EPCC-SS2000-07

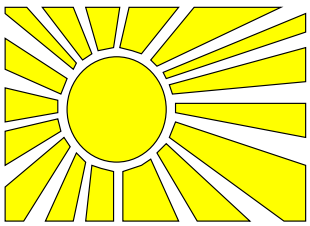
OpenMP Microbenchmarks

Darragh O'Neill

Abstract

OpenMP is the first standardisation of a set of compiler directives to implement shared memory parallel programming and as such it is important that we are aware of the cost of these implementations. Results are presented for the existing OpenMP Microbenchmarks for the Sun E6500 using different compilers. There are also results from an extension to the suite written as part of this project which measure the overhead associated with certain clauses and directives.

Another aspect of the project was the development of data processing scripts for the results of the benchmarking, involving statistical analysis and graphical interpretation.



Contents

1 Introduction

1.1 OpenMP

OpenMP provides a method of parallelising both C and Fortran code for shared memory parallel computers. It consists of compiler directives, environment variables and library routines which are more easily implementible than other parallel programming methods such as MPI. It is the first standardisation of shared-memory parallel programming directives that has been widely accepted (c.f. ANSI X3H5), is scalable and tackles many of the problems of previous attempts.

It has advantages over some of the other methods of parallel programming such as MPI and HPF. With respect to MPI, OpenMP is easier to program and more scalable. MPI was developed for distributed memory architecture and uses costly message passing semantics which are not required for modern scalable systems with globally distributed and cache coherent memories. It is also easier to parallelise existing serial code with OpenMP, whereas MPI requires much more modification of the code.
HPF never really took off with developers or hardware vendors and can sometimes perform well but other times it can be disappointing due to the limitations of the language.

1.2 Benchmarking

Benchmarking OpenMP allows evaluation of its performance with several things in mind. Firstly we can look at the implementation of OpenMP by different compilers to try to identify any means of improvement. We can also run the benchmarks on different systems to see what type of architecture best suits the use of OpenMP and also to see how the implementation scales with addition of more processors. Another important use is for the comparison of semantically equivalent directives such as CRITICAL and ATOMIC and to provide guidelines to developers as to the more efficient. It also allows developers to estimate the overheads present in their code by simply counting the number of directives and multiplying by the corresponding overhead.

1.3 The goals of this Project

The aims of this SSP project were as follows

- To write Perl scripts which process the output from the Benchmarking suite and generate more user friendly output such as HTML tables and Gnuplot graphs.
- To extend the Benchmarking suite in both Fortran and C to include the directive clauses PRIVATE, FIRSTPRIVATE, COPYIN and also to include the FLUSH directive.
- To run the suite on several different systems and compilers and to analyse and compare the results from these.

2 The OpenMP Microbenchmarking Suite

2.1 How the benchmarks work

The benchmarking of the directives are calculated simply by a comparison of the time taken for the same piece of code to be performed sequentially (Figure 1) and in parallel (Figure 2). The code used is a simple delay routine which increments a variable a given number of times. The code is performed in two loops. The inner one performs it `innerreps` times (usually 1000-10000) to reduce the error in the calculation of the overhead. The outer one performs it `outerreps` times (usually 10-50) to ensure there is a reasonable statistical distribution of samples.

```
      do k=0,outerreps
         start  = getclock()
         do j=1,innerreps
            call delay(dl)
         end do
         time(k) = (getclock() - start)* 1.0e6 / dble (innerreps)
      end do
```

Figure 1: Generation of reference time in serial

```
      do k=0,outerreps
         start  = getclock()
         do j=1,innerreps
!$OMP PARALLEL
            call delay(dl)
!$OMP END PARALLEL
         end do
         time(k) = (getclock() - start) * 1.0e6 / dble (innerreps)
      end do
```

Figure 2: Timing of directive

The overhead is then calculated using $T_p - T_s$, where T_p is the time taken to execute the code in parallel on p processors and T_s is the time taken to execute the code in serial.

2.2 Data Post-Processing

2.2.1 Outputted Data

The general format of the output from the benchmarking suite is shown in Figure 3.

```
Computing DIRECTIVE time

Sample_size       Average    Min         Max         S.D.      Outliers
         50       12.36719   12.10938    12.89062    0.20290          0

DIRECTIVE time =                          12.37 microseconds +/-  0.398

DIRECTIVE overhead =                       5.39 microseconds +/-  0.666
```

Figure 3: Sample benchmarking data

We would like to extract all of the information from the output files and display it in HTML tabular form and also the overhead into graphical form, plotted against some other value such as number of processors, chunksize or array size.

2.2.2 Using Perl to Process the Data

Extraction and output of the data using Perl is not a difficult task, but we wanted a very general form of extraction method and we also wanted to do some data processing as well. The desired script would take any number of input files, which could be concatenated runs or individual runs, and sort them by a given criteria. It would then take averages, two forms of standard deviations, minima and maxima and output these to HTML tables and Gnuplot graphs. The two standard deviations are, firstly over all samples and secondly over runs. The benchmarks output enough data (standard deviation, average and sample size) to recompute the sum of the squares over all samples and this in turn can be used to compute the true standard deviation over all samples. It can be seen more clearly in the maths. For the i^{th} run, we have n_i samples, so the standard deviation within the i^{th} run, σ_{ij}, is given by

$$\sigma_i = \sqrt{\frac{\sum_{j=1}^{n_i}(x_ij - \overline{x}_i)^2}{n_i - 1}}$$

By simple manipulation we can recover the sum of squares for the i^th run:

$$\sum_{j=1}^{n_i} x_{ij}^2 = \sigma_i^2(n_i - 1) + n_i\overline{x}_i^2$$

The standard deviation, σ, of all samples from all runs can then be computed as

$$\sigma = \sqrt{\frac{\sum_{i=1}^{R}\sum_{j=1}^{n_i} x_ij^2 - N\overline{x}^2}{N - 1}}$$

where $\overline{x}$ is the mean calculated over all runs, n_i and $\overline{x}_i$ are the number of samples and the mean respectively over i runs and N is the total number of samples. The other measure of interest is

the standard deviation of teh means of each run, σ_R, is given by

$$\sigma_R = \sqrt{\frac{\sum_{i=1}^{R}(\overline{x_i} - \overline{x})^2}{R-1}}$$

where R is the number of runs. The approach taken was simply to take all of the files and separate them all into temporary files containing the data from one run and record these filenames with an associated value (Number of Processors). These files were then sorted by this associated value and processed them in similar groups taking averages, standard deviations etc. Finally all of the processed data was tabulated into HTML form and gnuplot form.

2.2.3 Results Produced using Perl

An example of the HTML output from the Perl scripts is shown in Figure 4

Synchronisation Benchmarking Data

Hardware Platform: Sun Ultrasparc E6500

Operating System: Solaris

Compiler: Forte 6

Data for 1 threads
Number of Runs = 20
Number of Samples = 400

Directive	Overhead	Average	SD over Samples	SD over Runs	Max SD	Min	Max	Outliers
PARALLEL	7.194 +/- 0.116	11.059	0.059	0.021	0.100	10.974	11.393	5
DO	1.918 +/- 0.051	5.783	0.026	0.014	0.034	5.729	5.867	0
PARALLELDO	2.112 +/- 0.053	5.977	0.027	0.017	0.044	5.919	6.051	0
BARRIER	0.237 +/- 0.018	4.100	0.009	0.009	0.006	4.085	4.121	3
SINGLE	0.515 +/- 0.027	4.381	0.014	0.011	0.014	4.355	4.435	3
CRITICAL	0.105 +/- 0.006	3.969	0.003	0.003	0.004	3.965	3.980	1
LOCK/UNLOCK	0.131 +/- 0.006	3.993	0.003	0.003	0.001	3.990	4.002	0
ORDERED	0.342 +/- 0.018	4.208	0.009	0.005	0.017	4.189	4.239	2
ATOMIC	0.173 +/- 0.018	0.175	0.009	0.007	0.015	0.165	0.205	0
REDUCTION	3.787 +/- 0.057	7.712	0.029	0.018	0.037	7.624	7.811	0

Figure 4: HTML output from Perl Scripts

The Perl script takes the filenames to be processed as command line arguments. It then prompts the user for several pieces of information - the platform, operating system, compiler and clock speed. The clock speed is used in the gnuplot output so the overhead is outputted in clock cycles which makes comparisons of different systems easier. The gnuplot files, named automatically according to which directive they correspond to, are simply columns of tab separated numbers. The script also supplies a command file which can be loaded in gnuplot to format the graphs automatically. Examples of gnuplot graphs produced from these scripts can be seen in Section 3.

