Exam: B188633
Date: April 10, 2023

Parallel Implementations of Adaptive Quadrature using OpenMP

Course: Threaded Programming
Course Organiser: Dr Mark Bull
Assessment: 2

THE UNIVERSITY *of* EDINBURGH

Table of Contents

1. Introduction

Threaded Programming can be an efficient way to parallelise scientific computing applications. Our goal in this study is to implement two versions of a divide-and-conquer algorithm in OpenMP, an application programming interface (API) that supports multi-platform shared-memory parallel programming [1].

We are provided with two codes (developed in the C programming language), which implement the same algorithm in two different ways. The algorithm is an **adaptive quadrature** method that computes the integral of a function on a closed interval using a divide-and-conquer method [2]. The algorithm starts by applying two quadrature rules (3-point and 5-point Simpson's rules) to the whole interval [2]. If the difference between the integral estimates from the two rules is small enough (or the interval is too short), the result is added to the total integral estimate [2]. If it is not small enough, the interval is split into two equal halves, and the method is applied recursively to each half [2]. In the case supplied, evaluating the function requires the solution of an ordinary differential equation (ODE), which is relatively expensive in time [2].

We make all our codes in this study available with the associated code repository [3]. As supplied, the sequential code in **solver1.c** implements the algorithm using recursive function calls [3]. The sequential code in **section2.c** implements the algorithm using a last-in-first-out (LIFO) queue [3].

2. Parallel Implementations of the Algorithm

In this section, we describe the parallel implementations of the algorithm in three different versions. The first version, called Parallel Solver 1, parallelises the **solver1.c** sequential code [3]. The other two versions, Parallel Solver 2 and Parallel Solver 3, parallelise the **solver2.c** sequential code with two different synchronisation techniques [3].

2.1. Parallel Solver 1 with Task Constructs

This version parallelises the **solver1.c** sequential code, which uses recursive function calls. We use OpenMP task constructs to parallelise the code.

Our strategy here was to enclose the call to **simpson()** function (which uses recursion) in the **main()** function in a parallel region (refer to Code 1). Inside the parallel region, we only allow the **master** thread to package up the code block and its data for execution. All the other threads in the parallel region are available to execute the tasks. This strategy lends itself to **orphaned** directives. This is useful in our context as we can retain the modular programming style of the code, and we do not need to change the design of the sequential code significantly to parallelise it.

There is an implicit synchronisation (with an implicit **barrier**) at the end of the parallel region. At the end of the parallel region, the master thread waits for the other threads to finish and continues executing the next statements.

```
// call recursive quadrature routine
// create parallel region using orphaned directives
#pragma omp parallel
{
    // only master thread packages tasks
    #pragma omp master
    quad = simpson(func1, whole);
}
```

Code 1: Code extract showing usage of parallel region and master directives [3].

Inside the simpson() function, we create two tasks at every level (refer to Code 2). Each task contains one of the recursive calls to simpson(). We synchronise here using the **taskwait** directive. Each task must complete before they can be added together and returned. Although taskwait is a shallow wait (i.e., it only waits for its children), it is by implication a deep wait in this case because of the recursive structure. As the **quad1** and **quad2** variables are local variables, they are private to the current task. This is why we make them shared on child tasks, so they do not create their own **firstprivate** copies at this level.

```
// represent recursive calls to simpson() as tasks
#pragma omp task shared(quad1)
quad1 = simpson(func, i1);

#pragma omp task shared(quad2)
quad2 = simpson(func, i2);

#pragma omp taskwait
return quad1 + quad2;
```

Code 2: Code extract showing usage of task and taskwait directives [3].

2.2. Parallel Solver 2 with Critical Sections

This version parallelises the **solver2.c** sequential code, which uses a LIFO queue. We parallelise this version without using task constructs. Instead, we enclose the do while loop (inside the simpson() function) in a **parallel** region and allow all threads to enqueue and dequeue intervals (refer to Code 3).

