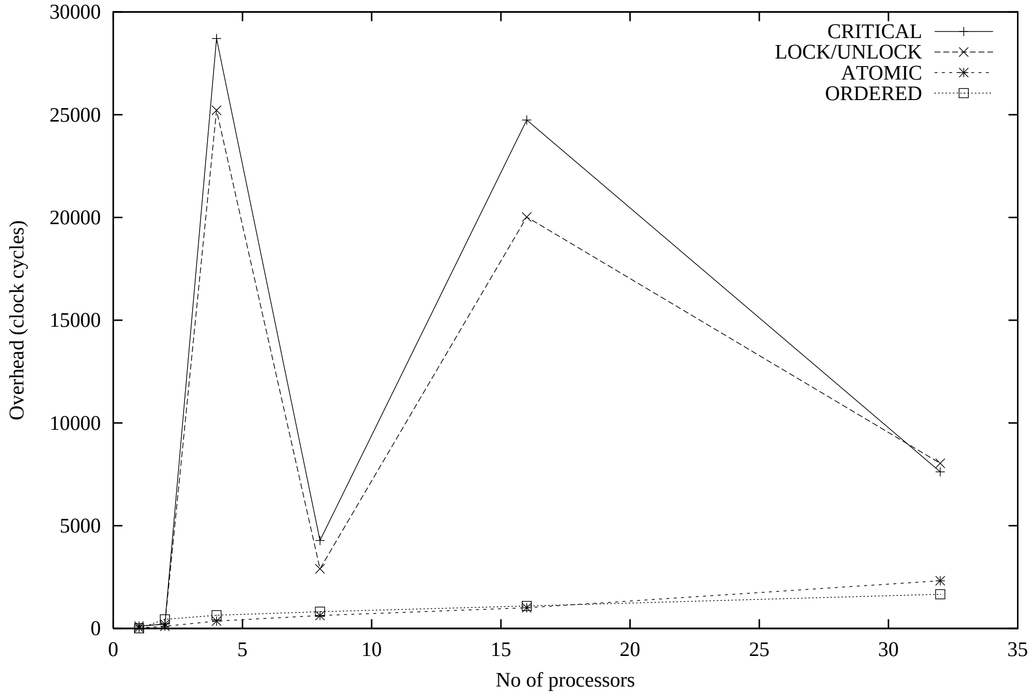

Fig. 7. Overhead of mutual exclusion synchronisation directives on a SGI Origin 3000