2.3 The New Benchmarks

2.3.1 The PRIVATE clause

The PRIVATE clause declares the variables in a list to be private to each thread in a team. On encountering this clause a new object of the same type is created for each thread. Any reference to a variable of the same name before the PARALLEL construct is ignored, i.e. the variables are undefined on entrance and exit of the PARALLEL construct.

2.3.2 The FIRSTPRIVATE clause

This is an extension of the PRIVATE clause in which any variables that existed before entry to the PARALLEL construct are initialised from those values and copied to each member of the team.

2.3.3 The THREADPRIVATE directive

The threadprivate directive makes named common blocks, in Fortran, and global variables, in C, private to a thread but global within that thread. The THREADPRIVATE directive must appear after the declaration of the common block or global variables. Each thread then has it's own copy of the named variables, but during sequential regions or MASTER regions the master threads copy is used. On entry to a PARALLEL construct the data in the THREADPRIVATE is undefined unless a COPYIN clause is present on the PARALLEL directive.

2.3.4 The COPYIN clause

The COPYIN clause applies only to common blocks or global variables declared in the THREADPRIVATE directive. The clause indicates that all of the threads private copies of the THREADPRIVATE data should be initialised with the master threads copy on entry to the PARALLEL region.

2.3.5 The FLUSH directive

The FLUSH directive indicates a point in the program where it must be ensured that all threads in the team have the same view of a list of particular variables. At this point all previous processes utilising these variables should have finished and any subsequent processes should not have begun. There is an implied FLUSH in several other directives, such as BARRIER and END PARALLEL.

2.4 Implementation of these Benchmarks

2.4.1 Fortran

The PRIVATE and FIRSTPRIVATE benchmarks were trivial to implement in exactly the same way as the previous benchmarks. The overhead was measured against the size of the array being

declared in the respective clauses. Fortran90 provides allocatable arrays which makes this task simple and efficient. Several issues arose in the coding of the THREADPRIVATE directive and COPYIN clause and also the FLUSH directive in Fortran.
Firstly with the declaration of the common block, it was noticed that there was a significant difference between declaring all of the arrays as members of one common block and declaring a separate common block for each array. It was decided that both warranted a place in the benchmark suite. In the case of the common blocks each array had to be declared individually as allocatable arrays are not permitted to be declared as part of a common block. This meant that the timing code had to be duplicated for each different array leading to a less streamlined, but still perfectly functional, code.
The FLUSH directive took a little more thought to measure. The directive will only flush variables which have been accessed before the flush occurs. Depending on whether a read, write or both have been performed the read or write buffers must be flushed. We also have to ensure that what is being flushed is sensible. For example, it would not be sensible to flush a private variable as only one thread has access and hence a consistent view of it. Equally we do not want all threads to write to and then flush the same variable as there would be non-deterministic overheads associated with false sharing. To overcome this we use a 2 dimensional array with one column associated with each thread and this column is written to in the delay subroutine and subsequently flushed (See figure 5).

```
         start  = getclock()
!$OMP PARALLEL PRIVATE(j,myid)
         myid = OMP_GET_THREAD_NUM()+1
         do j=1,innerreps
            call fldelay(dl,b(1,myid), sz)
!$OMP FLUSH(b)
         end do
!$OMP END PARALLEL
         time(k) = (getclock() - start) * 1.0e6 / dble (innerreps)
```

Figure 5: Implementation of the FLUSH benchmark

The `fldelay` subroutine writes a value to each element of the array `b` and also performs a delay loop.

2.4.2 C

The C version of the new benchmarks could not be coded as neatly as the Fortran code due to the way in which C deals with allocatable arrays. Dynamically allocating arrays in C can be easily accomplished using `malloc()`, but unfortunately this array can only be accessed using pointers. A problem arises here because when a pointer is declared in one of the clauses it takes the value of the pointer, i.e. the address, instead of the values of the variables to which the pointer refers. The pointer size is the same regardless of the size of the array and hence makes the test useless. The possibility of using linked lists was also investigated but it too depends on pointers. The only solution to this was to code all of the tests separately. The result was a bulkier and less elegant code than the Fortran version but this does not affect the results of the benchmarking. Implementation of the FLUSH directive in C also posed a problem as we

could not allocate the second dimension of an array as we did in Fortran. The solution which was used was, instead of using of assigning a column to each thread, to assign a section of a one dimensional array. This is equivalent to the Fortran version as both are simply contiguous portions of memory.

3 The Benchmarks

3.1 What was Benchmarked

Benchmarkng was performed on a Sun E3500, a Sun E6500 and a Sun E6000. The Sun E3500 contains 8 400 MHz Ultrasparc II processors, the Sun E6500 18 400 MHz Ultrasparc II processors and the Sun E6000 18 250 MHz Ultrasparc II processors.
Scheduling and synchronisation microbenchmarks were run on the E6500 using the KAI guidef90 compiler and the Sun Forte 6 Fortran compiler. The new benchmarks were run on all machines using the guidef90 and guidec compilers on the E6500 and E3500 and the Forte 6 compiler on the E6000.