The termination condition for the loop (i.e., an empty queue) is not sufficient in the parallel version, as the queue might be empty, but another thread may subsequently enqueue new intervals. The do while loop in the parallel version terminates only when the queue is empty and there are no intervals currently being processed. We achieve this by defining a new shared variable called **active_intervals**, which is incremented by a thread only after a successful dequeue operation from the queue, and decremented by a thread only after the tolerance criteria are met and the quadrature is increased.

```
// enclose do while loop in parallel region and
// allow all threads to enqueue and dequeue intervals
#pragma omp parallel default(none) \
shared(queue_p, active_intervals, quad, func)
```

Code 3: Code extract showing usage of parallel region directive enclosing the do while loop [3].

To avoid race conditions, we synchronise accesses to the shared variables with **critical** directives. Code 4 shows an example where the access to the queue is synchronised with **#pragma omp critical** directive. Both dequeue and enqueue operations to the queue have been synchronised. Similarly, we synchronise accesses to the **active_intervals** and **quad** variables.

```
// synchronise accesses to queue with critical section
#pragma omp critical
{
    if (!isempty(queue_p))
    {
        interval = dequeue(queue_p);
    }
}
```

Code 4: Code extract showing usage of critical directive [3].

2.3. Parallel Solver 3 with Lock Routines

This code parallelises the sequential code **solver2.c**, which uses a LIFO queue. The difference between this parallel version and the Parallel Solver 2 version is that instead of using critical sections to

synchronise accesses to shared variables, we use multiple (blocking) **locks**. Lock provides more flexibility than is provided by critical directive. So, we wanted to experiment with locks to observe whether they improve performance of the application.

We create and initialise three locks for three shared variables (i.e., **queue_p**, **active_intervals**, and **quad**). We set and unset the locks around places where these shared variables are updated. Outside the loop, we destroy the locks to free up memory resources. Code 5 shows the same dequeue operation from Code 4. The difference here is the usage of lock routines. We set the specific lock for the queue, then dequeue, and then unset the specific lock for the queue. We follow a similar process for the active_intervals and quad shared variables.

```
// synchronise accesses to queue with lock
omp_set_lock(&lock_queue_p);
if (!isempty(queue_p))
{
    interval = dequeue(queue_p);
}
omp_unset_lock(&lock_queue_p);
```

Code 5: Code extract showing usage of lock routines [3].

3. Discussion of Results

In this section, we discuss the hardware and software environments used in the study, analyse performance of the parallel implementations, and recommend optimisations for future studies.

3.1. Environment

We run the experiments on the standard compute (also called backend) nodes of the Cirrus HPC system [4]. We only use CPUs and we do not use any GPUs in this study. The hardware details of Cirrus are provided in Table 1. As OpenMP is a shared memory programming model, we only use one node per run.

We use the **Intel 20.4** compiler and compile the code with the **-O3 -qopenmp** flags to ensure a high level of sequential optimisation and to enable OpenMP. Also, we test our implementations with the **GNU 10.2** compiler with the **-O3 -fopenmp** flags. We run the codes in exclusive mode (i.e., by specifying **#SBATCH --exclusive** in the Slurm job script) on the compute nodes to ensure that other jobs cannot affect performance.

	Compute nodes
Nodes	*280 standard nodes*
CPUs	*2 Intel Xeon E5-2695 (per node)*
Cores per CPU	*18*
CPU core clock	*2.1 GHz*
Memory	*256 GB (2x128) main memory*
File system	*Lustre*

Table 1: Hardware details of the standard compute nodes of Cirrus [5].

The output of any code is only deemed correct if the **Result = 6.019096e-03** for any variant and thread configuration of the code. This is the default result that we observed when running the sequential codes. We confirm that all our variants and thread configurations return correct results, and we make the correctness testing results available in Appendix A.

