Performance Comparisons of Basic OpenMP Constructs *

Achal Prabhakar[1,2], Vladimir Getov[1,3], Barbara Chapman[2]

[1] Parallel Architectures and Performance Team, CCS-3
Los Alamos National Laboratory, New Mexico, USA
[2] Department of Computer Science, University of Houston, Texas, USA
[3] School of Computer Science, University of Westminster, London, UK

Abstract. OpenMP has become the de-facto standard for shared memory parallel programming. The directive based nature of OpenMP allows incremental and portable developement of parallel application for a wide range of platforms. The fact that OpenMP is easy to use implies that a lot of details are hidden from the end user. Therefore, basic factors like the runtime system, compiler optimizations and other implementation specific issues can have a significant impact on the performance of an OpenMP application. Frequently, OpenMP constructs can have widely varying performance on different operating platforms and even with different compilers on the same machine. This makes it very important to have a comparative study of the low-level performance of individual OpenMP constructs. In this paper, we present an enhanced set of microbenchmarks for OpenMP derived from the EPCC benchmarks and based on the SKaMPI benchmarking framework. We describe the methodology of evaluation followed by details of some of the constructs and their performance measurement. Results from experiments conducted on the IBM SP3 and the SUN SunFire systems are presented for each construct.

1 Introduction

OpenMP is increasingly being thought of as the industry standard for parallel programming on shared memory systems. The directive based interface and incremental parallelism are two major features of OpenMP that add ease of use and convenience to parallelising existing and new applications. As in any other programming model, the overheads of supporting the facilities for parallel execution can have a considerable impact on the overall application's performance. If not used with caution, the overheads due to the OpenMP programming model itself can dominate the overall execution time of the application, voiding the parallelisation effort. Although much depends on the application's structure and algorithm, the low-level performance of basic constructs in OpenMP can have

* This work was partially supported by the U.S. Department of Energy through Los Alamos National Laboratory contract W-7405-ENG-36 and by the Los Alamos Computer Science Institute under grant LANL 03891-99-23.

a significant effect. The performance of OpenMP constructs will vary across different architectures and with different compilers on the same machine; the runtime system and the compiler are key factors in the performance of a parallel program. Often there is a significant difference in overheads incurred by similar constructs which suggest the use of one over the other. Thus information on performance penalties and scalability of OpenMP constructs is very useful, both for understanding the underlying implementation and architecture as well as for creating efficient parallel programs using OpenMP. In this paper, we describe a benchmark suite that is designed to evaluate the overheads incurred by OpenMP constructs and that adds some innovative measurements.
Benchmarking and evaluation of high performance computers and programming models is a well established field. In addition to producing benchmark codes, a number of individuals and groups in the past have come up with techniques and tools for making, analyzing and presenting benchmark results. Notably the GENESIS [5], ParkBench [6], EuroBen, NAS parallel benchmarks [1] and the EPCC low-level OpenMP benchmarks [3] have been widely used. The nature of measurements, the methodology and the design of benchmarks suites keeps changing with the advent of new machines and architectures. The goal of our project was to evaluate and compare the raw performance of the low-level facilities offered by OpenMP without going into application level details. In addition, we aimed to provide an efficient benchmark suite so that it can execute in the shortest time possible and still deliver fairly accurate results. Given the high cost of supercomputer time, we should not spend more of it than necessary on benchmarking. Also, we assumed that it is not possible to include every possible measurement and method, and thus the benchmark suite should be extensible and modular, allowing for the easy addition of new measurements and methodologies, possibly even measurements for future programming models. At the same time, no single programming model should be dependent on the availability of any other one, giving the user freedom to exclude any of them. For instance, if one is not interested in benchmarking MPI, it should be possible to compile and execute the suite on a platform that does not have MPI installed. The starting point for our OpenMP benchmarks was the popular suite of low-level benchmarks developed at EPCC to measure the overheads of OpenMP constructs.