3.2 Results

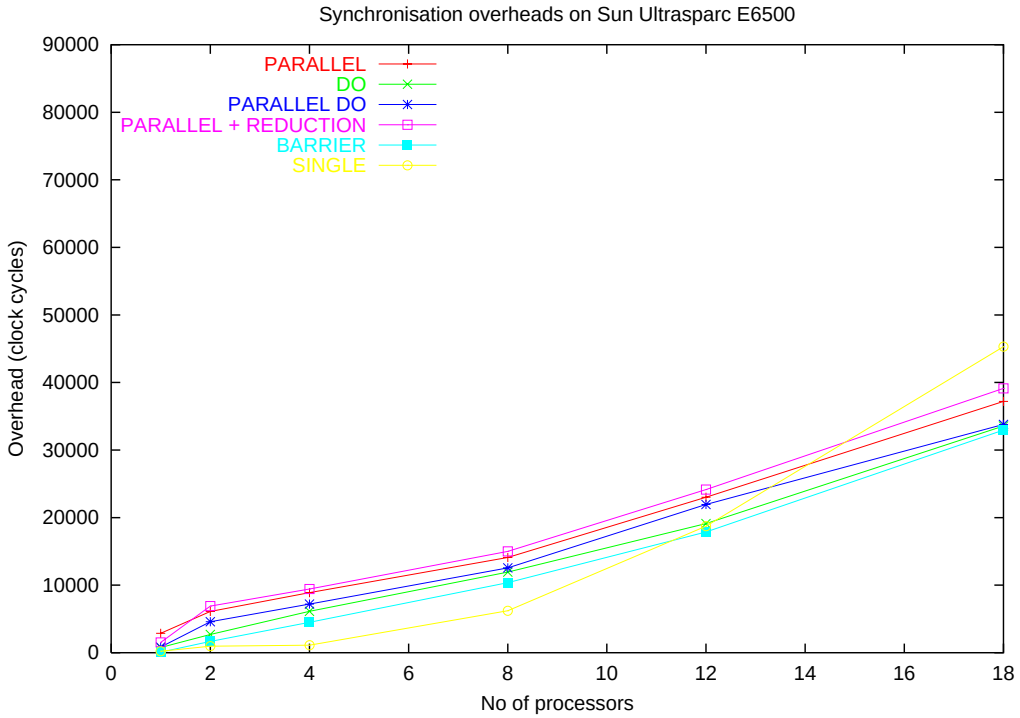


Figure 6: Synchronisation benchmark results from the E6500 using Forte compiler

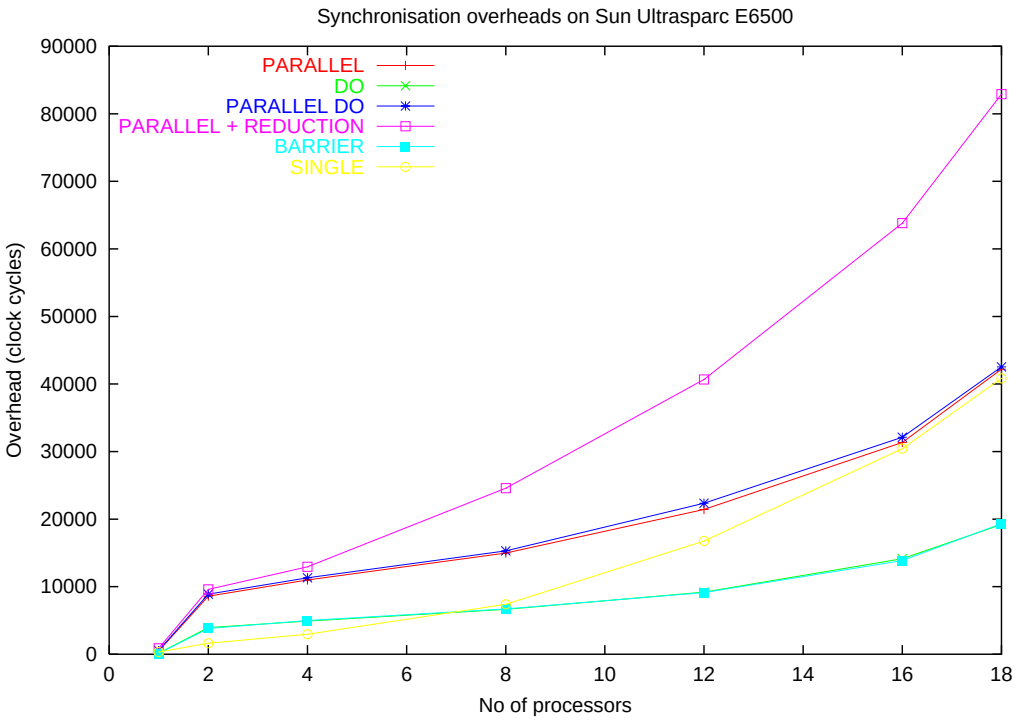


Figure 7: Synchronisation benchmark results from the E6500 using guidef90 compiler

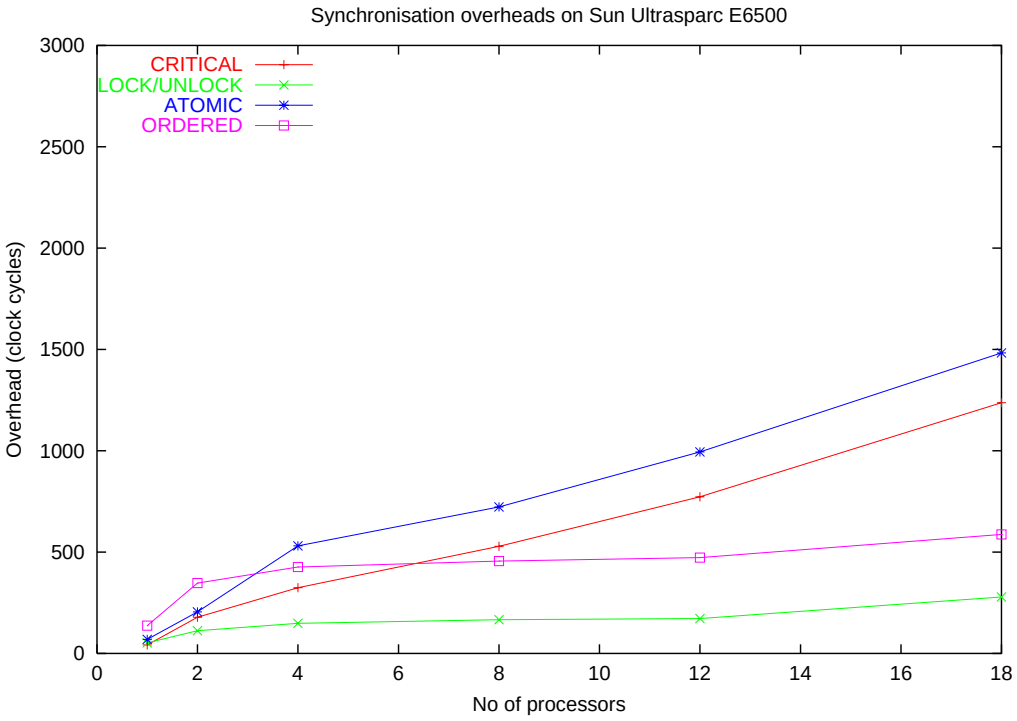


Figure 8: Synchronisation benchmark results from the E6500 using Forte compiler

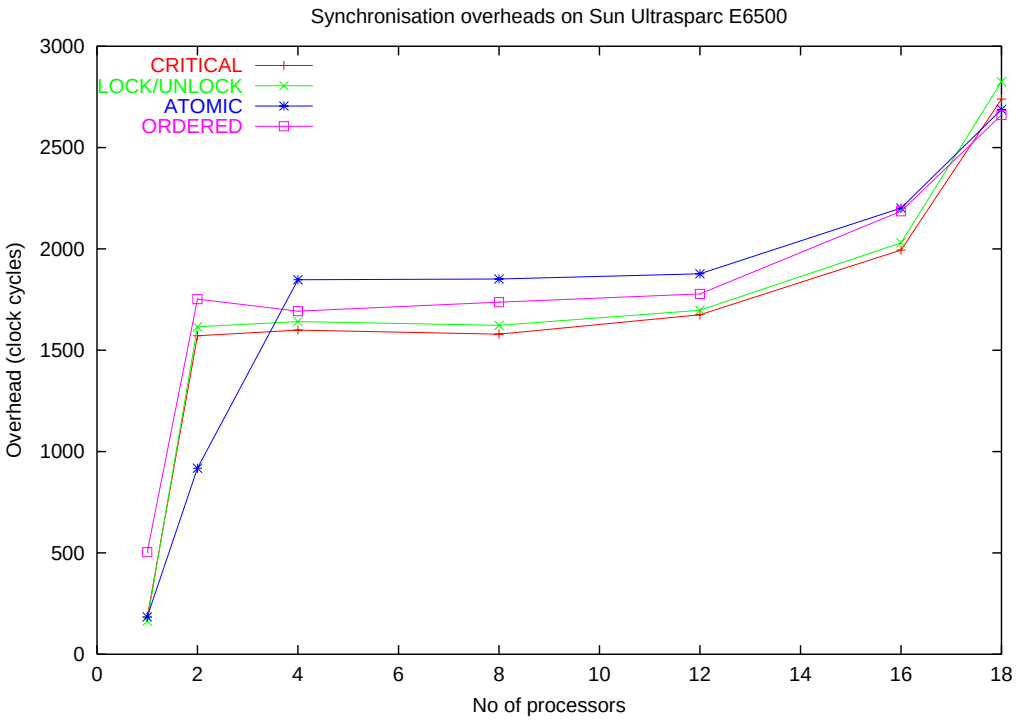


Figure 9: Synchronisation benchmark results from the E6500 using guidef90 compiler