We run the parallel implementations across several thread counts (i.e., 1, 2, 4, 6, 8, 12, 16, 24, and 32), and we run each thread configuration 3 times to determine mean execution time. We report the mean execution times in Appendix B. To observe all the raw execution times of all variants and configurations, please refer to the Excel file titled **coursework2_data.xlsx** in the provided code repository [3].

3.2. Performance Analysis

We plot speedups of the parallel implementations in Figure 1 and Figure 2 for the Intel and the GNU compilers respectively. Please note that we have used the execution times of the sequential codes as our baselines to compute speedups instead of the execution times of the parallel codes with a single thread. This is because having parallel regions (even if running with 1 thread) incur some overheads (refer to Appendix B). So, we want to measure how much faster our parallel implementations are compared to the sequential versions.

Across both Intel and GNU compilers, we observe that Parallel Solver 1 yields the best speedup, followed by Solver 3, and then Solver 2. Solver 1 benefits from the usage of task constructs. Tasks are independent units of work. They are composed of a block of code to execute, and data to compute with. We have per-task private copies of variables. After a thread completes a task, it is then free to work on another task. This is a beneficial design as the threads can keep working on tasks with little idle time. As the code has some irregular computations, using task constructs are helpful in this case because the runtime takes care of load balancing [6].

In Solver 1, there is not much synchronisation overhead as we only synchronise with the **taskwait** directive before the values of **quad1** and **quad2** variables are added and returned. On the other hand, there is relatively more synchronisation involved in Parallel Solver 2 and in Parallel Solver 3. In Solver 2,

we synchronise accesses to the queue, the active intervals counter, and the quadrature with critical sections. Threads can be idle waiting to access a critical section. In OpenMP, critical regions behave like a global lock [6]. Due to these synchronisations, parallelism is reduced. Hence, we observe limited speedup.

In Solver 3, we use multiple locks. This improves scalability slightly compared to the one observed with Solver 2 as we reduce the time spent in critical sections. However, the scalability is also limited here due to the synchronisation overheads of the shared variables. Every time we synchronise threads, there is some overhead, even if the threads are never idle [6].

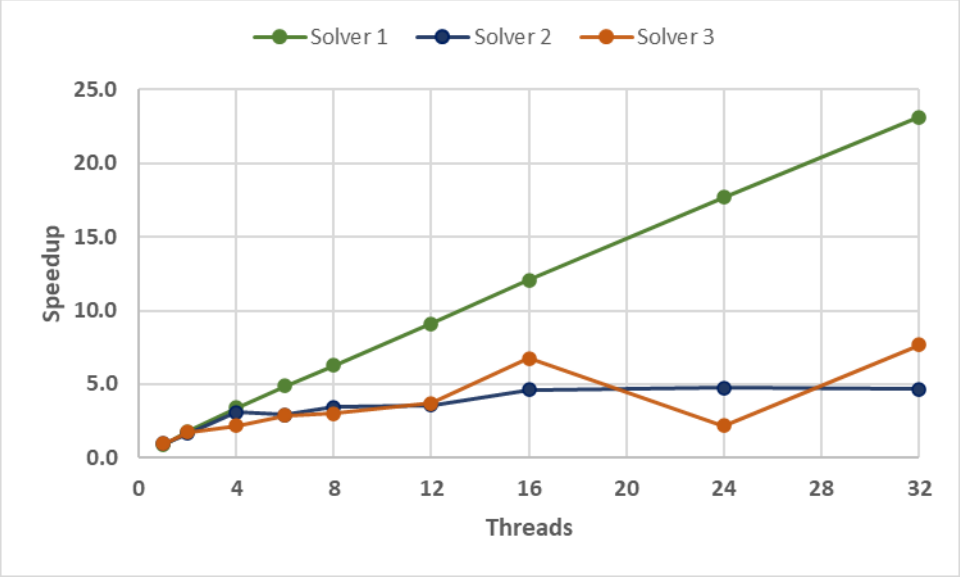

Figure 1: Speedups of the parallel implementations with **Intel 20.4 compiler** in a standard compute node of Cirrus.