The rest of the paper is organised as follows. Section 2 discusses the methodology and design of the benchmarks suite. Section 3 describes some of the important tests in the benchmarks suite. Space constraints do not permit us to cover all the contructs and the variants. Section 4 analyses the results from our experiments; it is followed by the conclusion in Section 5.

2 Evaluation Methodology

In general, the process of making a measurement, recording the results, collating them and writing to the output as well as accepting user input and constraints is similar for each measurement, regardless of the programming model used or the scope of the activity. The only difference in the whole process is the actual section of code that is executed and measured. As such it was obvious to consider the use of a framework that provides these basic facilities and allows a system for measurements routines to be “plugged” into the framework [7]. This way any new measurement will only require that the user writes the code that performs the operation to be measured and informs the framework of the new routine. This solves the problem of having an integrated and extensible suite.

SKaMPI, which stands for Special Karlsruhe MPI benchmarks [8, 9], from the University of Karlsruhe is one such benchmarking suite for MPI. SKaMPI provides a framework specifically for building extensible benchmarking suites. The important features of SKaMPI, such as automatic parameter refinement, automatic control of error, ability to merge the output results from multiple runs, and an intuitive user input mechanism, make it very useful for our work. The benchmark suite described here is based on the SKaMPI framework, with extensions made in both CASC at LLNL [4] and CCS-3 at LANL.

Since OpenMP [2] is directive based, it is not possible to directly measure the overheads imposed by each OpenMP construct. The methodology followed is to execute and measure a piece of code without an OpenMP directive (T_{ref}) and then measure the same code with the OpenMP directive of interest(T_m). The reference measurement is then subtracted from the OpenMP-enabled measurement to get the overhead for the particular construct (T_o).

$T_m = T_{ref} + T_o$, thus $T_o = T_m - T_{ref}$.

This is however not always necessary; for instance, the OpenMP `barrier` directive is an executable statement and can be measured without requiring a reference measurement. There are additional complications involved. As an example, consider the `lock-unlock` measurement. Should the measurement be the time taken by the master to perform the lock, or should it be the total time spent by all the threads? The decision in each such case is dependent on which method gives the most relevant information as regards the effect of the overheads on the total execution time of an application.

3 Basic constructs and evaluation results

3.1 Lock - Unlock

The OpenMP `lock-unlock` primitive provide fine grained control over synchronization between threads. As an alternative to using `critical` regions, there are instances when the explicit `lock-unlock` is advantageous over using `critical` regions. While the performance difference between `critical` and `lock-unlock` depends a lot on the underlying runtime system, in general, the `lock-unlock` combination is expected to incur less overhead than `critical`. The conventional approach to measure the lock/unlock overhead is to have all the threads make repeated invocations of lock followed by unlock and averaging the result to get a fairly accurate estimate of the overhead. However, if the runtime system does not guarantee lock-fairness, it may be possible that the thread which is timing

the loop can grab the lock in every iteration without giving a chance to other threads. In such a case, the results would reflect the `lock-unlock` overhead on a single thread, which is clearly not correct. In order to circumvent the fairness problem, our approach is to ensure that in every iteration, each thread gets an opportunity to grab the lock at-most once. This can be accomplished by making sure that all threads wait after each iteration for other threads. This is achieved by placing a OpenMP `barrier` after the unlock operation. The sequence of events is illustrated in Fig 1. Although it is possible that the same thread can grab the lock at the next iteration, the probability that the timing will be offset by more than the time for a single thread lock acquire is very low.
The reference time T_{ref} is calculated by the code in Program 1. Program 2 shows the pseudo code for the actual `lock-unlock` time T_m measurement.

```
for ( 1 to N )
  do_some_work
#pragma omp barrier
end for
```

Program 1.`Lock-unlock` reference time measurement.

```
for ( 1 to N )
  omp_set_lock()
   do_some_work
  omp_unset_lock()
#pragma omp barrier
end for
```

Program 2.`Lock-unlock` measurement (no assumptions of lock-fairness)

