|epcc|

Updating the EPCC OpenMP microbenchmark suite

Yifei Hu

B153668

Supervisor: Dr. Mark Bull

August 24, 2020

Contents

List of Figures

List of Tables

Listings

Abstract

We have extended the EPCC OpenMP Microbenchmark suite V3.0 by the latest specifications and usages in OpenMP 4.0, 4.5, and 5.0. The whole suite was successfully executed on the back end of Cirrus and compiled by both GNU and Intel compilers. There are mainly two significant contributions. On the one hand, we have compared the original benchmarks with the updated benchmarks both conceptually and statistically and investigated the underlying reasons that caused the differences, which provides reasonable hints to help OpenMP developers with their implementations. On the other hand, we have done the data pre-processing for raw benchmark results and parsed them to a reusable data visualisation module based on Java language.

1 Introduction

The EPCC OpenMP microbenchmark suite consists of a group of tests that measure the overhead of different OpenMP constructs. The tests includes array benchmarks, synchronisation, scheduling, and task benchmarks. By far, there are three versions of the EPCC OpenMP microbenchmark suite version 1.0, and version 2.0 were based on Fortran language, while version 3.0 was implemented in C language.

This dissertation aims to update the latest EPCC OpenMP Microbenchmark suite, which is based on version 3.0 to extend the coverage of OpenMP specifications. The update is divided into three steps. At first, we will focus on some known bugs and make some analysis to fix them and maintain the previous EPCC OpenMP microbenchmark suite. After that, the most significant upgrade is covering some new specifications of new versions of OpenMP(namely, OpenMP 4.0, OpenMP 4.5, and OpenMP 5.0), which are related to the new microbenchmark updating. Therefore, an upgraded EPCC OpenMP Microbenchmark suite based on OpenMP 5.0 is developed, which is up-to-date with the fresh usages of OpenMP. In addition, the updated EPCC OpenMP microbenchmark suite will process a list of experiments on the Cirrus back-end. Therefore we can make use of the experimental results to make some analysis towards different benchmarks. The last step of extending is to design a benchmark visualisation module to demonstrate the EPCC OpenMP microbenchmark suite's results. The new methods of results visualisation as well as the essential techniques, will be introduced.

The primary purpose of microbenchmark is to measure and execute the computational overhead based on different OpenMP directives. The result can be diverse in various hardware and compilers. After benchmarking, the results returned form the microbenchmark suite can indicate the performance of the designed implementation executed in typical architecture. Implementers who program in OpenMP can use these results to find out the defects of their implementation and explore the performance of OpenMP constructs, therefore enhancing the behavior in directive aspects. Additionally, a certain deficiency in the OpenMP program can be highlighted. The overhead measurements also help with the choice between specific semantically identical directives such as CRITICAL and ATOMIC.

This paper will go through six sections to illustrate the investigation. In chapter 2, the background of OpenMP will be introduced at first. By comparison, some other OpenMP microbenchmark suite will be discussed. Also, we will briefly review the previous editions of the EPCC OpenMP Microbenchmark suite. Furthermore, we demonstrate the complete process of benchmarks updating in section 3, which is the most significant part of the paper. We will fix the known bugs at first, and then process the benchmark extending. Section 4 covers the Cirrus experiment based on the updated EPCC OpenMP Microbenchmark suite, and the experimental results will be analysed. The implementation of result visualisation will be demonstrated in Section 5.

2 Background

2.1 OpenMP

The standard specification of OpenMP was established in 1997 by OpenMP ARB(Architecture Review Board), dominating the area of shared memory architectures. OpenMP is an Application Program Interface that can be utilised to construct the shared memory program, which is also called a multi-threaded program. The OpenMP model is SMP (stands for symmetric multi-processors), which means that all threads share both memory and data in OpenMP constructs. In terms of the major components of OpenMP, it contains the compiler directives, runtime library, and environment variables. The basic solution stack of OpenMP can be divided into four layers, which is demonstrated below in Figure1.

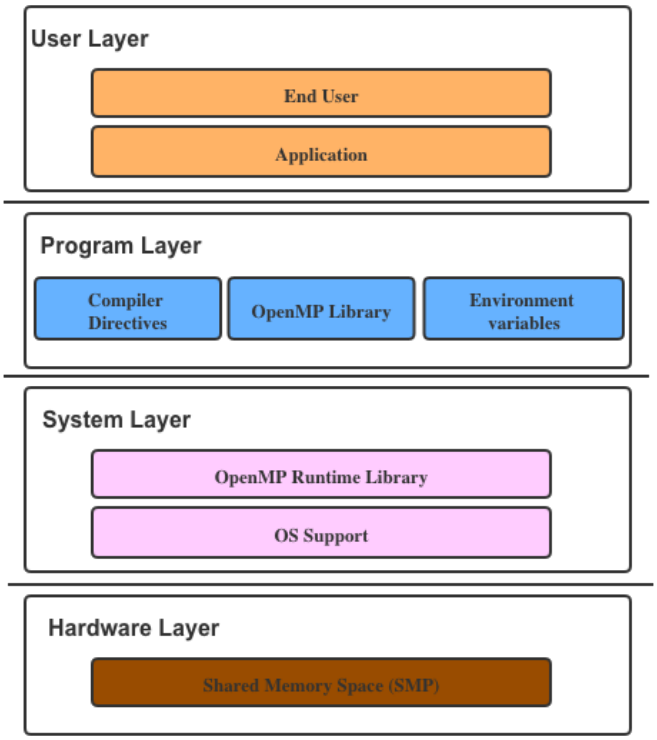

Figure 1: OpenMP Basic Stack

The most bottom layer of the OpenMP basic stack is the hardware layer. OpenMP constructs require hardware in which several identical processors are connected to shared main memory, which is also the central SMP concept. When it comes to the system layer, SMP operating system support is essential to take advantage of multi-threaded hardware. Besides, the OpenMP runtime library (`omp.h`) is also necessary because it can configure the execution environment like thread controlling. In terms of the program layer, compiler directives, OpenMP library, and environment variables are required elements to construct the OpenMP program. Thus, OpenMP applications can be built based on these fragments.

In terms of the parallel execution model, OpenMP adopts the Fork and Join model as fundamental. OpenMP program begins with a master thread(thread 0), executing sequentially until it reaches a parallel section at which the program forks into numerous worker threads. After all worker threads complete their tasks, they reach the end of the parallel section and join back to the master thread. It should be mentioned that all the threads share the same memory space. Figure 2 demonstrates the idea of the fork and join.

There are three essential components for the parallel region, including compiler directives, runtime li-

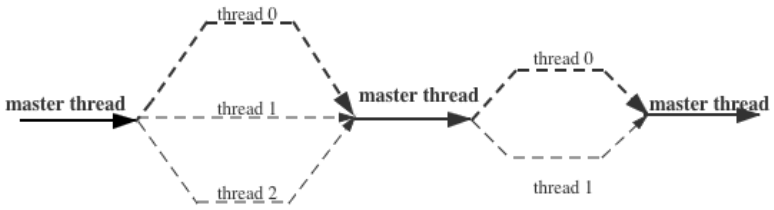

Figure 2: Fork-Join Model

brary routines, and environment variables. The example below shows the underlying construct of the OpenMP program for C.

```
#include <stdio.h>
//OpenMP runtime library
#include <omp.h>
int main() {
  int num;
  int count = 0;
  #pragma omp parallel private(num)
  {
   //Parallel directive
    #pragma omp atomic
    count++;
    //Environment variable
    num = omp_get_num_thread();
  }
   printf("Thread number: %d\n", num);
}
```

Listing 1: Simple OpenMP example

We can see from the example that `omp.h` as OpenMP **runtime library** needs to be included at the beginning of the program. The runtime library contains various runtime library routines. Therefore it defines the standard of OpenMP. In terms of **parallel directives**, they are used to comment on the source code and declare the parallel region. In the example program, `#pragma omp parallel private(count)` and `#pragma omp atomic` are representative parallel directives. **Environment variables** provided by OpenMP are exploited to control the execution of parallel code at runtime. For example, if the end-user sets the maximum number of threads to 8 threads in total(`export OMP_NUM_THREADS=8`), `omp_get_num_thread` can get the number of threads set by users. (The number of threads is 8 in this case).

OpenMP API has been growing for decades. It has been updated from the original version 1.0 to the latest version 5.0 released on November of 2018, which covers more advanced specifications and satisfies more complicated parallelism.

2.2 EPCC microbenchmarking

Generally, a benchmark can be defined as an executable program that evaluates and measures the performance of a fragment of computer architectures or computer systems. When it comes to microbenchmarks, they are designed to assess the behavior of a particular piece of code. Therefore, a microbenchmark can be esteemed as a low-level benchmarking. Microbenchmark generates a set of inputs for per-

formance assessment and reveals the upper bounds of a particular piece of code. Additionally, it captures or 'zooms in' the potential bugs in the code fragment, which helps the developers optimise the program.

2.2.1 EPCC Microbenchmark suite V1.0

EPCC OpenMP Microbenchmark suite was firstly built in 1999. The first version covered the specification of OpenMP 1.0, implemented in both Fortran and C. Soon after, Dr.Mark Bull proposed the additions of **loop scheduling** and **synchronisation** in *Measuring Synchronisation and Scheduling Overheads in OpenMP.*[5] Moreover, Mark aimed to build a technique to measure the overheads of original benchmarks as well as scheduling benchmarks and synchronisation benchmarks. A concise and accurate methodology was introduced to compute the overhead.

$$Overhead = T_p - T_s/p$$

As for a parallel program, T_p was defined as the parallel execution time of a particular piece of program on p processors, while T_s was the sequential execution time of the same fragment. Hence, the result of $T_p - T_s/p$ is the **ideal** parallel execution time. The difference between T_p and T_s/p reflects the overhead of a certain method of parallelism

In terms of the accuracy of the results, Mark made use of Fortran 90 system clock routine to ensure that the results returned by the clock were sufficiently precise, hence efficiently reducing the loss of precision. Besides, the statistic technique was introduced to guarantee consistency. Each benchmark was designed to repeat 50 times for overhead measurement, and 20 different runs. After that, all benchmarks were processed on three different hardware architectures, including **SGI Origin 2000, Sun HPC 3500** and **Compaq Alpha Server**. The SGI Origin 2000 was a sort of mid-range and high-end server computers designed by Silicon Graphics(SGI) in 1996. The Sun HPC 3500 belongs to the Sun Ultra series developed by Sun Microsystems in 1996. Besides, the Alpha Server was based on the DEC Alpha 64-bit microprocessor produced from 1994.

Mark analysed benchmarking results and proposed the potential hints to optimise the program, which could be partially listed:

- It was possible to remove the unnecessary barriers from the parallel region.
- The handling of reduction variables could be optimised.
- The tree-based scheme could be applied to the barrier, which only has the complexity of log p.
- It was possible to enable the compiler itself generating loop bounds without depending on the OpenMP Runtime Library.

2.2.2 EPCC Microbenchmark suite V2.0

Two years later, in 2001, the EPCC Microbenchmark suite was updated to the second version by Mark, which could be referred in his paper *A Microbenchmark Suite for OpenMP 2.0.*[6] The major concerns were put on covering the specifications of OpenMP 2.0. For example, the **WORKSHARE** and **PARALLEL WORKSHARE** directives was added to the synchronisation microbenchmark taking place the original **DO** and **PARALLEL DO** directives in OpenMP 1.0. These directives are in a similar fashion, and the specifications of them are listed below:

- **DO** - it specifies that the loop followed by **DO** must be run in parallel instantly.
- **PARALLEL DO** - it offers a shortcut expression for specifying **DO** directive in a parallel region.
- **WORKSHARE** - it divides the executing work into separate units of work.
- **PARALLEL WORKSHARE** - it offers a shortcut expression for specifying **WORKSHARE** directive in a parallel region.

Besides, the array benchmarks were added first in the second version, including PRIVATE, FIRSTPRIVATE, COPYIN, COPYPRIVATE, and REDUCTION clauses. The details benchmarks will be demonstrated in Section 2.3. The microbenchmark was processed on Sun HPC 6500 and SGI Origin 3000. As for Sun HPC 6500, It has 18400 MHz processors with 18Gb main memory, executing the Solaris 2.7 system. In terms of SGI Origin 3000, it has 128400MHz MIPS processors with 128Gb of main memory, executing IRIX 6.5 system.

After the microbenchmark on both two hardware platforms, the results were analysed based on the overheads. The significant results can be summarised below:

- The **DO** and **PARALLEL DO** directives had merely lower overhead than the overhead of the **WORKSHARE** and **PARALLEL WORKSHARE**.
- The overhead of the **COPYPRIVATE** clause was significant and scaled poorly with a number of threads. This result inspired that there were potentials to optimise the scope by parallelising array reductions over the elements of the array.
- The contention with **PRIVATE** and **FIRSTPRIVATE** clauses means that the private copies of the arrays were assigned to the heap rather than the stack.
- The better choice of multiple **PARALLEL DO** directives or a single **PARALLEL** directive consisting of multiple DO directives depended entirely on the system.
- The overhead of **SINGLE** directive was particularly high when all threads reach the directive at roughly the same time. By comparison, exploiting the **MASTER** directive with explicit barrier provided better performance behavior unless different threads arrive at different times.

It should be mentioned that the first two versions of EPCC OpenMP Microbenchmark Suite were developed by Fortran 90, while the C/C++ version built the third(latest) version.