We observe some NUMA effects (i.e., when increasing thread count from 16 to 24), especially with Solver 3 in the Intel compiler and Solver 1 in the GNU compiler. As Cirrus nodes are multi-socket systems, the location of the data in main memory is vital. The default policy for the operating system (OS) is to place data on the processor which first accesses it (i.e., first touch policy) [6]. This is a bottleneck for OpenMP programs. As the data is initialised in the master thread, it is all allocated on one processor [6]. Having all threads access data on the same processor reduces performance.

There is a possibility of cache space contention. As the L3 cache is shared on a Cirrus node, the codes may not scale well because a single core can access much of the shared cache.

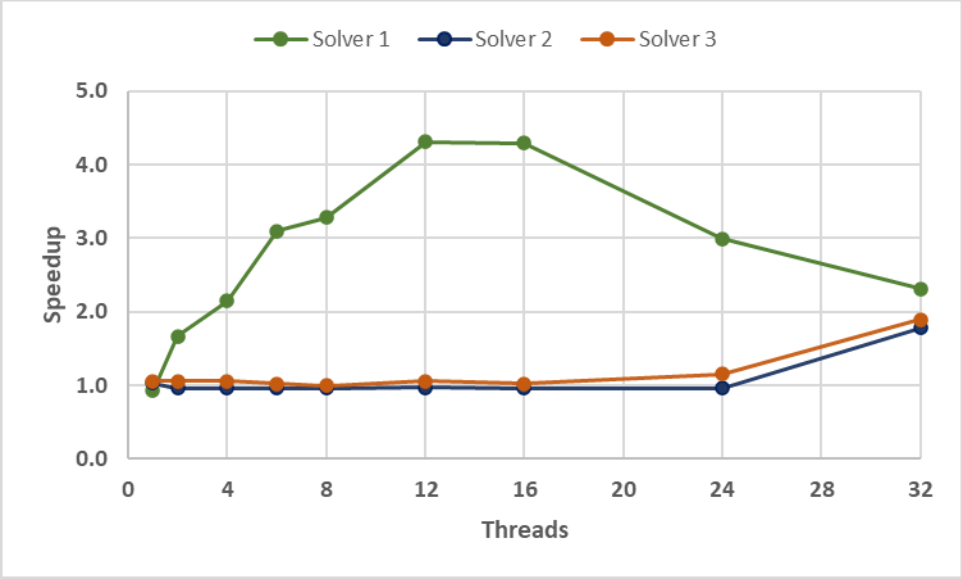

Figure 2: Speedups of the parallel implementations with **GNU 10.2 compiler** in a standard compute node of Cirrus.

We observe significant differences in speedups across the two compilers. The speedups observed with the Intel compiler are significantly higher than those observed with the GNU compiler. As the Intel compiler is optimised for Intel hardware (such as Cirrus), we observe higher performance with the Intel compiler in our case. **OpenMP 4.5** is fully supported in both Intel and GNU compilers [7]. So, the difference in performance is not due to the compilers using different OpenMP versions.

Even though we use **-O3** optimisation level across both compilers, it could be the case that the Intel compiler is performing more aggressive optimisations to the codes. For instance, Intel compiler performs compiler intrinsics, function inlining, loop unrolling, loop and memory access transformation, prefetching, and scalar replacement at the -O3 level [8]. GNU compiler performs function inlining and loop unrolling at the -O3 level [8]. Some of these additional optimisations performed by the Intel compiler could be improving the performance of the codes.

3.3. Review

In this subsection, we discuss some ways we could optimise the parallel implementations further in future studies. Although Parallel Solver 1 yields the best speedup across all the three implementations, we can try optimising the performance of it further. When we create tasks, we rely on the runtime to assign these to threads. This incurs some overheads (e.g., internal synchronisation in the runtime) [6]. We can try experimenting with the granularity of tasks, as too big tasks can result in idle threads or too small tasks can result in scheduling overheads [6]. In our case, we can try limiting the recursion depth at which tasks are created.

