

OpenMP Microbenchmarks Version 2.0

Fiona J. L. Reid and J. Mark Bull

July 14, 2004

Abstract

Overheads due to synchronisation, loop scheduling and array operations are an important factor in determining the performance of shared memory parallel programs. We present a set of benchmarks to measure these classes of overhead for the language constructs in OpenMP. Results are presented for a Sun Fire 15K, an IBM p690+ and an SGI Altix, each with its own implementation of OpenMP. Significant differences between the implementations are observed, which suggest possible means of improving performance.

Contents

1	Introduction	1
2	Methodology	2
3	Results	5
4	Analysis	6
4.1	Barrier type synchronisation	6
4.2	Mutual exclusion synchronisation	7
4.3	Loop Scheduling	8
4.4	Array operations: variation with array size	9
4.5	Array operations: variation with number of processors	10
5	Conclusions	10

1 Introduction

Synchronisation, loop scheduling and array operations can all be significant sources of overhead in shared memory parallel programs. In OpenMP ([11], [10]), the cost of these operations is dependent on their implementation in the OpenMP runtime library. In this paper we describe techniques for measuring the overheads associated with synchronisation directives, scheduling directives and array operations. We present results for OpenMP Fortran 90 implementations on a Sun Fire 15K, an IBM p690+ and an SGI Altix. Results for OpenMP C implementations on the Sun Fire 15K and IBM p690+ are given in the Appendix. The C results were found to be very similar to the Fortran results and therefore no discussion is given. The similarity between the Fortran and C results is unsurprising as both OpenMP implementations likely share the same runtime library.

In this paper we present version 2.0 of the OpenMP microbenchmark codes which represent a significant update to version 1.0 released in 1999. Version 1.0 of the microbenchmarks is described in [2]. An extension to version 1.0 incorporating some OpenMP 2.0 features is described by [3]. The extended microbenchmark suite presented in this paper allows the measurement of OpenMP 2.0 [11] features and also includes a new benchmark which measures the overheads associated with array operations. The benchmark codes presented here are freely available and can be downloaded from <http://www.epcc.ed.ac.uk/research/openmpbench>.

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 inside 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 runtime 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, scheduling and array operation 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], [5]) the historical lack of standardisation in shared memory programming has meant that little work has been carried out in this area. Barrier synchronisation is an important feature of this programming model, and previous studies of barrier performance include [4] and [9]. 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, [8], [12], [14]).

As the shared memory programming model and OpenMP in particular have become more popular a variety of new benchmarks have been written. Examples include, the NAS Parallel Benchmarks [6], the SPECComp suite [13] and the LLNL OpenMP benchmarks [7]. The NAS benchmarks compare the performance at kernel level whereas the SPECComp benchmarks compare the performance of different application type/size codes. The LLNL benchmarks are similar to those presented here, however they are currently not publicly available. The benchmarks presented in this paper allow direct comparison of the runtime libraries of different OpenMP implementations.

The remainder of this paper is organised as follows: Section 2 describes the techniques used to perform the overhead measurements, and Section 3 presents the results of the measurements on the three different systems. These results are analysed in Section 4 and conclusions drawn in Section 5. The results obtained from running the C versions of the benchmarks on the Sun Fire 15K and IBM p690+ are presented without discussion in the Appendix.

2 Methodology

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

$$O_p = T_p - T_s/p \quad (1)$$

To measure the overhead of OpenMP directives, the technique used is to compare the time taken for a section of code executed sequentially with 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)
    end do
!$OMP END DO
end do
!$OMP END PARALLEL
```

and 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 than 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/for`, `BARRIER` and `SINGLE`.

The `WORKSHARE` and `PARALLEL WORKSHARE` directives also include implied barriers and are benchmarked using a similar method to that described above. Rather than including a loop and a call to a subroutine inside the parallel region, an array operation is performed instead. For example, to measure the overhead of the `WORKSHARE` directive, we measure the execution time for

```

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

```

where `a` is a one dimensional array with the number of elements equal to the number of threads, and subtracting the reference time, that is the time taken to execute the same loop in serial, i.e.

```

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

```

For the serial loop, `a` is a one dimensional array containing a single element. N.B. `WORKSHARE` and `PARALLEL WORKSHARE` are Fortran 90 specific OpenMP 2.0 directives.

For the mutual exclusion directives, a similar approach is taken. 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 reference time as described for the `DO` overhead above, and divide by `innerreps`. This method is also applied to the `omp_set_lock` and `omp_unset_lock` pairs.

To measure the overhead of the `ORDERED` directive, we measure the execution time for

```

!$OMP PARALLEL DO ORDERED SCHEDULE(STATIC,1)
do j=1,innerreps
!$OMP ORDERED
    call delay(delaylength)
!$OMP END ORDERED
end do
!$OMP END PARALLEL DO

```

subtract the same reference time as above, and divide by `innerreps`. It should be noted that both an `ORDERED` directive and an `ORDERED` clause are required.

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, as this ensures that the compiler is generating the same code for both the parallel version and the sequential reference version of the loop body. However, restrictions on the syntax of `ATOMIC` prevents us from doing so in this case.

To measure the overheads associated with loop scheduling, we compare the execution time of

```

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

```

to the execution time for

```

do i=1,itiersperthr
    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 chunk size. For simplicity, however, we fix 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.

The array benchmark compares the overheads associated with various clauses when applied to arrays. The clauses considered are: `PRIVATE`, `FIRSTPRIVATE`, `COPYPRIVATE`, `COPYIN` and `REDUCTION`. The `PARALLEL` directive is used with the `PRIVATE`, `FIRSTPRIVATE`, `COPYIN` and `REDUCTION` clauses and the `SINGLE` directive is used with the `COPYPRIVATE` clause. A number of executables are generated for the array benchmark so that the variation of overhead with array size can be examined. Separate executables are used as dynamically allocated arrays are not permitted in some of the clauses. Note, that the use of arrays in the `REDUCTION` clause is a OpenMP 2.0 Fortran feature and so is not available in the C version of the code.

For example, to measure the overhead of the `COPYIN` array directive, we subtract the time taken for

```

do i=1,reps
    call dummy(a)
end do

```

from the time taken for

```

do i=1,reps
!$OMP PARALLEL COPYIN(a)
    call dummy(a)
!$OMP END PARALLEL
end do

```

where `dummy` is a routine which contains a dummy loop performing a simple operations on the array `a`. The argument of the clause, `a`, is a one-dimensional array the size of which is varied in powers of three, from one to $3^{10} = 59049$. For the `COPYIN` and `COPYPRIVATE` clauses, the array passed to `dummy` must be declared as `THREADPRIVATE`.

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 runtime 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 (preferably to 64-bit). In particular, clock routines returning 32-bit floating point clock values in seconds (e.g. `etime`) will result in significant loss of precision and so should not be used. All the results presented in this paper use the OpenMP runtime library routine `OMP_GET_WTIME()` for the timing calculations. This routine returns 64-bit floating point values and is accurate to the nearest microsecond on the systems tested.

To obtain statistically meaningful results each overhead measurement is repeated 20 times within each run and 20 different runs are performed which means that each measurement is obtained from the average of 400 individual measurements. The reason for this is that a much larger variability is observed between different runs than between different measurements within the same run. The cause of this is unknown but it may be due to the physical memory locations of synchronisation variables altering from run to run. Although every effort is made to ensure we have exclusive access to a given machine/processors, this is not always achievable. Background processes (e.g. OS related) may also be contributing the variability observed between runs.

Within each run we compute the mean and standard deviation σ of all 20 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 also for cases where there are many outliers (values more than 3σ above the mean).

3 Results

The synchronisation, scheduling and array operations benchmarks were run on the following systems: a Sun Fire 15K with 52 900MHz Ultrasparc III processors using Forte Developer, Fortran 95 version 7.0, an IBM p690+ with 1620 1700MHz POWER4+ processors using XL Fortran for AIX, version 8.1.1 and an SGI Altix with 256 1300MHz Itanium 2 processors using the Intel Fortran Compiler for Linux, version 7.1. Both the Sun Fire 15K and IBM p690+ had OpenMP 2.0 compatible compilers installed allowing overheads to be calculated for OpenMP 2.0 directives/clauses. The SGI Altix did not have an OpenMP 2.0 compiler installed and therefore only results for OpenMP 1.0 directives are presented for it. The specifications of the three machines are summarised in Table 1.

The scheduling and array benchmarks were run across 8 threads/processors only. The synchronisation benchmark and the array benchmark for a fixed array size were run for various numbers of threads/processors from one up to the maximum number possible for the particular machine. The maximum number of processors available to a shared memory code are respectively 48, 32 and 62 for the Sun Fire 15K, IBM p690+ and SGI Altix.

Although the Sun Fire 15K has a total of 52 processors, the front-end of the machine requires 4 processors so the maximum number of processors available to batch jobs is 48. The IBM p690+ is constructed from 50 frames of 32 processors and therefore the largest shared memory job that can be run is 32 processors. The SGI Altix consists of four frames of 64 processors, however the operating system requires two processors, so the maximum possible

Machine	Processor	Processor speed (MHz)	No. of processors		OS	Compiler
			Total	SM		
Sun Fire 15K	Ultrasparc III	900	52	48	Solaris	Forte Developer, Fortran 95 version 7.0
IBM p690+	POWER4+	1700	1620	32	AIX 5.2	XL Fortran for AIX, version 8.1.1
SGI Altix	Itanium 2	1300	256	62	SGI Linux	Intel Fortran Compiler for Linux, version 7.1

Table 1: Summary of machine specifications. The Sun Fire 15K and IBM p690+ compilers were fully OpenMP 2.0 compliant, whereas the SGI Altix compiler was OpenMP 1.0 compliant.

shared memory job size is 62 processors.

Results are presented for Fortran 90 versions of code only. Figures 1 to 3 show the measured overheads against number of processors for the implied barrier directives on the three systems. Measurements for the `WORKSHARE` and `PARALLEL WORKSHARE` directives could not be obtained for the SGI Altix (Figure 3) as they are OpenMP 2.0 specific features. Note the scale changes on both axes between the three figures.

Figures 6 to 8 show the measured overheads against the number of processors for the mutual exclusion constructs on the three systems. Again, note the changes of scale between the three figures.

Figures 9 to 11 show the measured overheads against chunk size for the different loop schedules on the three systems. In each case the results are given for eight processors. Since we keep the number of loop iterations per processor approximately constant, the results for other numbers of processors should be similar. Note the change in scale on the y-axis between Figures 9 and 10 and Figure 11.

Figures 12 to 14 show the overheads of the various clauses with array arguments against array size on the three systems. Note the changes in the scale on the y-axis between Figures 12 and 13 and Figure 14.

Figures 15 to 17 show the variation of the array clauses against the number of processors for the three systems. A fixed array size of 729 is used on all three systems. Again, note the changes in scale on both axes between the three figures. Measurements for the `COPYPRIVATE` and `REDUCTION` clauses could not be obtained on the SGI Altix (Figures 14 and 17) as they are both OpenMP 2.0 features.