2.2.3 EPCC Microbenchmark suite V3.0

In the third version of microbenchmark, the OpenMP 3.0 standard introduced in May 2008 was adopted. In the paper *A Microbenchmark Suite for OpenMP Tasks.*[7] Mark extended the new benchmarks with new versions of OpenMP constructs including **task** and **taskwait**. The basic benchmarking measurement technique remains the same as the previous versions.

The task directives were firstly introduced in OpenMP 3.0. The usage of task directives enables the implementer to get rid of some issues involving linked lists and recursive algorithms when parallelising. Till then, it was possible to implement certain kinds of parallelism both easily and efficiently. To test the performance of task directives, Mark added several benchmarks, including ParallelTasks, MasterTasks, MasterTasksBusySlaves, ConditionalTasks, NestedTasks, TreeBranchTasks, TreeLeafTasks, Taskwait, and TaskBarrier. The detail of each benchmark will be demonstrated in Section 2.3.

After the extending, the new suite was examined in the selected hardware platforms which are listed below:

- Cray XE6 (Magny-Cours node) - It contains two 12-core 2.1GHz AMD Magny-Cours processors with GCC, Cray, and PGI compilers.
- Cray XE6 (Istanbul node) - It contains two 16-core 2.3GHz AMD Istanbul processors with GCC, Cray, and PGI compilers.
- IBM Power7 server - It contains four 8-core 3.55GHz Power 7 processors with IBM XL C/C++ compiler
- AMD Magny-Cours server - It includes four 1.9GHz 12-core AMD Magny-Cours processors with GCC, Oracle Solaris suncc compilers.
- SGI Altix ICE 8200EX node - It contains two 2.66GHz 6-core Intel Westmere Xeon E560 processors with GCC and icc compilers.

- SGI Altix 4700 - It contains 128 1.6GHz Intel Montecito dual-core processors with icc compiler.

The conclusions analysed from the results returned by the new suite can be summarised:

- The benefits of parallelising task generation were uncertain, and the overhead of task generation did not depend on whether the slave threads are busy or idle.
- The overhead of task generation with a **false if** clause was quite low.
- The single level of task-nesting did not noticeably affect the overheads of the task directives.
- Checking the task completion could enhance the cost of the barrier.

2.3 Previous benchmarks summary

Based on the previous EPCC Microbenchmark Suite investigation, we figure out the main target is the same. The suite compares the execution time of sequential fragment with the fragment executed in the parallel region and parallel directives.

2.3.1 Overhead calculation

The suite's core is to result in the overhead of each benchmark constructed by different usage of OpenMP directives and clauses. Listing 2 and 3 show the main idea of the overhead.

```
void refer() {
   int i, j;
   for (j = 0; j < innerreps; j++) {
   for (i = 0; i < itersperthr; i++) {
      delay(delaylength);
   }
   }
}
```

Listing 2: Sequential version of the fragment

```
void teststatic() {
   int i, j;
#pragma omp parallel private(j)
   {
  for (j = 0; j < innerreps; j++) {
#pragma omp for schedule(static)
     for (i = 0; i < itersperthr * nthreads; i++) {
     delay(delaylength);
      }
  }
   }
}
```

Listing 3: Parallel version of the fragment

The **delay()** function keeps the processor continuous executing, which can be regarded as a **work**. It can be seen that each thread in Listing3 executes the same amount of work as sequential ones. To compute the overhead of typical parallel patterns with different combinations of parallel directives and clauses, we can subtract the execution time of the parallel version and the sequential version.

```
void printfooter(char *name, double testtime, double testsd, double
    referencetime, double refsd) {
```

```
    printf("%s time = %f microseconds +/- %f\n",
      name, testtime, CONF95*testsd);
    printf("%s overhead = %f microseconds +/- %f\n",
      name, testtime-referencetime, CONF95*(testsd+referencesd));//Overhead
          calculation
}
```

Listing 4: Overhead computation fragment

2.3.2 Test summary

In terms of the benchmarks, the current suite contains various types of benchmarks with different kinds of OpenMP usage within **array, scheduling, synchronisation** and **tasking**.

Benchmark name	**Parallel region**	**Description**
testpivnew	*#pragma omp parallel private(atest)*	It declares that the variable atest is private to a task.
testfirstprivnew	*#pragma omp parallel firstprivate(atest)*	It declares the variable atest is private to a task, and initialises atest with the original value before the parallel construct.
testcopyprivnew	*#pragma omp parallel private(atest)* *#pragma omp single copyprivate(atest)*	It declares that the variable atest is private to a task and specifies the associated code block being executed by only one thread. Besides, the value of private variable atest was broadcast in the data environment.
testthrprivnew	*#pragma omp parallel copyin(btest)*	It copies the value of a thread-private variable of the master thread to other threads in the team.

Table 1: Parallel construct for array tests

In terms of scheduling benchmarks, the **for** directive is used to note that associated loops can be executed in parallel by the member threads in the team. The work-sharing loop construct starts with `#pragma omp for`.

Benchmark name	**Parallel region**	**Description**
teststatic	*#pragma omp for schedule(static)*	If there is no chunk_size declared, the iteration space will be divided into roughly identical size, and one chunk is assigned to each thread at most.
teststaticn	*#pragma omp for schedule(static,cksz)*	The iterations will be divided into size of chunk_size, and each chunk has chunk_size of iterations. The chunk will be allocated to each thread in a round-robin fashion.[2]
testdynamicn	*#pragma omp for schedule(dynamic,cksz)*	The iterations are assigned to member threads in the team, every thread executes a chunk (with size of chunk_size)first and then requests another chunk till no chunks remain.
testguidedn	*#pragma omp for schedule(guided,cksz)*	The iterations are assigned to member threads in the team, every thread executes a chunk and then requests another chunk. But size of a chunk is the proportion of the number of unallocated iterations divided by the number of the threads causing the falls of the size of the chunks. Note that the minimum size of a chunk is configured by chunk_size.

Table 2: Parallel construct for scheduling tests

When it comes to sychronisation constructs, it regulates threads that bring a certain sequence when they execute implicit tasks. The synchronisation microbenchmark includes a variety of associated tests, which are listed below in Table 3.

Benchmark name	Parallel region	Description
testpr	*#pragma omp parallel*	It generates a team of OpenMP member threads and declares parallel region
testfor	*#pragma omp parallel private(j)* *#pragma omp for*	It specifies the iterations of associated loops executing in parallel by member threads.
testpfor	*#pragma omp parallel for*	It is a shortcut for declaring a parallel region that contains work-sharing loops and no other statements.
testbar	*#pragma omp barrier*	All member threads in the team should complete their executions before the following execution beyond the barrier.
testsing	*#pragma omp single*	It specifies the associated code block being executed by exclussively one thread in the team, while the other threads wait until the barrier.
testcrit	*#pragma omp critical*	It limits the execution of the code block to a single thread once.
testlock	*omp_set_lock(&lock);* *omp_unset_lock(&lock);*	A lock guarantees the data element lock can only be handled with by only one process.
testorder	*#pragma omp ordered*	It declares the associated loop being executed in the order of the original iteration of the loop .
testatom	*#pragma omp atomic*	It ensures that a particular memory location is enforced to be updated atomically instead of exposing it to simultaneous or multiple reads and writes that leads to some intermediate values.
testred	*#pragma omp parallel reduction(+:aaaa)*	The reduction clause creates a private copy for the variable aaaa for each thread, and at the end of the reduction, the variable aaaa is applied to all private copies of shared variable.

Table 3: Parallel construct for synchronisation tests

After OpenMP 3.0, the usage of task directives was included. Therefore, the third version of the suite added tasking benchmarks that examined the platform's performance of tasking directives. The benchmarks can be summarised in Table 4.

Benchmark name	Parallel region	Description
testParallelTaskGeneration	*#pragma omp task*	It declares an explicit task.
testMasterTaskGeneration	*#pragma omp master* *#pragma omp task*	It defines an explicit task to the master thread.
testMasterTaskGenerationWithBusySlaves	*#pragma omp task*	The master thread generates the tasks while keeping other member threads busy
testNestedTaskGeneration	*#pragma omp task private* *#pragma omp task untied* *#pragma omp taskwait*	It defines nested tasks with nested loops, and all threads generate outer tasks.
testNestedMasterTaskGeneration	*#pragma omp master* *#pragma omp task private* *#pragma omp task* *#pragma omp taskwait*	It defines nested tasks with nested loops, and only master thread generates outer tasks.
testTaskWait	*#pragma omp task* *#pragma omp taskwait*	It defines a halt on the completetion of child tasks for current task.
testTaskBarrier	*#pragma omp task* *#pragma omp barrier*	It declares an explicit barrier at the barrier point and requires all threads reach the point of barrier.[10]
testConditionalTaskGeneration	*#pragma omp task if(returnfalse())*	It generates an immediate execution task and suspends all other threads when encountering the task region. Tasks on other threads will be resumed after the accomplishment of the undeferred task.
testBranchTaskGeneration	*#pragma omp task*	Each thread generates tasks in a binary tree recursively. The delay function is executed in every task.
testLeafTaskGeneration	*#pragma omp task*	Each thread generates tasks in a binary tree recursively.

Table 4: Parallel construct for tasking tests

3 Benchmarks updating

In this chapter, the paper focuses on extending the current version of the EPCC OpenMP Microbenchmark suite, which is based on the OpenMP 3.0 standard. Since OpenMP has upgraded from 3.0 to 5.0

with more usage of directives and clauses, we will do some investigations and find out the potential additions that can be added to the current suite. Therefore, we can use the extended suite to measure new OpenMP constructs' performance based on the new specifications of the latest OpenMP edition and compare the results with the original benchmarks' results. Before the extension of the current suite, the existing bug should be removed to enhance the correctness.

3.1 Bugs handling

According to the user's report, there is one issue with **ATOMIC** test in the fresh suite in the `syncbench.c` file.

```
void testatom() {
   int j;
   double aaaa = 0.0;
   double epsilon = 1.0e-15;
   double b,c;
   b = 1.0;
   c = (1.0 + epsilon);
#pragma omp parallel private(j) firstprivate(b)
   {
  for (j = 0; j < innerreps / nthreads; j++) {
#pragma omp atomic
      aaaa += b;
      b *= c;
  }
   }
   if (aaaa < 0.0)
  printf("%f\n", aaaa);
}
```

Listing 5: Current ATOMIC benchmark

The issue here is that the expression `innerreps/nthreads` involves the shared variables, which are both declared globally, where `j` and `b` are the private variables in the parallel region. As a result, the computation of the **upper bound** of the loop will be processed in each iteration, causing the execution latency. Based on the analysis, the ATOMIC test's real overhead will be lower than the current construct of the ATOMIC benchmark. To fix this issue, we naturally move the 64-bit integer division outside the loop to avoid the extra execution of the ATOMIC test. The corrected ATOMIC test is illustrated below in Listing 6.

```
void testatomnew() {
   int j;
   double aaaa = 0.0;
   double epsilon = 1.0e-15;
   double b,c;
   b = 1.0;
   c = (1.0 + epsilon);
   #pragma omp parallel private(j) firstprivate(b)
      {
        int ub = innerreps / nthreads;
    for (j = 0; j < ub; j++) {
   #pragma omp atomic
        aaaa += b;
        b *= c;
    }
      }
   if (aaaa < 0.0)
```

```
  printf("%f\n", aaaa);
}
```

Listing 6: Corrected ATOMIC benchmark

To verify the changes to the atomic test, we process a brief experiment on the Cirrus platform(which will be introduced later) using the Intel compiler to compare the overhead of the previous atomic test and modified atomic test. The verification of the atomic test modification is designed to process by 1, 2, 4, 8, 16, 32 OpenMP threads on the Cirrus backend to demonstrate the effect of removing the division of 64-bit integer global variables.

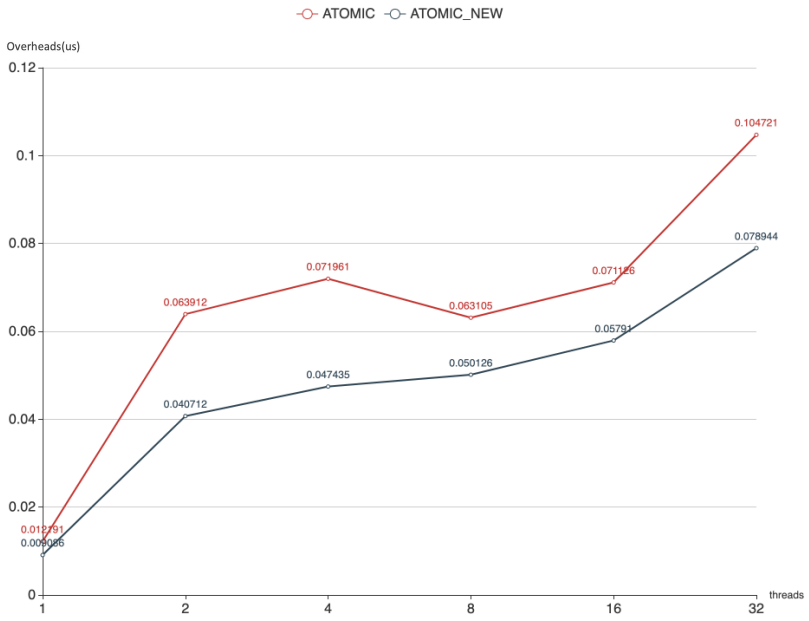

Figure 3: Overheads of new atomic and original atomic tests executed by Intel-compilers-18

We can see from Figure 3 that the new atomic test's overhead is a bit lower than the data of the original atomic test. The evidence proves that moving the division of global variables out of the parallel region enables the atomic overhead measurement much closer to reality. Thus, we replace the previous version of the atomic test with the new one to enhance the suite's correctness and accuracy.

