Extending the EPCC OpenMP Microbenchmarks for OpenMP 3.0

Joseph Harkness

Supervisor: Dr. Mark Bull

August 27th, 2010

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2010
Acknowledgements

Many thanks to Dr. Mark Bull for providing guidance and insight while supervising this project. Thank you also to all of the EPCC teaching staff on the MSc course.
Abstract

Version 2.0 of the EPCC OpenMP Microbenchmark suite has been updated to include the latest features of the specification in OpenMP 3.0. Both the newly developed and pre-existing benchmarks were then run on a variety of different hardware platforms, including the latest upgrade to the HECToR supercomputer, with a number of different compilers. Differences in compiler implementations of the specification have successfully been highlighted, suggesting some areas for improvement, while providing a survey of the performance of a large range of OpenMP directives. The effects of the different hardware architectures were also observed, often with strong performance gains to be made with larger numbers of cores per processor.
## Contents

Chapter 1  Introduction ........................................................................................................... 1

Chapter 2  Background ........................................................................................................ 3

2.1 OpenMP ......................................................................................................................... 3

2.2 OpenMP 3.0 ................................................................................................................... 5

2.2.1 Example Programs ...................................................................................................... 6

2.3 Microbenchmarking ...................................................................................................... 9

2.3.1 Caveats ....................................................................................................................... 9

2.3.2 Examples ................................................................................................................... 10

2.4 EPCC OpenMP Microbenchmarks V2.0 ....................................................................... 10

2.4.1 Methodology ............................................................................................................. 11

2.5 List of Benchmarks ...................................................................................................... 12

2.5.1 Scheduling ................................................................................................................ 12

2.5.2 Synchronisation ....................................................................................................... 13

Chapter 3  Implementation and Test Platforms ................................................................. 16

3.1 Additional Benchmarks Implemented ............................................................................ 16

3.1.1 Parallel and Serial Task Generation ......................................................................... 16

3.1.2 Parallel Generation + Barrier/Taskwait .................................................................... 17

3.1.3 Tree Structured Task Generation ............................................................................. 18

3.1.4 Immediate Execution ............................................................................................... 21

3.2 Note on Memory Profiling ............................................................................................ 22

3.3 Hardware Platforms .................................................................................................... 23
List of Figures

Figure 1 – Diagram of the tree created by the first tree-based task generation microbenchmark. ................................................................. 19
Figure 2 – Diagram of the tree generated by the second tree-based task generation microbenchmark. .................................................. 20
Figure 3 – Magny-Cours architecture diagram. ................................................................. 25
Figure 4 – PGI scheduling overheads (Ness, 16 threads, pgcc 10.0-0) ............................... 33
Figure 5 – Intel scheduling overheads (Ness, 16 threads, icc 11.1). ............................... 33
Figure 6 – GNU scheduling overheads (Ness, 16 threads, gcc 4.4.3). ............................. 34
Figure 7 – PGI scheduling overheads (Ness-GPU, 8 threads, pgcc 10.0-0). ...................... 34
Figure 8 – Intel scheduling overheads (Ness-GPU, 8 threads, icc 11.1). ......................... 35
Figure 9 – GNU scheduling overheads (Ness-GPU, 8 threads, gcc 4.4.3). ....................... 35
Figure 10 – PGI scheduling overheads (XT4, 4 threads, pgcc 10.3-0) ............................. 36
Figure 11 – Cray scheduling overheads (XT4, 4 threads, craycc 7.1.6). ......................... 36
Figure 12 – Pathscale scheduling overheads (XT4, 4 threads, pathcc 3.2.99). ................. 37
Figure 13 – GNU scheduling overheads (XT4, 4 threads, gcc 4.4.3). .............................. 37
Figure 14 – PGI scheduling overheads (XT6, 24 threads, pgcc 10.3-0). ............................ 38
Figure 15 – Cray scheduling overheads (XT6, 24 threads, craycc 7.2.1.105). ................. 38
Figure 16 – Pathscale scheduling overheads (XT6, 24 threads, pathcc 3.2.99). ............... 39
Figure 17 – GNU scheduling overheads (XT6, 24 threads, gcc 4.4.3). ........................... 39
Figure 18 – PGI synchronisation overheads (Ness, pgcc 10.0-0). .................................. 40
Figure 19 – Intel synchronisation overheads (Ness, icc 11.1). ....................................... 41
Figure 20 – GNU synchronisation overheads (Ness, gcc 4.4.3). ................................... 41
Figure 21 – PGI locking overheads (Ness, pgcc 10.0-0). .............................................. 42
Figure 22 – Intel locking overheads (Ness, icc 11.1). ................................................... 42
Figure 23 – GNU locking overheads (Ness, gcc 4.4.3). .................................................. 43
Figure 24 – PGI synchronisation overheads (Ness-GPU, pgcc 10.0-0). ......................... 43
Figure 25 – Intel synchronisation overheads (Ness-GPU, icc 11.1). .............................. 44
Figure 26 – GNU synchronisation overheads (Ness-GPU, gcc 4.4.3). ............................ 44
Figure 27 – PGI locking overheads (Ness-GPU, pgcc 10.0-0). ...................................... 45
Figure 28 – Intel locking overheads (Ness-GPU, icc 11.1). ........................................... 45
Figure 29 – GNU locking overheads (Ness-GPU, gcc 4.4.3). ........................................ 46
Figure 30 – PGI synchronisation overheads (XT4, pgcc 10.3-0). ................................... 46
Figure 31 – Cray synchronisation overheads (XT4, craycc 7.1.6). .................................. 47
Figure 32 – Pathscale synchronisation overheads (XT4, pathcc 3.2.99). ....................... 47
Figure 33 – GNU synchronisation overheads (XT4, gcc 4.4.3). .................................... 48
Figure 34 – PGI locking overheads (XT4, pgcc 10.3-0). ........................................ 48
Figure 35 – Cray locking overheads (XT4, craycc 7.1.6). ........................................ 49
Figure 36 – Pathscale locking overheads (XT4 3.2.99). ........................................ 49
Figure 37 – GNU locking overheads (XT4, gcc 4.4.3). ........................................ 50
Figure 38 – PGI synchronisation overheads (XT6, pgcc 10.3-0). ............................... 50
Figure 39 – Cray synchronisation overheads (XT6, craycc 7.2.1.105) ....................... 51
Figure 40 – Pathscale synchronisation overheads (XT6, pathcc 3.2.99). .................... 51
Figure 41 – GNU synchronisation overheads (XT6, gcc 4.4.3). ............................... 52
Figure 42 – PGI locking overheads (XT6, pgcc 10.3-0). ........................................ 52
Figure 43 – Cray locking overheads (XT6, craycc 7.2.1.105). .................................. 53
Figure 44 – Pathscale locking overheads (XT6, 3.2.99). ........................................ 53
Figure 45 – GNU locking overheads (XT6, gcc 4.4.3). ........................................ 54
Figure 46 – PGI task generation overheads (Ness, pgcc 10.0-0). .............................. 55
Figure 47 – Intel task generation overheads (Ness, icc 11.1). .................................. 56
Figure 48 – GNU task generation overheads (Ness, gcc 4.4.3). .............................. 56
Figure 49 – PGI task generation overheads (Ness-GPU, pgcc 10.0-0) ....................... 57
Figure 50 – Intel task generation overheads (Ness-GPU, icc 11.1). ........................... 57
Figure 51 – GNU task generation overheads (Ness-GPU, gcc 4.4.3) ......................... 58
Figure 52 – PGI task generation overheads (XT4, pgcc 10.3-0). ............................... 58
Figure 53 – Cray task generation overheads (XT4, craycc 7.1.6). ............................ 59
Figure 54 – GNU task generation overheads (XT4, gcc 4.4.3). .............................. 59
Figure 55 – PGI task generation overheads (XT6, pgcc 10.3-0). ............................... 60
Figure 56 – Cray task generation overheads (XT6, craycc 7.2.1.105) ....................... 60
Figure 57 – GNU task generation overheads (XT6, gcc 4.4.3). .............................. 61
Figure 58 – PGI, Intel and GNU immediate execution overhead
                        (Ness, pgcc 10.0-0, icc 11.1, gcc 4.4.3). ........................................ 61
Figure 59 – PGI, Intel and GNU immediate execution overhead
                        (Ness-GPU, pgcc 10.0-0, icc 11.1, gcc 4.4.3). .............................. 62
Figure 60 – PGI, and GNU immediate execution overhead
                        (XT4, pgcc 10.3-0, icc 11.1, gcc 4.4.3). ........................................ 62
Figure 61 – PGI, and GNU immediate execution overhead
                        (XT6, pgcc 10.3-0, icc 11.1, gcc 4.4.3). ........................................ 63
Chapter 1

Introduction

The purpose of this project is to update the existing set of EPCC OpenMP Microbenchmarks in order to test the new features introduced in version 3.0 of the OpenMP specification.

The purpose of the microbenchmark suite is to measure the computational overhead (in clock cycles) associated with each OpenMP compiler directive. By running the benchmarks on different machines and repeating the tests with a range of compilers it is possible to use the resulting data to compare the performance of different implementations and architectures. This has a twofold purpose: it highlights any deficiencies which may exist in a particular implementation, allowing implementers to improve performance in that area, and it also allows programmers using OpenMP to make informed decisions about which directives should be avoided if possible.

In order to remain relevant, it is necessary to update the microbenchmarks to include features from the latest version of the OpenMP specification, as well as to run them and investigate the performance of current generation compilers and hardware platforms.

This document will firstly introduce key concepts such as the specifics of what OpenMP is and how it is implemented; as well defining what constitutes a microbenchmark and the issues surrounding microbenchmarking in general. The OpenMP tasking model which is new in OpenMP 3.0 will be explained and discussed with the aid of code samples. After these details have been covered, the methodology of version 2.0 of the EPCC OpenMP Microbenchmarks will be examined, followed by an analysis of the directives that it covers.

Subsequent to the background material, Chapter 3 covers the new microbenchmarks that have been developed and gives details of their implementation and why they will provide useful data. As a necessary precursor to the analysis of the results from these benchmarks, the different architectures of the test platforms the benchmarks were run on will then be elucidated, with an overview of the compilers used on each of them.
Chapter 4 contains graphs of all of the data output from running both the pre-existing microbenchmarks as well as the newly designed task system benchmarks. Results are presented by benchmark and secondarily by hardware platform.

Finally, Chapter 5 analyses each individual microbenchmark, comparing implementations and platforms and attempting to extract any general characteristics that become apparent themes, in addition to pointing out instances where it is clear there is room for optimisation.
Chapter 2

Background

2.1 OpenMP

OpenMP is a specification for an API that allows programmers to easily implement thread-based parallelism using the shared variables model. The specification covers the C, C++ and Fortran programming languages.

The specification is designed to be easily portable and implementations exist for both Microsoft Windows and Unix/Linux. Since the majority of modern supercomputers use some variation of Linux, other operating systems will not be investigated in this report. However it should be noted that the microbenchmarks themselves are also portable so doing so would not be a difficult exercise.

OpenMP may be considered a fairly high-level method of achieving multi-threaded parallelism, especially compared with other APIs such as POSIX Threads\(^1\). Specifically, OpenMP does not expose the programmer to any per-thread micromanagement such as creation, job assignment, control and destruction of individual threads; all of which would have to be handled explicitly by the programmer in the case of pthreads and other lower-level methods.