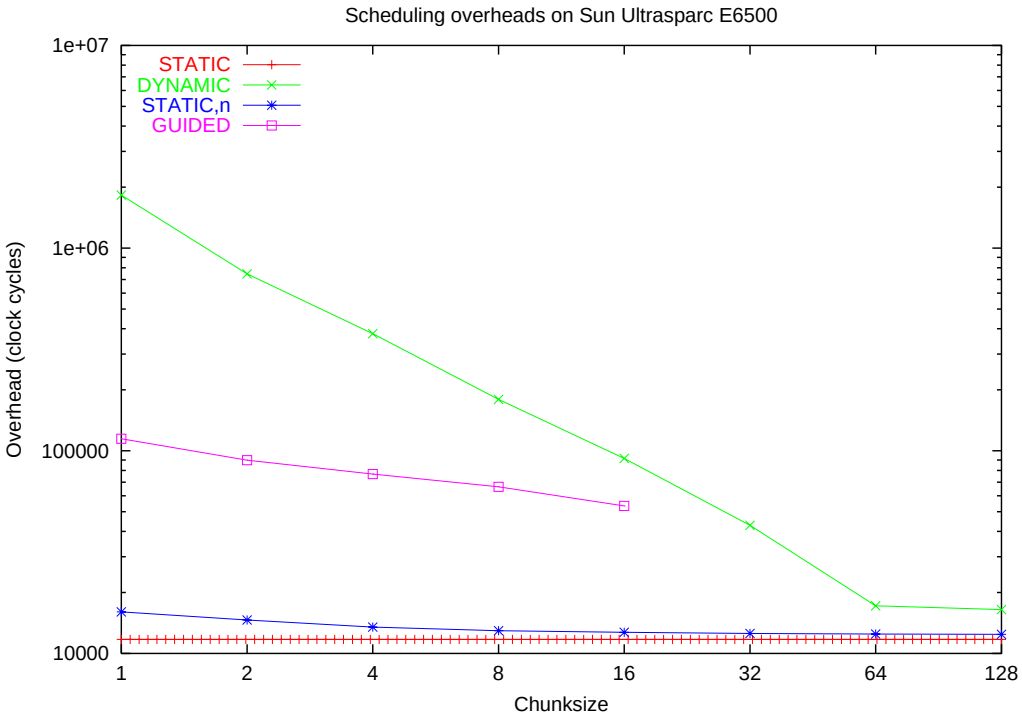


Figure 10: Scheduling benchmark results from the E6500 using Forte compiler

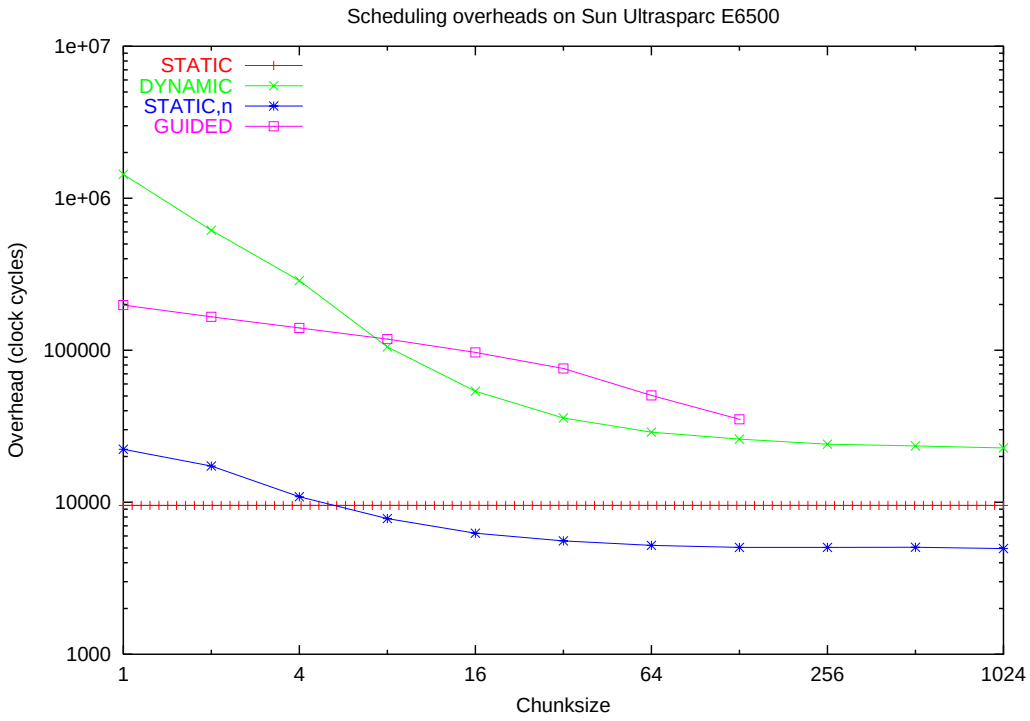


Figure 11: Scheduling benchmark results from the E6500 using guidef90 compiler

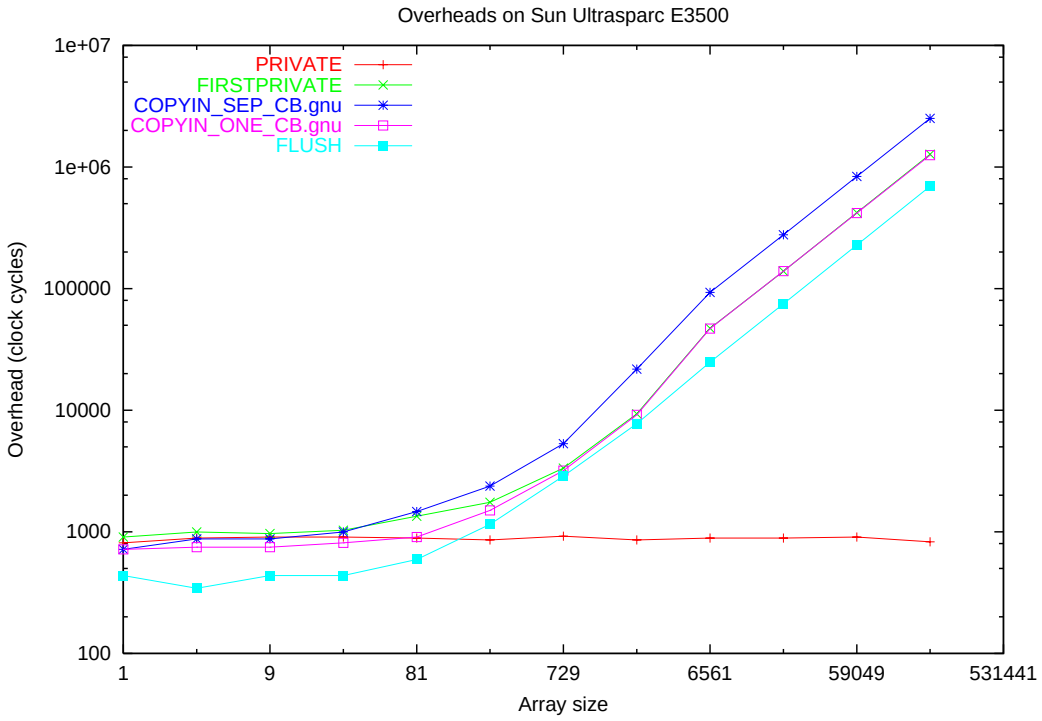


Figure 12: Clause Benchmarks for E3500 on one processor using guidef90