3.2 OpenMP specifications updating

The significant concern should be put on the latest OpenMP specifications after correcting the existing bugs. Before we implement the new benchmarks, necessary investigation about the latest OpenMP standard, including 4.0, 4.5, and 5.0, should be done first. We will go through each microbenchmark and discuss the current benchmarks in detail at first and figure out the potentials of how pre-existing benchmarks can be upgraded to embrace the new specifications.

3.2.1 Scheduling additions

Since there are little updates on array benchmarks, we start the extending from OpenMP scheduling usages. The `schedule` clause is used for specifying the scheduling type of the parallel loops. The current scheduling microbenchmark involves four types of benchmarks which are **Static**, **Static,n**, **Dynamic** and **Guided**.

Static and Static,n

`Static` is the default schedule type which straightforwardly assigns the loop iterations to threads by an equal distribution which can be regarded as a low-cost schedule.[4] When entering the loop, every thread determines which chunk of the loop they should execute by itself. Similarly, the schedule `Static,n` divides the iteration space equally as well. The only difference is that `Static,n` equally divides the iteration space into `n`-chunk size of total iterations. It should be mentioned that the round-robin fashion is applied here to adjust the iteration division. Due to the difference in the schedule pattern, the expense of `static,n` schedule should be comparably greater than the data of `static` schedule. Figure 4 and 5 below illustrate the basic idea of `static` and `static,n`

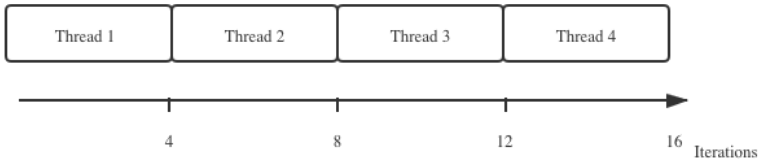

Figure 4: Static scheduling

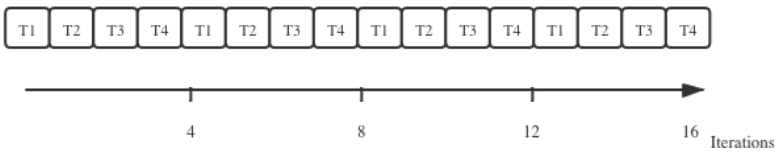

Figure 5: Static,n scheduling

Based on the investigation of OpenMP 4.5 specifications, we figure out that new modifiers can be applied to schedule clause.[13] The new construct of loop scheduling is declared as
`#pragma omp parallel for schedule([modifier [modifier]:]kind[,chunk_size])`
where `modifier` is defined below:

- **monotonic**: If the **monotonic** modifier is declared then every thread executes the assigned chunks in increasing order.
- **nonmonotonic**: If the **nonmonotonic** modifier is declared, then every thread executes the assigned chunks in any order.
- **simd**: If the **simd** modifier is declared and the loop is constructed in SIMD, the chunk_size will be a multiple of the SIMD width. But when the loop is not associated with a SIMD construct, the **simd** modifier will be neglected.

Since there are no SIMD constructs on the current scheduling benchmarks, we skip this specification and focus on **monotonic** and **nonmonotonic** modifiers. We follow the construct guide and simply add the modifiers, therefore, generating new benchmarks. Listing 7 is an example of static scheduling with a monotonic modifier.

```
void teststaticmono() {
   int i, j;
#pragma omp parallel private(j)
   {
  for (j = 0; j < innerreps; j++) {
#pragma omp for schedule(monotonic:static)
     for (i = 0; i < itersperthr * nthreads; i++) {
     delay(delaylength);
      }
  }
   }
}
```

Listing 7: Static scheduling benchmark with monotonic modifier

For example, if there are four chunks and two OpenMP threads in total, the first thread `thread 1` is assigned with chunk 1 and 2, and the other thread is assigned with chunk 3 chunk and chunk 4. If monotonic is specified, thread 1 must execute chunk 1 - > chunk 2, and thread 2 must execute chunk 3 - > chunk 4 in increasing order. When nonmonotonic is specified (which is a default modifier), thread 1 can execute iteration in any order like chunk 2 -> chunk 1, the situation of thread 2 is similar. The idea should be the same for `static,n` schedule. The idea is demonstrated below in Figure 5 and Figure 6.

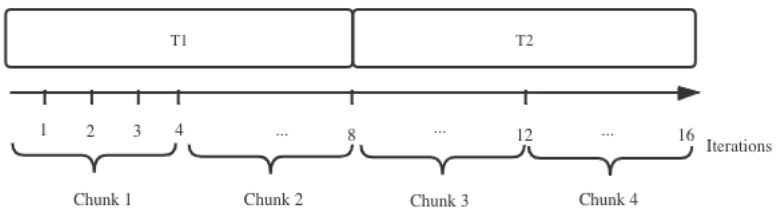

Figure 6: Static with monotonic scheduling

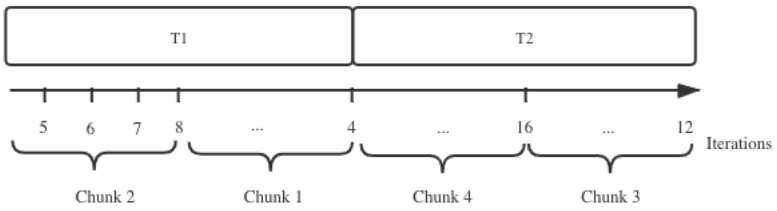

Figure 7: Static with nonmonotonic scheduling

According to the analysis, the new benchmarks thereby generated:

- `static (monotonic)`
- `static (nonmonotonic)`
- `static,n (monotonic)`
- `static,n(nonmonotonic)`

Similarly, the monotonic and nonmonotonic modifiers might be applied to the `guided` and `dynamic` tests.

Guided, Dynamic

When `dynamic` is defined, the iteration space will be divided into chunks, and each chunk has the size of chunk_size. Every OpenMP thread process a chunk of iterations execution, and then require another chunk of the rest. Chunks allocate all the thread in first-come-first-served order. [17] Likewise, `guided` requests threads in a first-come-first-served order as well. Yet, the size of the chunk keeps deducting due to that the chunk of each size is the proportion of unallocated iterations divided by the total number of OpenMP threads. Since the number of unassigned iterations falls, the size of chunk falls as well.

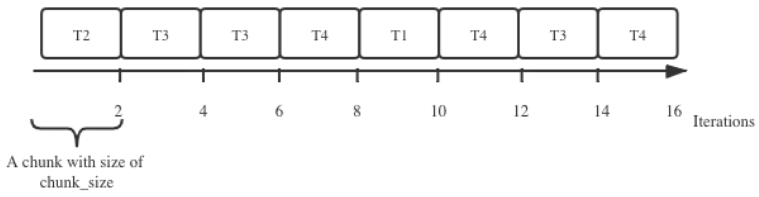

Figure 8: dynamic scheduling with 2 chunksize

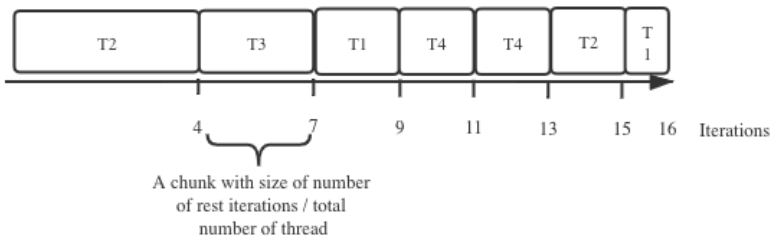

Figure 9: guided scheduling with 2 chunksize

We can see the difference between Figure 8 and Figure 9, the size of the chunk keeps unchanged for `dynamic` with chunk_size of 2 while the size of chunk reducts in `guided` scheduling. It should be mentioned that the minimal size of the chunk is set greater than chunk_size.

As it is mentioned before, we can apply the monotonic and nonmonotonic modifiers to the `guided` and `dynamic` directives. The monotonic modifier enables the chunks to iterate in increasing order, while the nonmonotonic specifies the random execution order of chunks.[8] Therefore We generate four new benchmarks including:

- `guided (monotonic)`
- `guided (nonmonotonic)`
- `dynamic (monotonic)`
- `dynamic (nonmonotonic)`

Taskloop

Referred to OpenMP 4.5, the `taskloop` construct was introduced the first time.[13] The motivation of introducing the `taskloop` directive is that the work-sharing of the OpenMP loop and nested parallelism may result in an unexpected excessive fork and join overhead. Thereby, the usage of the taskloop construct was introduced, which splits the loop iterations into chunks, and executes the chunks by exploiting OpenMP tasks in parallel. By using `taskloop`, the parallel region can wait on completion of the implicit tasks, and every generated tasks is allocated by distributed iterations of the loop.[11] The number of iterations for each task is the proportion of the total iteration number divided by task number. Listing 8 is an example of a basic taskloop construct.

```
#pragma omp taskloop num_tasks (50)
   for (int i = 0; i < 1000; i++)
      foo(i);
```

Listing 8: Taskloop example

The construct above generates 50 tied tasks and distributes 20 iterations (which is the proportional of 1000 iterations divided by 50 tasks) to each tied task. When other implicit tasks encounter, they will wait for the completion of all tasks declared in the `taskloop` construct. Based on the idea of `taskloop` construct, we thereby generate a benchmark to test the performance of basic `taskloop` directive.

```
void taskloopn() {
    int i, j;
#pragma omp parallel private(j)
    {
#pragma omp master{
```

```
    for (j = 0; j < innerreps; j++) {
#pragma omp taskloop num_tasks((itersperthr*nthreads)/cksz)
       for (i = 0; i < itersperthr * nthreads; i++) {
       delay(delaylength);
       }
       }
    }
    }
}
```

Listing 9: Taskloop benchmark

The total number of iterations here is `itersperthr * nthreads`, and the number of tasks is set by the total number of iterations divided by chunk size `cksz`, which assigns each task with a chunk of iteration. By far, we have accomplished the updating of scheduling benchmarks. The extending of synchronisation benchmarks follows in the next section.

3.2.2 Synchronisation additions

The `synchronisation` constructs bring OpenMP threads specific order to the sequence when they execute the work. Originally, the current suite for synchrnosation microbenchmark supports several benchmarks, including `parallel, for, parallel-for, barrier, single, critical, locking, ordered, atomic, reduction` tests. The related directives which have potentials to be extended by new OpenMP specifications will be discussed below.

Barrier

All member threads of the team in the parallel region are forced to execute the `barrier` region. More specifically, a `barrier` declares a point at which all operating threads that have already finished their work should wait till the rest threads have reached the point. The original `barrier` test aimed to benchmark the performance of `barrier` directive.

```
void testbar() {
   int j;
#pragma omp parallel private(j)
   {
  for (j = 0; j < innerreps; j++) {
     delay(delaylength);
#pragma omp barrier
  }
   }
}
```

Listing 10: Current barrier test

After analysis, we find that the current construct of the barrier may occur in an unexpected circumstance. The same thread may arrive last at the barrier of all iterations. Figure 10 demonstrates the special situation.

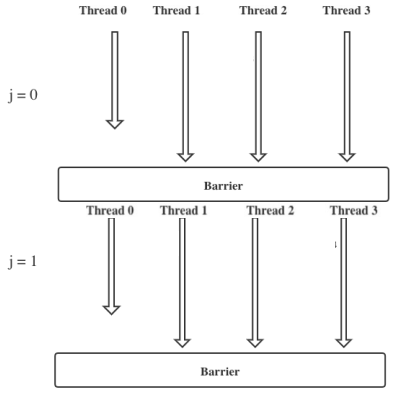

Figure 10: Special situation on current barrier test

Thus, the original barrier test is probably an 'unfair' barrier test. In order to update the original barrier benchmark, we aim to enable every thread an **equal** chance to arrive at the barrier last. Hence, we generate a new barrier test illustrated below in Listing 11.

```
void testbarvar() {
   int j;
#pragma omp parallel private(j)
   {
   int thread_num = omp_get_thread_num();
  for (j = 0; j < innerreps; j++) {
   if(thread_num==(j%nthreads))
     delay(delaylength);
#pragma omp barrier
  }
   }
}
```

Listing 11: New barrier test

The barrier benchmark is designed to guarantee that in each iteration, only one thread executes `delay` function and the rest keep idle and wait on the barrier. The busy thread keeps changing, followed by the number of iteration. The idea of the 'fair' barrier benchmark is illustrated below in Figure 11.

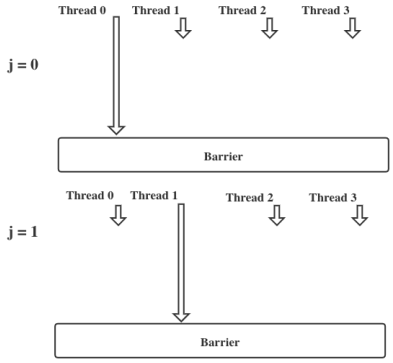

Figure 11: Situation of extended barrier test

The performance comparison between these two benchmarks will be processed in Chapter 4 based on the

experimental results.