4 Analysis

4.1 Barrier type synchronisation

Figure 1 shows that the `BARRIER`, `DO`, `SINGLE` and `WORKSHARE` directives have similar performance on the Sun Fire 15K, suggesting that nearly all the cost of the `DO`, `SINGLE` and `WORKSHARE` directives is in the implied barrier. All four directives appear to scale well with increasing numbers of processors. The `PARALLEL`, `PARALLEL DO` and `PARALLEL WORKSHARE` directives take between 2-9 times as long as the `BARRIER` directive, with cost increasing steadily

with the number of processors. A slight increase in cost is observed at around 44 processors which may be spurious; high standard deviations were observed for these results. The addition of a `REDUCTION` clause to the `PARALLEL` directive significantly increases its cost and makes it less scalable, especially for more than eight processors.

Figure 2 shows that the `BARRIER`, `DO`, `SINGLE` and `WORKSHARE` directives behave in a similar way on the IBM p690+ as on the Sun Fire 15K except that the number of clock cycles are up to 12 times higher on the IBM p690+. Even allowing for the difference in clock speed between the machines these differences are somewhat surprising. The `PARALLEL` and `PARALLEL DO` directives are more costly (approx 30% greater for 24 processors) than the `BARRIER` directive. Unlike the Sun Fire 15K, the `PARALLEL + REDUCTION` directive costs approximately the same as the `PARALLEL` directive. This means that the cost associated with the `REDUCTION` is nearly all attributed to the cost of the `PARALLEL` directive. The `PARALLEL WORKSHARE` directive on the IBM p690+ scales very poorly, costing more than double the cost of `PARALLEL`. All the overheads on the IBM p690+ show a distinct change in gradient at 8 processors. This is not entirely unexpected as each 32 processor frame of the IBM p690+ comprises of four eight processor Multi-Chip Modules (MCM). It appears that communications within an MCM are significantly less costly than those between MCM's where the communications need to go via the bus interconnect. The Sun Fire 15K behaves much more like a true shared memory machine.

On the SGI Altix (Figure 3) the `BARRIER`, `DO` and `PARALLEL DO` directives have similar performance and cost. As with the Sun Fire 15K and IBM p690+, the `PARALLEL` directive is more costly (up to a factor of two) than the `BARRIER` directive. The main difference is that the `PARALLEL + REDUCTION` and `SINGLE` directives have almost the same overheads associated with them whereas for the other two systems the `SINGLE` directive showed similar performance to the `BARRIER` directive. For 1 to 32 processors, the performance of the `PARALLEL + REDUCTION` directive is poorest on the SGI Altix (see Figure 4). Beyond 32 processors the Sun Fire 15K has poorest performance and no data is available for the IBM p690+. The cost of the `SINGLE` directive is always greatest on the SGI Altix, e.g. for 32 processors the Altix is over 10 times slower than the Sun Fire 15K and nearly twice as slow as the IBM p690+. The huge overhead associated with the `SINGLE` directive can be avoided by using `MASTER` with an explicit barrier. However, this method will only be successful in situations where all threads reach the directive at approximately the same time. If the threads arrive at different times then no benefit will be gained from using the `MASTER` directive.

For the IBM p690+ the main limiting factor seems to be the cost of the `BARRIER` directive, which for more than eight processors (i.e. as soon as communications take place outside the MCM) is significantly higher than for the other two machines. Figure 5 shows the overhead associated with the `BARRIER` directive for the three systems. Any improvement to the implementation of the barrier on the IBM p690+ would give rise to a significant improvement in performance.

4.2 Mutual exclusion synchronisation

Figure 6 shows that for more than eight processors, the `ATOMIC` directive is the most costly on the Sun Fire 15K. Up to eight processors the cost of `ATOMIC` and `ORDERED` directives are approximately the same. A sharp increase in cost is observed around 44 processors which may be spurious as large standard deviations were obtained for these results. The `CRITICAL` and `LOCK/UNLOCK` directives cost roughly the same on the Sun Fire 15K and have roughly linear

scaling.

On the IBM p690+ (Figure 7) the `ORDERED` directive is the most expensive for all numbers of processors. The cost of `ORDERED` directive relative to the other mutual exclusion directives is staggering, e.g. for 24 processors the `ATOMIC` directive takes 1788 clock cycles, whereas the `ORDERED` directive takes 27173 clock cycles (more than 15 times larger). The `CRITICAL`, `ATOMIC` and `LOCK/UNLOCK` directives show a steady, nearly linear, increase in overhead time with number of processors. As with the Sun Fire 15K the `CRITICAL` and `LOCK/UNLOCK` directives have almost the same cost and roughly linear scaling. No obvious change in behaviour/gradient is observed around eight processors meaning that the mutual exclusion constructs are not sensitive to the MCM configuration.

Figure 8 shows that the `CRITICAL` and `LOCK/UNLOCK` directives behave in a similar way to both the Sun Fire 15K and IBM p690+, except that they are over two times slower on the SGI Altix. The `ATOMIC` directive is by far the least expensive and when compared to the other two systems it is found to be ten times faster than the Sun Fire 15K and four times faster than the IBM p690+. The `ATOMIC`, `CRITICAL` and `LOCK/UNLOCK` directives exhibit non linear scaling. Rather than a linear increase in overhead time with the number of processors, a sharp initial increase is observed from 1-4 processors followed by a constant overhead time for eight or more processors. This behaviour is perhaps unsurprising as each frame of the SGI Altix is comprised of sixteen nodes of four CPU compute modules. It appears that there is greater contention for resources within a single node than between nodes. The `ORDERED` directive does not follow this pattern, it has non-linear scaling where the relative cost improves as the number of processors increases.

The results show that in general these directives are well implemented, with the obvious exceptions of the `ORDERED` directive on the IBM p690+ and the `CRITICAL` and `LOCK/UNLOCK` directives on the SGI Altix. The `ATOMIC` directive shows poor scaling on the Sun Fire 15K but this may be spurious due to the large errors obtained for 44 and 48 processors.

4.3 Loop Scheduling

The results in Figure 9 show that on the Sun Fire 15K the block cyclic scheduling (`STATIC,N`) costs the same as a block schedule (`STATIC`) for chunk sizes greater than eight. For small chunk sizes (from 1 to 8) the overhead decreases rapidly (much faster than exponential, a degree six polynomial fits the data) with increasing chunk size toward the `STATIC` value. `DYNAMIC` scheduling is several orders of magnitude more expensive and shows a logarithmic decrease in overhead time with chunk size. `GUIDED` scheduling is over ten times as expensive as the `STATIC,N` schedule and shows a logarithmic decrease in cost with increasing chunk size. The rate of decrease of the `GUIDED` schedule is much slower than the `DYNAMIC` schedule. This is probably to be expected as the chunk size here is the minimum size of the chunk - the majority of the iterations are executed in large chunks. The fact that the `DYNAMIC` and `GUIDED` schedules never approximate to the `STATIC` or `STATIC,N` values is disappointing and should be improved upon. Note that results are not shown for large chunk sizes (>16) with the `GUIDED` schedule as this would result in load imbalance.

On the IBM p690+ (Figure 10) the overhead of the block cyclic schedule (`STATIC,N`) does not converge to that of the block schedule until the chunk size exceeds 32. Unlike the Sun Fire 15K, the overhead of the `DYNAMIC` schedule converges to around that of the block cyclic schedule for chunk sizes of 64 or greater. The `GUIDED` schedule behaves in a similar way to the Sun Fire 15K. The `DYNAMIC` and `GUIDED` schedules on the IBM p690+ are around three times

cheaper than on the Sun Fire 15K.

Figure 11 shows that on the SGI Altix, the overheads of the block cyclic schedule and block schedule cost approximately the same for all chunk sizes - meaning that the implementation of the `STATIC,N` schedule is particularly good on the SGI Altix. Curiously, and unlike the other two systems the `GUIDED` schedule is always more expensive than the `DYNAMIC` schedule and shows no sign of converging toward the `STATIC` values. Comparing with the other two systems, the overhead of the `GUIDED` schedule on the SGI Altix is more than twice that of the IBM p690+ and four times that of the Sun Fire 15K. The `DYNAMIC` schedule behaves in a similar manner to the other two systems and is consistently smaller, particularly for small (<32) chunk sizes.

The principal observation from these results is that schedules with smaller chunk sizes incur substantial overheads and can exceed 200,000 clock cycles on all three systems. The overheads are particularly large for `DYNAMIC` and `GUIDED` scheduling. As a result, if small chunk sizes are required then `STATIC` or `STATIC,N` scheduling should be used. The block cyclic scheduling (`STATIC,N`) is extremely poor on the IBM p690+ and would therefore be a good area to attempt optimisation. The `GUIDED` scheduling on the SGI Altix requires further investigation as it really should not be more expensive than the `DYNAMIC` scheduling. Comparison of the block schedule (`STATIC`) across the three systems shows the Sun Fire 15K to have the best overall performance (4339 clock cycles), followed by the SGI Altix (9699 clock cycles) and then the IBM p690+ (13818 clock cycles).

4.4 Array operations: variation with array size

Figure 12 shows the overheads of the various clauses with array arguments for the Sun Fire 15K. All the overheads except `PRIVATE` show a steady, linear increase with array size. The cost of the `PRIVATE` clause is approximately the same as the `PARALLEL` directive and does not vary with array size. The overhead of the `COPYPRIVATE` clause exhibits rather erratic behaviour for small (<729) array sizes, beyond which a steady increase is observed. The reason for this behaviour is unknown. The `REDUCTION` clause is always the most expensive and costs more than two million clock cycles for the array size of 59049.

On the IBM p690+ (Figure 13) the overhead associated with the `PRIVATE` clause behaves in a similar way to the Sun Fire 15K, except that it is 50% less expensive on the IBM p690+. The `COPYPRIVATE` clause is significantly cheaper on the IBM p690+ and does not show the erratic behaviour observed on the Sun Fire 15K for small array sizes. The `FIRSTPRIVATE` and `COPYIN` clauses have nearly identical costs which increase linearly with array size. A very sharp change in gradient is observed at array size 19683 suggesting that we may have encountered some architecture or implementation related limit. The overhead of the `REDUCTION` clause increases linearly with array size until 6561 and then levels out for array sizes of 6561 and greater.

Figure 14 shows the corresponding results for array clauses on the SGI Altix. As the Intel compiler does not support OpenMP 2.0 extensions only results for OpenMP 1.0 clauses, `PRIVATE`, `FIRSTPRIVATE` and `COPYIN` are shown. The `PRIVATE` clause shows somewhat erratic jumps at array sizes 27, 729 and 2187; again the reason for these jumps is currently unknown. The `FIRSTPRIVATE` and `COPYIN` clauses behave in much the same way as the Sun Fire 15K and IBM p690+, except that the number cycles required is less - up to 85 times faster than the IBM p690+ and up to 11 times faster than the Sun Fire 15K.

4.5 Array operations: variation with number of processors

Figure 15 shows the overheads of the various clauses with array arguments, where the size of the array is fixed at 729 elements. For comparison, we also show the `PARALLEL` directive overhead. The graph is dominated by the overheads of the `REDUCTION` clause, which totally swamp the other measurements. For 24 processors, the cost of the `REDUCTION` clause exceeds one million clock cycles. The overheads of all the clauses increase linearly with the number of processors. A sharp rise in cost is observed at 48 processors, however this value should be treated with skepticism as the standard deviations obtained were extremely large.