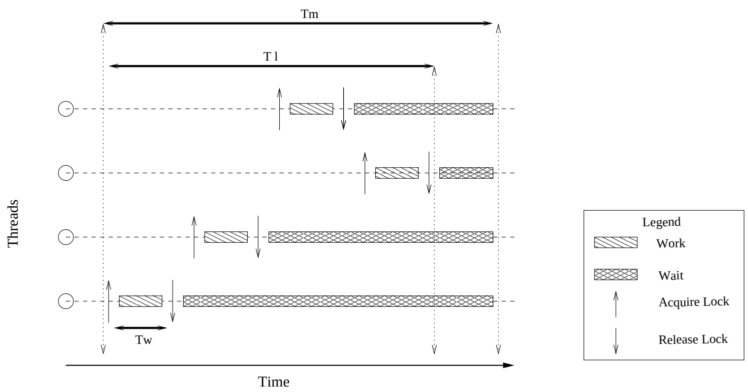

Figure 1. Timing Diagram for the Lock/Unlock Measurement

It is subtracted from the reference time to offset the `barrier` cost and dummy work cost. The measured quantity now is the sum of times that each thread had spent in performing the `lock-unlock` operation, regardless of whether the runtime system ensures lock fairness or not. This gives an accurate estimate of the effect that the number of threads has on the lock performance.

$T_m = T_b + T_l + T_w$ or $T_m = T_{ref} + T_l$

where $T_{ref} = T_b + T_w$ Thus $T_l = T_m - T_{ref}$

3.2 Critical

The OpenMP `critical` directive is similar to the `lock-unlock` primitive: both guarantee exclusivity to a thread in a region. A thread waits at the beginning of a `critical` region until no other thread is executing the same `critical` region. In principal a `critical` region is equivalent to enclosing the structured block with calls to `omp_set_lock` and `omp_unset_lock`, however the performance difference is highly dependent on the underlying runtime implementation of both `critical` and `lock-unlock` used by the operating platform. In general, the `lock-unlock` primitives are expected to provide better performance than the `critical` directive, but the actual difference in performance and scalability depends on the implementation and might vary with the number of threads.
The measurement approach is similar to that for the `lock-unlock` primitives. A dummy work routine enclosed in a `critical` region is called repeatedly by all the threads in the team with the master thread responsible for timing the loop. The final result is averaged over the number of iterations to get an estimate of the overhead for a single `critical` region. The pseudo code for the measurement routine is shown in Program 3.
Since the compiler decides how to translate a `critical` directive into more basic

```
for ( 1 to N )
  #pragma omp critical
      do_some_work()
end for
```

Program 3. Pseudo code for measuring overhead of `Critical` directive assuming lock fairness.

primitives, the behavior of the `critical` directive with respect to lock-fairness is implementation dependent. In case the runtime system does not guarantee that a thread can not acquire a mutex repeatedly, the code segment in program 3 will not yield the correct overhead. Like the `lock-unlock` measurement, the worst case scenario would be that the master thread, doing the timing, acquires the mutex for all iterations of the loop. The result would be the overhead for executing a `critical` region with only one thread.
Our solution is to force every thread to wait at the end of the `critical` region by having an OpenMP `barrier` directive. This guarantees that for any single iteration, all the threads would have executed the `critical` region at least once. The timing diagram for this measurement is similar to the `lock-unlock` (Fig. 1.). It should be noted that the reference measurement is done over the same number of threads as for the actual measurement. The result from the reference measurement for n threads is subtracted from the actual measurement for the

same number of threads. Program 4 and 5 show the pseudo code for measuring the `critical` overhead and the reference time respectively.

```
for ( 1 to N )
#pragma omp critical
  do_some_work();
#pragma omp barrier
end for
```

Program 4.`Critical` directive measurement (no assumption of lock fairness.)

```
for ( 1 to N )
  do_some_work()
#pragma omp barrier
end for
```

Program 5. Reference measurement for `critical` directive.