Locking

OpenMP Library possesses the traditional technique *lock* as well. The function of a lock is similar to the critical directive to some extent. The critical section guarantees the inner code is performed by exclusively one thread, while a lock guarantees the lock variable can only be accessed by lock routine. There are three status of an OpenMP lock: *uninitialised*, *unlocked* and *locked*. We should declare it first using `omp_lock_t lock;` to create the lock variable, and the status of the lock is uninitialised. Then we should initialise the lock with `omp_init_lock(lock)`. Therefore the status of lock turns to be unlocked. In order to analyse the lock routine, we take the previous lock test as an example:

```
omp_lock_t lock;
omp_init_lock(&lock);
void testlock() {
   int j;
#pragma omp parallel private(j)
   {
  for (j = 0; j < innerreps / nthreads; j++) {
     omp_set_lock(&lock);
     delay(delaylength);
     omp_unset_lock(&lock);
  }
   }
}
```

Listing 12: Current lock test

The lock constructs guarantee only one thread executes `delay` function at a time. It should be mentioned that the lock routine provides a more flexible and explicit technique to avoid the particular code regions being accessed by other threads simultaneously. When a thread encounters a locked code section, it should keep waiting until the code section is unlocked.

In terms of lock test updating, we find that the pattern of lock initialisation is extended. The `lock_hint` trait firstly defined in OpenMP 4.5 standard and is detailed described in OpenMP 5.0 specification.[12] Thus, we come up with ideas of generating new lock tests with hints, and we cover two hints for extending this time.

- contended: More threads tend to perform typical operation simultaneously, which is called high contention.
- uncontended: Fewer threads tends to perform typical operation simultaneously, which is called low contention.

Hence, we generate contended lock benchmark and uncontended lock benchmark by simply changing the lock initialisation to `omp_init_lock_with_hint(&lock, omp_lock_hint_contended);` and `omp_init_lock_with_hint(&lock, omp_lock_hint_uncontended);`

Atomic

The `atomic` region guarantees atomic access to a specific memory space. Each atomic operation is completely performed on a typical memory location, which guarantees no data races on these locations. After investigation, the *memory-order-clause* is added in the atomic construct to indicate the memory ordering behavior of the atomic region, which includes `seq_cst, acq_rel, release, acquire, and relaxed`. Besides, the *atomic-clause* is also extended to four types in detail: `read, write, update and capture`. As for the current atomic test, both load and assert operations involve, therefore the *atomic-clause* should be set `update`. By default, the *atomic-clause* is set to `update` hence

we can remain the original atomic construct for atomic-clause. Then, we put our main focus on memory ordering, and Table 5 compares the rules of different memory orders.[16]

Memory order	Rules
relaxed	It only guarantee that load() and store() are atomic operations, without guarantees of the sequence of execuction.
acquire	In this thread, all subsequent read operations must be executed after the completion of the atomic operation.
release	In this thread, all subsequent write operations must be executed after the completion of the atomic operation.
acq_rel	It combines both acquire memory order and release memory order.
seq_cst	All stores and loads are executed sequentially.

Table 5: Memory ordering

Based on the summary of memory orders, we can conclude the `relaxed` memory order is the least strict memory order. On the opposite side, `seq_cst` regulates the most strict atomic execution, while `release` and `acquires` lie in the middle. It is natural to ask a question of whether the setting of strict memory order influences the performance. Hence, we generate a new benchmark to find out the result.

```
void testatomseqcst() {
   int j;
   double aaaa = 0.0;
   double epsilon = 1.0e-15;
   double b,c;
   b = 1.0;
   c = (1.0 + epsilon);
#pragma omp parallel private(j) firstprivate(b)
   {
     int ub = innerreps / nthreads;
  for (j = 0; j < ub; j++) {
#pragma omp atomic seq_cst
      aaaa += b;
      b *= c;
  }
   }
   if (aaaa < 0.0)
  printf("%f\n", aaaa);
}
```

Listing 13: Atomic with seq_cst test

3.2.3 Tasking additions

In OpenMP, tasks are independent units of work which contains both code and data. A new task is generated when a thread encounters a task construct. In the third version of the suite different types of task benchmarks were introduced according to various usages of task directives.[14] Based on OpenMP 4.0 standard, the `depend` clause can be added to the `task` directive to enforce extra constraints on tasks scheduling. The constraints limit that the dependencies are only between sibling tasks. Listing 14 demonstrates the basic idea of tasks with dependency.

```
#pragma omp task depend(out:t)
 t = foo()
#pragma omp task depend(in:t)
 p = goo(t)
```

Listing 14: Example of dependent tasks

It is feasible to set a partial ordering to the tasks that guarantee the second task executing before the first task, avoiding the incorrect result. Thereby, we generate new benchmarks according to this mechanism.

```
void testParallelTaskGenerationWithDeps() {
    int j, id;
    int *a;
    a = (int *) malloc (omp_get_max_threads() * sizeof(int));
#pragma omp parallel private( j, id )
    {
    id = omp_get_thread_num();
    for ( j = 0; j < innerreps; j ++ ) {
#pragma omp task depend(inout:a[id])
       {
       delay( delaylength );
       }
    };
  };
}
```

Listing 15: Parallel task with dependencies

We dynamically define the memory size of pointer `a` by the total amount of threads and make the pointer `a` as a dependent. It can be seen from Listing 15 that each iteration generates a task. By using `pragma omp task depend(inout:a[id])`, the execution of each task should be forced to start with receiving the output of the last task. [9] Hence we guarantee the partial ordering of the tasks. Similarly, we generate a task-depend version for **MasterTaskGeneration** benchmark.

```
void testMasterTaskGenerationWithDeps() {
    int j, id;
    int *a;
    a = (int *) malloc (omp_get_max_threads() * sizeof(int));
#pragma omp parallel private(j, id)
    {
#pragma omp master
    {
       for (j = 0; j < innerreps * nthreads; j++) {
       id = j%nthreads;
#pragma omp task depend(inout: a[id])
       {
          delay(delaylength);
       }
       }
    }
    }

}
```

Listing 16: Master task with dependencies

By far, we have finished the extending of the third version of suite by adding new OpenMP features and fixing the known bug. The benchmarking results of new EPCC OpenMP Microbenchmark suite will be demonstrated in next chapter.

4 Benchmark results and analysis

In this chapter, we aim to examine the performance of new benchmarks and make comparisons with the previous benchmarks. It should be mentioned that we omit the benchmarking of array benchmarks due to no updates applied to array benchmarks.

4.1 Experimental environment

We will process the benchmarking on the back end of the Cirrus, which guarantees the correctness and accuracy of the experiments. After investigation, Cirrus's architecture is HPE SGI Apollo 8600, which has 72 Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz. Every processor consists of 18 cores with a 45MB cache block and 64 bytes cache line. Cirrus is based on the Intel AVX2 instruction set extension, which is suitable for floating computations and parallelism.

We plan to compile the whole suite by two compilers, including GNU compiler and Intel compiler. The general description of two compilers is listed below:

- GNU Compiler Collection(GCC) - It is a compiler system generated by the GNU Project which supports varieties of programming languages. In our case, we adopt GCC 8.2 which is the latest version installed on the Cirrus by far.
- Intel C++ Compiler - It generates optimised code for IA-32 and IA-64 platforms, and it also supports compiling non-optimised code for non-intel processors. In our case, we adopt Intel-compilers-18 for OpenMP code compiling.

The compiler flags for suite compiling are decided below:
GCC:
`gcc -fopenmp -O3 -lm`
Intel Compiler:
`icc -openmp -O1 -lm`

Then, we run the new suite on the back end of Cirrus and get lists of each benchmark's results. All the experimental results will be imported as raw data to the designed data visualisation module. (which will be demonstrated in the next Section). Thereby we generate plots to illustrate the behavior of the new benchmarks.

4.2 Scheduling benchmarks

We have updated the scheduling microbenchmark mainly in two aspects. One is the monotonic/nonmonotonic modifier update. The other is the taskloop, where we divide the loop iterations into tasks. After the execution of Cirrus, we demonstrate the results below. It should be mentioned that the graphs below are all generated by the designed visualisation mechanism, which will be introduced later in chapter 6.

4.2.1 Monotonic/Nonmonotonic update

We update the original scheduling benchmarks, including `static, staticn, dynamicn, guidedn` with `monotonic/nonmonotonic` modifiers and compare the behavior with original ones.

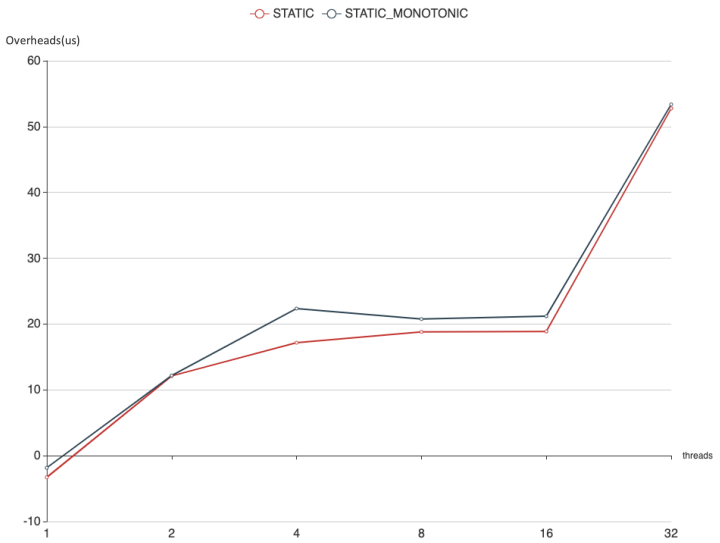

Figure 12: Static tests compiled by GCC

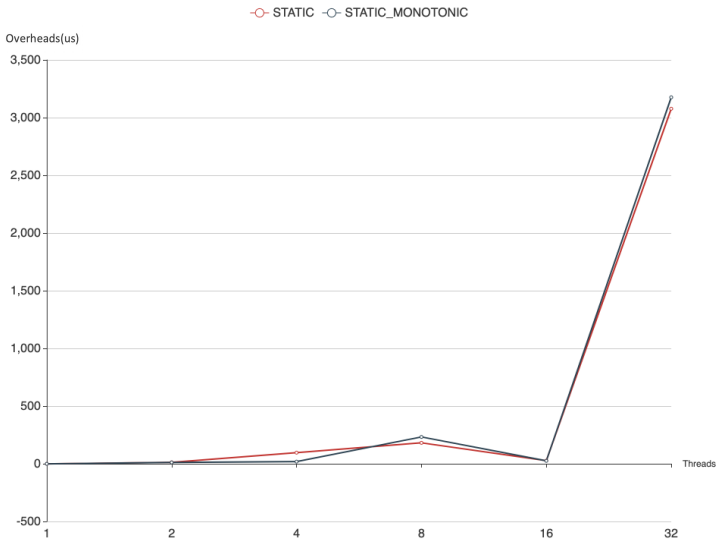

Figure 13: Static tests compiled by Intel

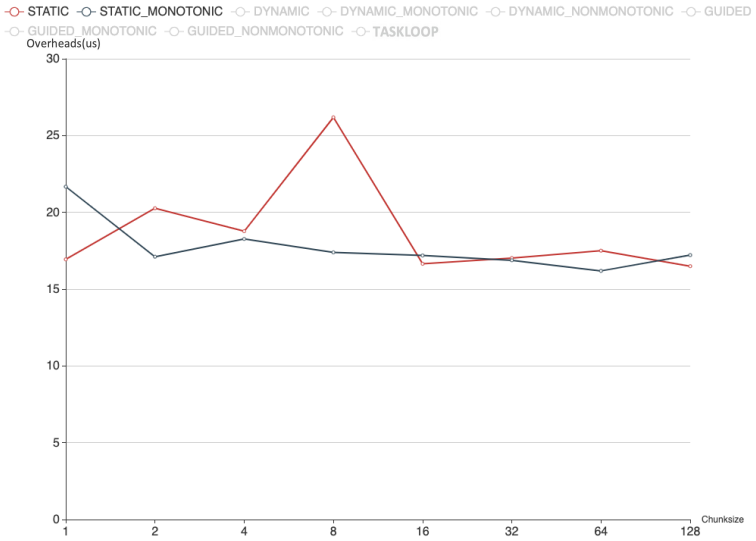

Figure 14: Staticn tests compiled by GCC

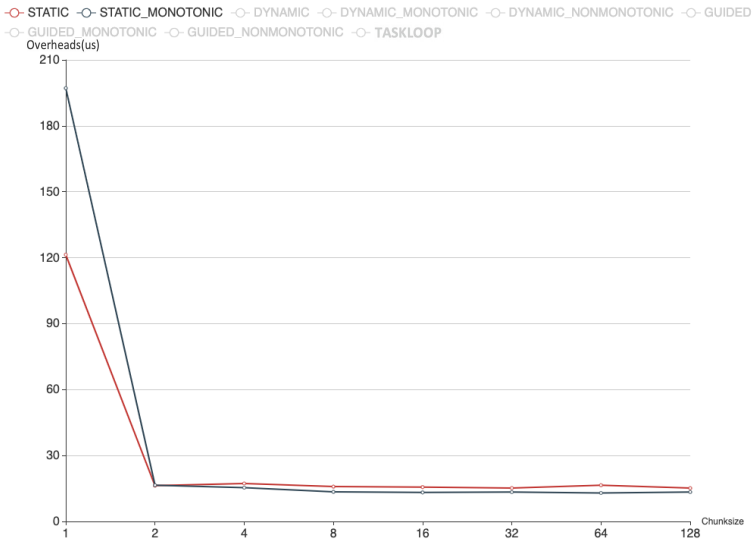

Figure 15: Staticn tests compiled by Intel

It should be mentioned that we delete the static with nonmonotonic modifiers combinations. It is due to static with nonmonotonic modifiers that make no sense due to static directive executes fixed chunks assigned to all threads equally. In addition, the monotonic version of `STATIC` directive acts the same as the original `STATIC` tests. We expect the `STATIC_MONOTONIC` and `STATICN_MONOTONIC` to share similar overheads with original ones. We can see from Figures 14 and 15 that the overheads of both the original static and monotonic static tests increase, followed by the increment of thread numbers. It can be seen that most cases meet our assumptions.