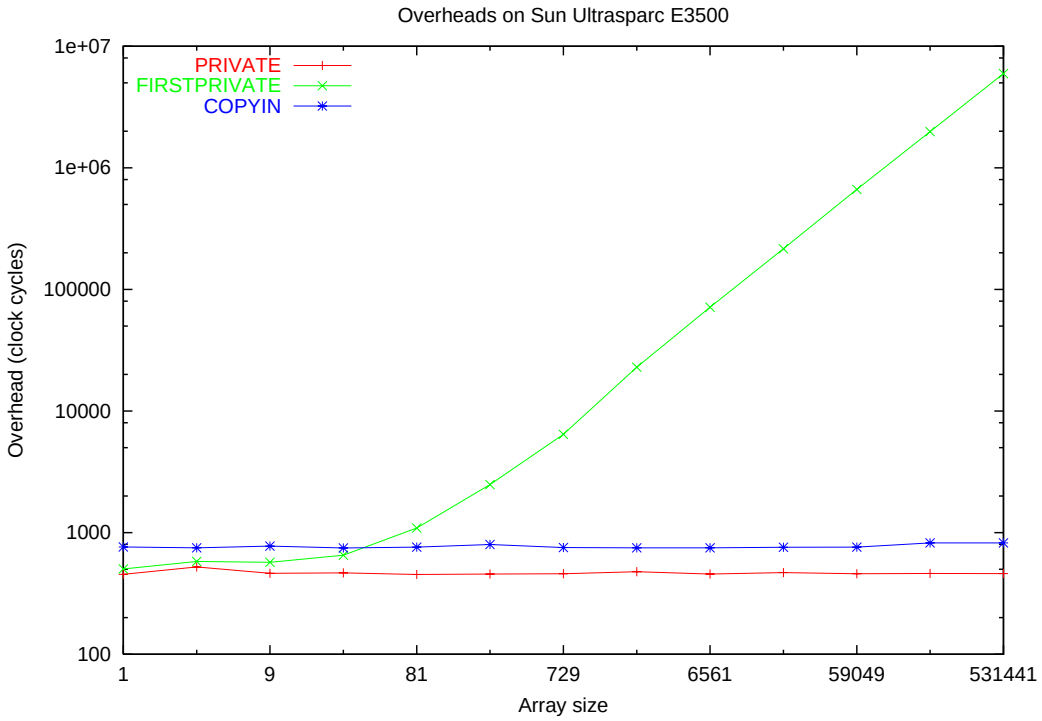


Figure 13: Clause Benchmarks for E3500 on one processor using guidec

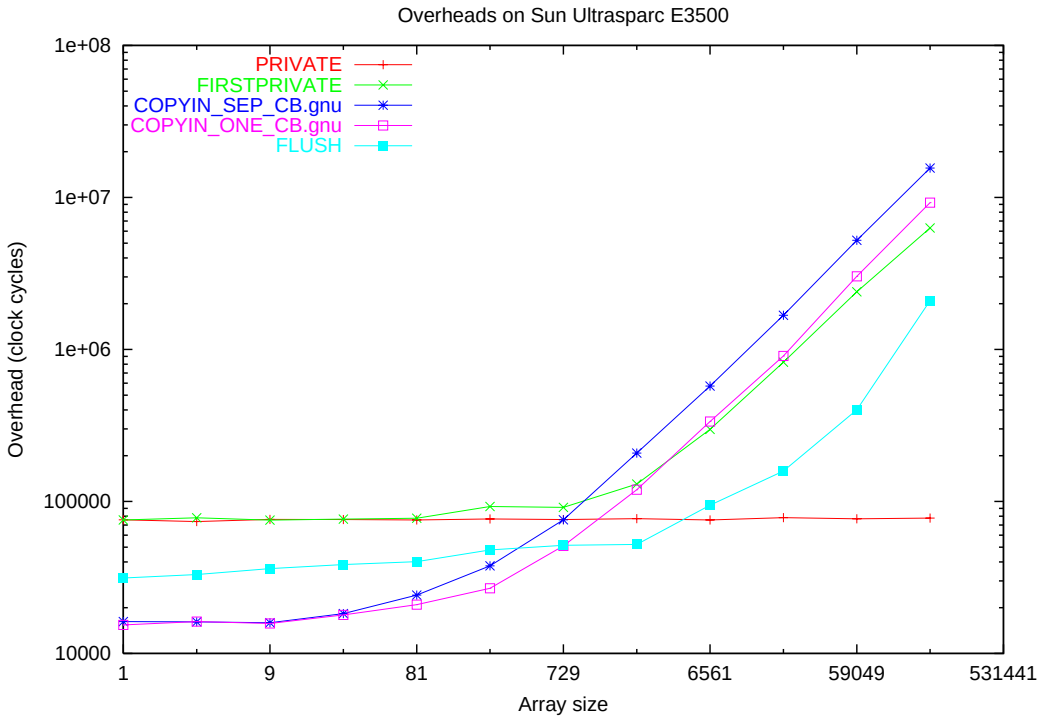


Figure 14: Clause Benchmarks for E3500 on eight processors using guidef90

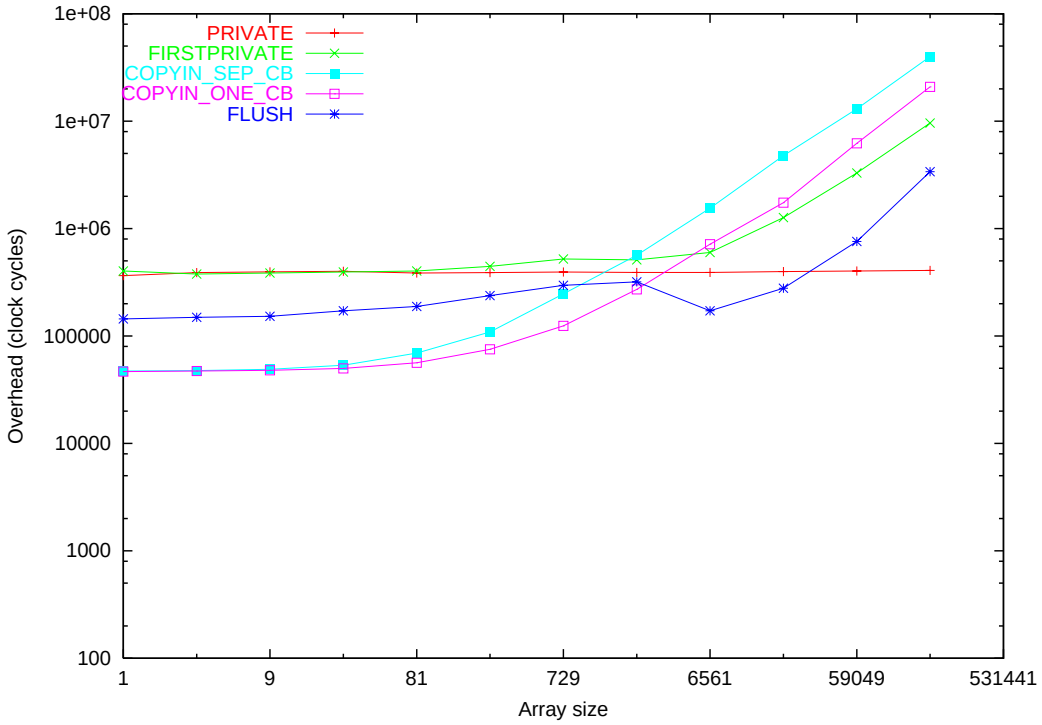


Figure 15: Clause Benchmarks for E6500 on sixteen processors using guidef90

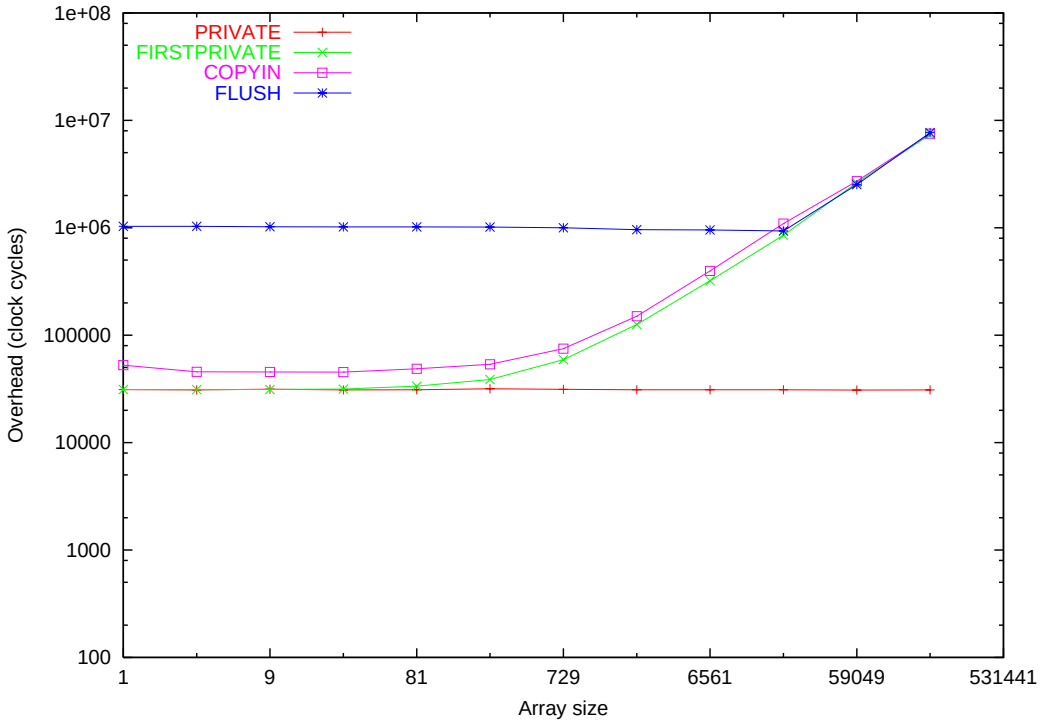