For the IBM p690+ (Figure 16) again the overhead associated with array `REDUCTION` is the largest. However, unlike the Sun Fire 15K, its cost is comparable to the overheads obtained for the `COPYPRIVATE` operation. Comparison with the Sun Fire 15K shows that the `REDUCTION` clause is over five times cheaper on the IBM p690+. The `FIRSTPRIVATE` and `COPYIN` clauses have the same costs associated with them. The same is true for the overheads associated with the `PARALLEL` and `PRIVATE` operations. For more than 16 processors the `FIRSTPRIVATE` and `COPYPRIVATE` clauses are roughly twice as expensive on the IBM p690+. With the exception of the `REDUCTION` clause, all overheads exhibit a change in gradient at 8 processors. Similar behaviour was observed for the barrier type synchronisation directives (see Figure 2). The change in gradient at 8 processors is most likely related to the MCM configuration discussed previously.

For the SGI Altix (Figure 17) the standard deviations obtained for measurements made on sixteen or more processors were very large and therefore comparison of these results with other two systems is meaningless. Considering only results from 1 to 8 processors, a linear relationship between overhead and number of processors is observed. The cost of the `PRIVATE` clause is comparable to that observed on the Sun Fire 15K. The cost of the `FIRSTPRIVATE` and `COPYIN` clauses are similar to those obtained on the IBM p690+. No comment can be made regarding the scaling for more than 8 processors as the errors obtained were too large (often larger than the measurement itself).

5 Conclusions

We have presented an updated set of benchmarks for measuring the overheads of synchronisation, loop scheduling and array operations in OpenMP 2.0. Particular emphasis has been placed on obtaining statistically meaningful measurements. The benchmarks have been run on three distinct shared memory platforms. Differences are observed both between the individual directives and between the same directive examined on each system. Some potential areas for optimisation have also 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] Bull, J.M. (1999) *Measuring Synchronisation and Scheduling Overheads in OpenMP*, Proceedings of the First European Workshop on OpenMP, Lund, Sweden, pp 99–105.

- [3] Bull, J.M. and O’Neill D. (2001) *A Microbenchmark Suite for OpenMP 2.0*, Proceedings of the Third European Workshop on OpenMP (EWOMP’01), Barcelona, Spain.
- [4] Grunwald, D. and Vajracharya S. (1994) *Efficient Barriers for Distributed Shared Memory Computers* in Proceedings of 8th International Parallel Processing Symposium, April 1994.
- [5] Hockney, R.W and Berry, M., (eds) (1991) *Public International Benchmarks for Parallel Computers*, PARKBENCH Committee: Report—1.
- [6] Jin, H. *Programming Baseline for NAS Parallel Benchmarks*, www.nas.nasa.gov/Software/NPB/
- [7] *LLNL OpenMP benchmarks*, www.llnl.gov/CASC/RTS_Report/openmp_perf.html
- [8] 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.
- [9] 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.
- [10] OpenMP Architecture Review Board, *OpenMP C and C++ Application Program Interface*, Version 2.0, March 2002.
- [11] OpenMP Architecture Review Board, *OpenMP Fortran Application Program Interface*, Version 2.0, November 2000.
- [12] 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.
- [13] Standard Performance Evaluation Corporation (SPEC), SPEC OMP2001 www.spec.org/hpg/omp2001
- [14] 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.