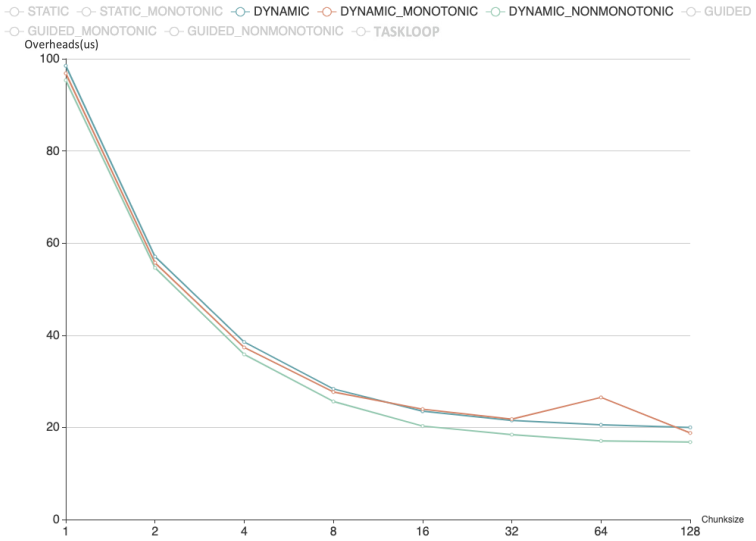

Figure 16: Dynamicn tests compiled by GCC

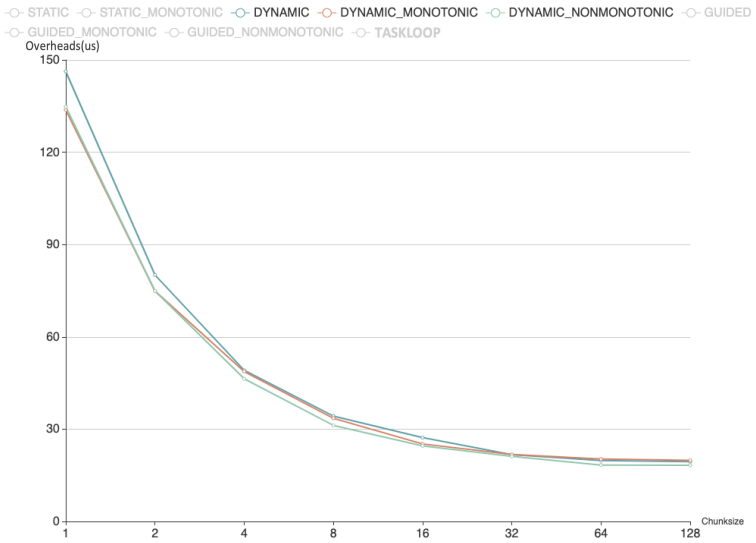

Figure 17: Dynamicn tests compiled by Intel

As for new dynamic tests, we expect `DYNAMIC_NONMONOTONIC` test shares similar results with the original one due to the usage of `nonmonotonic:dynmaic` is the same as default dynamic directive. Besides, we suppose that `DYNAMIC_MONOTONIC` has larger overhead than others due to the extra computation to guarantee the sequential order of chunk execution. However, our expectation does not come true based on the results. We can see from Figures 16 and 17 that the overheads of three dynamic tests are quite close, and we cannot gain a specific conclusion based on this experiment. The situation might be caused by noise when executing.

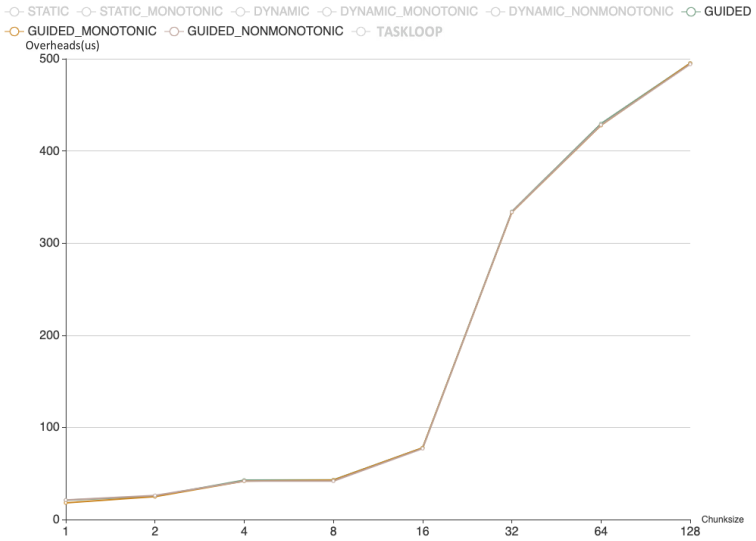

Figure 18: Guidedn tests compiled by GCC

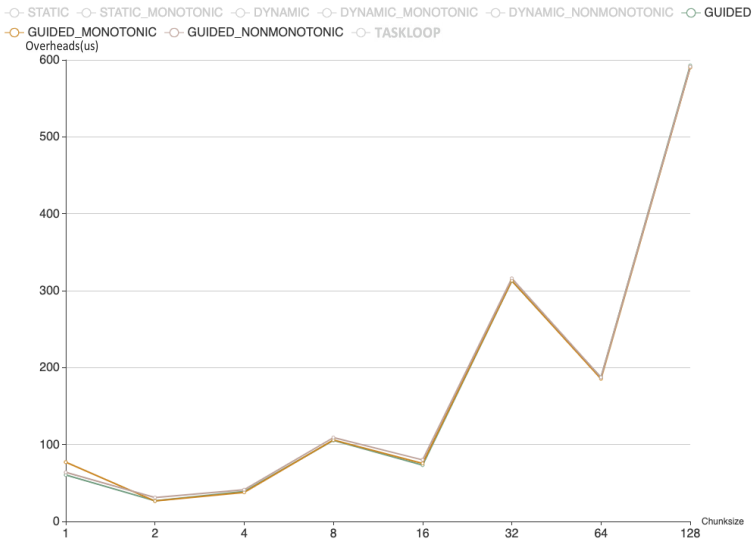

Figure 19: Guided tests compiled by Intel

When it comes to guided tests, we expect to see the three benchmarks have closer overheads followed by the increment of the chunk size. When the chunk size is comparably big, each thread will be assigned with less number of chunks. Thus, the effects of a monotonic/nonmonotonic modifier will be reduced. In terms of the graphs above, we figure out that the gap among three tests gets smaller followed by the increment of chunk_size.

4.2.2 Taskloop update

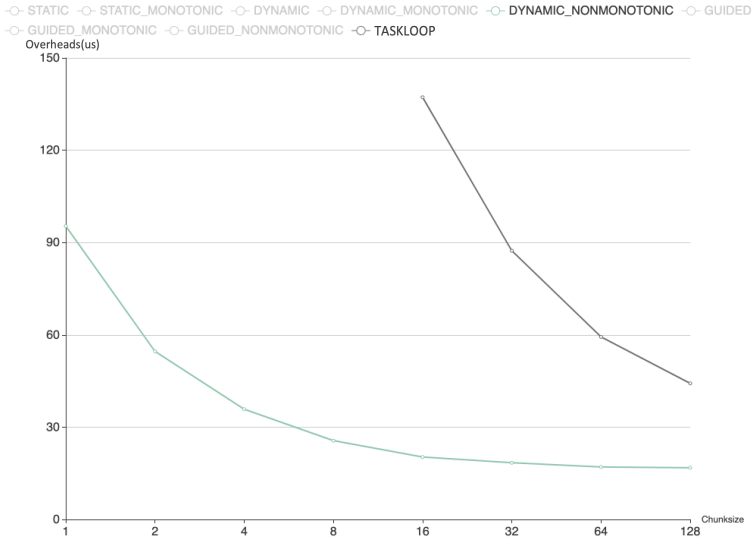

Figure 20: Taskloop tests compiled by GCC

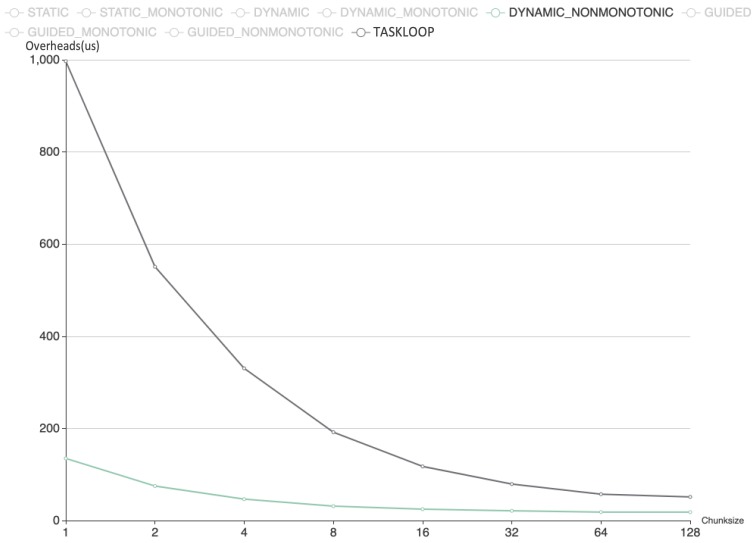

Figure 21: Taskloop tests compiled by Intel

We compare `Taskloop` tests with `DYNAMIC_NONMONOTONIC` tests here, because the parallel ideas are quite similar. Taskloop directives separate the iterations to different tasks and assign tasks to each thread randomly. Similarly, `nonmonotonic: dynamic` directive enables all threads allocated by chunks in random (i.e., first-come-first) order. We guess that taskloop tests have worse behavior than the other because of the extra execution of task generation. We can see from both Figure 20 and Figure 21 that the overheads of taskloop are much larger than the other. Therefore, our assumption is proved.

4.3 Synchronisation benchmarks

In terms of synchronisation microbenchmark, we have updated the atomic test with a new clause and have extended the lock test with hints. Besides, the variant barrier benchmark was also generated to guarantee different thread reaches the barrier last. Hence, we examine the mentioned extending here.

4.3.1 Atomic update

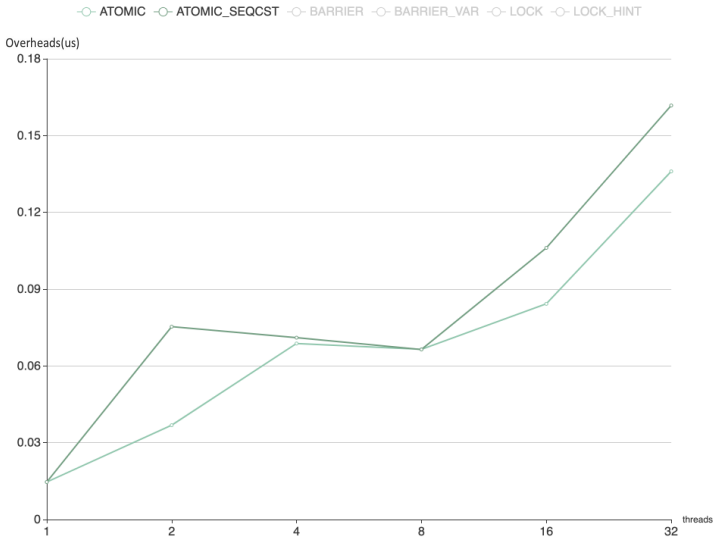

Figure 22: Atomic tests compiled by gcc

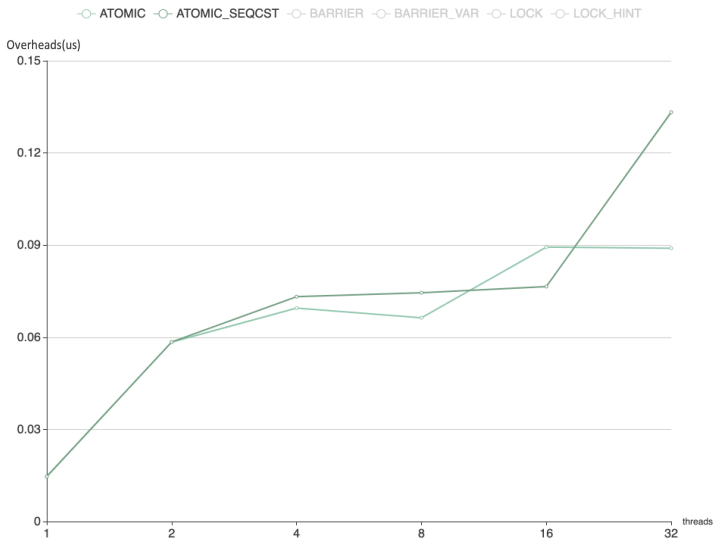

Figure 23: Atomic tests compiled by Intel

With `seq_cst` memory-order-clause, we set more strict memory order to atomic parallelism, which enhances the execution cost. Thereby, we expect that `ATOMIC SEQ_CST` tests have worse perfor-

mance than the original tests. Based on the results in Figure 22 and Figure 23, we can see the `ATOMIC SEQ_CST` tests indeed have higher overheads than the original ones for both GCC execution and Intel execution. It should be mentioned that the atomic seq_cst possibly does not have much extra overhead here since the Cirrus platform based on Intel Xeon(R) CPU E5-2695 v4 has a strong memory consistency model. We expect to see larger overheads on other hardware like ARM platforms. The Cirrus back end's noise may cause the special case in Figure 23 when the thread number is 8.