3.3 Barrier

The `barrier` is a global synchronization directive. Each thread waits at the `barrier` until all the other threads have arrived at it. Once all threads have reached the `barrier`, each thread continues to execute the code after the `barrier` in parallel. There is an implied flush of shared variables for the `barrier` directive.
The `barrier` measurement is similar to the `atomic` directive measurement. The work routine is placed in the `barrier` scope and is executed repeatedly by all the threads (see Program 7). The measured time is for the execution of the n thread `barrier` plus the time taken by the work routine. It should be noted that since the `barrier` has an implied flush, its overhead is automatically included in the `barrier` overhead. The reference measurement calculates the execution time of the work routine for a single thread, which is then subtracted from the `barrier` measurement to get the overhead for the `barrier` directive (see Program 6).

```
for ( 1 to N )
  do_some_work()
end for
```

Program 6.`Barrier` directive reference measurement.

```
for ( 1 to N )
#pragma omp barrier
  do_some_work()
end for
```

Program 7.`Barrier` directive measurement.

3.4 Single

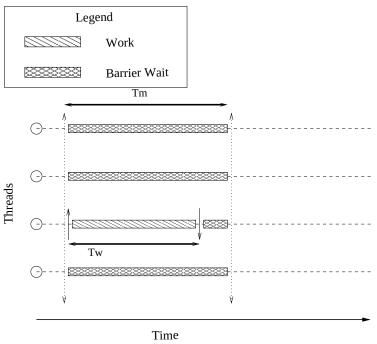

Figure 2. OpenMP `Single`

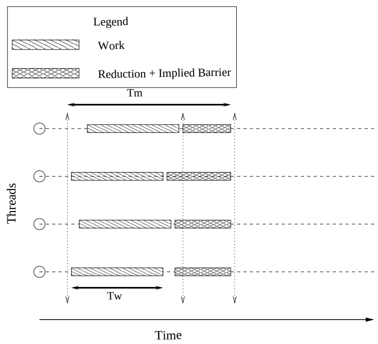

Figure 3. OpenMP `For-Reduction`

The OpenMP `single` directive is another one of the many synchronization directives available; it ensures that only a single thread will execute the block of code in the scope of the directive. The executing thread need not necessarily be the master thread. There is an implied `barrier` after the `single` directive. The approach for measuring the overhead of the `single` directive is illustrated in the timing diagram in Figure 2. The block of code containing the `single` directive and the work routine is executed for a number of times. The master thread performs the timing, which is averaged over the number of iterations performed for the loop (see pseudo code in Program 8). The reference time calculation is similar to that for the `barrier` directive (Prog. 6).

```
for ( 1 to N )
#pragma omp single
  do_some_work()
end for
```

Program 8.`Single` directive measurement.

```
for ( 1 to N )
#pragma omp for reduction(+:red_var)
 for( 1 to N * num_threads)
   red_var += do_some_work()
 end for
end for
```

Program 9.`reduction` clause with for directive measurement.

3.5 Reduction

A `reduction` clause is used in conjunction with work-sharing constructs to perform reduction on the shared variable according to the specified reduction operator. Since this clause is used in conjunction with the work sharing constructs, there is always an implied `barrier` unless the enclosing construct has a nowait clause, in which case the reduction computation is not guaranteed until a `barrier` is encountered. The measurement of the `reduction` clause is important as it is one of the most frequent operations in any parallel program. Also the use of the `reduction` clause, rather than a manual implementation of a reduction, is usually expected to provide opportunities to the compiler for optimization.
Since it is always used in conjunction with a work-sharing construct, the reference time measurement for this clause has to estimate the overhead of the enclosing construct. Consider the case of `reduction` with the OpenMP for directive. We provide tests for measuring the overhead of the `omp for reduction` as well as the overhead of just introducing the `reduction` clause in the `omp for` directive. Program 9 shows the pseudo code for measuring the overhead of the `omp for reduction` directive. In this case the reference time is simply the overhead of the loop and the work routine (Program 10). In order to measure the overhead of the `reduction` clause itself, the reference routine has to incorporate the `omp for` directive also, as shown in Program 11.

```
for( 1 to N )
  red_var += do_some_work()
end for
```

Program 10. `Reduction` clause reference measurement.

```
#pragma omp for
for( 1 to N * Num_threads )
     red_var += do_some_work()
end for
```

Program 11. `Reduction` clause without for directive reference measurement.