There are two major components in an implementation of OpenMP; compiler support and a runtime library. [1]

Built-in compiler support is necessary in order to transform OpenMP compiler directives into parallel code. This is the case since OpenMP utilises compiler directives in order to specify which code segments are to be run in parallel, in addition to other necessary functions such as thread synchronisation. Compilers without OpenMP support will simply ignore all OpenMP directives and compile the code serially. The fact that the same source code can be compiled either with or without OpenMP enabled is advantageous from a maintainability perspective and also allows for easy verification that a given code is correct when run in serial, for

\(^1\) A.K.A. ‘pthreads’.
parallel debugging purposes. Typically, OpenMP support is enabled by compiler flags such as -mp (PGI) or -fopenmp (GNU).

```c
#include <omp.h>
#include <stdio.h>

void main() {
    #pragma omp parallel
    {
        printf("Hello world, from thread %d.\n", omp_get_num_threads());
    }
}
```

Listing 1 – A simple OpenMP ‘Hello World’ program in C.

The runtime library consists of a number of utility functions as defined in the OpenMP standard, in addition to any internal implementation-specific code. For example, the `omp_get_thread_num()` function is a part of the standard and is accessible to the programmer (in C by including the `omp.h` header file (listing 1) and in Fortran via ‘use omp_lib’), however the runtime library will also need to include internal functions for forking new threads and assigning work.

Environment variables provide a method for allowing the user to affect aspects of the program execution at runtime. The most prominent example of this is the `OMP_NUM_THREADS` variable, which determines the number of threads that each parallel region should fork. Another notable example is the `OMP_SCHEDULE` variable, which allows the user to set a scheduling scheme for all parallel `for` loops where the clause has been set to runtime.

Another critical aspect of programming in OpenMP and the shared variables model more generally is data scoping. The two main types of scope in an OpenMP parallel region are shared and private. A shared variable is visible and accessible for both read and write at any time by all threads. It is this facility that allows for threads to combine partial results of a parallelised calculation or to implicitly communicate by modifying shared data. It is the programmer’s responsibility to ensure that these variables are not written to by multiple threads simultaneously, as the fact that even simple operations such as addition are not atomic means that the final result can often be unexpected and lead to unexpected behaviour in a program. Private variables are replicated for each thread and are created upon entry to the parallel region. This means that each thread has its own copy of a private variable in its own stack, which is completely inaccessible to other threads. Examples of private variables include loop indices and temporary variables for storing intermediate results.

```c
#pragma omp parallel private(i, j) shared(sum)
```

Listing 2 – Example of data scoping in OpenMP. Variables `i` and `j` are designated as private, while `sum` is shared.
2.2 OpenMP 3.0

The key addition to the specification in OpenMP 3.0 is that of task-based parallelism. This feature allows the programmer to designate multiple arbitrary sections of code as ‘tasks’ that may be executed in parallel with other code. Adding tasks to OpenMP allows irregular execution paths to be more easily parallelised than was previously possible, for example recursive functions and traversal of some abstract data structures in a loop.

When a thread encounters a task directive, the enclosed code is designated as a task, the execution of which may be deferred. This means that the data environment of the task is defined (variables initialised as firstprivate\(^2\) unless otherwise stated, for an orphaned task) and the task goes into a pool of incomplete tasks. Upon entering a taskwait or barrier construct a thread will enter the pool of idle threads which may be allocated tasks to complete. The way in which this occurs is implementation defined. [15]

Task directives may include an optional if clause, which, if the enclosed statement evaluates to false, causes the task to be executed immediately by the encountering thread rather than deferred. Use of the if clause can therefore help to reduce overheads created by use of the tasking system by choosing to execute a task immediately rather than put it into the pool, which may be beneficial if there is only a small amount of work to perform within the task (local data may already exist in cache).

\[
\text{#pragma omp task if}\left(\text{statement}\right)
\]

Listing 3 – Adding an immediate execution condition with an if clause.

A given task may be tied (default) or untied. Tied tasks are only ever executed by the first thread to begin executing it and may be suspended at an implicit or explicit barrier or a taskwait. Untied tasks may be partially executed by any number of different threads until they are completed. In addition to the points defined for a tied task, an untied task may be suspended at any point, depending upon the implementation. When a task is suspended the thread executing the task is free to switch execution to another incomplete task, as long as that task is either tied to it or designated as untied. While setting a task as tied increases the chances of associated data remaining in cache for the assigned thread, it may sometimes be beneficial to use an untied task. For example, in the case where there exists number of load imbalanced tasks, it is conceivable that a single thread ends up becoming tied to a few of the tasks containing the majority of the work. If these tasks were untied then other threads would be able to resume them instead of sitting idle while one thread

---

\(^2\) Essentially the same as private, but initialised to the value of the variable when the directive is encountered.
attempts to complete them all. It should be noted that untied tasks should not make use of threadprivate variables since any such data would be lost if the task switched to another thread. [12]

#pragma omp task untied

Listing 4 – Specifying a task as untied.

2.2.1 Example Programs

```c
#include <omp.h>
#include <stdio.h>

int foo(void);
int bar(void);

int main() {
    int result1, result2, total;
    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp task
            {
                result1 = foo();
            }
            #pragma omp task
            {
                result2 = bar();
            }
            #pragma omp taskwait
            printf("Total = %d\n", result1 + result2);
        }
    }
    return 0;
}
```

Listing 5 – A simple C program using OpenMP tasks. For illustration purposes only; a real program with a similar structure may incur less overhead by using sections in place of tasks.

Listing 5 shows a simple C program that uses tasks in order two execute two arbitrary functions (foo and bar) in parallel. The parallel region is entered on line 9 and the number of threads specified in the OMP_NUM_THREADS environment variable are created. At line 11 all threads other than the master thread (thread ID = 0) step over the master region and enter an implicit barrier at the end of the parallel region (line 26), the master thread continues to line 13 where it reaches the first task directive. At this point a task corresponding to the enclosed code is generated, including the data environment of the task [2], because this task directive is not orphaned the data-sharing attributes for the variables inside the task are inherited from the parallel directive. In this particular case there are no explicitly stated data sharing clauses on the parallel directive so the default of shared is applied to
result1 and inherited by the task. With the first task now created it may either be immediately executed by the encountering thread (in this case the master thread) or have its execution deferred to some later point in time, with this choice being determined by the implementation. If the task is executed immediately then the master thread will call foo() and assign its value to result1 before continuing to line 17, where the process is repeated as the second task is created. Once the second task has been dealt with (either generated and executed or generated and deferred) the master thread will enter the taskwait directive on line 22, which it may exit as soon as all outstanding tasks have been completed. The master thread will now output the sum of result1 and result2 before synchronising with other threads in the implicit barrier on line 26 and reaching the end of the program.

What has not been mentioned here is the existence of ‘task scheduling’ points throughout the code. Task scheduling points exist at the following points: upon encountering a task directive, when completing execution of a task, at implicit and explicit barriers, and in taskwait directives [3]. They may also be inserted at any arbitrary point within a task which is designated as untied. A task scheduling point gives the opportunity to the thread at the point to be assigned any outstanding task (that has not been bound to another thread) for immediate execution. Additionally, if a scheduling point is reached whilst executing a task it is possible to suspend the task and pick up a different one. For the case of the example code in listing 5 task scheduling points therefore exist at lines 13, 17, 22 and 26, as well as after lines 15 and 19, that is, interior to but at the end of each of the two tasks. This means that while the master thread generates both of the tasks, any threads waiting in the barrier at line 26 may begin executing these tasks once they have been created. One possible scenario for the case of two threads where both tasks are deferred by the master thread is thus: thread 1 begins execution of one of the two tasks, and when thread 0 reaches the taskwait directive it picks up the remaining task, moving out of the barrier once both tasks are complete. Since much is left up to the implementation and further variability is introduced by threads being asynchronous there are a great deal of possible execution paths.
#include <stdio.h>

int fib(int);

int main() {
    int n = 10;
    int result;
    #pragma omp parallel firstprivate(n)
    {
        #pragma omp master
        {
            result = fib(n);
            printf("Fibonacci No. %d is %d.\n", n, result);
        }
        return 0;
    }

    int fib(int n) {
        int i, j;
        if (n < 2)
            return n;
        else {
            #pragma omp task shared(i)
            i = fib(n-1);
            #pragma omp task shared(j)
            j = fib(n-2);
            #pragma omp taskwait
            return i + j;
        }
    }
}

Listing 6 – Parallel calculation of the n\textsuperscript{th} Fibonacci number using tasks. Taken from the OpenMP 3.0 Specification’s examples section [4] and [5].

Listing 6 details a parallel recursive function for calculating Fibonacci numbers. This function is more complicated in structure than the simple code in listing 5 and demonstrates how tasks allow codes with complex execution paths to be parallelised; whereas the simple program could have easily been parallelised in previous versions of OpenMP by using sections or even simply by wrapping each ‘task’ in an if statement (i.e. if (omp_thread_num() == 1) etc.), it is unclear how the recursive Fibonacci algorithm could be tackled. One approach would be to use nested parallel regions, however this has a much greater overhead than using tasks, and on top of this the behaviour of nested parallel regions was not necessarily consistent between implementations in previous versions of OpenMP.

In terms of semantics it should be noted that variables \( i \) and \( j \) in listing 6 are declared as shared in the task directives, in contrast with result1 and result2 in listing 5 which are not. This is due to the fact that the task directives in the Fibonacci function are orphaned, i.e. not enclosed by a specific parallel construct; if the shared clause was not present then both variables would default to firstprivate.
2.3 Microbenchmarking

Generally speaking, a benchmark is a program that is written to measure the relative performance of a particular aspect of a computer system, be it hardware or software. The Linpack benchmark which is commonly used to assess the relative performance of supercomputers \[6\] is an example of a hardware benchmark: the same code is run on multiple platforms and the number of FLOPS\(^3\) achieved is recorded and used to compare them. While this only determines which hardware platform is best at running the benchmark, the benchmark is designed to be similar to typical real-world use cases, in this case, solving a system of dense linear equations (a common scientific computing problem). In contrast, a software benchmark is typically run on one hardware platform a repeated number of times whilst varying some aspect of software\(^4\), for example, the time taken to encode a given media file may be measured using different algorithms in order to determine which is optimal.

A microbenchmark is a benchmark that is designed to measure the performance of a small, specific piece of code, isolated from any particular real-world use case (the code in question typically has no use in itself). Microbenchmarking contrasts with application-level benchmarking, which attempts to replicate a real-world program albeit in a form from which repeatable measurements can be extracted, such as rendering a predetermined scene as a test of graphical performance. For this reason microbenchmarks may be considered as a ‘low-level’ class of benchmarking, a real world application will essentially be built from numerous components which may individually be suited to microbenchmarking. This hints at the fact that a suite of microbenchmarks are necessary in order gain a comprehensive view of a particular area of interest.

The advantage of a microbenchmark suite over an application level benchmark is that the microbenchmarks will have a wider range of applicability – many different types of program may all use the sections of code tested by the suite, but they may do so in different proportions. Instead of writing many application benchmarks it is possible to get a good indication of performance for a specific program by looking at the most important sections of code and determining their overall contribution to run time and consulting the set of microbenchmarks for their performance characteristics when using different tools (i.e. compilers) or platforms.

2.3.1 Caveats

When designing and running a microbenchmark it is important to ensure that the code under investigation is the same code that actually runs on the hardware. Since code in a given language must actually be compiled to machine code it will undergo numerous transformations and optimisations, with the aim of the compiler being to reduce the total runtime while still producing a correct output, it is possible that a sophisticated compiler will be able to recognise redundant code and to bypass it. While this is highly desirable under

---

\(^3\) Floating Point OPerations Per Second.