4.3.2 Barrier update

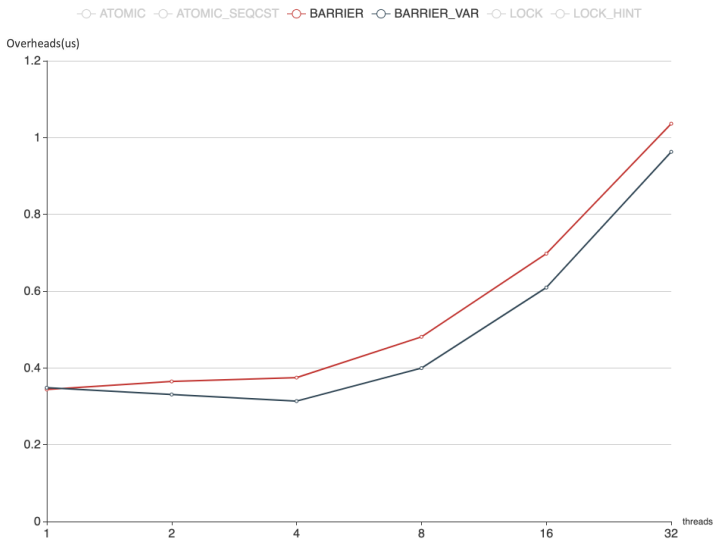

Figure 24: Barrier tests compiled by GCC

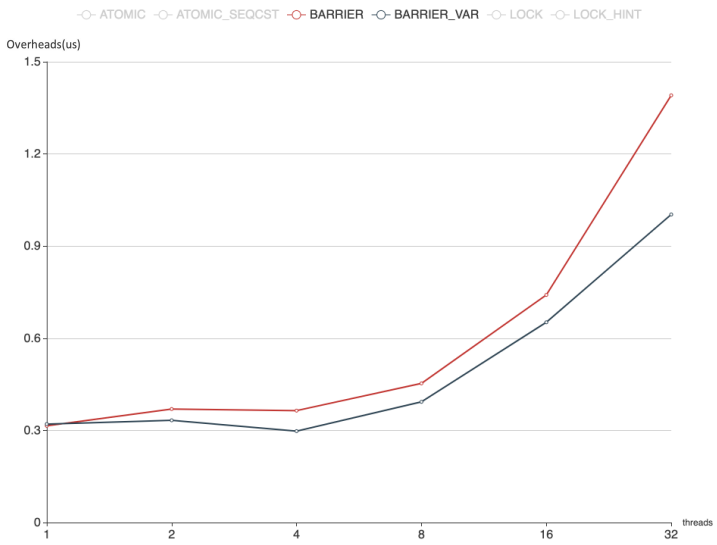

Figure 25: Barrier tests compiled by Intel

When it comes to new barrier tests, the variant-barrier tests guarantee different threads last to reach the barrier. In contrast, the original barrier tests have the potential of the same thread last to arrive at the barrier. In the barrier variant test, the idle threads that have no work to do and can execute part of the barrier code before the last thread arrives, which can make some progress of the performance. Thereby, the overheads of variant-barrier tests should be lower than the original barrier tests. As a result, we figure out that both runs compiled by GCC and Intel compilers prove our assumption. The behavior of variant-barrier tests is a bit better than the previous ones.

4.3.3 Lock update

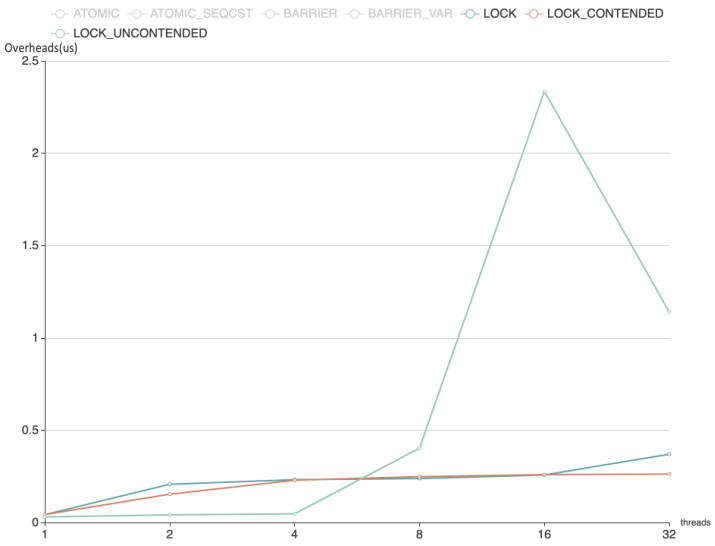

Figure 26: Lock tests compiled by Intel

We should mention that the contended/uncontended lock and simple lock depend on the particular parallelism implementations. Figure 26 tells that when the number of threads is below 4, the uncontended lock initiation provides the best performance. When the number of threads becomes bigger, the behavior of uncontended lock execution soon gets the worst, while the simple lock parallelism and contended lock parallelism share similar behavior.

4.4 Task benchmarks

In terms of task benchmarks updates, we extend the task generation with dependencies to manage the task execution sequence. The results of the experiments are demonstrated below.

4.4.1 Task updates

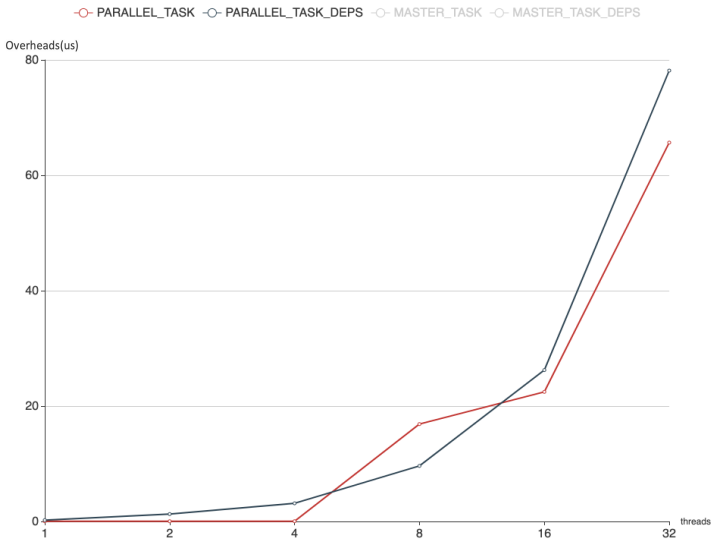

Figure 27: ParallelTask tests compiled by GCC

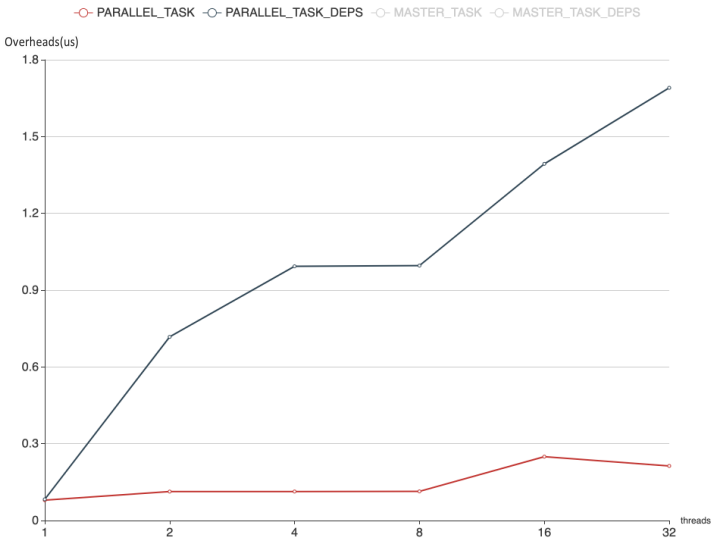

Figure 28: ParallelTask tests compiled by Intel

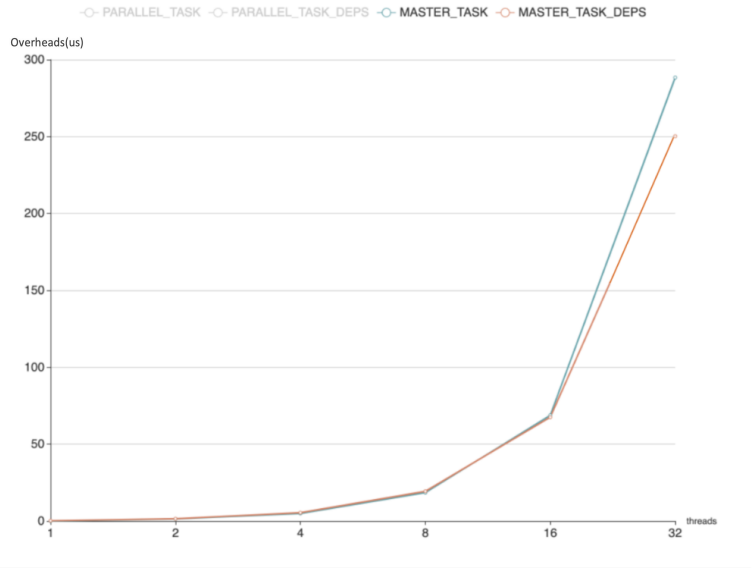

Figure 29: MasterTask tests compiled by GCC

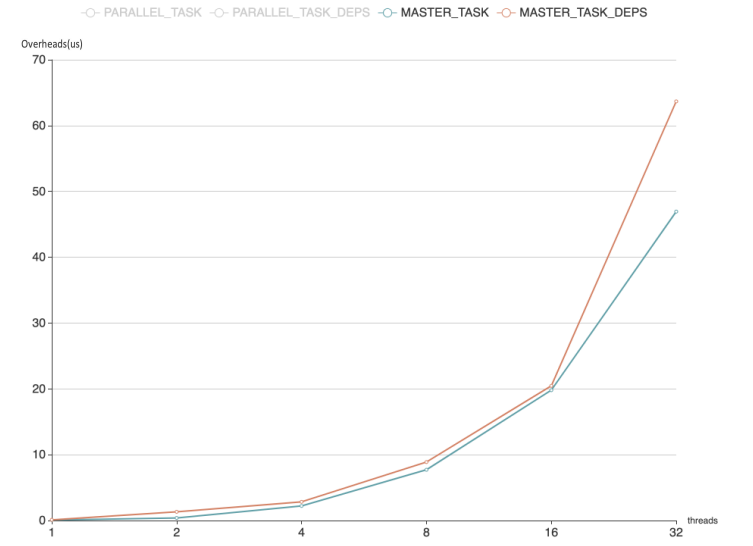

Figure 30: MasterTask tests compiled by Intel

We added dependencies to the original `MasterTask` tests and `ParallelTask` tests. The task-dependent requires extra computation for managing the sequence of task execution, therefore increasing the execution cost. We expect the task-dependent tests have higher overheads than the original `MasterTask` and `ParallelTask` benchmarks. We can see from the four figures above that task-dependent performance tests are worse than the original ones. Note that most of the overheads are probably from the task generation's cost rather than the task execution for MasterTask tests. Due to `ParallelTask` tests enable all worker threads to call the `delay` function, while `MasterTask` tests only enable master thread to call the `delay` function, which causes the former tests having better performance.

5 Result Visualisation

Developing the visualisation module to demonstrate the benchmarking results is another essential part of the dissertation. We plan to implement the raw data pre-processing at first. After that, we post the handled benchmarking results to the frontend interface. Figure 31 demonstrates the workflow of the design of visualisation.

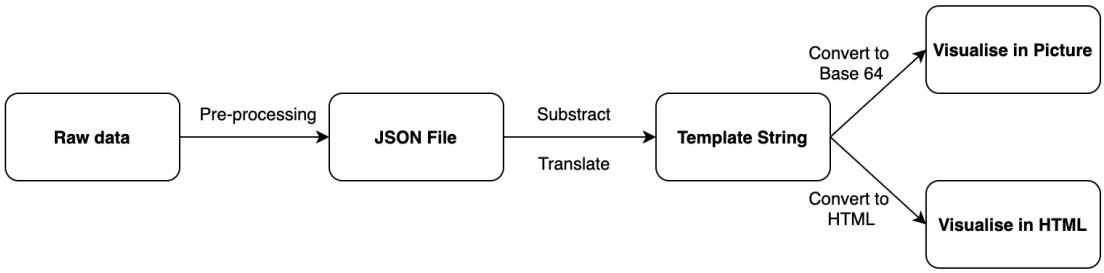

Figure 31: Visualisation workflow

We pre-process the raw benchmark results to the JSON file first. The reason why we adopt tidy the data to JSON format is that JSON is a lightweight format of data-interchange. It has high re-usability and is simple to parse and generate by machines. After that, the target data in JSON are subtracted and translated to the visualisation template strings. Finally, we provide two options for visualisation. One is converting the template strings to picture files by encoding to base64. The other is converting to HTML, which permits users to select which benchmark to demonstrate, and the webpage supports the local picture downloading.