3.6 Parallel & Parallel-for

The `parallel` construct is the only way to start a parallel region in OpenMP. When a thread encounters a parallel region, it creates a new team of threads and itself becomes the master of the team. The code immediately following the `parallel` construct and within the dynamic scope of the directive is executed in parallel by all the threads in the team, including the thread that created the team. There is an implied `barrier` at the end of the `parallel` clause.Measurement of the `parallel` region is simpler than for the other directives. The measurement routine consists of a work routine within a `parallel` region which is repeatedly invoked and the final result is averaged by the master thread. The reference time is the execution time of the work routine for a single thread.

`Parallel for` is the compound parallel work share directive. It specifies a parallel region with only a single work sharing directive. When a thread encounters a `parallel-for` directive, it creates a new team of threads. The work is shared among the threads according to the schedule policy in effect. After completion of the work all the threads wait at the implied `barrier`.

Measuring the overhead of the `parallel for` directive involves repeatedly timing the work routine contained within the `parallel for` directive (Prog.12) . The reference time is the time taken for the execution of the work routine for a single thread (Prog.13).

```
for ( 1 to N )
#pragma omp parallel for
 for ( 1 to N * Num_threads )
   do_some_work()
 end for
end for
```

Program 12.`Parallel-for` directive measurement.

```
for ( 1 to N )
  do_some_work()
end for
```

Program 13.`Parallel-for` directive reference measurement.

4 Presentation and Discussion of Results

The experiments were conducted on an IBM SP3 8-way SMP with 375 MHz Power3 processors, using the VisualAge C++ Professional / C for AIX Compiler, Version 5 and on a SUN SunFire 6800 24-way UltraSPARC-III, 750 MHz using the Sun WorkShop 6 update 2 C 5.3 2001/05/15 compiler. All the times mentioned in the results are in microseconds.

On the IBM SP3, the `lock-unlock` overheads increase rapidly from 5 to 6 processors (Fig. 4). Strangely, on the SUN the overheads of the `critical` directive (Fig. 5) are lower than those for the `lock-unlock` primitives, even though there is an implied flush at the end of the `critical` region, suggesting that it will incur additional overheads. The difference can be attributed to better optimization in the implementation of the `critical` directive. On the IBM, the cost of `lock-unlock` is less than that of the `critical` directive, with the gap increasing with increasing number of processors, most probably due to increasing

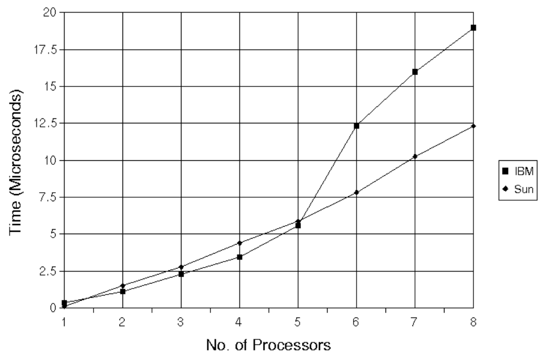

Figure 4. `Lock-unlock` overhead on the IBM SP3 and SUN SunFire

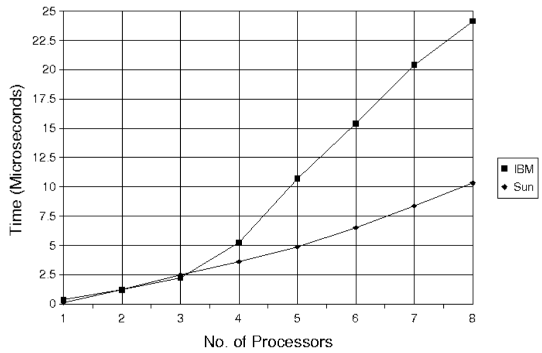

Figure 5. `Critical` overhead on the IBM SP3 and SUN SunFire

cost of the implied flush. The lower cost of `critical` on the SUN SunFire and the `lock-unlock` on the IBM SP3 suggest alternate implementations and / or difference in cost of the flush on the two systems. However, for both constructs the SUN machine shows better scalability than the IBM.