In Solver 2 and in Solver 3, we can experiment with multiple queues. This may reduce the load on the single queue used in those versions and may return better performance. Also, we could use atomics (instead of critical sections or locks) in places where they are applicable. Atomics may have less overhead than critical sections or locks [9]. But we must be careful using atomics as it has a quality of implementation problem [6]. For instance, there is no mutual exclusion between critical and atomic directives [9]. We cannot protect updates to shared variables in one place with atomic directive and another with critical directive if they might contend [6].

Placement of threads on the hardware can have a considerable effect on performance [6]. We could try reducing some of the NUMA effects. In some operating systems, there are options to control data placement. For instance, in Linux, we can use **numactl** to change policy to round-robin (instead of first touch) [6]. Also, the first touch policy can be used to control data placement indirectly by parallelising data initialisation [6].

4. Conclusion

Our goal in this study was to parallelise two versions of a divide-and-conquer algorithm in OpenMP. We developed three parallel implementations (one parallel implementation for **solver1.c** and two parallel implementations for **solver2.c**) each using different OpenMP techniques (i.e., task constructs, critical sections, and lock routines), and analysed scalability of the implementations.

We observed that task constructs applied to the sequential version with recursion yields the best speedup compared to the other two implementations, which apply a parallel region on the main loop (with two different synchronisation techniques). The lower synchronisation overhead of the task constructs in Parallel Solver 1 limits scalability of the code relatively less than those observed with the other versions (with more synchronisations).

We discussed some recommendations to optimise the implementations in future studies. All our code and experimental data have been made available in the associated code repository of this study [3].

References

[1] OpenMP, OpenMP, 14-Mar-2023. [Online]. Available: https://www.openmp.org/. [Accessed: 10-Apr-2023].

[2] EPCC, “Coursework for Threaded Programming Part 2.” EPCC, Edinburgh.

[3] B188633, “Parallel Implementations of Adaptive Quadrature using OpenMP (Code).” The University of Edinburgh.

[4] EPCC, Cirrus. [Online]. Available: https://cirrus.readthedocs.io/en/main/. [Accessed: 10-Apr-2023].

[5] EPCC, “Cirrus Hardware,” Cirrus. [Online]. Available: https://www.cirrus.ac.uk/about/hardware.html. [Accessed: 10-Apr-2023].

[6] EPCC, “OpenMP Performance,” Archer. [Online]. Available: http://www.archer.ac.uk/training/course-material/2019/07/sgl-node-imp/L06-openmp.pdf. [Accessed: 10-Apr-2023].

[7] OpenMP, “OpenMP Compilers & Tools,” OpenMP, 22-Nov-2022. [Online]. Available: https://www.openmp.org/resources/openmp-compilers-tools/. [Accessed: 10-Apr-2023].

[8] B188633, “Optimising Molecular Dynamics Application with the Compiler on HPC System.” The University of Edinburgh, Edinburgh, 28-Oct-2022.

[9] A. Duran, “Parallel programming with OpenMP,” University of Washington. [Online]. Available: https://courses.cs.washington.edu/courses/csep524/13wi/omp_tutorial_full.pdf. [Accessed: 10-Apr-2023].

Appendix A: Correctness Data

In this appendix, we include the correctness testing data from all configurations run in our experiments.

Compiler	***Intel***			***GNU***		
Threads	***Solver 1***	***Solver 2***	***Solver 3***	***Solver 1***	***Solver 2***	***Solver 3***
1	*pass*	*pass*	*pass*	*pass*	*pass*	*pass*

Table 2: Correctness testing results of the sequential codes across Intel and GNU compilers.