Figure 16: Clause Benchmarks for E6500 on sixteen processors using guidec

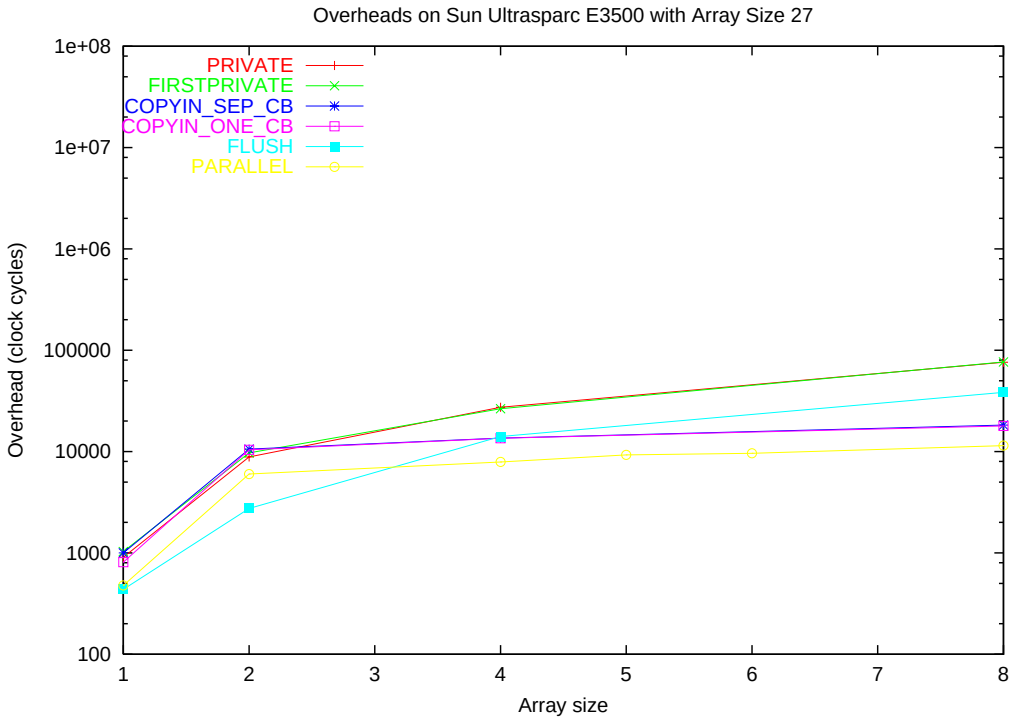


Figure 17: Clause Benchmarks for E3500 on varying numbers of processors using an array size of 27

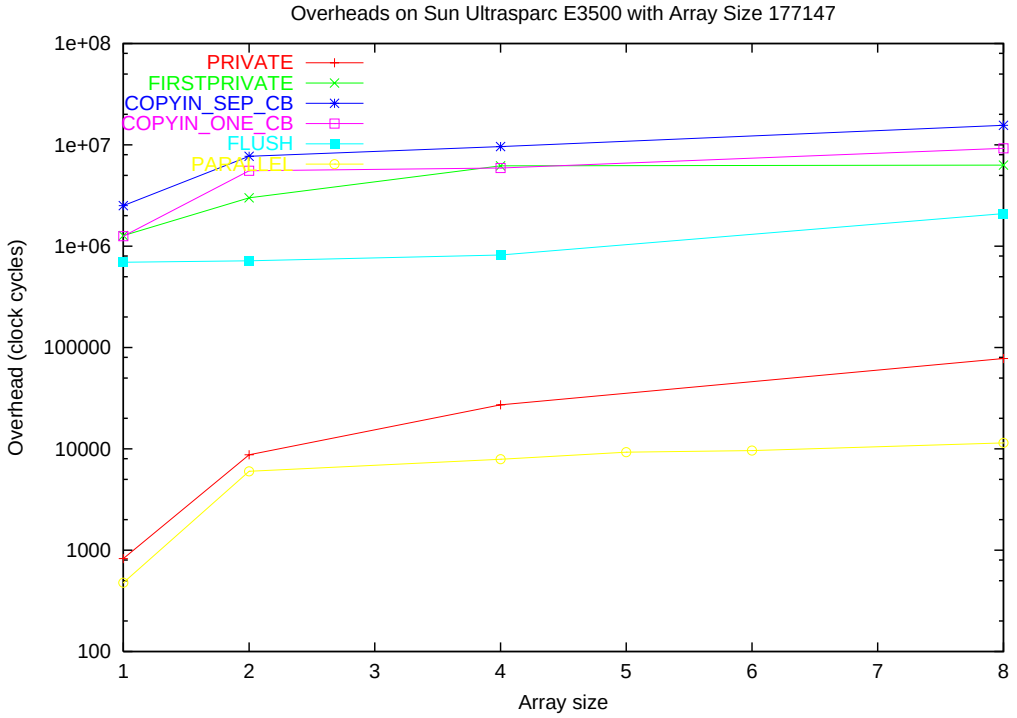


Figure 18: Clause Benchmarks for E3500 on varying numbers of processors using an array size of 177147 and guidef90