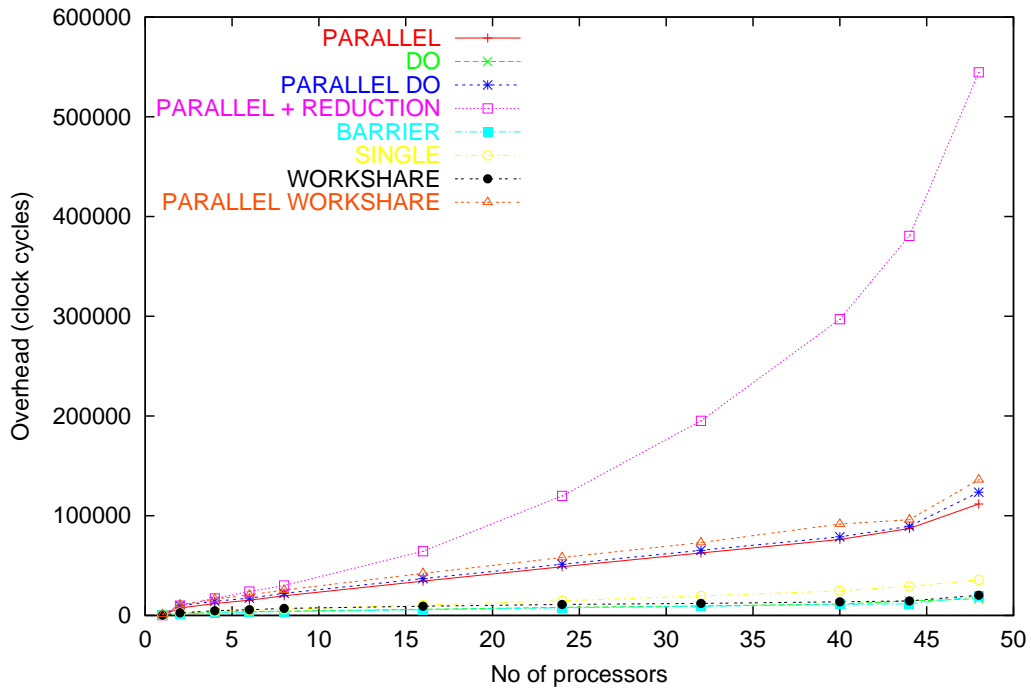


Figure 1: Synchronisation overheads on Sun Fire 15K

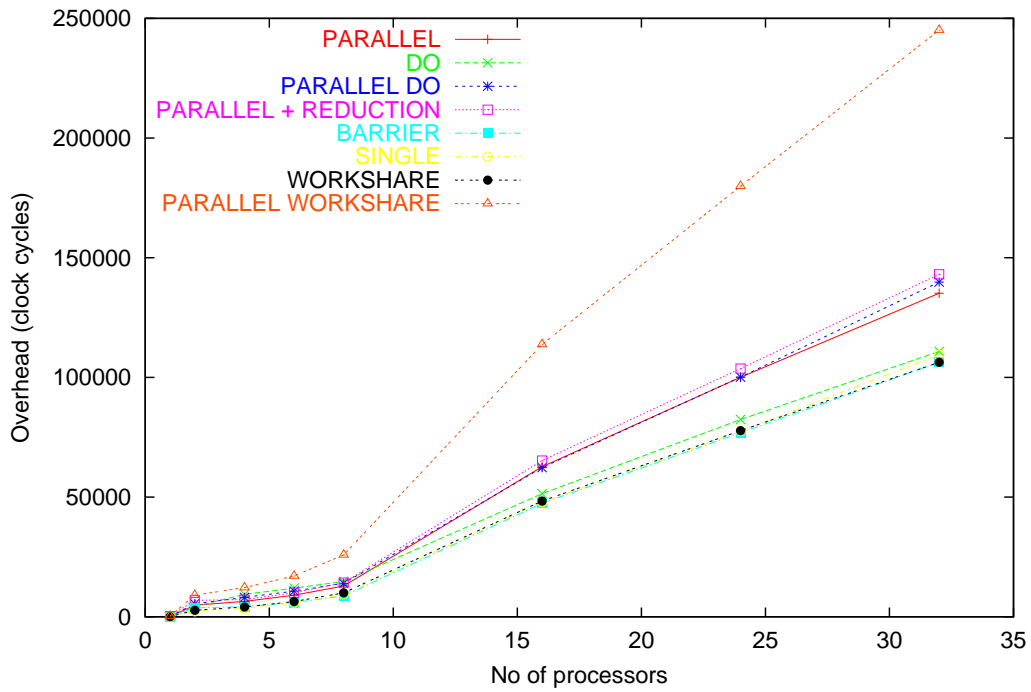


Figure 2: Synchronisation overheads on IBM p690+

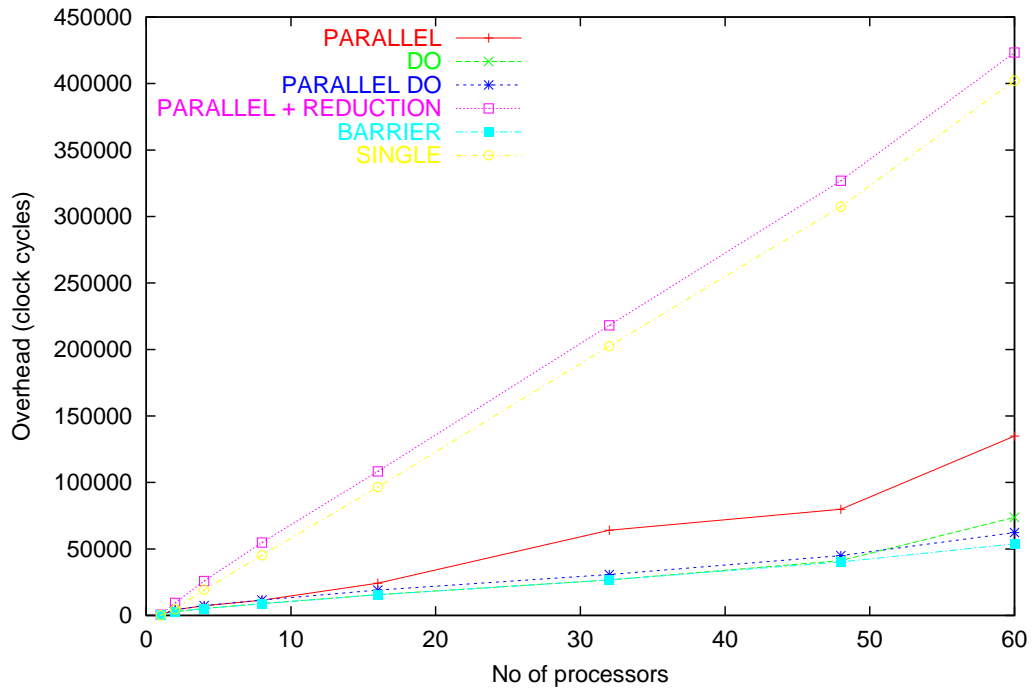


Figure 3: Synchronisation overheads on SGI Altix

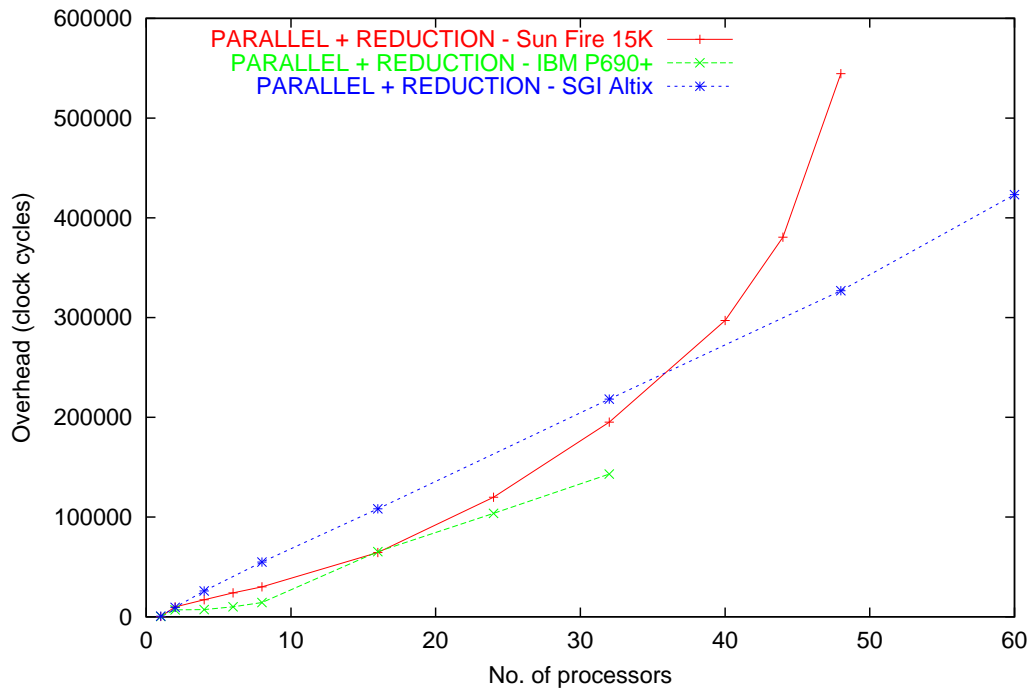


Figure 4: Overhead of PARALLEL + REDUCTION directive for the three systems

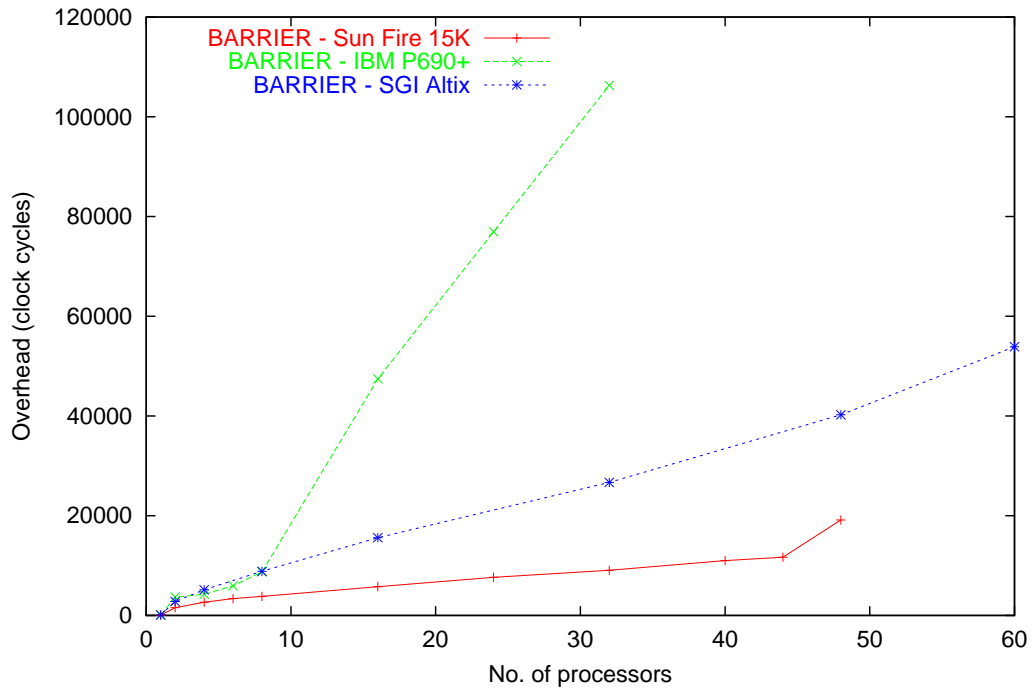


Figure 5: Overhead of BARRIER directive for the three systems

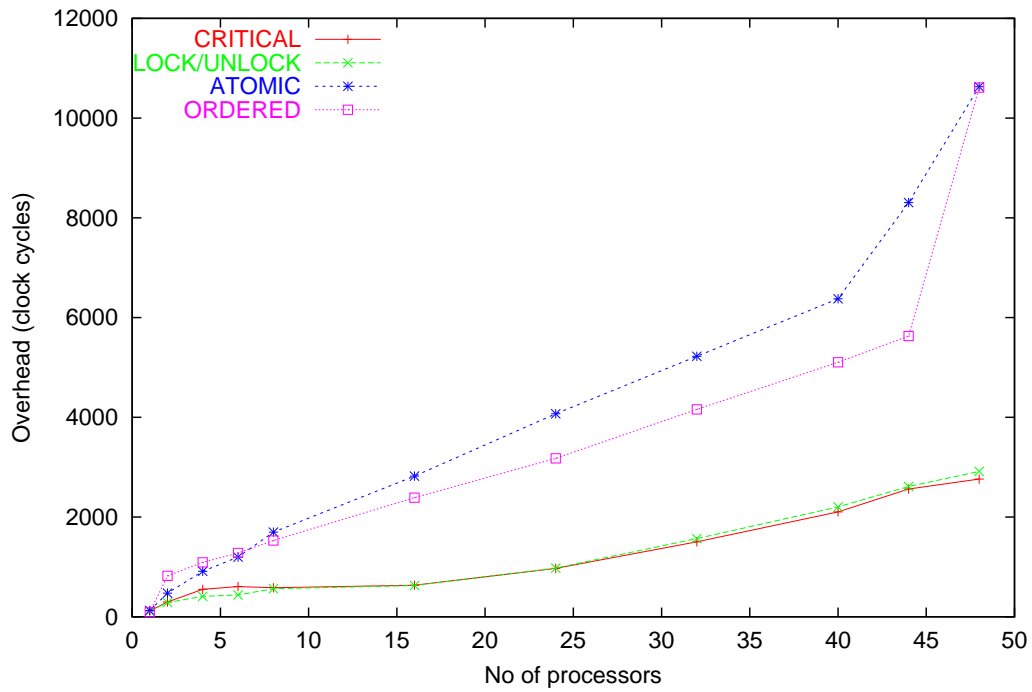


Figure 6: Mutual exclusion overheads on Sun Fire 15K

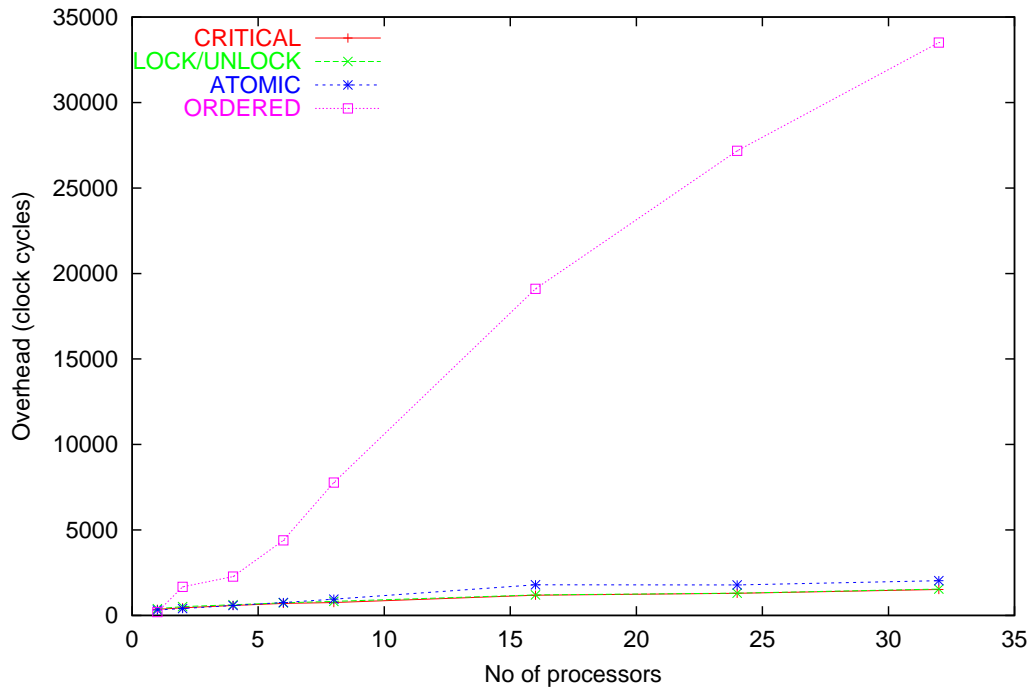