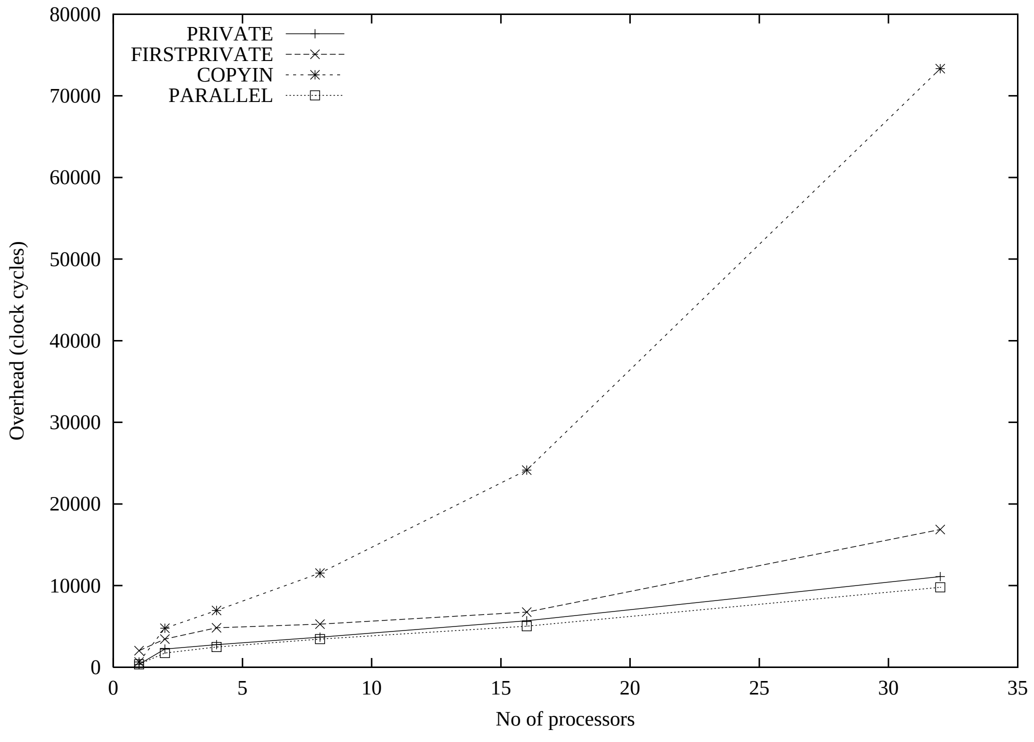

Fig. 8. Overhead of array clauses (array size = 729) on a SGI Origin 3000

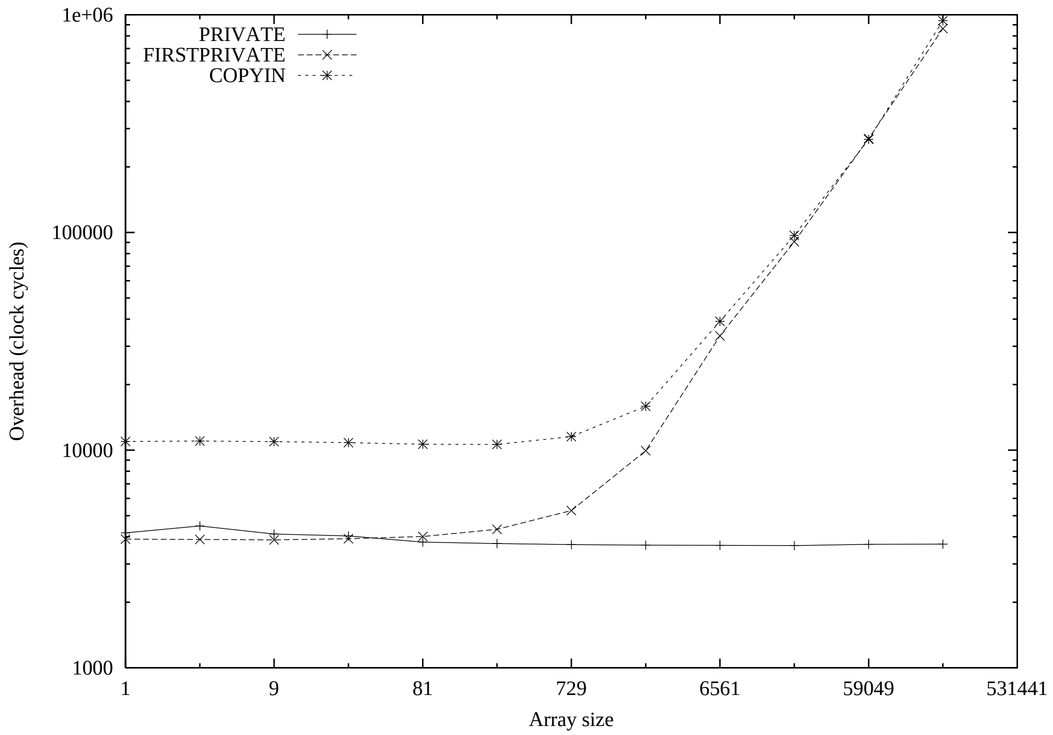

Fig. 9. Overhead of array clauses on 8 processors of a SGI Origin 3000

array arguments, where the dimension of the array is 729 elements. Figure 9 shows the same overheads for 8 threads and varying array sizes. In general, these overheads are an order of magnitude smaller than on the Sun, with almost no additional overhead for `PRIVATE` and a modest additional overhead for `FIRSTPRIVATE`. The `COPYIN` overhead is much larger than the `FIRSTPRIVATE` overhead, and shows worse-than-linear scaling. This suggest a memory allocation contention problem here as well, but we have not investigated this further.

IV. Conclusions and Future Work

We have presented extensions to a microbenchmark suite for OpenMP to include clauses with array arguments and features of OpenMP 2.0. We have presented results which demonstrate the utility of these benchmarks, both to programmers and to implementors. Although some advances have been made towards highly efficient OpenMP implementations, performance bugs remain on both the systems we examined here. Future work will consist of benchmarking other OpenMP 2.0 compilers as they become available, and producing a C/C++ version of the suite as the updated standard emerges.

References

[1] R. Berrendorf and G. Nieken, *Performance Characteristics for OpenMP Constructs on Different Parallel Computer Architectures*, Proceedings of First European Workshop on OpenMP, Lund, Sweden, Sept. 1999.

[2] J. M. Bull, *Measuring Synchronisation and Scheduling Overheads in OpenMP*, Proceedings of First European Workshop on OpenMP, Lund, Sweden, Sept. 1999, pp. 99-105.

[3] H. Jin *Programming Baseline for NAS Parallel Benchmarks*, `www.nas.nasa.gov/~hjin/PBN.html`

[4] OpenMP Architecture Review Board. OpenMP Fortran Application Program Interface, Version 1.1. Available from `www.openmp.org`, November 1999.

[5] OpenMP Architecture Review Board. OpenMP Fortran Application Program Interface, Version 2.0. Available from `www.openmp.org`, November 2000.

[6] Standard Performance Evaluation Corporation (SPEC), *SPEC OMP2001*, `www.spec.org/hpg/omp2001`