The `barrier` implementation on the SUN appears to be quite efficient and scales well with an increasing number of processors. The overhead of a `barrier` is very small, around 3 microseconds for 8 processors (Fig. 6). Markedly different is the IBM implementation which is costly and also scales poorly. The poor `barrier` performance on the IBM may be responsible for the low performance of all other `barrier` synchronized directives, such as the `parallel`, `parallel-for`, and `reduction` constructs. Once again, we notice a sharp increase for 2 and 8 processors on the IBM.

For fewer than 7 processors, the overhead of the `single` directive (Fig. 7) is lower than the overhead of the `barrier` on the IBM SP3, even though there is an implied `barrier` at the end of a `single` directive. However, with more than 7 processors the cost is higher than that of `barrier`, which is the expected

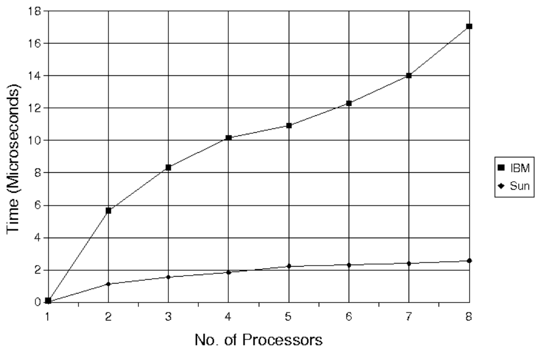

Figure 6. Overhead of the `barrier` directive on the IBM SP3 and SUN SunFire

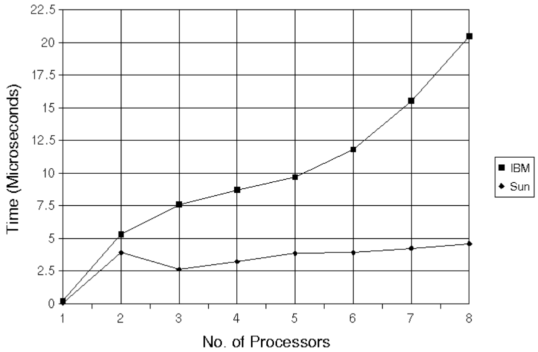

Figure 7. Overhead of the `single` directive on the IBM SP3 and SUN SunFire

behaviour. A reason for the discrepancy could be the overlap between the first phase (check-in) of the `barrier` with the computations involved with the `single` directive. On the SUN SunFire however, the cost of `single` is always greater than that of the `barrier` and closely follows the `barrier` curve, suggesting that most of the cost of the `single` directive is in the implied `barrier`. There is a sharp increase in the overhead of the `single` directive for 2 processors. We noticed this increase for a majority of runs performed and decided to keep it as being more statistically representative. However, this might have been due to disturbances when the experiments were conducted. In general, the cost of `single` on the SUN appears to be scalable and as efficient as the implied `barrier`. On the IBM though, the cost of `single` increases rapidly with an increasing number of processors, with a sharp increase from 1 to 2 and 6 to 8 processors.

The `reduction` operation is extremely expensive on the IBM SP3, with the overhead being as high as 72 microseconds for 8 processors (Fig. 8). The poor scalability suggests an implementation that is based on atomic updates of the shared reduction variable. Also the costly `barrier` adds to the overheads of

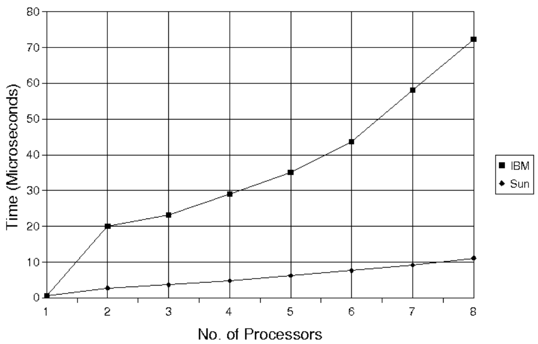

Figure 8. Overhead of the `for-reduction` directive on the IBM SP3 and SUN SunFire

`reduction`. The SUN SunFire shows a much better performance, both in terms of overhead and scalability. The slope is very similar to that of the `barrier` overhead curve, pointing towards a tree based reduction algorithm that is combined with the execution of the `barrier`.