Figure 7: Mutual exclusion overheads on IBM p690+

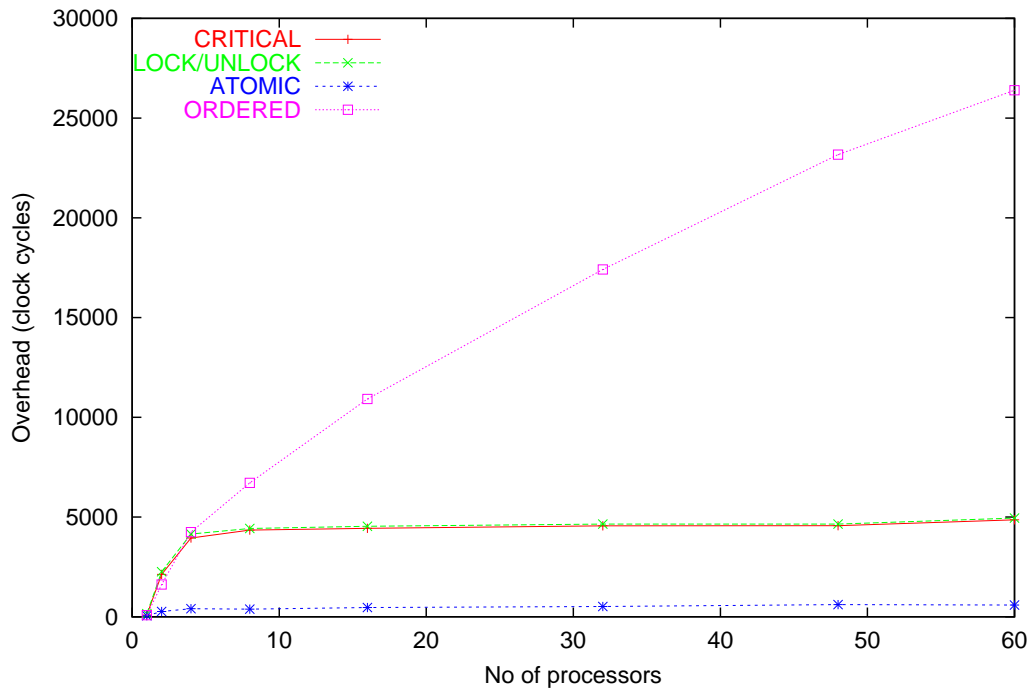


Figure 8: Synchronisation overheads on SGI Altix

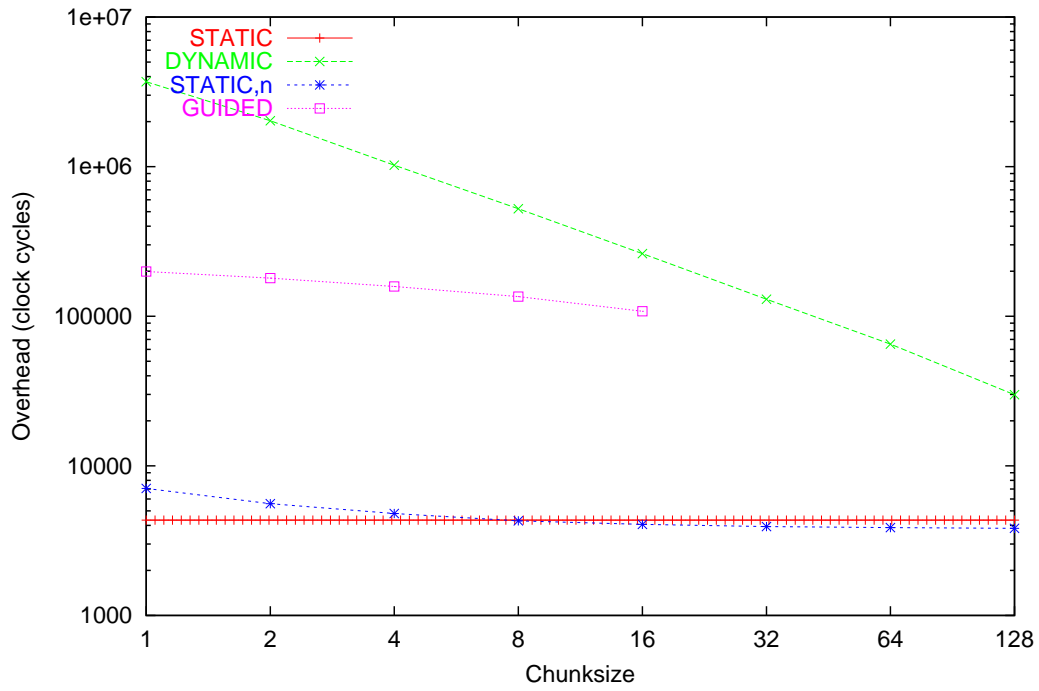


Figure 9: Scheduling overheads on Sun Fire 15K for 8 processors

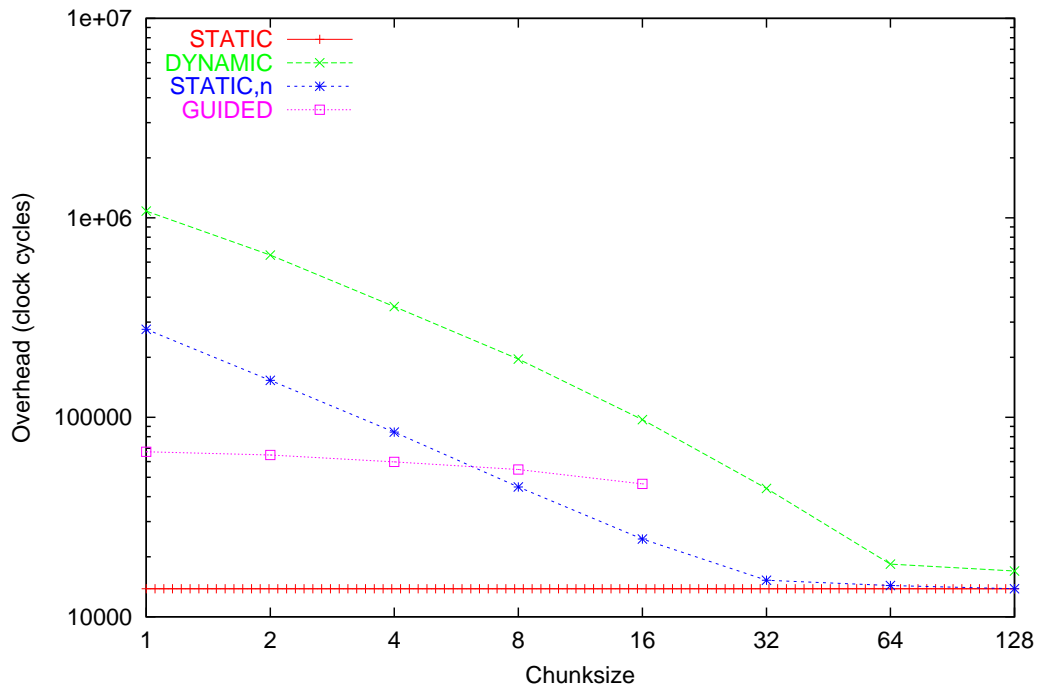


Figure 10: Scheduling overheads on IBM p690+ for 8 processors

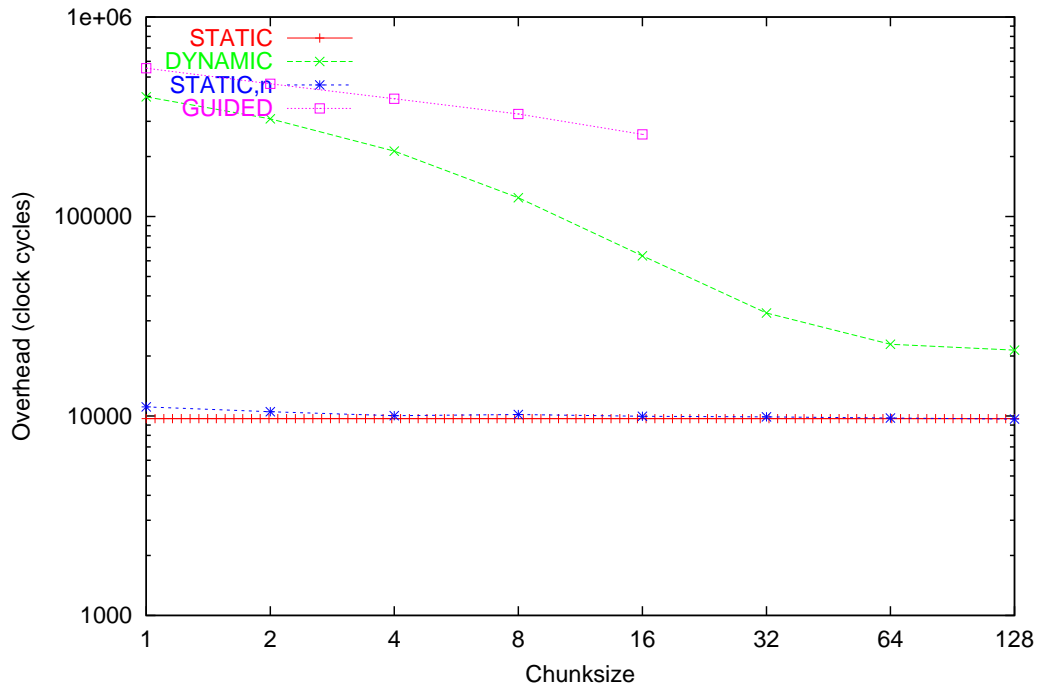


Figure 11: Scheduling overheads on SGI Altix for 8 processors

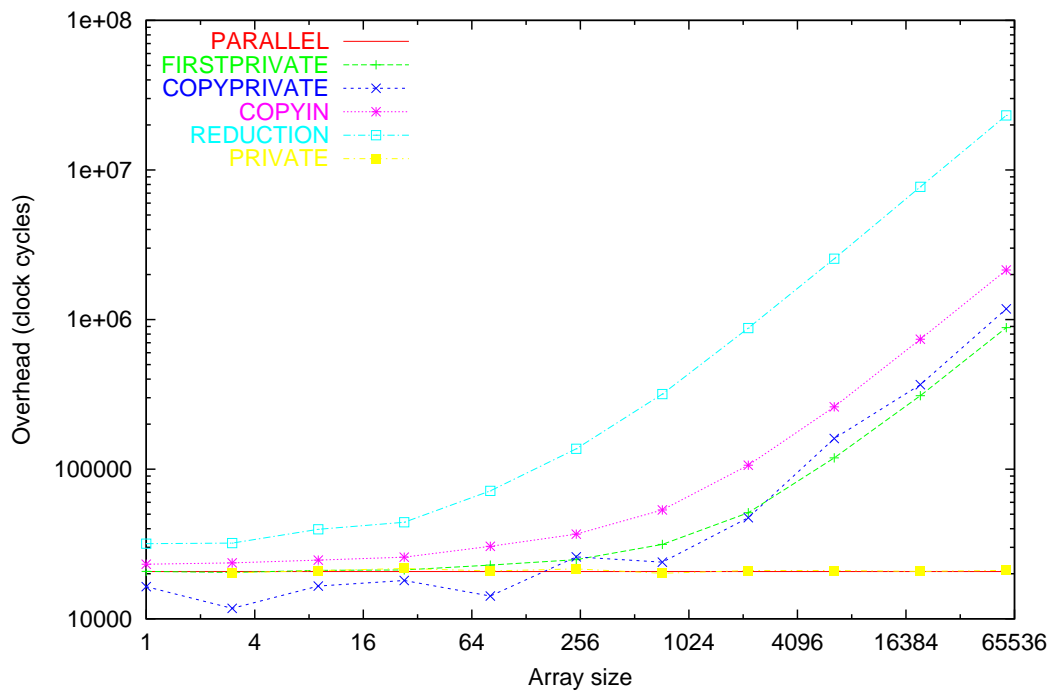


Figure 12: Array overheads on Sun Fire 15K for 8 processors

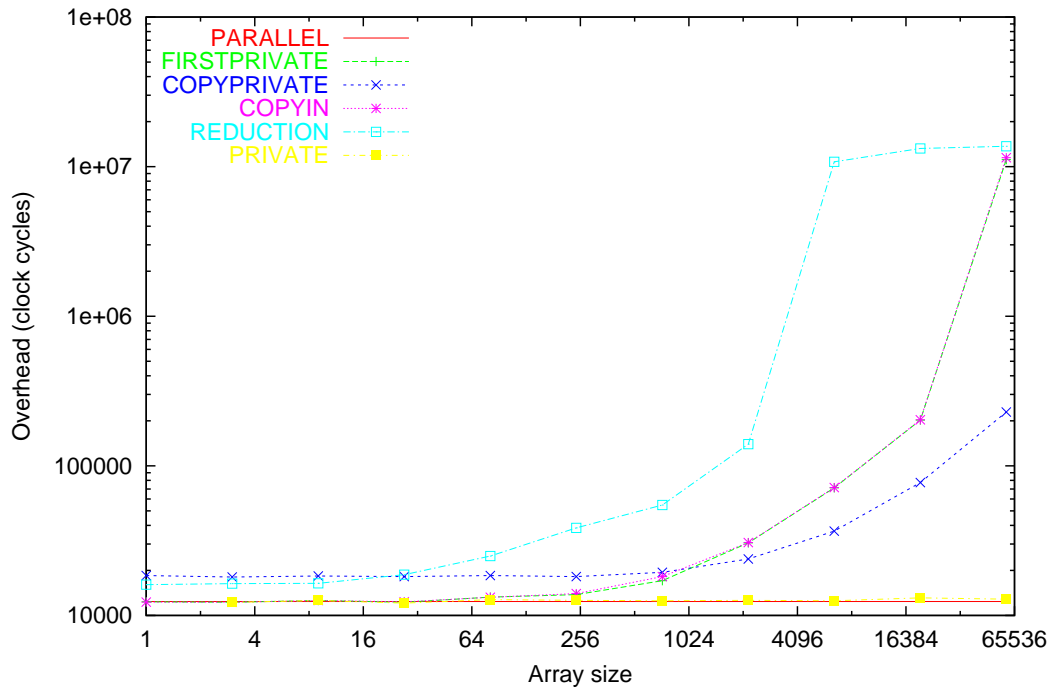