\(^4\) Of course, the whole suite of tests may be run again on another platform.
normal circumstances, for the purposes of benchmarking it can be highly destructive as there is a risk that the code being measured may be optimised away, leading to erroneous results. Luckily, most modern compilers will only perform the more disruptive types of optimisation such as inter-procedural analysis at more aggressive optimisation settings, when specific flags are set. In addition to this, for the purposes of this project, OpenMP introduces complexity to the code which serves to makes optimisation more difficult for the compiler. Nevertheless care must be taken to avoid such things occurring. It should be noted that these concerns are less of an issue for application level benchmarks, since these produce some useful result any optimisations are welcome, as well as the fact that these benchmarks may consist of hundreds or thousands of lines of code, compared with tens of lines for a microbenchmark. Some further discussion is located in section 3.1.4 for the design of the immediate execution benchmark.

A microbenchmark may be extremely sensitive to any interruption of its execution, for example by the operating system or any other running programs that are in contention for system resources. Again, with an application level benchmark this is less of a problem due to the fact that it is designed to measure a real world scenario and the benchmark itself will typically run for a much greater amount of time compared with a microbenchmark. If an interruption of some sort does occur during microbenchmarking, such as a context switch, it may throw off a test by a factor several times its ‘true’ run time. This is discussed further in section 3.4 on statistical significance.

2.3.2 Examples

SKaMPI\(^5\) is a microbenchmark suite designed to measure the performance of MPI\(^6\) library functions, allowing for performance comparisons between MPI implementations (such as OpenMPI and MPICH) and different hardware platforms. It is somewhat comparable to the EPCC’s OpenMP Microbenchmark suite in terms of aim, though applied to MPI rather than OpenMP.

2.4 EPCC OpenMP Microbenchmarks V2.0

Results from the first version of the EPCC microbenchmarks were first presented in 1999 at the first European Workshop on OpenMP\(^7\). This version of the benchmarks related to version 1.0 of the C and Fortran OpenMP standards, which were at the time separate documents. The paper compared three systems from Sun, SGI and Compaq, using different compilers on each system. The current version of the microbenchmarks had results presented EWOMP 2004 and related to version 2.0 of the OpenMP specifications. The main new benchmarks incorporated related to array data clauses and there was also a restructuring of the Fortran code to use the free-format style. Since then versions 2.5 and 3.0 of the specification have

\(^5\) Special Karlsruher MPI benchmark. See http://liinwww.ira.uka.de/~skampi/index.html

\(^6\) Message Passing Interface. See http://www.mpi-forum.org/

\(^7\) EWOMP
been released, with 2.5 marking a merger between the separate C and Fortran documents with few other changes. OpenMP 3.0 is discussed in section 2.2.

2.4.1 Methodology

The overhead associated with a given directive, \( O_p \), can be calculated according to the formula:

\[
O_p = T_p - \frac{T_s}{p}
\]  

(1)

Where:

- \( O_p \) – Parallel overhead
- \( T_p \) – Parallel benchmark execution time
- \( T_s \) – Serial benchmark execution time
- \( p \) – Number of threads

\( T_p \) and \( T_s \) relate to the time taken to execute a dummy function a set number of times, in parallel and in serial respectively. For example, the overhead of the parallel directive may be measured by subtracting the execution time of listing 7 from that of 8 and dividing by the number of repetitions in the loop. [7]

```c
for (i = 0; i < reps; i++) {
    delay(delaylength);
}
```

Listing 7 – Reference time loop.

```c
for (i = 0; i < reps; i++) {
    #pragma omp parallel
    {
        delay(delaylength);
    }
}
```

Listing 8 – Parallel directive benchmark loop.

It should be noted that the length of the delay function is chosen such that it takes approximately 100 clock cycles, this is to try to minimise loss of precision caused by subtracting numbers differing significantly in magnitude. Additionally, by running the parallel code for the same number of iterations as the serial reference code, thereby multiplying the total amount of work done by the number of threads, the need to divide by \( p \) in equation (1) may be neglected and the magnitude of the two times are further kept similar.
2.5 List of Benchmarks

2.5.1 Scheduling

2.5.1.1 Static and Static,n

The static and static,n schedules are the simplest scheduling schemes in OpenMP and they should therefore incur the least amount of overhead. The plain static schedule allocates loop iterations to threads simply by dividing the total number of iterations by the number of threads, this is a one-off cost that only needs to be performed when the loop is entered, and which may be partially performed at compile-time depending on whether the iteration space is known or dynamic. The static,n schedule is similar, however instead of dividing the iteration space by the number of threads it is divided into chunks containing chunksize iterations\(^8\). All of the chunks are then pre-assigned to threads in a cyclic fashion. Therefore, while the static schedule has a constant cost, the static,n has a cost that decreases as the chunksize decreases due to the fact that the number of chunks increases and there is a cost associated with switching between chunks (even though they are already allocated).

2.5.1.2 Dynamic and Guided

The guided and dynamic schedules are similar in that they both require work to be performed at runtime in order to allocate loop iterations to threads. For the dynamic schedule, as can be read in the OpenMP specification [8], the total number of iterations is divided into chunks of a size equal the specified chunksize. The chunks are then allocated to threads on a first-come-first-served basis, i.e. as soon as a thread becomes idle it will be allocated the next incomplete chunk. This means that the smaller the chunksize is, the greater the number of total chunks, and therefore the greater the amount of overhead incurred in the process of allocation. We would therefore expect the overhead of the dynamic schedule to be highest at a chunksize of 1 and to decrease as chunksize approaches 128, which as can be seen at line 5 in listing 9 is the point where the loop is split into the same number of chunks as the number of threads, i.e. one chunk per thread.

\(^8\) If the iteration space does not divide exactly by chunksize then the final chunk may be smaller than the rest.
For the guided schedule the chunksize sets the minimum size of a chunk rather than setting the size of all chunks as with dynamic. Apart from this restriction determined by the chunksize, the size of a chunk is calculated by dividing the number of remaining iterations by the number of threads, as a result of this chunks start large and get smaller. This means that there will always be fewer chunks than with a dynamic schedule and therefore the overhead resulting from allocation should be lower (although calculating the number of iterations to allocate adds some additional work to guided).

2.5.2 Synchronisation
2.5.2.1 Parallel, For, Barrier and Single

The first set of graphs for the synchronisation microbenchmarks contains the parallel, for, parallel for, parallel with reduction, barrier, and single directives. The parallel construct is fundamental to OpenMP as it is this directive that is responsible for defining areas of code to be executed in parallel. In OpenMP 3.0 a parallel region generates a number of tasks equal to the number of threads requested (either specified by the OMP_NUM_THREADS environment variable or by a num_threads clause on the parallel directive), and then each thread is tied to one of these tasks [9]. The end of a parallel region contains an implicit barrier so the overhead of the parallel directive must be greater than that of the barrier directive. Since the parallel directive is only usually used a relatively small number of times in a program, the overhead associated with it is not of great importance for typical scientific applications as it will be a very small one-off cost in comparison to the main body of the code. For any programs wishing to utilise OpenMP in a realtime application however, the parallel directive could ostensibly be encountered many times per second, in which case any excessive overhead in implementation of the directive could be crippling to performance. At the time of writing there are no well-known examples of OpenMP being used in such a context, a case study of using OpenMP to parallelise a game [10] made use of only one parallel construct that was only encountered at initialisation, meaning that overhead of the parallel directive would not have a significant impact in this case either. Even so, it is still worthwhile to examine a microbenchmark of the directive in order to ensure that there are no major failings in any of the implementations being tested.

---

9. The parallel directive enclosed the main function call used to start the game, and was immediately followed by a single directive. The majority of threads would then wait in the implicit barrier at the end of the parallel region, where they would pick up tasks created by the main thread.
The for directive, used to parallelise for loops, is one of the most commonly used OpenMP directives. The cost associated with this directive will determine the minimum amount of work in a loop that is worth parallelising in this manner; for parallelisation to be profitable the equation \( x + t < xt \) must be satisfied\(^{10}\).

The single directive is used to make sections of code in a parallel region be executed only once. An encountering thread will set a flag and execute the code if it has not yet been executed and step over the block if the thread sees that the flag has already been set. There is an implicit barrier at the end of a single region unless the nowait clause is specified. For this reason the overhead of the single directive should have a lower limit of that of the barrier directive.

2.5.2.2 Locking

The second set of graphs for the synchronisation microbenchmarks contains those directives used to prevent sections of code from being executed by multiple threads at once: critical, lock/unlock, atomic and ordered.

When a piece of code is designated as a critical section using the critical directive it may only ever be executed by one thread at a time, if a thread encounters a critical section that another thread is currently executing then it will wait until that thread exits the section before continuing. The mechanism for implementing this is implementation defined.

In addition to the critical directive, the OpenMP runtime routines omp_set_lock and omp_unset_lock offer another slightly more explicit and flexible method for protecting regions of code from being accessed simultaneously by multiple threads, at the cost of increased micromanagement due to the necessity of lock variables. A thread encountering a locked section of code behaves identically to a thread encountering busy critical section, and must wait for the section to be unlocked before continuing. As mentioned, the advantage of using lock routines over critical sections is increased flexibility for the programmer; a lock may be set or unset from any section of code with access to the associated lock variable, whereas a critical section must be a single contiguous block of code.

The atomic directive differs from the other directives here in that it may only be applied to a single statement rather than a block of code, furthermore that statement must modify the value of a variable.

---

\(^{10}\) For overhead cost \( a \), number of threads \( t \), work cost \( xa \), parallelisation is profitable for: \( a + \frac{xa}{t} < xa \) (i.e. \( \frac{overhead + \frac{work}{threads}}{work} < work \)). Thus: \( x + t < xt \)
int x = 0;
#pragma omp parallel shared(x)
{
    x += foo();
}
return x;

Listing 10 – Example of code with a data race that should be avoided with use of an atomic directive.

In listing 10 the value of the variable x will differ when the return statement is reached, depending upon the order in which line 4 was evaluated by the threads in the parallel region. This occurs since the statement is essentially split into four segments; load x from memory, evaluate foo(), add the value of foo() to x, and finally, store x in memory. The key point is that the load and store of x may be performed by each thread in any order, meaning that if all of the threads in the parallel region in this example loaded the value 0, then the last thread to issue a store would overwrite the value already there, resulting in the final value of x being equal to 0 + foo() for that thread only.

One method of making this code produce the correct result (i.e. the sum of the values of foo() for each thread) would be to enclose line 4 in a critical directive. While this would work perfectly ok in terms of getting the right answer, the code would be entirely serialised as each thread would have to wait its turn to execute the statement. If an atomic directive was utilised on the other hand, the code would be mostly parallel as well as returning the correct result (assuming foo() takes significantly longer than loading and storing x). This is because the atomic directive ensures that loads and stores of x are serialised and must be performed immediately one after the other, hence atomic. The resulting program is parallel as threads may all evaluate foo() at any time with only the load and store being protected, unlike if a critical section was used. For this reason the atomic directive should incur the least overhead of this set of benchmarks.

The ordered construct is designed to be useful for sequentially ordering output from parallel code in a loop [10]. Code placed within an ordered region is executed only once per loop iteration by the first thread that encounters it, while the section is being executed, subsequent encountering threads will step over the region and continue until the region is reached once more in the next loop iteration. At this point all encountering threads will wait until the ordered region has been completed from the previous loop iteration, in similar manner to threads waiting at a busy critical section. For a loop containing a large amount of work and an ordered section containing relatively little work it is possible that the overhead of the ordered section may be largely hidden since one ordered block will usually be completed by the time the next one is reached. For a loop containing a large proportion of work inside an ordered block, the loop effectively becomes serialised as this proportion increases.
Chapter 3

Implementation and Test Platforms

3.1 Additional Benchmarks Implemented

3.1.1 Parallel and Serial Task Generation

The serial and parallel task generation microbenchmarks are designed to show which strategy of task generation produces the least amount of overhead.