Overheads for the `parallel` directive on the IBM SP3 increase sharply for 2 processors, remain almost constant up to 5 processors and then increase rapidly for 7 and 8 processors (Fig. 9). Since there is an implied `barrier` at the end of the `parallel` region, subtracting the cost of the `barrier` from the overhead of the `parallel` region, gives us an estimate of the overhead due to `parallel` region creation and management. This simple calculation shows a large overhead for 2 processors which reduces up to 6 processors and rises again for 7 processors. On the SUN SunFire machine, a `parallel` region incurs lower overheads and they scale better than on the IBM SP3. However the overhead is always more than twice the cost of the `barrier`, suggesting that on the SUN there is a complete `barrier` at the start as well as the end of the `parallel` region. Unlike the IBM, subtracting 2 times the cost of the `barrier` from the `parallel` region overhead yields a more uniform curve.

5 Conclusions

The importance of benchmarks in analysing and tuning the performance of parallel programs cannot be overemphasized. We have presented an enhanced and exhaustive set of benchmarks for evaluating the performance of OpenMP constructs that used the work carried out at EPCC on OpenMP overheads measurement as a starting point. The implementation of the benchmarks is based on the extensible SKaMPI framework and allows great flexibility for tailoring the benchmark both externally and internally. We have shown the results for two different machines, the IBM SP3 and the SUN SunFire 6800. Critical discussion of the results has been presented.

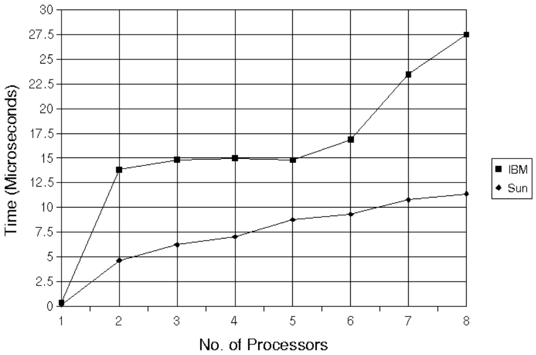

Figure 9. Overhead of the `parallel` directive on the IBM SP3 and SUN SunFire

6 Acknowledgements

We convey our sincere thanks to the staff at the San Diego Supercomputing Center and at the University of Ulm in Germany for allowing use of their computing facilities and for the technical support while running the experiments presented in this paper.

References

[1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[2] OpenMP Architecture Review Board. OpenMP Fortran Application Program Interface, Version 1.1, November 1999.

[3] J. M. Bull. Measuring Synchronisation and Scheduling Overheads in OpenMP. In *EWOMP '99, Lund, Sep., 1999.*, 1999.

[4] Bronis R. de Supinski and John May. Benchmarking Pthreads Performance. In *Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '99)*, June 1999.

[5] A. J. G. Hey. The genesis distributed memory benchmarks. *Parallel Computing*, 17(10–11):1275–1283, 1991.

[6] R. Hockney and M. Berry. Public international benchmarks for parallel computers report. Technical report, Parkbench Committee, 1994.

[7] R. W. Hockney and V. S. Getov. Low-level benchmarking: Performance profiles. In *Proc. Euromicro Workshop on PDP,IEEE CS Press*, Jan. 1998.

[8] R. Reussner, P. Sanders, L. Prechelt, and M. Mueller. SKaMPI: A detailed accurate MPI benchmark. In *Springer Lecture Notes in Computer Science.*, volume 1497, pages 52–59, 1998.

[9] R.H. Reussner. Skalib: Skampi as a library - technical reference manual. Technical report, Department of Informatic, University of Karlsruhe, Germany, 1999.