Figure 13: Array overheads on IBM p690+ for 8 processors

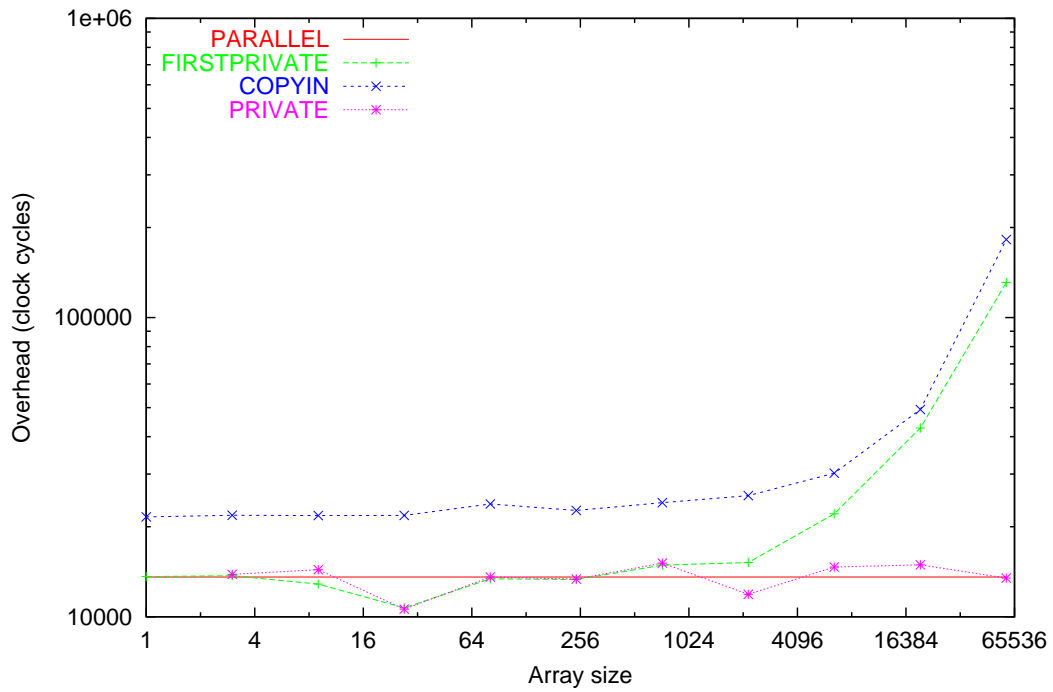


Figure 14: Array overheads on SGI Altix for 8 processors

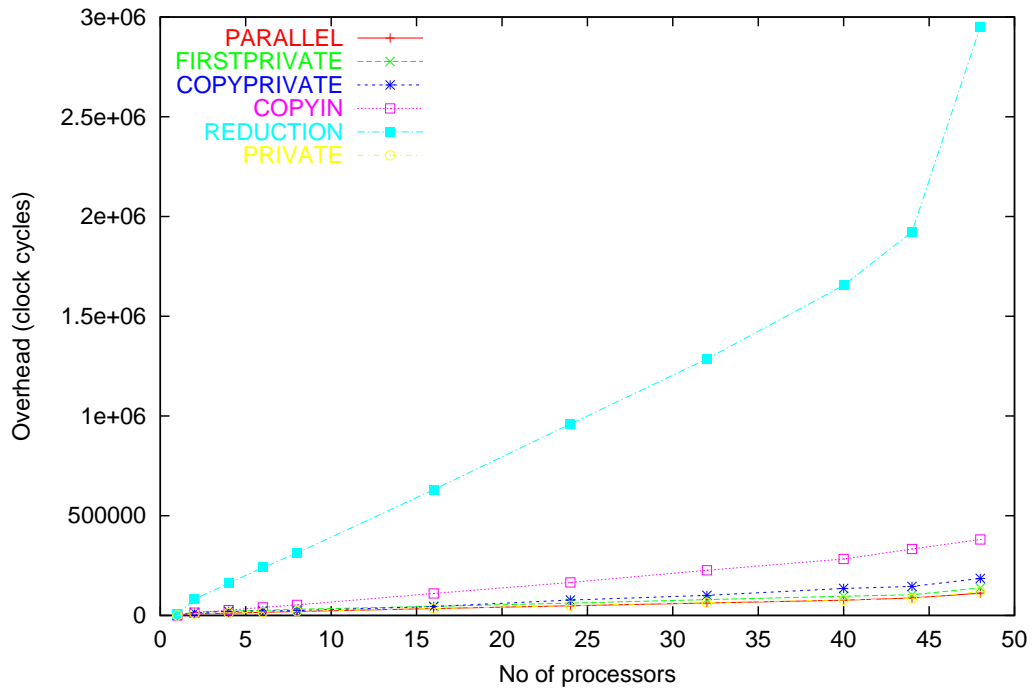


Figure 15: Array overheads on Sun Fire 15K for a one dimensional array of 729 elements

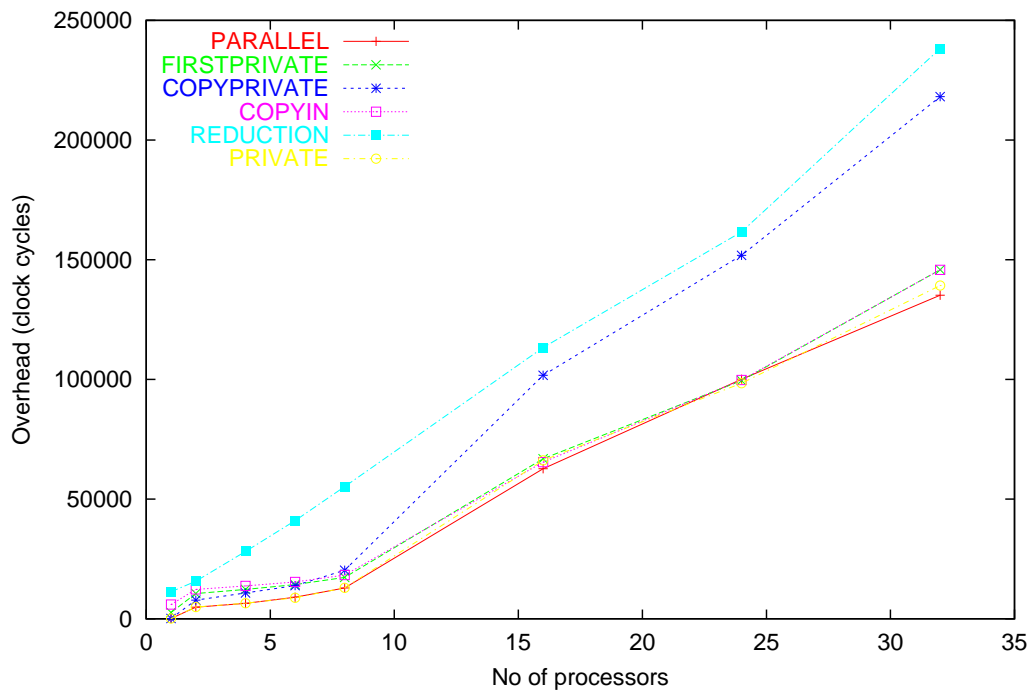


Figure 16: Array overheads on IBM p690+ for a one dimensional array of 729 elements

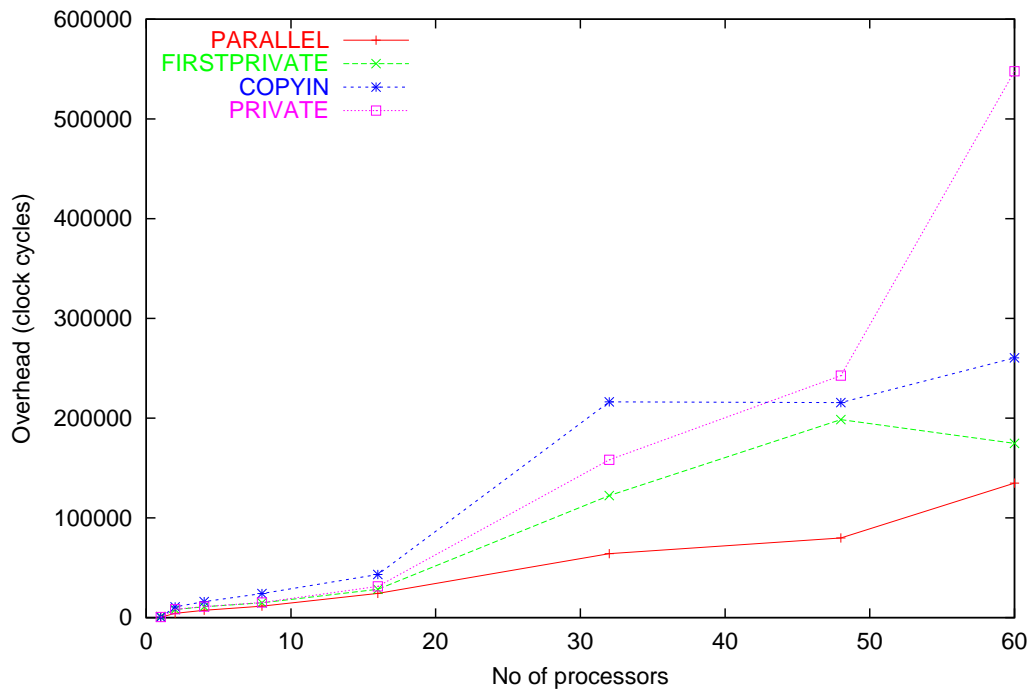


Figure 17: Array overheads on SGI Altix for a one dimensional array of 729 elements

Appendix

Results from the C versions of the benchmarks on the Sun Fire 15K and IBM p690+ are included for reference only.