Compiler	***Intel***			***GNU***		
Threads	***Solver 1***	***Solver 2***	***Solver 3***	***Solver 1***	***Solver 2***	***Solver 3***
1	*pass*	*pass*	*pass*	*pass*	*pass*	*pass*
2	*pass*	*pass*	*pass*	*pass*	*pass*	*pass*
4	*pass*	*pass*	*pass*	*pass*	*pass*	*pass*

6	*pass*	*pass*	*pass*	*pass*	*pass*	*pass*
8	*pass*	*pass*	*pass*	*pass*	*pass*	*pass*
12	*pass*	*pass*	*pass*	*pass*	*pass*	*pass*
16	*pass*	*pass*	*pass*	*pass*	*pass*	*pass*
24	*pass*	*pass*	*pass*	*pass*	*pass*	*pass*
32	*pass*	*pass*	*pass*	*pass*	*pass*	*pass*

Table 3: Correctness testing results of the parallel codes across Intel and GNU compilers.

Appendix B: Performance Data

In this appendix, we include the performance testing data from all configurations run in our experiments.

Threads	***Solver 1 (mean)***	***Solver 2 (mean)***	***Solver 3 (mean)***
1	*22.670*	*24.890*	*24.890*

Table 4: Execution times (in seconds) of the sequential codes with the Intel compiler.

Threads	***Solver 1 (mean)***	***Solver 2 (mean)***	***Solver 3 (mean)***
1	*23.536*	*24.314*	*24.314*

Table 5: Execution times (in seconds) of the sequential codes with the GNU compiler.

Threads	***Solver 1 (mean)***	***Solver 2 (mean)***	***Solver 3 (mean)***
1	*25.283*	*25.318*	*25.302*
2	*12.626*	*15.011*	*14.179*
4	*6.634*	*8.062*	*11.470*
6	*4.651*	*8.560*	*8.618*
8	*3.610*	*7.190*	*8.256*
12	*2.488*	*6.931*	*6.699*
16	*1.877*	*5.389*	*3.678*
24	*1.282*	*5.228*	*11.492*
32	*0.980*	*5.314*	*3.244*

Table 6: Execution times (in seconds) of the parallel codes with the Intel compiler.

Threads	***Solver 1 (mean)***	***Solver 2 (mean)***	***Solver 3 (mean)***
1	*25.339*	*23.622*	*22.926*
2	*14.086*	*25.176*	*22.895*
4	*10.946*	*25.163*	*22.927*

6	*7.601*	*25.186*	*23.675*
8	*7.166*	*25.181*	*24.464*
12	*5.461*	*25.065*	*22.902*
16	*5.487*	*25.203*	*23.667*
24	*7.874*	*25.167*	*21.014*
32	*10.168*	*13.677*	*12.834*

Table 7: Execution times (in seconds) of the parallel codes with the GNU compiler.

Threads	***Solver 1***	***Solver 2***	***Solver 3***
1	*0.9*	*1.0*	*1.0*
2	*1.8*	*1.7*	*1.8*
4	*3.4*	*3.1*	*2.2*
6	*4.9*	*2.9*	*2.9*
8	*6.3*	*3.5*	*3.0*
12	*9.1*	*3.6*	*3.7*
16	*12.1*	*4.6*	*6.8*
24	*17.7*	*4.8*	*2.2*
32	*23.1*	*4.7*	*7.7*

Table 8: Speedups of the parallel codes with the Intel compiler.

Threads	***Solver 1***	***Solver 2***	***Solver 3***
1	*0.9*	*1.0*	*1.1*
2	*1.7*	*1.0*	*1.1*
4	*2.2*	*1.0*	*1.1*
6	*3.1*	*1.0*	*1.0*
8	*3.3*	*1.0*	*1.0*
12	*4.3*	*1.0*	*1.1*
16	*4.3*	*1.0*	*1.0*
24	*3.0*	*1.0*	*1.2*
32	*2.3*	*1.8*	*1.9*

Table 9: Speedups of the parallel codes with the GNU compiler.