In the case of serial task generation, one thread iterates through a loop which generates one task per iteration while the other threads wait at an implicit barrier (and therefore task scheduling point) and pick up work as it becomes available. The efficiency of the underlying task creation and allocation system is key here, if it takes longer to generate a task than it does to pick it up and execute the enclosed work then the code becomes serialised as the work from one iteration of the loop will be completed before the next set of work can be made available. This scheme should perform progressively worse as the number of threads increases as the rate at which tasks can be generated will remain constant as the number of threads that must be kept busy increases.

```c
#pragma omp parallel private(j,i)
{
    #pragma omp master
    {
        for (j = 0; j < innerreps; j++) {
            /* Since this is executed by one thread we need
            * (itersperthr * nthreads) iterations.
            */
            for (i = 0; i < itersperthr * nthreads; i++) {
                #pragma omp task
                {
                    delay(delaylength);
                }
            }
        }
    } /* End master */
} /* End parallel */
```

Listing 11 – The serial task generation microbenchmark. Extract from `taskbench.c`. 
As can be seen in the code for the serial task generation benchmark (listing 11 above), the structure of the code is similar to that described in listing 5, the master thread is responsible for generating tasks while other threads wait in the implicit barrier at the end of the parallel region, picking up tasks as they are created.

For parallel task generation every thread takes part in generating tasks, with each thread iterating over its own loop. This could mean that a task generation phase takes place before any work is executed and therefore work is commenced later than with serial task generation. However, this may not be the case; due to the fact that a task scheduling point occurs when creating a task it is possible that an implementation could decide to begin execution of any outstanding task. It is speculated that such behaviour may be beneficial if it is decided that the number of outstanding tasks is too high, as each outstanding task will consume memory resources in order to store its data environment. It is possible that the additional logic for such decisions may add further overhead however. Regardless of these factors, parallel task generation should theoretically scale better than serial task generation as every additional thread will always have work to do generating tasks.

```c
#pragma omp parallel private(j,i)
{
  for (j = 0; j < innerreps; j++) {
    /*
     * Since this is executed by all threads we need
     * (itersperthr) iterations.
     */
    for (i = 0; i < itersperthr; i++) {
      #pragma omp task
      {
        delay(delaylength);
      }
    }
  }
}
```

Listing 12 – The parallel task generation microbenchmark. Extract from taskbench.c.

The parallel task generation benchmark is similar to the serial generation code, albeit without a master directive. It should be noted that the iteration bounds of the innermost loop must be changed from the serial task generation code in order to produce the correct number of total tasks; all threads will iterate over the generation loop so itersperthr iterations are required rather than itersperthr*nthreads iterations.

### 3.1.2 Parallel Generation + Barrier/Taskwait

Both the barrier and taskwait directives will halt all encountering threads until child tasks created so far by the current task have been completed. Each of these directives also contains a task scheduling point so it is possible for a thread to continue executing tasks whilst inside either of these constructs. The fact that a taskwait construct does not require all threads to reach it and synchronise at that point differentiates it from the barrier directive, which has this requirement; for this reason it is expected that the taskwait directive will be the lighter-weight of the two, inducing less overhead.
In order to measure the overhead of the barrier and taskwait directives in this context, similar code to the parallel task generation microbenchmark was employed (listing 12), with the directive in question being inserted after the task generation loop.

```c
#pragma omp parallel private(j,i)
{
    for (j = 0; j < innerreps; j++) {
        /*
        * Since this is executed by all threads we need
        * (itersperthr) iterations.
        */
        for (i = 0; i < itersperthr; i++) {
            #pragma omp task
            {
                delay(delaylength);
            }
        }
    #pragma omp taskwait
}
```

Listing 13 – Parallel task generation with taskwait directive. Note that for the equivalent barrier benchmark the taskwait directive is simply replaced with a barrier directive.

Placing the directive at this point allows for a significant number of tasks to be generated, thus creating ‘something to wait for’: waiting for a very small number of tasks to complete would not provide a good metric of performance. Also, at this position the directive is still inside the innerreps loop, resulting in the directive being encountered many times so that it can be averaged over and not lost due to insufficient numerical precision: if only a single taskwait/barrier was inside the outerreps loop then its overhead would likely be extremely small compared to the total number of cycles spent on the task generation loop and the work itself.

### 3.1.3 Tree Structured Task Generation

Serial and parallel task generation as discussed in section 3.1.2 are two common schemes for parallelisation using tasks likely to occur in a real program\(^{11}\). A further task generation pattern is the recursive case, where a thread generates a task which then itself will create more tasks, the Fibonacci code in listing 6 is an example of such a program. Recursive generation of tasks in this manner represents a significant use-case for tasks since it is at least partly what the task system was created for, that is, to allow for parallelisation of code with more complex execution patterns rather than primarily looping structures.

To test the performance of recursive task generation two slightly differing microbenchmarks were written in which tasks are generated in a tree structure; each task generates two child

\(^{11}\) A real program would most likely not use tasks in a manner where usage of tasks could be replaced with an OpenMP `for` directive; however the pattern of task generation would remain the same.
tasks and each of these child tasks generate their own child tasks, and so on until the required number of tasks have been created.

The first tree structured generation function places work (i.e. the dummy delay function) on both the branches and leaves of the tree structure. This is illustrated in figure 1.

![Diagram of the tree created by the tree gen branch microbenchmark.](image)

Figure 1 – Diagram of the tree created by the tree gen branch microbenchmark. Nodes containing work are shaded.

This structure is created by calling a recursive function, tree_gen_branch that takes which level of the tree the call is on as its argument. Once the tree function is called with a level that satisfies the pre-computed required level to generate the correct amount of work, the function generates no further tasks and the tree terminates. For example, if itersperthr is equal to 64 then before the benchmark starts, tree_term_level will be calculated as 6. The if statement in the tree_gen_branch function will then evaluate to false when the tree_gen_branch function is called with a tree_level argument of 6.

```c
#pragma omp parallel private(j)
{
    #pragma omp master
    {
        for (j = 0; j < innerreps*nthreads; j++) {
            #pragma omp task
            {
                delay(delaylength);
            }
            tree_gen_branch(1);
        }
    }
}
```

Listing 14 – The first tree-structured task generation microbenchmark. Note that one additional task containing work is placed before the call to tree_gen_branch in order to make the total amount of work executed equal to that performed by the second tree-structured task generation microbenchmark.
/* Compute maximum binary tree depth */

```c
int tree_term_level = (int) (log10((float)itersperthr) / log10(2.0));
```

tree_term_level++;

### Listing 14 --

```c
/* * Work takes place at each branch and each leaf */

void tree_gen_branch(int tree_level) {
    if ( tree_level < tree_term_level ) {
        #pragma omp task
        {
            delay(delaylength);
            tree_gen_branch(tree_level + 1);
            tree_gen_branch(tree_level + 1);
        }
    }
}
```

### Listing 15 --

As a variation on the previous tree an additional tree was written which places all of the work (delay calls) on the leaf nodes rather than on branches as well. To achieve this the total number of tasks is greater since the tree has to have one additional layer of depth in order to create an equivalent number of work items. While this should introduce more overhead in the form of additional tasks, it also means that each task is faster to execute as none of the branch nodes will contain work other than that of generating more tasks, meaning that the full tree can be generated faster. The function for the recursive generation of each node in this tree is presented in listing 16, note that the code for the full microbenchmark is similar to that in listing 15, but with the function call changed to `tree_gen_leaf` and the additional task removed.

![Diagram of the tree generated by the tree gen leaf microbenchmark. Nodes containing work are shaded.](image)

### Figure 2 – Diagram of the tree generated by the tree gen leaf microbenchmark. Nodes containing work are shaded.
/**
 * All work takes place on leaf nodes
 */

void tree_gen_leaf(int tree_level) {
    if ( tree_level == tree_term_level ) {
        delay(delaylength);
    } else {
        #pragma omp task
        {
            tree_gen_leaf(tree_level + 1);
            tree_gen_leaf(tree_level + 1);
        }
    }
}

Listing 16 --

It should be noted that while tasks are generated in a tree pattern, the nature of the tasks system means that the order in which tasks are executed depends largely on the underlying implementation. It is quite possible that while some section of the tree has been generated all the way down to the leaf nodes, another section may have outstanding tasks close to the top of the tree, preventing any generation of tasks further down. From the perspective of parallel computation, for a balanced tree of unknown depth it is generally favourable to prioritise tasks higher up the tree, if this is done then it will take longer to reach a leaf node, which is undesirable since a leaf node creates no additional work. This affects the amount of parallelism available due to the fact that with a tree structure there is a delay before enough tasks are generated to occupy all of the available threads; for a binary tree with eight threads available for example, the first second and third levels would need to be completed before there were eight tasks outstanding.

3.1.4 Immediate Execution

One item of interest with regards to performance in OpenMP 3.0 is the overhead induced by a task construct with an if clause that evaluates to false, which results in the task being executed immediately rather than deferred. The result of this microbenchmark will determine how practical it is to use tasks liberally throughout a program, switching each of them to immediate execution (or not) individually, depending upon the situation. If the overhead of using tasks with an if clause that evaluates to false is high then the aforementioned use case will not be of much use, any potential gain from the situational parallelism will probably be outweighed by this overhead. If the overhead is found to be very low however it might be worth encouraging such practices.

Rather than if(<false>) resulting in a task construct being removed altogether, the OpenMP specification requires that the task is still generated, just that when generation has been
completed that the task is executed immediately by the current thread. The parent task may not be resumed until this task has been completed. The task still being generated is necessary as otherwise the meaning of the enclosed code may differ depending upon the truth value of the if clause, due to the effects of any data environment clauses for variables used inside the task region. In this manner behaviour of the task in relation to lock ownership and synchronisation is also kept consistent [12].

```c
#pragma omp parallel private(j,i) 
{ 
    for (j = 0; j < innerreps; j++) {
        for (i = 0; i < itersperthr; i++) {
            #pragma omp task if(return_false_arg(i))
            { 
                delay(delaylength);
            }
        }
    }
}
```

Listing 17 – The immediate execution microbenchmark. The content of the if clause varies slightly with each test. Extract from `taskbench.c`.

In order to check whether or not any of the compilers on test optimise-out a task construct with a compile-time-evaluate-able if clause that evaluates to false, such as `if(0)`, the benchmark was run with three of different statements with the same effect. The first case uses the previously mentioned `if(0)`, the second case uses a function call `return_false()` that simply returns 0 and the third cases uses a function call taking an argument that always returns 0 (but is not just ‘return 0’). These second two cases should prevent a compiler from determining at compile-time that the if clause is false (case two could possible be determined at compile time by a sophisticated compiler with inter-procedural optimisation or if the function was in-lined and re-evaluated, however the third case makes it extremely unlikely for something like this to occur).

### 3.2 Note on Memory Profiling

Although not technically a microbenchmark, one further aspect of an OpenMP implementation worth measuring is memory usage. In scientific computing one of the constraints on the size and complexity of the simulations that can be run on a computer is the amount of memory available. With this in mind, if an implementation of tasks uses an excessive amount of memory as the number of outstanding tasks increases then that implementation may be precluded from being used.