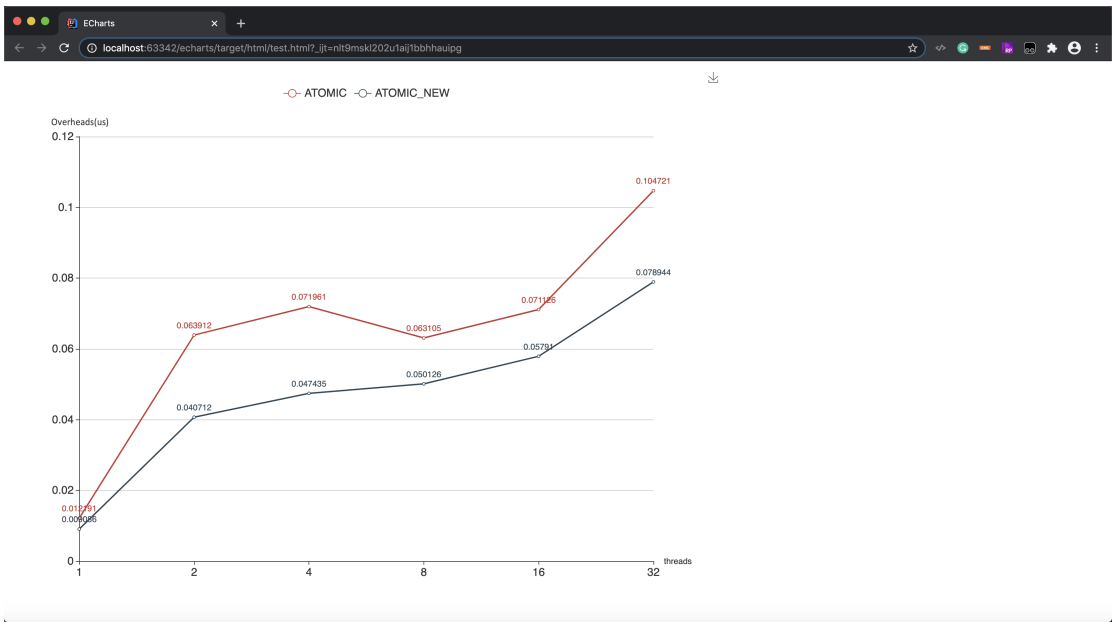

Figure 32: Visualisation screenshot

Figure 32 is a screenshot of HTML visualisation. After we run the compiled **App.java** file, both graph files and Html files are generated. The interface of HTML allows users to select a particular benchmark to illustrate. For example, a user can click *ATOMIC_NEW* in the legend area to hide the results of it. What is more, the HTML page allows users to download the picture to the local disk. Users can simply

click the download icon on the top of the page to download the generated graph. Details of visualisation are shown below.

5.1 Data Processing

We parse the experimental results to JSON files for further data visualisation. In practice, we add some data-preprocessing code to the original `common.c` to output the experimental results into files. Listing 17 demonstrates the changes of the original `common.c`

```
void printfooter(char *name, double testtime, double testsd,
     double referencetime, double refsd) {
   FILE *f;
   f = fopen(name,"a+");
   if(f == NULL)
   {
   printf("Error!");
   exit(1);
   }
   printf("%s time = %f microseconds +/- %f\n",
    name, testtime, CONF95*testsd);
   printf("%s overhead = %f microseconds +/- %f\n",
    name, testtime-referencetime, CONF95*(testsd+referencesd));
   fprintf(f,
       "{\"Name\":\"%s\",\"Threads\":\"%d\",\"Overhead\":\"%f\",\"Error
   \":\"%f\"}\n",name,nthreads,testtime-referencetime,
       CONF95*(testsd+referencesd));
   fclose(f);
}
```

Listing 17: Generate JSON files

Hence we gain lists of benchmark results in JSON format. Then, we need to merge the separate JSON files into only one JSON file, which is required by Front-end chart generating. To accomplish further data processing, we build Java scripts to merge single JSON files into one JSON file.

Besides, we need to create `Benchmark` entity class to manage each benchmark result like parsing `"Name":"PARALLEL TASK","Threads":"1","Overhead":"0.028726","Error": "0.000460"` to a benchmark entity where we can handle the elements including `Name, Threads, Overhead` so far and so forth. The scripts of JSON merging and Benchmark entity are illustrated in the appendix.

5.2 Chart Generating

When it comes to data plotting, the chart template called Apache Echarts is adopted to plot the benchmark results through the JSON file. Apache Echarts is a data visualisation mechanism based on JavaScript, which is high performance and easy-to-use. Echarts is compatible with vast latest browsers like Firefox, Chrome, IE, Safari so on and so forth.[3]

Echarts provides the styles and visualisation effects of the charts. We can modify the parameters in the template file to build the expected chart style. To fill the data to the plot, we need to parse our generated JSON into the Echarts template. The template reserves a declared object **option** for users to fill the figures and manage the plot formatting. Therefore, the pattern of plotting the benchmark results is dynamically parsing the benchmarking results from the `Benchmark` entities to the object **option**. On the one side, we build a set template file prepared to receive the related information from benchmark entities. The code of the template sample is shown in Listing 25 in Appendix.

On the other side, we post the required data (which are managed by `Benchmark` entities) from the back-end to fill the data gaps. Listing 18 illustrates an example to generate the template string.

```
HashMap<String, Object> datas = new HashMap<>();
      datas.put("names", JSON.toJSONString(names));
      datas.put("Threads", JSON.toJSONString(Threads));
      datas.put("barrier_overhead",JSON.toJSONString(barrier_overhead));
      datas.put("barrier_var_overhead",JSON.toJSONString(barrier_var_overhead));
      String option = FreemarkerUtil.generateString("sync.ftl, "/template/",
          datas);
```

Listing 18: Data Posting

By far, we gain the complete **option** object. After that, we simply export the generated Echarts template to HTML file or encode the template file to the base64 file for data visualisation.

5.3 Source code summary

The whole visualisation developing is based on Java language, which is an object-oriented and class-based language. Due to its portability and re-usability, we choose to utilise Java language to implement result visualisation. Figure 33 illustrates the overview of the source code.

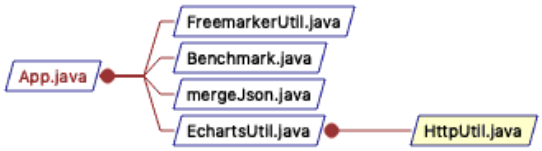

Figure 33: Visualisation workflow

- FreemarkerUtil.java - It generates visualisation template strings by data lists.
- HttpUtil.java - It provides HTTP *post* function and generate the response entity.
- EchartsUtil.java - It provides the functions of *base64 encoding* and *Html generating* based on the template strings as well as the response entity.
- mergeJSON.java - It merges the multiple single JSON files to only one JSON file.
- Benchmark.java - It is an entity class for benchmark results, which contains benchmark constructor as well as *get* and *set* functions.
- App.java - It glues all the components together and does the initialisations.

We demonstrate all the visualisation scripts later in the Appendix.

6 Conclusion

We have successfully updated the EPCC OpenMP Microbenchmark suite v3.0 to cover the latest OpenMP specifications. To begin with, the ideas of previous versions of EPCC OpenMP Microbenchmark suites, as well as the overhead calculation and the concrete tests, were briefly introduced, which helps with the understanding of the background.

Moreover, we corrected the known bug for `ATOMIC` benchmark capturing closer overhead measurement to the reality, which increases the accuracy and correctness of the suite. Then we start our core extending through reasonable steps: original benchmarks analysis, latest OpenMP specification discussion, new benchmark implementation, and comparisons. We firstly detailed analysed the original benchmarks to investigate the potentials of each benchmark. After that, we discussed the latest OpenMP specifications based on the investigation. What is more, several new benchmarks were generated based on the latest usages of the OpenMP standard. Last but not least, we theoretically compare the difference between the previous benchmarks with the designed benchmarks.

To learn about the performance of new benchmarks in practice, we designed sets of experiments to examine the overheads of the extended benchmarks and the original ones. The experiments were processed on the back end of the Cirrus with up to 32 OpenMP threads. The designed benchmarks and the corresponding benchmarks were plotted together to illustrate the performance differences between them. To explore the behavior of new benchmarks and understand the inner mechanisms, we made general result assumptions theoretically and analysed the underlying reasons. Finally, we compared the guesses with the reality to prove our understandings. Generally, the experimental results meet our expectations, which proved the correctness and effectiveness of our benchmark extending.

Additionally, the data visualisation module was successfully implemented to visualise the benchmark results. We accomplished the data cleaning step, which parses the original results to the standard JSON file. Then, we extracted the effective data elements and posted the data elements to the front. Hence, the result of plotting is successfully generated.

To conclude, the goal of the dissertation project has been generally achieved. We successfully make two significant contributions, including previous suite extending and visualisation module implementation.

Future work

As for future work, the accelerator offloading capability can be involved in the suite. After a brief investigation, we could translate some of the EPCC OpenACC benchmarks to the OpenMP suite. In practice, the translation is feasible because the construct model of OpenMP implementations can be mapped to similar usages and concepts in OpenACC. [15] Table 6 demonstrates some similar directives shared by both OpenMP and OpenACC.

OpenMP	OpenACC	Description
parallel	kernals	They both initialise the parallel regions in the code. OpenMP generates a team of threads executing on a multi-threaded platform, and OpenACC offloads the computation to the GPU.
for	loop	They both specify that the iterations associated with loops will be run in parallel. OpenMP assigns the iterations to the member threads, and OpenACC declares the type of parallelism to execute the loop.
task	async & wait	They both do the task creation. OpenMP declares an explicit task. As for OpenACC, it is feasible to combine both async and wait clauses to generate tasks.
sections	async & wait	As for OpenMP, it is a non-iterative work-sharing construct consisting of structured blocks assigned among the member threads. When it comes to OpenACC, it is potential to combine both async and wait to create sections.

Table 6: Selective comparison between OpenMP and OpenACC directives

Besides, there have been some researches on OpenACC translation porting several OpenACC constructs to OpenMP constructs including OpenMP scheduling translations, barrier translations, tasking translations, so on and so forth.[15] Therefore, we could mitigate some benchmarks in the OpenACC suite to the current OpenMP suite. Moreover, we might evaluate the performance of translated benchmarks and the original OpenACC benchmarks and calculate the speedup. The speedup can be roughly defined like

$$Speedup = T(OpenACC)/T(OpenMP)$$

where T(OpenACC) is the execution time of the original OpenACC benchmark, and T(OpenMP) is the execution time of the OpenMP benchmark translated from OpenACC. We could make some analysis based on the benchmark results to investigate the underlying reasons.

Besides, the current benchmark experiments were processed on Cirrus and exclusively executed by the Intel compiler and GNU compiler due to the time limit. We can mitigate the suite to a variety of different compilers and hardware platforms in the future.