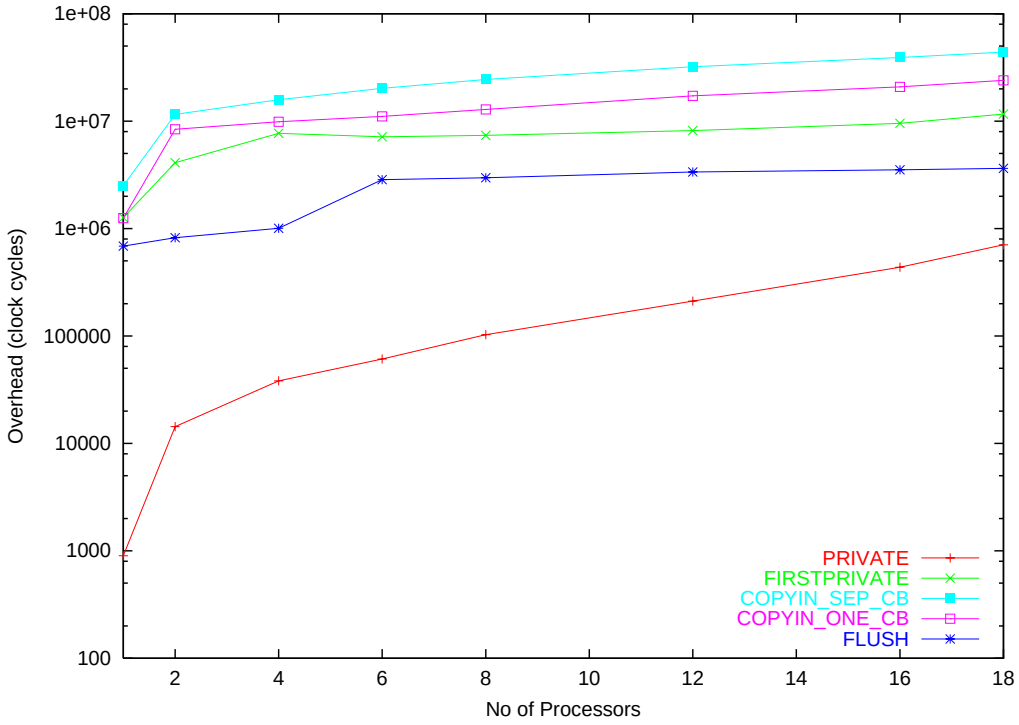


Figure 19: Clause Benchmarks for E6500 on varying numbers of processors using an array size of 177147 using guidef90

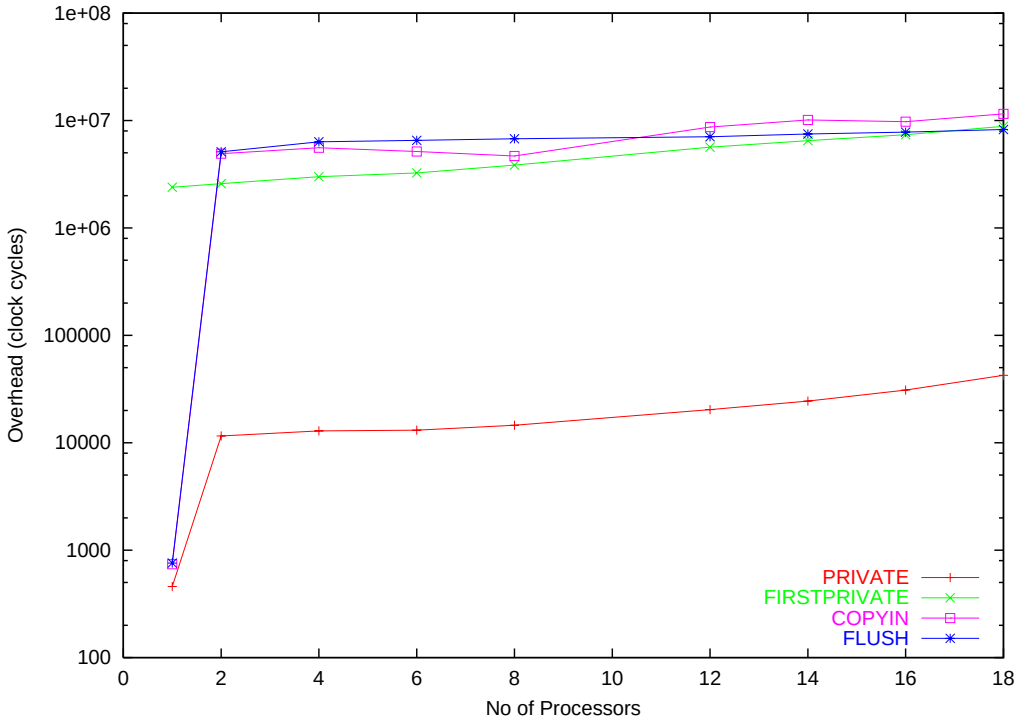


Figure 20: Clause Benchmarks for E6500 on varying numbers of processors using an array size of 177147 using guidec

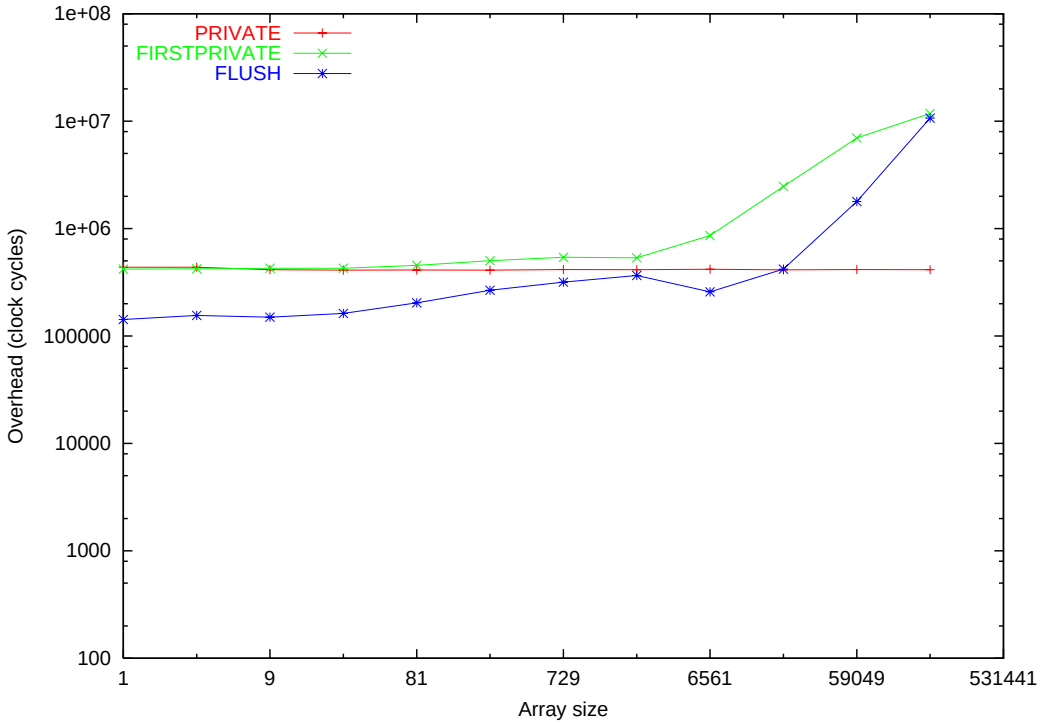


Figure 21: Clause Benchmarks for E6000 on 16 processors for varying array sizes using Forte 6