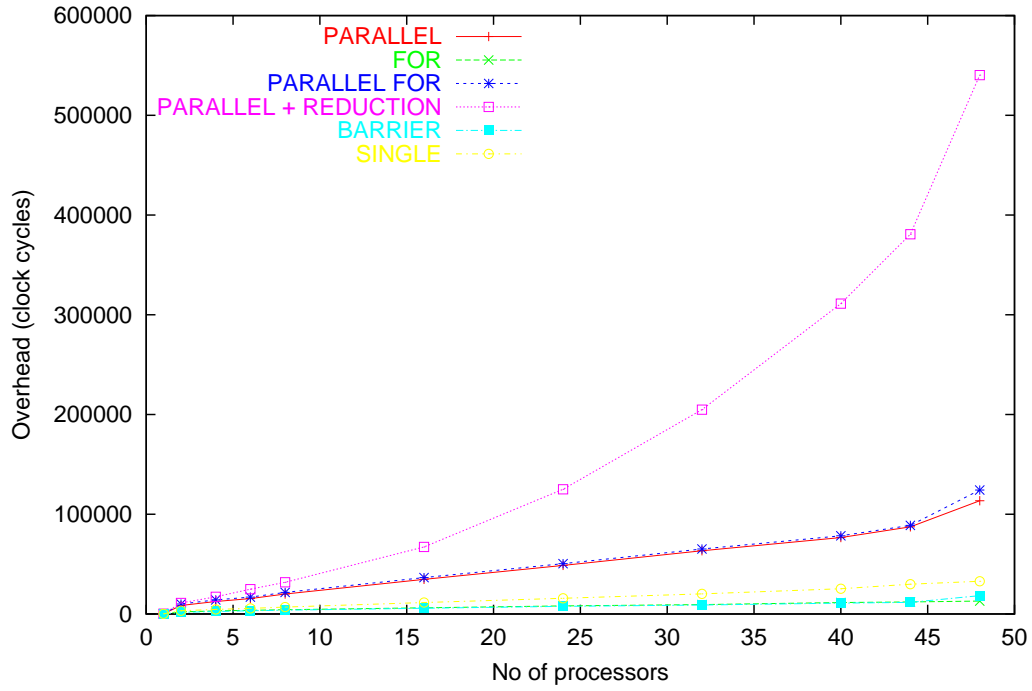


Figure 18: Synchronisation overheads on Sun Fire 15K

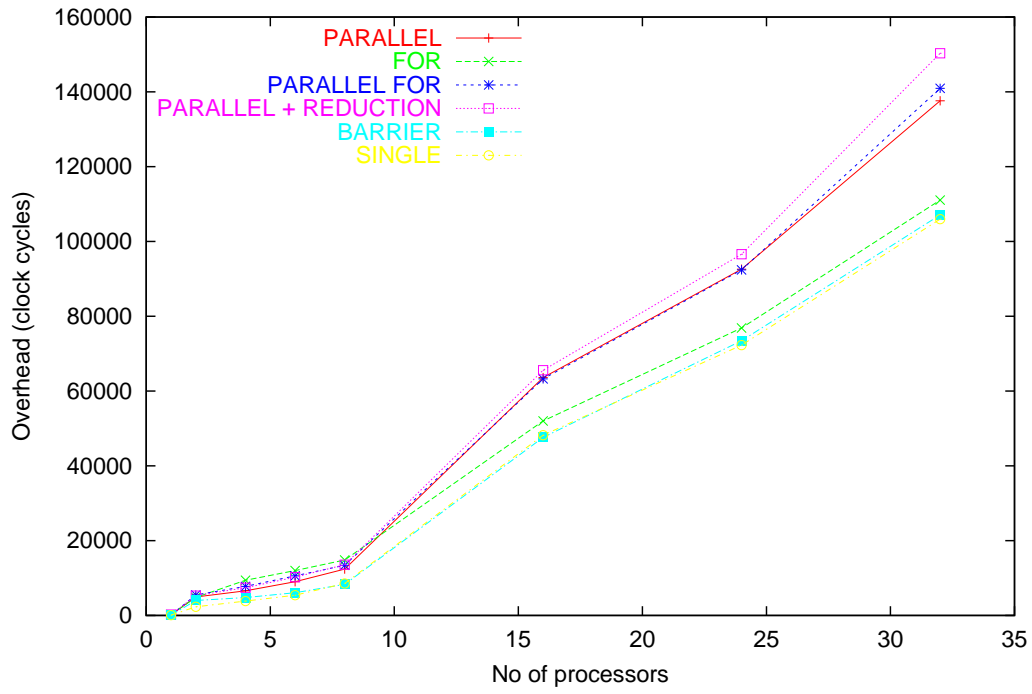


Figure 19: Synchronisation overheads on IBM p690+

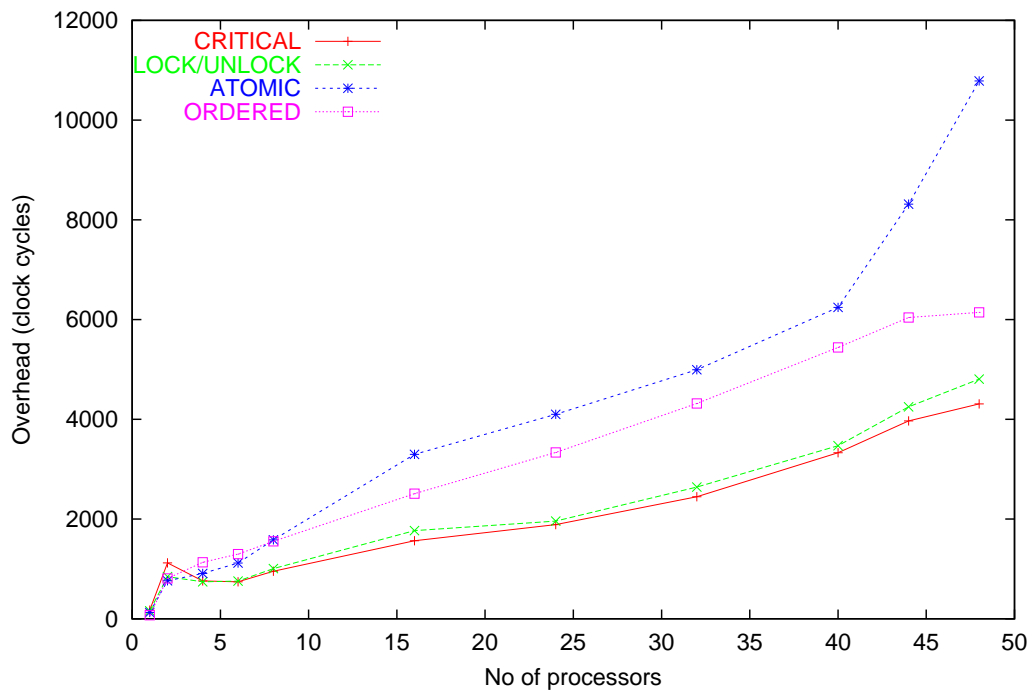


Figure 20: Mutual exclusion overheads on Sun Fire 15K

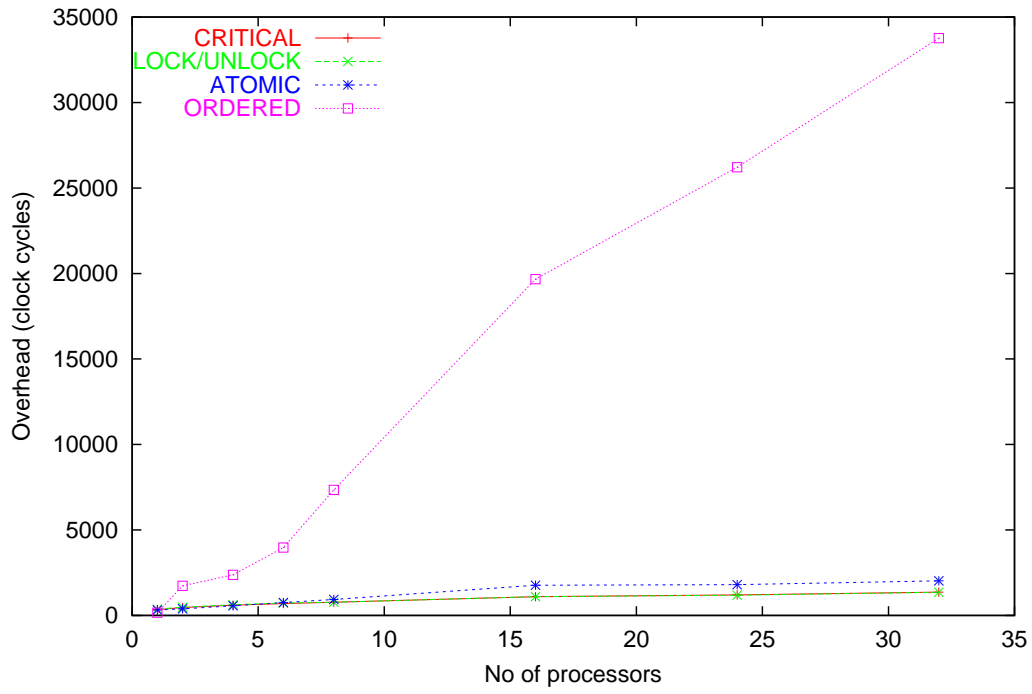


Figure 21: Mutual exclusion overheads on IBM p690+

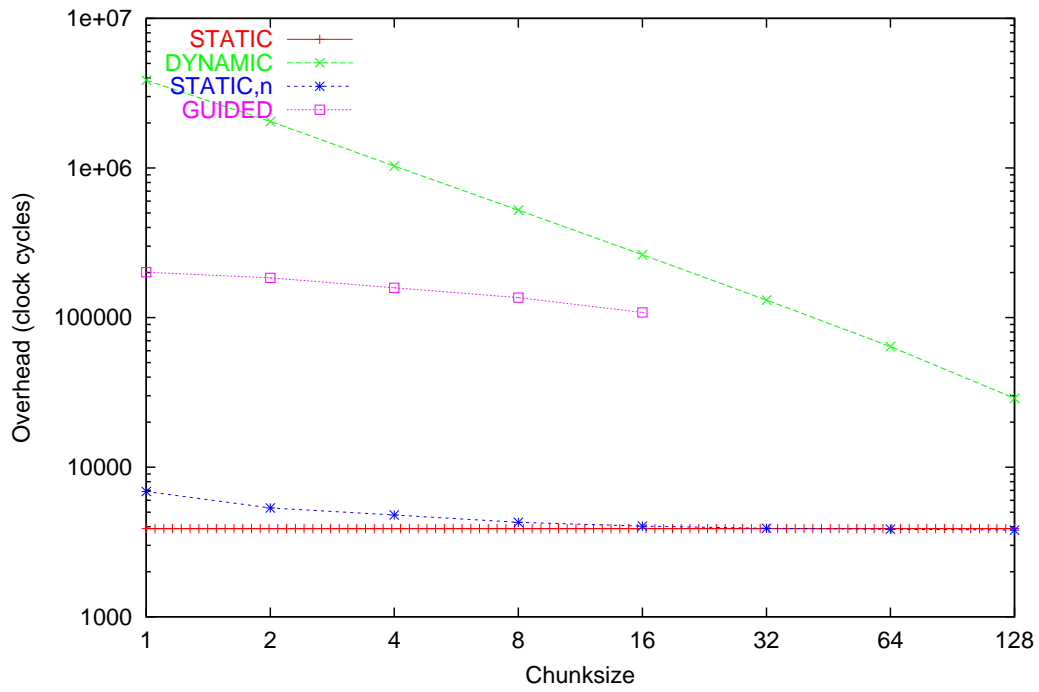


Figure 22: Scheduling overheads on Sun Fire 15K for 8 processors

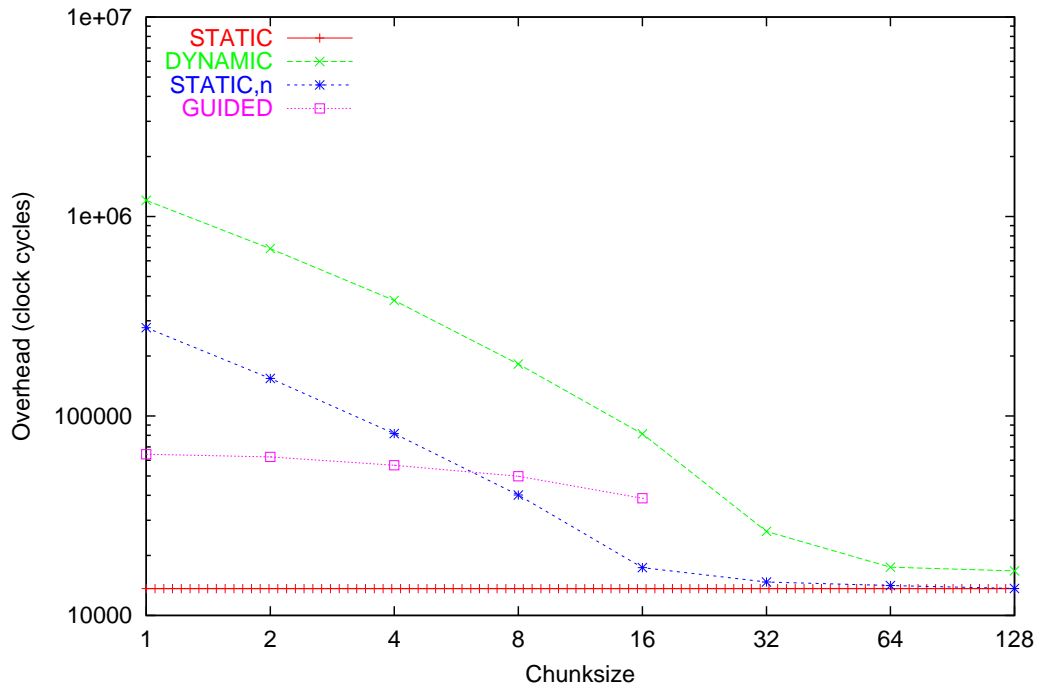