In an attempt to investigate the memory usage of each implementation being evaluated the ‘Massif’ tool was used, which is part of the Valgrind suite of performance analysis tools. Valgrind works by taking a standard executable binary file and running it on a virtual CPU, inserting its own instrumentation in the process [13]. In the case of Massif, memory allocation calls are intercepted and instrumented. Traditional performance analysis tools often work by modifying a binary at compilation by adding calls to their own instrumentation code, unavoidably altering the program being examined. With Valgrind this can be avoided due to the fact that the analysis happens in software – as long as the software simulation of the CPU
is accurate, data collected will be accurate and instrumentation calls can be specifically excluded from the results. Due to the fact that Valgrind is a mature tool, this is usually true for serial code (data collected is very accurate), however for parallel code it is impossible to avoid interfering with the results as a result of non-determinacy.

Despite this, tests were run to see if any information could be gleaned. Unfortunately it was found that code compiled by the PGI compiler running under Valgrind resulted in consumption of all system memory due to some unknown bug, preventing the gathering of any data. Code compiled by the GNU compiler ran correctly, but with a prohibitively large run time. As a result of these failures this line of investigation was abandoned.

### 3.3 Hardware Platforms

#### 3.3.1 Ness

Ness is an EPCC research computer that provides a similar environment to that of large scale supercomputers such as HECToR, thus also lending itself well to training. The system consists of two separate Sun Fire X4600 nodes containing 8 dual-core AMD Opterons each. As a result of the fact that Ness has the largest number of individual processors out of the systems being tested it may be expected that it performs comparatively poorly in some of the benchmarks due to an increased need to access data stored off-chip.

<table>
<thead>
<tr>
<th>System Name</th>
<th>Ness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td>Scientific Linux</td>
</tr>
<tr>
<td>Processor</td>
<td>AMD Opteron</td>
</tr>
<tr>
<td>Model No</td>
<td>1218</td>
</tr>
<tr>
<td>Codename</td>
<td>Santa Ana</td>
</tr>
<tr>
<td>Cores Per Processor</td>
<td>2</td>
</tr>
<tr>
<td>Clock Speed</td>
<td>2.6 GHz</td>
</tr>
<tr>
<td>Cache</td>
<td>L2: 1MB per Core, L3: None</td>
</tr>
<tr>
<td>Total No of Processors</td>
<td>8</td>
</tr>
<tr>
<td>Total RAM</td>
<td>32 GB</td>
</tr>
<tr>
<td>RAM Per Core</td>
<td>2 GB</td>
</tr>
</tbody>
</table>

#### 3.3.2 Ness-GPU

The Ness-GPU system is designed for the testing of GPGPU codes, however since it uses Intel ‘Nehalem’-based chips it is an interesting candidate to compare performance with the other systems and investigate the effects that the differences in architecture have on the microbenchmarks.
### 3.3.3 HECToR XT4

While the HECToR XT4 service has a large number of total cores, each individual node contains only a single quad-core processor. Nevertheless, using the system allows for usage of some additional compilers not available on Ness, as well as investigation of the effect of having 4 cores on the same die and a shared L3 cache, compared with 4 cores necessarily being on two separate processors, each without L3 cache, as is the case with Ness. Both the XT4 and XT6 use the specialised ‘Compute Node Linux’ operating system from Cray, designed to minimise the chances of the operating system adversely affecting performance (section 3.4). The fact that each benchmark is run a repeated number of times on each hardware platform as described in section 3.4 means that this difference should have no observable effect on the results, apart from perhaps a reduced incidence of outlying runs occurring.

<table>
<thead>
<tr>
<th><strong>System Name</strong></th>
<th><strong>Ness-GPU</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td>Scientific Linux</td>
</tr>
<tr>
<td>Processor</td>
<td>Intel Xeon (Nehalem-Based)</td>
</tr>
<tr>
<td></td>
<td>Model (N^2)</td>
</tr>
<tr>
<td></td>
<td>Codename</td>
</tr>
<tr>
<td></td>
<td>Cores Per Processor</td>
</tr>
<tr>
<td></td>
<td>Clock Speed</td>
</tr>
<tr>
<td></td>
<td>Cache</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Total (N^2) of Processors</td>
<td>2</td>
</tr>
<tr>
<td>Total RAM</td>
<td>24 GB</td>
</tr>
<tr>
<td>RAM Per Core</td>
<td>3 GB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>System Name</strong></th>
<th><strong>HECToR XT4 Node</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td>Cray Compute Node Linux</td>
</tr>
<tr>
<td>Processor</td>
<td>AMD Opteron</td>
</tr>
<tr>
<td></td>
<td>Model (N^2)</td>
</tr>
<tr>
<td></td>
<td>Codename</td>
</tr>
<tr>
<td></td>
<td>Cores Per Processor</td>
</tr>
<tr>
<td></td>
<td>Clock Speed</td>
</tr>
<tr>
<td></td>
<td>Cache</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Total (N^2) of Processors</td>
<td>1</td>
</tr>
<tr>
<td>Total RAM</td>
<td>8 GB</td>
</tr>
<tr>
<td>RAM Per Core</td>
<td>2 GB</td>
</tr>
</tbody>
</table>

### 3.3.4 HECToR XT6

The XT6 (phase 2B upgrade) greatly increases the number of available cores from 4 per node to 24 per node, although the amount of memory available per core actually decreases. Since the microbenchmarks will not use a large amount of memory this will have no effect on the results.
<table>
<thead>
<tr>
<th>System Name</th>
<th>HECToR XT6 Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td>Cray Compute Node Linux</td>
</tr>
<tr>
<td>Processor</td>
<td>AMD Opteron</td>
</tr>
<tr>
<td>Model No</td>
<td>6172</td>
</tr>
<tr>
<td>Codename</td>
<td>Magny-Cours</td>
</tr>
<tr>
<td>Cores Per Processor</td>
<td>12 (2x6 Core NUMA regions)</td>
</tr>
<tr>
<td>Clock Speed</td>
<td>2.1 GHz</td>
</tr>
</tbody>
</table>
| Cache | L2: 512KB per Core  
| | L3: 2x6MB Shared  
| | (10MB Available) |
| Total No of Processors | 2 |
| Total RAM | 32 GB |
| RAM Per Core | 1.3 GB |

An interesting feature of the XT6 nodes is that the ‘Magny-Cours’ 12-core CPU is actually a Multi-Chip Module (MCM) consisting of two 6-core CPUs linked via HyperTransport links [14].

![Figure 3 – Magny-Cours architecture diagram. [14]](image)

This architecture results in a hierarchy of core to core communications costs, in order of speed, fastest first: both cores on same 6-core die, cores on separate dies within same MCM; cores on separate MCMs, directly across; cores on separate MCMs, diagonally across. When running the microbenchmarks with varying numbers of threads the assigned cores fill up each 6-core die in turn, filling an entire MCM before placing any threads on cores located on the
second MCM. As a result of this pattern it may be possible to see increased overheads in the microbenchmarks when crossing these boundaries, i.e. 6→7 cores, 12→13 cores and 18→19 cores. It remains to be seen whether or not this is true since the increased latency may be hidden by other factors.

### 3.4 Statistical Significance

As mentioned in section 2.3.1 it is possible for other programs running on a system or the operating system itself to interfere with a microbenchmarking run that is in progress. There are a number considerations that can be taken into account to minimise the chances of this happening.

Firstly, the machine on which the test is taking place should be otherwise idle, with no other users and a minimal number of background tasks. Obviously, the greater the number of programs running, the more likely it is that one of them will wake and require system resources during a given time period. Unless the microbenchmark is highly underutilising the system (and even then there will still be some effect) the program waking up will divert resources from the benchmark, resulting in an increased time being recorded for the benchmark currently running. On HECToR the operating system running on each node is a specially modified version of Linux, Cray ‘Compute Node Linux’, which is designed to reduce the overall footprint of the OS and to reduce the chances of interfering with the program running on the node.

Secondly, on a shared computing resource such as Ness\(^\text{12}\) an entire node should always be reserved if possible, even if not all cores are being utilised. This eliminates the chances of another program being scheduled on the same node, which, although it will not utilise the same cores as the benchmark will usually share memory bandwidth to RAM.

As a side note it is possible to increase the OS scheduling priority of the benchmark, which will decrease the likelihood of the OS deciding to interrupt it. On unix-like systems this is achieved by setting a negative ‘niceness’ value for the program with -20 being the highest priority. This can be achieved by running the program via the `nice` command line utility. It should be noted that this is not applicable to the systems used in this project due to the fact they are only indirectly accessible via a batch system (PBS Pro on HECToR and Sun Grid Engine on Ness).

While what has been discussed so far reduces the chances of interference in the execution of the benchmark, it is inevitable that there will be at least a few instances where interference does occur. In order to identify and eliminate these cases it is necessary to run a microbenchmark many times over and then examine the distribution of timings recorded. This can be seen in listings 18 and 19 below.

---

\(^\text{12}\) On HECToR it is not possible to use any less than one whole node.
for (k = 0; k <= OUTERREPS; k++){
    start = getclock();
    #pragma omp parallel private(j)
    {
        for (j = 0; j < innerreps; j++){
            delay(delaylength);
            #pragma omp barrier
        }
    }
    times[k] = (getclock() - start) * 1.0e6 / (double) innerreps;
}

Listing 18 – Microbenchmark to measure the barrier directive. Extract from syncbench.c from version 2 of the EPCC OpenMP Microbenchmarks.

while (actualtime < targettime) {
    delaylength = delaylength * 1.1 + 1;
    start = getclock();
    for (i=0; i< reps; i++) {
        delay(delaylength);
    }
    actualtime  = (getclock() - start) / (double) reps;
}

Listing 19 – delaylength-tuning loop. Extract from schedbench.c.

3.5 Compilers Used

3.5.1 Portland Group

The Portland Group compiler is amongst the most widely used compilers in the scientific computing and wider high performance computing communities. As such it is of key importance to test this compiler against the updated set of benchmarks in order to identify any weak points in its implementation of OpenMP.

3.5.2 GNU

Due to the fact that the GNU Compiler Collection (gcc) is free software it sees widespread use across all areas of programming, furthermore it supports a number of programming languages and a wide range hardware platforms. The ubiquitous nature of gcc makes it a worthwhile candidate for testing.

Although gcc fully supports OpenMP 3.0, a possible bug in the implementation was discovered when running one of the newly written task system microbenchmarks. When running the task generation benchmark with a barrier directive (section 3.1.3) a gcc-compiled binary will hang indefinitely the majority of the time and not complete execution of the benchmark, though if the binary is run multiple times it will occasionally complete. This behaviour is suggestive of a race condition being present in the code that causes one or more threads to stall waiting to complete synchronisation with the other threads. Since this behaviour was not observed for any of the other compilers on test when compiling identical code, it must be concluded that it is likely that there is a bug in gcc’s implementation of tasks
rather than there being a bug in the microbenchmark code. Pre-processor conditionals were therefore added to the final code to allow a compile-time disable of this specific test.

3.5.3 Intel

Despite not being as commonly used as either the PGI or GNU compilers, the Intel compiler is worth investigating if only due to Intel’s position as the world’s largest CPU manufacturer. Not being available natively on HECToR, problems getting the compiled binaries to run prevented testing on this platform.

3.5.4 Cray

The Cray compiler is only available on the HECToR system. Although the compiler supports OpenMP 3.0 it was found to be missing support for if clauses on task directives when attempting to compile the new benchmarks. For this reason the immediate execution benchmarks could not be performed with this compiler.

3.5.5 Pathscale

Out of the platforms investigated the Pathscale compiler is only available on the HECToR XT4 and XT6 systems. In addition to this, the current Pathscale compiler only supports OpenMP up to version 2.5 and hence it can only be used for the scheduling and synchronisation benchmarks.