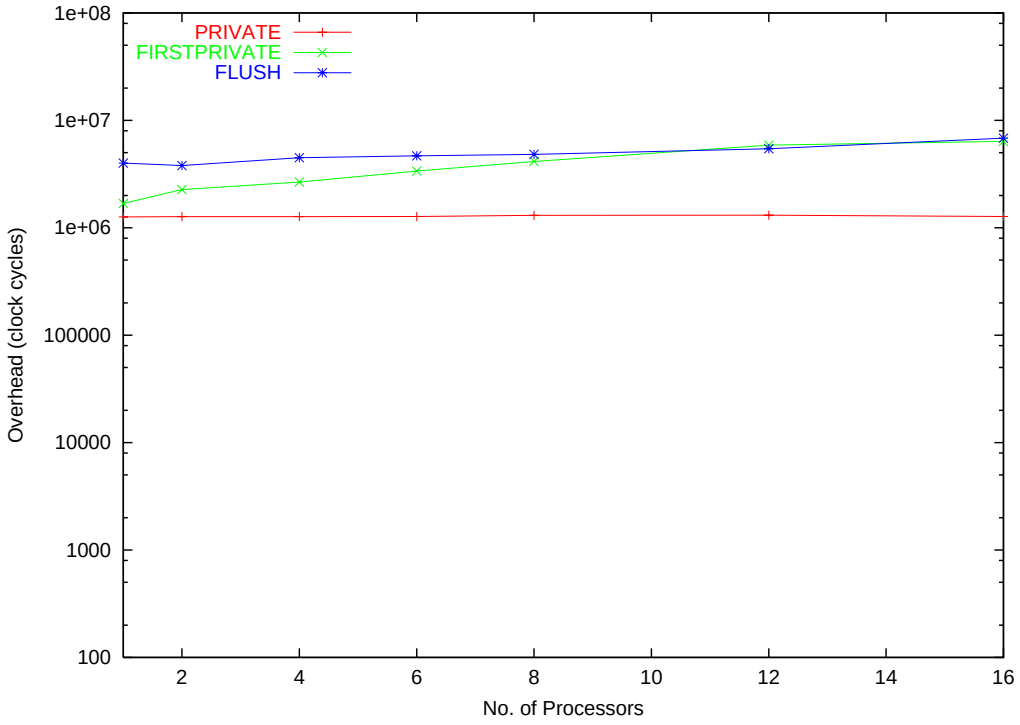


Figure 22: Clause Benchmarks for E6000 on varying numbers of processors using an array size of 177147 using Forte 6

3.3 Analysis

Comparison of the synchronisation results from the guide compiler and the Forte compiler (figures 7-11) reveal that in general the new Forte compiler is implementing the OpenMP directives better than the guide compiler. The guidef90 reduction operation is particularly inefficient and scales very badly. The rest of the directives in figures 6 and 7 perform marginally better using the Forte compiler, except the BARRIER and DO directive, on which guidef90 is significantly superior.

As for the second set of synchronisation directives the Forte compiler performs consistently better. There are some peculiarities with the results from the Forte implementation (figure 8), however. It is seen that LOCK/UNLOCK and CRITICAL perform significantly better than the ATOMIC directive. We would not expect this as it is trivial to implement an ATOMIC directive using either of these other ones, so one would expect that ATOMIC is open to more optimisation than either of them.

Using the guidef90 compiler resulted in a bad implementation of the STATIC scheduling. This was seen in the statistical data for this benchmark. There were large standard deviations over samples and runs showing how inconsistent the behaviour of this implementation is. The Forte compiler performed better in this benchmark and gave very consistent results over samples and runs.

In all three implementations of the array benchmarks (figures 12-22), the overhead of the `PRIVATE` clause does not depend on array size: this is to be expected as the new copies of the array can be stack allocated. In the KAI implementations, the increase in overhead with numbers of threads simply reflects the increasing cost of the barrier(s) associated with the `PARALLEL` directive. In the Sun Forte 6 implementation, the overhead is constant with number of threads, but significantly larger than in the KAI implementations.

In all implementations the overhead of the `FIRSTPRIVATE` clause is almost the same as that of the `PRIVATE` clause for small array sizes. For larger array sizes, the overhead increases as the cost of the copy becomes significant. As more threads are added the overhead tends to increase. A likely explanation for this is that all the threads are copying simultaneously, making additional demands on memory bandwidth.

In the guidef90 implementation, the `COPYIN` overhead is signifi cantly smaller than the `PRIVATE` overhead. This suggests that there is an additional fixed cost associated with the `PRIVATE` clause. This additional cost does not appear to be present in the guidec implementation. guidec also has optimisations for one thread (where a copy is not required) which are not present in guidef90. As with `FIRSTPRIVATE`, the cost of the copy becomes increasingly important as the array size gets larger. There is also an additional cost associated with having all the arrays in a single common block. Scalability with thread number is similar to the `FIRSTPRIVATE` clause. We were unable to run this benchmark on the Sun Forte 6 compiler.

For the `FLUSH` directive the two Fortran implementations show very similar behaviour, including a local maximum at an array size of 2187, which may be associated with the size of the level one cache. For larger array sizes, the cost inreases approximately linearly with array size. The guidec implementation shows a much larger, constant overhead for all but the largest array sizes.

4 Conclusions and Further Work

The results of this project have shown how the performance of OpenMP depends on both the system being used and the compiler implementation. Differences were seen between both all three compilers used with the Forte 6 compiler generally performing better than either of the KAI guide compilers, although guide did fare better on some of the benchmarks.
Development of microbenchmarks for OpenMP will continue for as long as OpenMP is still of interest as a method of parallelisation. New Fortran compilers supporting the new OpenMP 2.0 API will be released in the near future and will provide several new directives and clauses , all of which will require benchmarking.

References

[1] Randal L. Schwartz. *Learning Perl*. O'Reilly and Associates.

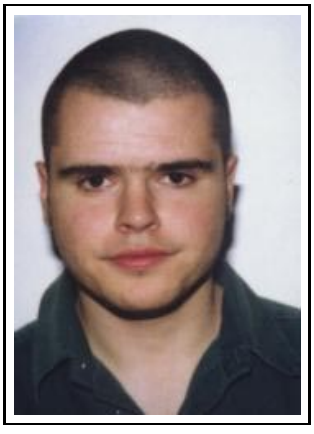
[2] Brian D. Hahn. *Fortran 90 for Scientists & Engineers*. Edward Arnold.

[3] Michael Metcalf and John Reid. *Fortran 90/95 explained*. Oxford University Press.

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

[5] OpenMP Fortran Application Program Interface Version 1.1

[6] OpenMP C and C++ Application Pragram Interface



Darragh O'Neill is in his final year of studying Computational Chemistry at Trinity College Dublin. His final year research project will involve studying Fischer-Tropsch synthesis over a {111} cobalt surface using Density Functional Theory to try to identify likely intermediates and a possible mechanism.

My supervisor Dr. Mark Bull is a principal consultant at Edinburgh Parallel Computing Centre.