Figure 23: Scheduling overheads on IBM p690+ for 8 processors

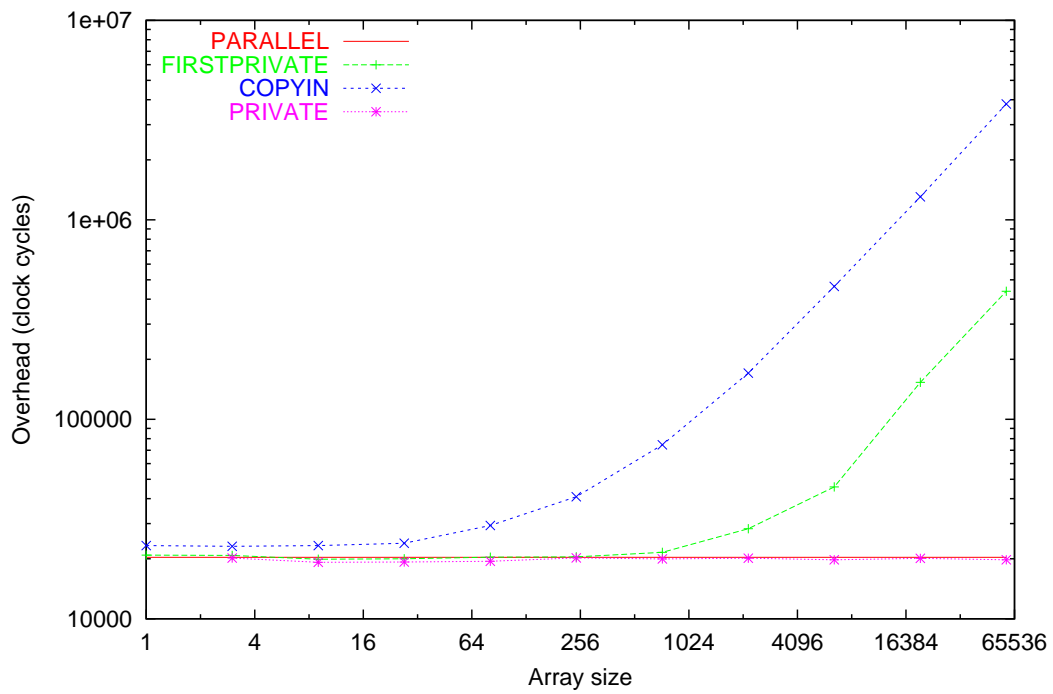


Figure 24: Array overheads on Sun Fire 15K for 8 processors

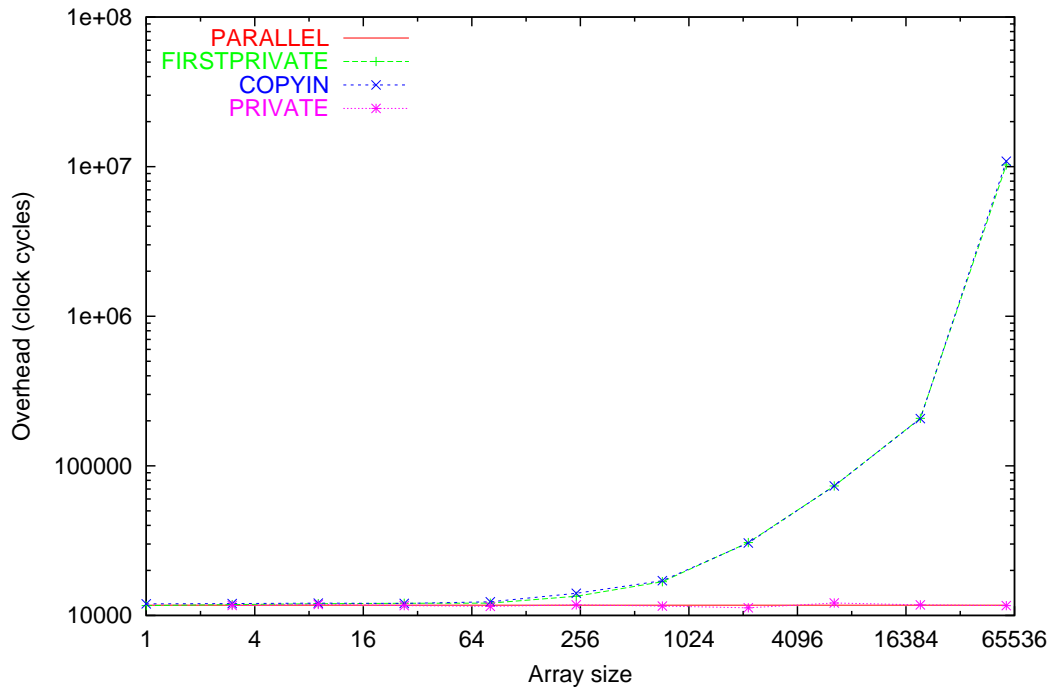


Figure 25: Array overheads on IBM p690+ for 8 processors

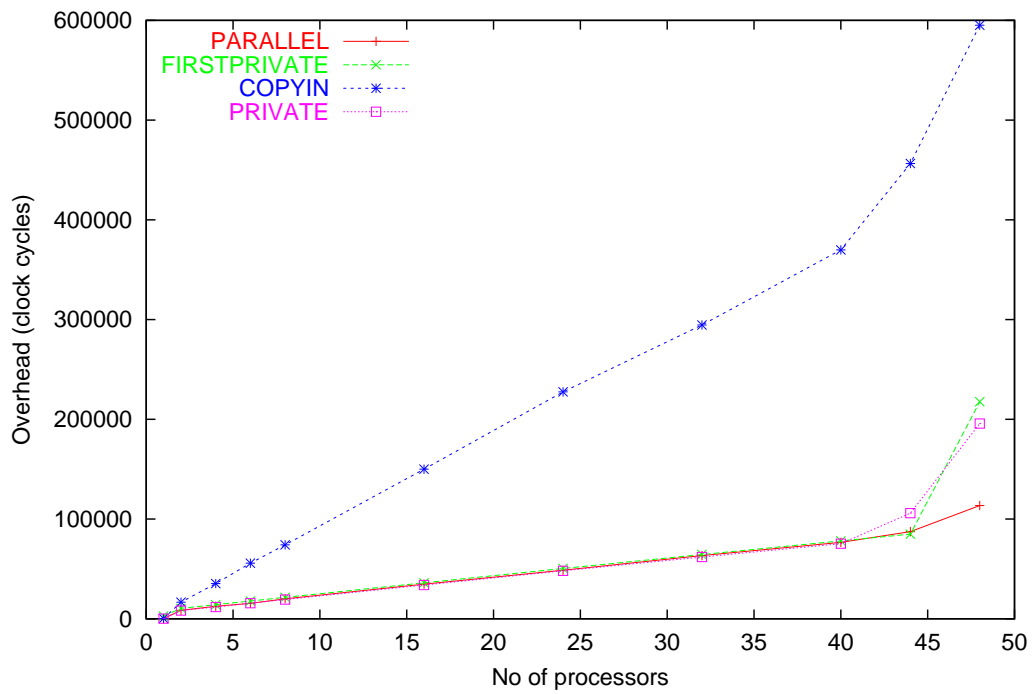


Figure 26: Array overheads on Sun Fire 15K for a one dimensional array of 729 elements

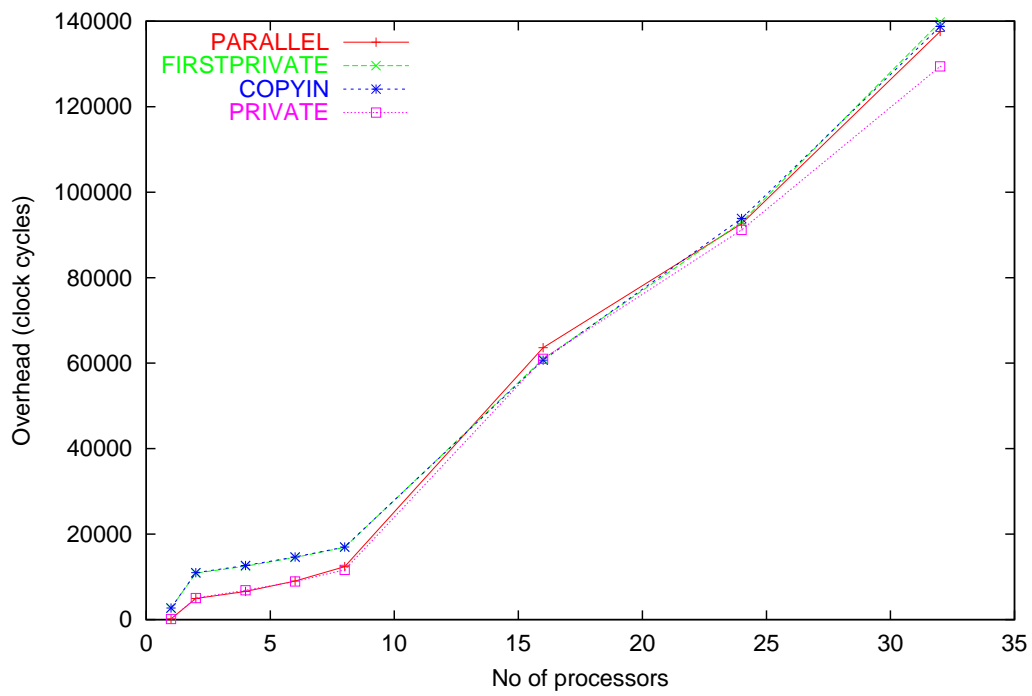


Figure 27: Array overheads on IBM p690+ for a one dimensional array of 729 elements