3.5.6 Compilation Note for HECToR

Compilation of code on HECToR using different compilers is handled by a Cray-provided wrapper script, cc, and a system of modules. In order to use the system, cc is always used to refer to the (C) compiler when compiling, then, to change which compiler cc invokes, the relevant ‘programming environment’ module must be loaded. An example of how to compile a simple C program (i.e. not using a Makefile) using the GNU C compiler is provided in listing 20.

```
$ #Unload the PGI compiler (loaded by default).
$ module unload PrgEnv-pgi

$ #Load the desired compiler’s environment.
$ module load PrgEnv-gnu

$ #Perform compilation using the ‘cc’ wrapper.
$ cc hello_world.c -o hello
```

Listing 20 – Compilation on HECToR.

The cc wrapper automatically includes all compiler flags required for a program to run correctly on HECToR’s back-end nodes. In order to optimise the output executable for the specific CPU architecture on HECToR another module must be loaded before compilation that will add further flags to the cc wrapper. On the XT4 (phase 2a) system the module
xtpe-barcelona should be loaded, on the XT6 (phase 2b) system the xtpe-mc12 module needs to be loaded.

### 3.5.7 Compiler Flags Used

It was decided that for each compiler an ‘average’ level of serial optimisation should be turned on when compiling the benchmark code, for several reasons; lack of optimisation is unrealistic, the benchmarks should not take an excessive amount of computer time, higher level optimisation increases the likelihood of it disrupting the test (see section 2.3.1), and finally, the higher the optimisation level, the greater the divergence in options between compilers. Since the available options and the definition of those options that are common varies substantially from compiler to compiler it is not possible to provide a very meaningful definition for ‘average’, in addition to this a disproportionate amount of time could be spent in any attempt to ensure that each compiler had equivalent features enabled. In the end it was decided to allow common optimisations that don’t require an in-depth knowledge of the individual compiler as well as optimisation for the architecture used.

The compiler flags decided upon may be found in the Makefile provided as part of the code as well as in the (less comprehensive) listing 21 below.

**gcc:**
- `O2 -fopenmp -lm -Wall -march=native -DDISABLE_TGEN_BARRIER_TEST`

**pgcc:**
- `fast -mp -lm -tp k8-64e`
  (`-tp nehalem-64 on Ness-GPU`)

**icc:**
- `m64 -O2 -openmp -openmp-link=static -lm`

**craycc:**
- `O2 -lm -DDISABLE_IMMEDIATE_TEST`

**pathcc:**
- `mp -lm -O2 -OPT:Ofast`

**Listing 21 – List of compiler flags used to compile the benchmarks. It should be noted that architecture-specific optimisation flags are not used on HECToR due to the module system.**

Notice that the Intel compiler does not include any architecture optimisation flags. This is a result of the compiler opting to provide the facility to turn on individual extensions to the x86 ISA\(^{13}\) (such as SSE3 for example) rather than choosing a particular CPU. While an attempt was made to use this facility, the binaries produced refused to execute on Ness citing lack of

---

\(^{13}\) Instruction Set Architecture
instruction support on the platform, despite the AMD Opterons on Ness supporting the enabled extensions in reality\textsuperscript{14}. Therefore the decision was made not to use these flags.

The pre-processor definitions included with \texttt{gcc} and \texttt{craycc} are to turn off certain tests as described in 3.3.2 and 3.3.4, respectively.

Despite what has been said here it should be noted that the code actually being measured is contained within precompiled libraries and therefore optimisation will not have any large effect on the actual results.

\textsuperscript{14} List of supported extensions found via the /\texttt{proc/cpuinfo} file and cross referenced with product specifications.
### 3.6 Summary of Microbenchmarks Run

<table>
<thead>
<tr>
<th>Type and Filename</th>
<th>Internal Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scheduling</strong></td>
<td>STATIC</td>
<td>Static for loop scheduling clause.</td>
</tr>
<tr>
<td>schedbench.c</td>
<td>STATIC,n</td>
<td>Static for loop scheduling clause with a specified chunksize.</td>
</tr>
<tr>
<td></td>
<td>DYNAMIC</td>
<td>Dynamic for loop scheduling clause.</td>
</tr>
<tr>
<td></td>
<td>GUIDED</td>
<td>Guided for loop scheduling clause.</td>
</tr>
<tr>
<td><strong>Synchronisation</strong></td>
<td>PARALLEL</td>
<td>Parallel construct.</td>
</tr>
<tr>
<td>syncbench.c</td>
<td>FOR</td>
<td>For loop.</td>
</tr>
<tr>
<td></td>
<td>PARALLEL FOR</td>
<td>Parallel for loop.</td>
</tr>
<tr>
<td></td>
<td>PARALLEL + REDUCTION</td>
<td>Parallel construct with reduction clause.</td>
</tr>
<tr>
<td></td>
<td>BARRIER</td>
<td>Barrier directive.</td>
</tr>
<tr>
<td></td>
<td>SINGLE</td>
<td>Single directive.</td>
</tr>
<tr>
<td></td>
<td>CRITICAL</td>
<td>Critical directive.</td>
</tr>
<tr>
<td></td>
<td>LOCK/UNLOCK</td>
<td>Calls to omp_set_lock and omp_unset_lock.</td>
</tr>
<tr>
<td></td>
<td>ATOMIC</td>
<td>Atomic directive.</td>
</tr>
<tr>
<td></td>
<td>ORDERED</td>
<td>Ordered directive with ordered clause on parallel construct.</td>
</tr>
<tr>
<td><strong>Tasks</strong></td>
<td>PARALLEL TASK GEN</td>
<td>Generation of tasks by all threads.</td>
</tr>
<tr>
<td>taskbench.c</td>
<td>SERIAL TASK GEN</td>
<td>Generation of tasks by a single thread.</td>
</tr>
<tr>
<td></td>
<td>PARALLEL TASK GEN TASKWAIT</td>
<td>Generation of tasks by all threads, with a taskwait directive outside the innermost loop.</td>
</tr>
<tr>
<td></td>
<td>PARALLEL TASK GEN BARRIER</td>
<td>Generation of tasks by all threads, with a barrier directive outside the innermost loop.</td>
</tr>
<tr>
<td></td>
<td>PARALLEL TASK GEN TREE BRANCH</td>
<td>Generation of tasks in parallel via recursive binary tree function, work placed on all nodes.</td>
</tr>
<tr>
<td></td>
<td>PARALLEL TASK GEN TREE LEAF</td>
<td>Generation of tasks in parallel via recursive binary tree function, work placed only on leaf nodes.</td>
</tr>
<tr>
<td></td>
<td>IMMEDIATE EXEC EVAL</td>
<td>Task directive with if clause containing ‘0’.</td>
</tr>
<tr>
<td></td>
<td>IMMEDIATE EXEC FUNCTION</td>
<td>Task directive with if clause containing a call to a function returning 0.</td>
</tr>
<tr>
<td></td>
<td>IMMEDIATE EXEC ARG FUNCTION</td>
<td>Task directive with if clause containing a call to a function with an argument, returning 0.</td>
</tr>
</tbody>
</table>
Chapter 4

Results

4.1 Pre-existing Benchmarks

4.1.1 Scheduling

The scheduling benchmarks (schedbench.c in the code) cover all of the scheduling clauses available for determining how loop iterations are allocated in a #pragma omp parallel for or #pragma omp for compiler directive.

Results of this benchmark are presented below for the PGI, Cray, Pathscale and GNU C compilers\(^{15}\), on the HECToR XT4 and XT6 systems and the PGI, Intel\(^ {16} \) and GNU C compilers on the Ness and Ness-GPU systems.

\(^{15}\) pgcc, craycc, pathcc and gcc, respectively.

\(^{16}\) icc
4.1.1.1 Ness

Figure 4 – PGI scheduling overheads (Ness, 16 threads, pgcc 10.0-0).

Figure 5 – Intel scheduling overheads (Ness, 16 threads, icc 11.1).
4.1.1.2 Ness-GPU

Figure 6 – GNU scheduling overheads (Ness, 16 threads, gcc 4.4.3).

Figure 7 – PGI scheduling overheads (Ness-GPU, 8 threads, pgcc 10.0-0).
Figure 8 – Intel scheduling overheads (Ness-GPU, 8 threads, icc 11.1).

Figure 9 – GNU scheduling overheads (Ness-GPU, 8 threads, gcc 4.4.3).
4.1.1.3 HECToR XT4

Figure 10 – PGI scheduling overheads (XT4, 4 threads, pgcc 10.3-0).

Figure 11 – Cray scheduling overheads (XT4, 4 threads, craycc 7.1.6).
Figure 12 – Pathscale scheduling overheads (XT4, 4 threads, pathcc 3.2.99).

Figure 13 – GNU scheduling overheads (XT4, 4 threads, gcc 4.4.3).
4.1.1.4 HECToR XT6

Figure 14 – PGI scheduling overheads (XT6, 24 threads, pgcc 10.3-0).

Figure 15 – Cray scheduling overheads (XT6, 24 threads, craycc 7.2.1.105).
Figure 16 – Pathscale scheduling overheads (XT6, 24 threads, pathcc 3.2.99).

Figure 17 – GNU scheduling overheads (XT6, 24 threads, gcc 4.4.3).
4.1.2 Synchronisation

The synchronisation benchmarks are split into two graphs per system-compiler combination. The first graph shows the results for the directives involving the synchronisation of all threads whilst the second graph is for directives with the purpose of ensuring a section of code is only ever executed by one thread at a time, i.e. directives involving some form of locking.

Results are presented below for the same compilers and systems as for the scheduling benchmarks.

4.1.2.1 Ness

![Graph showing PG synchronous overheads (Ness, pgcc 10.0-0).](image)

Figure 18 – PGI synchronisation overheads (Ness, pgcc 10.0-0).
Figure 19 – Intel synchronisation overheads (Ness, icc 11.1).

Figure 20 – GNU synchronisation overheads (Ness, gcc 4.4.3).
Locking Benchmarks:

Figure 21 – PGI locking overheads (Ness, pgcc 10.0-0).

Figure 22 – Intel locking overheads (Ness, icc 11.1).
4.1.2.2 Ness-GPU

Figure 23 – GNU locking overheads (Ness, gcc 4.4.3).

Figure 24 – PGI synchronisation overheads (Ness-GPU, pgcc 10.0-0).
Figure 25 – Intel synchronisation overheads (Ness-GPU, icc 11.1).

Figure 26 – GNU synchronisation overheads (Ness-GPU, gcc 4.4.3).

Locking Benchmarks
Figure 27 – PGI locking overheads (Ness-GPU, pgcc 10.0-0).

Figure 28 – Intel locking overheads (Ness-GPU, icc 11.1).
4.1.2.3 HECToR XT4

Figure 29 – GNU locking overheads (Ness-GPU, gcc 4.4.3).

Figure 30 – PGI synchronisation overheads (XT4, pgcc 10.3-0).
Figure 31 – Cray synchronisation overheads (XT4, craycc 7.1.6).

Figure 32 – Pathscale synchronisation overheads (XT4, pathcc 3.2.99).
Figure 33 – GNU synchronisation overheads (XT4, gcc 4.4.3).

Figure 34 – PGI locking overheads (XT4, pgcc 10.3-0).
Figure 35 – Cray locking overheads (XT4, craycc 7.1.6).

Figure 36 – Pathscale locking overheads (XT4 3.2.99).
Figure 37 – GNU locking overheads (XT4, gcc 4.4.3).

4.1.2.4 HECToR XT6

Figure 38 – PGI synchronisation overheads (XT6, pgcc 10.3-0).
Figure 39 – Cray synchronisation overheads (XT6, craycc 7.2.1.105).

Figure 40 – Pathscale synchronisation overheads (XT6, pathcc 3.2.99).
Figure 41 – GNU synchronisation overheads (XT6, gcc 4.4.3).

Locking Benchmarks

Figure 42 – PGI locking overheads (XT6, pgcc 10.3-0).
Figure 43 – Cray locking overheads (XT6, craycc 7.2.1.105).

Figure 44 – Pathscale locking overheads (XT6, 3.2.99).
Figure 45 – GNU locking overheads (XT6, gcc 4.4.3).
4.2 New Benchmarks

4.2.1 Parallel, Serial and Tree-structured Task Generation

4.2.1.1 Ness

Figure 46 – PGI task generation overheads (Ness, pgcc 10.0-0).
Figure 47 – Intel task generation overheads (Ness, icc 11.1).

Figure 48 – GNU task generation overheads (Ness, gcc 4.4.3).
4.2.1.2 Ness-GPU

Figure 49 – PGI task generation overheads (Ness-GPU, pgcc 10.0-0).

Figure 50 – Intel task generation overheads (Ness-GPU, icc 11.1).
Figure 51 – GNU task generation overheads (Ness-GPU, gcc 4.4.3).

4.2.1.3 HECToR XT4

Figure 52 – PGI task generation overheads (XT4, pgcc 10.3-0).
Figure 53 – Cray task generation overheads (XT4, craycc 7.1.6).

Figure 54 – GNU task generation overheads (XT4, gcc 4.4.3).
4.2.1.4 HECToR XT6

Figure 55 – PGI task generation overheads (XT6, pgcc 10.3-0).

Figure 56 – Cray task generation overheads (XT6, craycc 7.2.1.105).
4.2.2 Immediate Execution

4.2.2.1 Ness

Figure 57 – GNU task generation overheads (XT6, gcc 4.4.3).

Figure 58 – PGI, Intel and GNU immediate execution overhead (Ness, pgcc 10.0-0, icc 11.1, gcc 4.4.3).
4.2.2.2 Ness-GPU

Figure 59 – PGI, Intel and GNU immediate execution overhead
(Ness-GPU, pgcc 10.0-0, icc 11.1, gcc 4.4.3).

4.2.2.3 HECToR XT4

Figure 60 – PGI, and GNU immediate execution overhead (XT4, pgcc 10.3-0, gcc 4.4.3).
4.2.2.4 HECToR XT6

Figure 61 – PGI, and GNU immediate execution overhead (XT6, pgcc 10.3-0, gcc 4.4.3).
Chapter 5

Analysis

Results for the version 2 and version 3 microbenchmarks have been presented in section 4, including some notes on the observed performance. This section will expand upon these observations and attempt to extract some general trends.

5.1 Scheduling

5.1.1 Static and Static,n

On the whole the static schedules have the lowest overhead for each compiler on all four test platforms, as would be expected (see section 11). On Ness the static performance of all implementations appears to be equivalent, with an overhead of around 15 000 clock cycles. This situation is repeated on Ness-GPU albeit with a significantly reduced overhead of approximately 2 600 cycles, making Ness-GPU incur one sixth of the overhead of Ness for this schedule. This performance difference between platforms may be explained by the fact Ness-GPU is using less threads (8 vs. 16) and therefore has less work to allocate, coupled with the fact that it has four cores per CPU versus two cores per CPU on Ness, leading to decreased communication costs between threads (less need to make off-chip data accesses). On the HECToR XT4 system there is more variability between implementations: the PGI and Pathscale compilers are similar at around 1 500 cycles whilst the Cray compiler achieves 900 cycles. The performance increase of the PGI implementation relative to Ness-GPU may once more be credited to a reduced number of threads, the XT4 having the same number of threads per CPU as Ness-GPU. Strangely the GNU compiler does not follow suit and there is a massive performance decrease on the XT4, with an overhead around 12 000 cycles (figure 13). Given that there is no hardware reason for this it can only be assumed that there is some kind of software bug causing this performance regression. Of note is the fact that the guided schedule actually manages to outperform static in this case. On the XT6 system the performance of the PGI compiler is actually faster than that on Ness, despite the increase from 16 to 24 threads. This indicates that the more tightly coupled CPU architecture of the XT6 (figure 14) is of great benefit in this area. The Cray compiler is around 600 cycles slower than PGI, showing room for improvement. Both the GNU and Pathscale compilers exhibit similar behaviour to that of GNU on the XT4, but with even greater detrimental impact. Each compiler is an order of magnitude slower than the PGI and Cray implementations. It is speculated that some aspect of their design is preventing proper utilisation of the hardware, considering that each compiler has shown that is has the ability to perform reasonably well on other platforms.
The difference between static and static,n is most readily visible on HECToR XT4; overhead increases as the chunksize decreases due to an increased need for threads to switch between chunks.

5.1.2 Dynamic and Guided

The dynamic schedule overhead follows a similar pattern with each of the compilers; overhead decreases steadily with chunksize and begins to level off towards the point at which chunksize = 128 is reached. Due to the way the code is written a chunksize of 128 is the point at which each thread is allocated one chunk, i.e. the distribution of iterations to threads is more or less identical to the static schedule (threads are not guaranteed to be allocated chunks in order of thread ID). This means that at a chunksize of 128 we observe the minimum cost of the schedule by definition. Looking at the results from Ness it is clear that the Intel compiler has the best implementation with a maximum overhead of 600 000 for a chunksize of 1 and a minimum overhead of 20 000 for a chunksize of 128. In comparison the GNU implementation runs from 450 000 to 4 000 and the PGI implementation from over 2 000 000 to 90 000. Thus both the GNU and in particular PGI dynamic implementations could do with being improved. On Ness-GPU the results are similar but with the overheads much reduced overall.

As previously discussed in the description of the guided schedule it is expected that its performance should generally be superior to that of the dynamic schedule due to the fact that it results in fewer chunks to allocate in total.

Examining the Ness and Ness-GPU data it is immediately apparent that the Intel guided schedule is significantly worse than PGI and GNU with roughly twice the overhead. On Ness this makes guided incur more overhead than the dynamic schedule for all chunksizes. Of the PGI and GNU implementations GNU is slightly faster.

On the XT4 system dynamic and guided performance is improved from that on Ness-GPU, most likely due to the reduced number of threads creating reduced demand on the mechanism for chunk allocation; rather than the change of CPU, which has been observed to be worse than Ness-GPU in several of the other benchmarks. Out of the different compilers available on HECToR the Pathscale and Cray compilers have similar guided performance whilst the Cray implementation of guided is faster for larger chunksizes. Once again the PGI compiler lags behind the others in this test with a larger amount of overhead in both the guided and dynamic schedules. As observed for the static benchmarks the GNU implementation curiously suffers on the XT4 system and at a chunksize of around 8 the guided and dynamic schedules actually begin increasing in overhead where they converge with the very poorly performing static schedules. This cannot reasonably be explained without viewing the source code but suffice to say the implementation seems to have an issue that is for some reason manifested on HECToR and not on either of the Ness systems. On the XT6 the Pathscale compiler exhibits similar behaviour, with the performance of guided and dynamic being comparable to other compilers until a chunksize of around 8, at which point performance decreases and levels off. The PGI compiler on the XT6 node is consistent with results from the other platforms with results very similar to those on Ness, leaving the Cray compiler as the only reasonable implementation of dynamic for chunksizes beyond 8.
5.2 Synchronisation

5.2.1 Parallel, For, Barrier and Single

The parallel and parallel for directives are highly coupled in most implementations, indicating that no significant overhead is induced by adding a for directive to a parallel directive. This can be explained by the fact that the for and barrier directives are also highly coupled; the overhead of the for directive lies almost exclusively in the implicit barrier at the end of the region and therefore, since the parallel construct also contains an implicit barrier, when the for and parallel directives are combined into a parallel for they may share the implicit barrier, resulting in the observed relationship.

Strangely, the single directive manages to have less overhead than the barrier directive under the PGI implementation as well as for some sections of other implementations. As described in section 4.1.2, this should theoretically be impossible due to the single directive containing an implied barrier at its exit.

The PGI compiler consistently achieves the best overall performance in these microbenchmarks, with the familiar pattern of improved performance as the number of cores per processor is increased. The change from Ness to Ness-GPU for instance yields an approximately 8 000 cycle decrease in overhead in the parallel and parallel for directives for 8 threads.

One striking result of this set of benchmarks is the behaviour of the Pathscale compiler on the XT6 (figure 40). There is an incredibly pronounced increase in overhead that is very clearly associated with the move from one Magny-Cours CPU to two. Some aspect of the tuning of the optimisation must not account for the increased communication cost of data moving to the second chip.

5.2.2 Locking

The critical directive and the lock/unlock runtime routines appear to be very tightly coupled in the majority of tests performed, indicating that most compilers use an identical underlying implementation, which is not surprising. The only compiler where the two differ significantly appears to be Pathscale, which is particularly visible on the XT6 graph (graph 44). The PGI compiler has the best implementation of both critical and lock/unlock pairs with overheads usually many times less than other implementations across all platforms, other than the Cray compiler which has a similarly performing implementation on the XT4 and XT6.

For the atomic directive, the PGI compiler once again has a significantly faster implementation than other compilers. The Cray compiler has similar performance but is slower by a small degree. On the XT6 system the PGI compiler’s atomic overhead seems to level off until around 6 threads, at which point overhead steadily rises with thread count. This is no doubt due to the architecture of the system, with 6 cores per NUMA region as discussed in section 3.3.4. It should be noted that the overhead on the XT6 is less than that on Ness and Ness-GPU for the same number of threads (XT4 performance is approximately equivalent).
This must be due to the reduced inter-core communication times resulting from the greater number of cores per chip.

In almost every benchmark performed, the ordered construct has a much greater overhead than the other directives. Whether or not this is inherent to the directive itself or whether it is down to the fact that it is one of the less commonly used and therefore less well optimised directives is unclear. The GNU implementation of the directive appears to be different to the others in that it scales very well; as the number of threads increases there is very little change in the overhead of the directive. While this makes the GNU implementation the best performing on Ness, the Pathscale compiler seems to have the superior implementation overall since the overhead is much lower, particularly on the XT6 system, where the GNU implementation is the poorest on test. On the XT4 and Ness-GPU systems the GNU implementation seems to fare worse than others perhaps due to the small number of physical CPUs (1 on the XT4 and 2 on the Ness-GPU system) giving rise to negligible negative scaling effects for other compilers, indeed, the GNU compiler performs worse than others on Ness as well for low numbers of threads. The PGI, Cray and Intel implementations appear to be similar in performance, doing well on the XT4 and Ness-GPU systems whilst performing poorly on the Ness and XT6 systems.

5.3 Tasks

5.3.1 Task Generation

For all compilers except PGI, serial task generation incurred more overhead than parallel task generation, as might be expected (section 3.1.1). Serial task generation ranged between 10 times and 100 times worse than standard parallel task generation. As with the other benchmarks, performance increased when moving to a hardware platform with a greater number of cores per processor.

The performance of the recursive tree structured task generation loops varies by an extremely large amount between compilers. On the XT6 for example, performance of the two benchmarks converges on that of serial task generation under the Cray compiler, ending up with an overhead of approximately $1 \times 10^7$ for 24 threads, with the GNU implementation having an overhead in between the serial and parallel task generation loops with an overhead of just over $1 \times 10^6$. Finally the PGI implementation is an order of magnitude faster still with an overhead below $1 \times 10^5$. In all of these cases placing work at the leaf nodes of the tree rather than at the leaves as well as branches appears to give a significant reduction in overhead, despite the fact that the trees containing only work at leaf node contains one extra level of tasks as illustrated in figure 2. It is believed that this is caused by increased speed in generating the entire tree structure in the leaf-node-only case as this makes all work items available for processing sooner – when work is placed on branch nodes that work must be completed before generating the subtree belonging to the branch, potentially leaving some threads idle if one side of the tree has been completed and not the other.

Whereas the PGI implementation seems to be extremely poor with respect to regular parallel task generation, it excels when applied to the tree-based benchmarks. Upon first consideration this does not appear to make sense, generating the tree is quite similar to parallel loop task generation since all threads are working to generate tasks simultaneously,
and so the PGI implementation should not be able have such a low overhead compared with the parallel loop benchmark. Furthermore, intuitively the tree-based benchmark would be expected to have greater overhead since it is like the parallel loop but with additional dependencies between tasks. Despite the fact that it is not possible to say for certain due to the nature of the benchmark as a black-box test, one possible explanation is that the internal data structure of the task system on the PGI compiler is itself some form of tree structure and therefore the pattern of dependencies between tasks in the benchmark meshes well with this internal representation, resulting in fast data access when evaluating tasks for execution. This could also potentially explain the poor performance of the other task generation benchmarks with the PGI compiler; parallel task generation may end up being stored internally as a tree where every task generated is a leaf, which would be inefficient compared with some simpler data structures. When thought of more carefully, internally representing outstanding tasks and their relationships to each other as a tree makes sense, the primary reason that tasks were added to OpenMP was to enable that parallelisation of programs with more complex execution patterns than simple loops, with the recursive Fibonacci function (listing 6) as a good example. It therefore makes sense to optimise for this case even if there is a performance penalty with parallel task generation.

As a general rule GNU was observed to be the fastest compiler for parallel task generation, while the PGI work-on-leaves recursive tree was the fastest function overall on each test.

The purpose of benchmarking parallel task generation loops with inserted barrier and taskwait directives was to attempt to compare the cost of these directives in a context where they are almost semantically identical. Originally the plan was to subtract the overhead time of the regular parallel task generation loop (listing 12) from the overhead time of both of these loops (listing 13) in order to determine separate values for the overhead of the barrier and taskwait directives in this context. After examining the results however it is clear that this is not valid; in some instances the code containing a taskwait is actually faster than the code only containing an implicit barrier, so in this case it would not make sense to subtract one from the other to derive a time for the taskwait directive. Such behaviour is a result of the fact that there is much room for implementations to vary significantly in execution paths when making decisions as to when to execute tasks when entering scheduling points. Despite this fact these benchmarks still provide interesting information about how the implementation of tasks differs between compilers.

On the PGI compiler both the barrier and taskwait loops are more or less equivalent in overhead to the standard parallel task generation loop, this can be seen on figures 46, 49, 52 and 55. The fact that both barrier and taskwait directives contain task scheduling points means that any overhead they introduce can be masked by the fact that threads in these constructs may continue to work on outstanding tasks. This masking of overhead is compounded as if a particular thread is behind the others then tasks that it creates will be executed by the threads waiting for it, effectively nullifying the fact that they are waiting. It is possible that differences between these three parallel generation loops would be exaggerated if the number of tasks was reduced in favour of fewer tasks containing more work, in which case not all threads waiting in a barrier may be able to be assigned a task.

With both the Intel and GNU compilers the taskwait and barrier loops follow a similar path to either the performance of the serial task generation code in the case of GNU, or the parallel
task generation code, in the case of Intel; though each has a definite offset from the loop being followed. This is particularly clear in figure 48, the performance of the taskwait loop almost perfectly follows that of the serial generation loop, though maintaining the same distance beneath it (an increasing number of clock cycles since this is a logarithmic plot). This implies that for the taskwait loop on the GNU compiler the execution pattern is characteristically similar to that of the serial loop. On the other hand the Intel implementation ends up with these loops more closely resembling the parallel case, this is clearest in figure 47.

In figures 53 and 56 it is clear that the Cray compiler has more overhead for the taskwait directive than it does for the barrier. As was discussed in section 3.1.2 this should not be the case; a taskwait does not require all threads to reach it before threads waiting there can leave and so there should be a slightly less overhead than a barrier. With the barrier having greater performance there is in fact no reason to use a taskwait directive. Clearly there is some scope for improvement of the taskwait directive in the Cray implementation, especially seeing as the Intel compiler manages to achieve the expected behaviour.

5.3.2 Immediate Execution

On Ness all compilers show a gradual increase in the overhead associated with immediate execution as the thread count increases. There are significant gaps in performance between compilers with GNU being fastest and Intel performing very poorly compared to the others.

As has been seen with other benchmarks the overheads on Ness-GPU are much lower than those on Ness. While the ranking of the compilers remains the same (GNU fastest, Intel slowest), the trends are different to those observed on Ness; the Intel and PGI compilers do not increase in overhead with the number of threads and the GNU compiler has peaks in overhead at 3 and 6 threads (figure 59). Furthermore, all implementations see a drop in overhead when the maximum number of threads is reached. Although it is not clear what could cause these peaks to appear in the GNU implementation, the rise in overhead between 4 and 5 threads may be a result of the move to two processors, poorer scaling of the GNU implementation compared with other compilers was also observed in the task generation benchmarks – see figure 48. It should, however, be noted that the GNU implementation is extremely efficient on one processor, with overheads below 100 cycles.

On both HECToR XT4 and XT6 the Cray compiler cannot be tested as previously mentioned in section 3.5. On the XT4 the PGI compiler performs similarly to on Ness, with a gradual increase in overhead with thread count; performance is in fact slightly worse than on Ness-GPU which is understandable given that the Intel Nehalem chip on Ness-GPU is more sophisticated than the Opteron on the XT4 system. Surprisingly the GNU implementation encounters significant performance issues on the XT4 with overhead more than ten times greater than 1-4 threads on Ness-GPU and also worse performance than that observed on Ness. For the XT6 GNU performance is similar to the XT4, however overhead steadily increases with thread count, ending up with over double the overhead of the PGI compiler at 24 threads.

A number of things can be discerned from this data. The Intel compiler performs very poorly in comparison with other implementations in this test and would benefit from optimisation
efforts in this area. To put the overhead in perspective immediate execution on the Intel compiler is slightly higher than for a barrier on Ness and significantly slower than a barrier on Ness-GPU. Though the PGI implementation isn’t the fastest, it performs reliably, with similar amount of overhead on all platforms used; this contrasts with the GNU compiler which is significantly faster in most cases but with more variable overhead levels on the same machine and between machines, with poor performance on the XT6. If making frequent use of \( i f \) clauses on tasks it is recommended that the Intel compiler is avoided and that it is kept in mind that overhead is usually a few thousand clock cycles when evaluating whether or not having a conditional task is worthwhile. Of course, the Cray compiler is simply not an option at all since it does not implement task \( i f \) clauses.

No significant differences are observed between the three variations on the benchmark so compiler optimisation has not dramatically altered the test. The PGI compiler has more overhead for the variant with a function with an argument, which is to be expected as there is a small amount of extra code compared with the other two variants.
Chapter 6

Conclusions

Several new benchmarks have successfully been written as part of the ongoing effort to keep the EPCC OpenMP Microbenchmark suite up to date and therefore relevant to modern compilers and parallel computing. In addition to this the full range of benchmarks, including those pre-existing, were run on four different hardware platforms and with a range of compilers. From these results a number of interesting pieces of information were extracted.

The PGI compiler, which is one of the most use compilers in the scientific computing community, has the least amount of overhead for many directives out of those compilers on test. However there are several weaknesses that users should be wary of. The dynamic loop scheduling directive has large amounts of overhead compared with other compilers, as well as the guided loop schedule.

Having many cores on the same chip offers many performance gains in a variety of benchmarks over systems with an equivalent number of cores but distributed over more physical CPUs. The HECToR XT6 system is a prime example of this, while the Ness-GPU system also demonstrated solid performance gains over Ness. It should be noted however that occasionally a compiler providing good performance on one machine may have trouble scaling with a very large number of cores, notably the GNU compiler.

Methods for generating large numbers of tasks efficiently were investigated with the new task generation benchmarks. It was found that for most compilers generating tasks in parallel with all available threads produced the least amount of overhead. It is speculated that for the PGI compiler, which performs poorly in standard parallel task generation, the internal representation of incomplete tasks may resemble a tree structure, rendering it extremely fast at processing tasks created by recursive algorithms, which are quite probably the primary use case for tasks.

Measuring the overhead introduced by the immediate execution of tasks produced two interesting results. Firstly, it is possible to for this process to be quite efficient as evidenced by the GNU compiler, while some implementations have unacceptable amounts of overhead introduced (Intel), indicating that some optimisation effort would be worthwhile in order to make the if clause truly useful. Secondly, despite the fact that the Cray compiler is supposed to support OpenMP 3.0, it is missing this feature.
In addition to the findings outlined here the tests that have been run also illustrate more minor differences between compilers that suggest areas for improvement in those compilers with higher overheads.
Chapter 7

References

[3] Ibid. p8
[4] Ibid. p182
[8] OpenMP API Specification, OpenMP ARB. p43
[9] Ibid. p34
[10] Ibid. p215
[14] HECToR Phase 2b Documentation. http://www.hector.ac.uk/cse/documentation/Phase2b/#arch
Appendix A

A.1 Batch System Job Submission Scripts

A.1.1 Ness

#!/bin/bash
#$-cwd
#$-V
#$-l h_rt=0:30:0
#$-pe omp 16

binary=taskbench #Choose from syncbench, schedbench, taskbench
compiler=pgcc    #e.g. pgcc, gcc, icc
queue=ness-gpu   #Choose ‘ness’ or ‘ness-gpu’

mkdir -p ./results-final/$binary/$queue/$compiler/

echo "=== BINARY:     $(md5sum $binary)==="
echo "=== COMPILER:   $compiler ==="
echo "=== QUEUE:      $queue ==="
echo "=== START TIME: $(date) ==="

#for threadcount in 1 2 4 6 8 10 12 14 16; do
#    echo " === Running with $threadcount threads.. ==="
#    for run in $(seq 1 1 20); do
#        echo " === Run $run ==="
#        OMP_NUM_THREADS=$threadcount ./$binary > ./results-final/$binary/$queue/$compiler-threads:$threadcount-run:$run-date:$ (date +%d-%m-%y).out
#done

echo "===           D O N E               ==="
echo "=== END TIME:   $(date) ==="
#!/bin/bash
--
#PBS -N ompmb3
#PBS -l mppwidth=24
#PBS -l mppnppn=24
#PBS -l walltime=00:30:00
#PBS -A <RESOURCE BUDGET CODE>

binary=taskbench
compiler=craycc-long
queue=hector2b

# Necessary for correct thread binding when using the pathscale
# compiler (harmless if not).
export PSC_OMP_AFFINITY=FALSE
# Work in the correct directory.
cd $PBS_O_WORKDIR

# Ensure directory exists for storing output data.
mkdir -p ./results-final/$binary/$queue/$compiler/

echo "=== BIN: $binary ($(md5sum $binary)) ==="
echo "=== COMPILER: $compiler ==="
echo "=== QUEUE: $queue ==="
echo "=== START TIME: $(date) ==="
for threadcount in 1 2 3 6 9 12 15 18 21 24; do
    export OMP_NUM_THREADS=$threadcount
    echo "  === Running with $threadcount threads.. ==="
    for run in $(seq 1 1 20); do
        echo "    === Run $run ==="
        aprun -n 1 -N 1 -d $OMP_NUM_THREADS ./binary > ./results-final/$binary/$queue/$compiler/$binary-threads:$threadcount-run:$run-date:$(date +%d-%m-%y).out
done
    done
echo "=== DONE ==="
echo "=== END TIME: $(date) ==="