7 Appendix

Listings of visualisation source code

```
package service;
import java.io.*;
import java.util.ArrayList;

public class mergeJson {
   public static void generateJson() throws IOException {
      ArrayList<String> list = new ArrayList<String>();
      mergeJson test = new mergeJson();
     //Get the benchmark names.
      list = test.getAllFileName("./raw");
     //Copy the single strings to one string.
      String output ="[";
      for(int i = 0; i < list.size(); i++){
         output = output + test.getContent(list.get(i))+",\n";
      }
      output = output.substring(0,output.length()-2);
      output = output+"]";
     //Output the processed string to single JSON file.
      File writename = new File("output.json");
      writename.createNewFile();
      BufferedWriter out = new BufferedWriter(new
          FileWriter(writename,true));
      out.append(output);
      out.flush();
      out.close();
   }
   public static ArrayList<String> getAllFileName(String path) {
      ArrayList<String> fileNameList = new ArrayList<String>();
      boolean flag = false;
      File file = new File(path);
      File[] tempList = file.listFiles();
      for (int i = 0; i < tempList.length; i++) {
         if (tempList[i].isFile()) {
            fileNameList.add(tempList[i].getName());
         }
      }
      return fileNameList;
   }
   public String getContent(String filename) throws IOException {
      FileInputStream s = new FileInputStream(new File("./raw/" + filename));
      byte[] b = new byte[fis.available()];
      s.read(b);
      fis.close();
      String str2 = new String(b);
      return str2;
   }
}
```

Listing 19: mergeJSON.java

```
package pojo;
public class Benchmark {
   private String name;
   private int threads;
```

```
    private double overhead;
    private double error;
    private double Chunksize;
    public Benchmark(String name, int threads, double overhead, double error,
        double Chunksize) {
        this.name = name;
        this.threads = threads;
        this.overhead = overhead;
        this.error = error;
        this.Chunksize=Chunksize;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getThread() {
        return threads;
    }

    public void setThreads(int threads) {
        this.threads = threads;
    }

    public double getOverhead() {
        return overhead;
    }

    public void setOverhead(double overhead) {
        this.overhead = overhead;
    }

    public double getError() {
        return error;
    }

    public void setError(double error) {
        this.error = error;
    }

}
```

Listing 20: Benchmark.java

```
package util;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.PrintStream;
import java.util.HashMap;
import java.util.Map;
import org.apache.http.client.ClientProtocolException;
import com.alibaba.fastjson.JSON;
import com.alibaba.fastjson.JSONObject;
```

```
public class EchartsUtil {
   private static String url = "http://localhost:7777";
   private static final String SUCCESS_CODE = "1";
   public static String generateEchartsBase64(String option) throws
       ClientProtocolException, IOException {
      String base64 = "";
      if (option == null) {
         return base64;
      }
      option = option.replaceAll("\\s+", "").replaceAll("\"", "'");
      // post template strings to echartsConvert server
      Map<String, String> params = new HashMap<>();
      params.put("opt", option);
      generateHTML(option);
      String response = HttpUtil.post(url, params, "utf-8");

      // Parse echartsConvert response entity.
      JSONObject responseJson = JSON.parseObject(response);
      String code = responseJson.getString("code");
      if (SUCCESS_CODE.equals(code)) {
         base64 = responseJson.getString("data");
      }
      else {
         String string = responseJson.getString("msg");
         throw new RuntimeException(string);
      }
      return base64;
   }
   public static void generateHTML(String option) throws
       FileNotFoundException {
      StringBuilder stringHtml = new StringBuilder();
         PrintStream printStream = new PrintStream(new
             FileOutputStream("./target/html/test.html"));
//Generate HTML file
      stringHtml.append("<!DOCTYPE html>\n" +
            "<html>\n" +
            "<head>\n" +
            "   <meta charset=\"utf-8\">\n" +
            "   <title>ECharts</title>\n" +
            "   <script src=\"echarts.min.js\"></script>\n" +
            "\n" +
            "\n" +
            "   <script src=\"jquery.min.js\"></script>\n" +
            "</head>\n" +
            "<body>\n" +
            "   <div id=\"main\" style=\"width:
                1024px;height:768px;\"></div>\n" +
            "   <script type=\"text/javascript\">\n" +
            "      var myChart =
                echarts.init(document.getElementById('main'));\n"
                +"myChart.showLoading();"+
            " myChart.setOption("+option+");\n" +
            "\n" +" setTimeout(function() { myChart.hideLoading(); }, 3000);
                "+
            "   </script>\n" +
            "</body>\n" +
            "</html>");
      try{
         printStream.println(stringHtml.toString());
      }catch (Exception e) {
```

```
            e.printStackTrace();
        }
    }
}
```

Listing 21: EchartUtil.java

```
package util;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;
import org.apache.http.HttpEntity;
import org.apache.http.NameValuePair;
import org.apache.http.client.entity.UrlEncodedFormEntity;
import org.apache.http.client.methods.CloseableHttpResponse;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClients;
import org.apache.http.message.BasicNameValuePair;
import org.apache.http.util.EntityUtils;


public class HttpUtil {
    public static String post(String url, Map<String, String> params, String
        charset)
            throws IOException {
        String responseEntity = "";
        // Create CloseableHttpClient object
        CloseableHttpClient client = HttpClients.createDefault();
        // Exploit post function
        HttpPost httpPost = new HttpPost(url);
        // Generate parameters
        List<NameValuePair> nameValuePairs = new ArrayList<>();
        if (params != null) {
            for (Entry<String, String> entry : params.entrySet()) {
                nameValuePairs.add(new BasicNameValuePair(entry.getKey(),
                    entry.getValue()));
            }
        }
        // Add parameters to post request
        httpPost.setEntity(new UrlEncodedFormEntity(nameValuePairs, charset));
        // Send the post request
        CloseableHttpResponse response = client.execute(httpPost);
        // Acquire the response entity
        HttpEntity entity = response.getEntity();
        if (entity != null) {
            responseEntity = EntityUtils.toString(entity, charset);
        }
        // Release and consume the entity.
        EntityUtils.consume(entity);
        response.close();
        return responseEntity;
    }
}
```

Listing 22: HttpUtil.java

```java
package util;

import java.io.File;
import java.io.IOException;
import java.io.StringWriter;
import java.util.Map;
import freemarker.template.Configuration;
import freemarker.template.Template;
import freemarker.template.TemplateException;

public class FreemarkerUtil {
   private static final String path =
       FreemarkerUtil.class.getClassLoader().getResource("").getPath();
   public static String generateString(String templateFileName, String
       templateDirectory, Map<String, Object> datas)
          throws IOException, TemplateException {
      Configuration configuration = new
          Configuration(Configuration.VERSION_2_3_0);
      // Set the default encoding
      configuration.setDefaultEncoding("UTF-8");
      // Set the template folder
      configuration.setDirectoryForTemplateLoading(new File(path +
          templateDirectory));
      // Generate the template object
      Template template = configuration.getTemplate(templateFileName);
      //Write datas to the template and return it
      try (StringWriter stringWriter = new StringWriter()) {
         template.process(datas, stringWriter);
         stringWriter.flush();
         return stringWriter.getBuffer().toString();
      }
   }
}
```

Listing 23: FreemarkerUtil.java

```java
import Decoder.BASE64Decoder;
import com.alibaba.fastjson.JSON;
import freemarker.template.TemplateException;
import org.apache.commons.io.IOUtils;
import org.apache.http.client.ClientProtocolException;
import pojo.Benchmark;
import service.mergeJson;
import util.EchartsUtil;
import util.FreemarkerUtil;

import java.io.*;
import java.text.ParseException;
import java.util.*;

public class App {
   public static void main(String[] args) throws ClientProtocolException,
       IOException, TemplateException, ParseException {
      mergeJson.generateJson();
      InputStream inputStream = new FileInputStream("atomic.json");
      String text = IOUtils.toString(inputStream,"utf8");
      List<Benchmark> list = JSON.parseArray(text, Benchmark.class);
      List<String> names = new ArrayList<>();
```

```
        List<Integer> threads = new ArrayList<>();
        List<Double> overheads = new ArrayList<>();
        List<Double> atom_overhead = new ArrayList<>();
        List<Double> atomnew_overhead = new ArrayList<>();
        for(Benchmark d:list){
            names.add(d.getName());
            threads.add(d.getThread());
            switch (d.getName()){
                case "ATOMIC": atom_overhead.add(d.getOverhead()); break;
                case "ATOMIC_NEW": atomnew_overhead.add(d.getOverhead()); break;
            }
        }
        removeDuplicateWithOrder(names);
        removeDuplicateWithOrder(threads);
        // Template parameters initialisation
        HashMap<String, Object> datas = new HashMap<>();
        datas.put("names", JSON.toJSONString(names));
        datas.put("threads", JSON.toJSONString(threads));
        datas.put("atom_overhead",JSON.toJSONString(atom_overhead));
        datas.put("atomnew_overhead",JSON.toJSONString(atomnew_overhead));
        // Generate option template strings
        String option = FreemarkerUtil.generateString("atomic.ftl",
            "/template/", datas);
        // Generate base64 string by template strings
        String base64 = EchartsUtil.generateEchartsBase64(option);
        generateImage(base64, "test.png");
    }

    // Delete duplicated elements in the ArrayList
    public static void removeDuplicateWithOrder(List list) {
        Set set = new HashSet();
        List newList = new ArrayList();
        for (Iterator iter = list.iterator(); iter.hasNext();) {
            Object element = iter.next();
            if (set.add(element))
                newList.add(element);
        }
        list.clear();
        list.addAll(newList);
    }
    public static void generateImage(String base64, String path) throws
        IOException {
        BASE64Decoder decoder = new BASE64Decoder();
        try (OutputStream out = new FileOutputStream(path)){
            // Decode
            byte[] b = decoder.decodeBuffer(base64);
            for (int i = 0; i < b.length; ++i) {
                if (b[i] < 0) {
                    b[i] += 256;
                }
            }
            out.write(b);
            out.flush();
        }
    }
}
```

Listing 24: App.java

```
{
   legend: {
      textStyle: {
      fontSize:'16'
         },
   show:true,
   top:'25px',
   data: ${names}
         },
   tooltip: {
      trigger: 'axis'
          },
   toolbox: {
      feature: {
         saveAsImage: {}
             }
          },
   grid:{
      top:'100px',
      left:'100px',
      right:'100px'
       },
   xAxis: {
      axisLabel: {
      textStyle: {
      fontSize:'16'
               }
         },
   type: 'category',
   boundaryGap: false,
   name: 'threads',
   data: ${threads}
},
   yAxis: {
      axisLabel: {
          textStyle: {
            fontSize:'16'
         }
},
   type: 'value',
   name: 'Overheads(Ms)'
},
   series: [
   {
      name: 'ATOMIC',
      type: 'line',
      itemStyle: {normal:{label:{show:true}}},
      data: ${atom_overhead}
   },
   {
      name: 'ATOMIC_NEW',
      type: 'line',
      itemStyle: {normal:{label:{show:true}}},
      data: ${atomnew_overhead}
   }
         ]
}
```

Listing 25: atomic.ftl

Experimental results

	Thread 1	Thread 2	Thread 4	Thread 8	Thread 16	Thread 32
static	-3.277	12.116	17.163	18.809	18.882	52.762
static_monotonic	-1.829	12.174	22.360	20.761	21.187	53.370

Table 7: Schedbench_static execution by GCC

	cksz 1	cksz 2	cksz 4	cksz 8	cksz 16	cksz 32	cksz 64	cksz 128
staticn	16.940	20.267	18.773	26.183	16.652	17.026	17.506	16.496
staticn_monotonic	21.671	17.113	18.268	17.396	17.192	16.880	16.190	17.218
dynamicn	98.474	57.061	38.539	28.368	23.507	21.511	20.550	19.989
dynamicn_momonotonic	96.829	55.731	37.343	27.686	23.951	21.795	26.499	18.789
dynamicn_nonmonotonic	95.335	54.640	35.834	25.620	20.292	18.427	17.058	16.808
guidedn	21.023	25.326	42.796	42.943	77.814	334.109	429.548	495.142
guidedn_monotonic	18.120	24.784	41.761	42.843	77.816	333.368	427.659	495.133
guidedn_nonmonotonic	21.134	26.064	41.837	41.745	77.020	333.658	428.201	494.048
Taskloop	Broken	Broken	Broken	Broken	Broken	87.347	59.344	44.231

Table 8: Schedbench_chunksize execution by GCC with 8 threads

	Thread 1	Thread 2	Thread 4	Thread 8	Thread 16	Thread 32
barrier	0.343	0.364	0.374	0.481	0.697	1.036
barrier_varient	0.348	0.330	0.313	0.399	0.609	0.962
lock	0.023	0.070	0.092	0.207	0.265	0.425
atomic	0.014	0.036	0.068	0.066	0.084	0.136
atomic_seqcst	0.014	0.075	0.071	0.066	0.106	0.161

Table 9: Syncbench execution by GCC

	Thread 1	Thread 2	Thread 4	Thread 8	Thread 16	Thread 32
parallel_task	0.079	0.113	0.113	0.113	0.249	0.213
parallel_task_deps	0.082	0.717	0.993	0.995	1.393	1.690
master_task	0.078	0.392	2.215	7.711	19.807	46.936
master_task_deps	0.101	1.328	2.839	8.884	20.463	63.679

Table 10: Taskbench execution by GCC

	Thread 1	Thread 2	Thread 4	Thread 8	Thread 16	Thread 32
static	-0.721	12.329	96.299	181.373	27.055	3076.985
static_monotonic	-0.667	11.137	18.509	232.411	25.238	3174.795

Table 11: Schedbench_static execution by Intel

	cksz 1	cksz 2	cksz 4	cksz 8	cksz 16	cksz 32	cksz 64	cksz 128
staticn	121.373	16.290	17.288	15.850	15.626	15.212	16.499	15.159
staticn_monotonic	197.159	16.490	15.373	13.468	13.204	13.361	12.964	13.382
dynamicn	146.243	80.093	49.124	34.235	27.274	21.687	19.794	19.440
dynamicn_momonotonic	133.768	74.885	48.714	33.535	25.195	21.756	20.280	19.873
dynamicn_nonmonotonic	134.752	74.883	46.376	31.214	24.571	21.152	18.336	18.284
guidedn	60.475	26.908	39.001	105.416	73.325	312.716	187.280	592.701
guidedn_monotonic	77.097	26.594	37.864	105.978	75.469	313.009	185.500	590.164
guidedn_nonmonotonic	63.849	31.116	41.261	109.045	80.059	316.061	187.326	591.073
Taskloop	996.902	550.812	330.306	191.737	117.454	79.219	57.187	51.278

Table 12: Schedbench_chunksize execution by Intel with 8 threads

	Thread 1	Thread 2	Thread 4	Thread 8	Thread 16	Thread 32
barrier	0.314	0.369	0.364	0.452	0.740	1.390
barrier_varient	0.320	0.332	0.297	0.392	0.651	1.002
lock	0.042	0.206	0.231	0.238	0.256	0.369
lock_contended	0.042	0.153	0.228	0.248	0.259	0.261
lock_uncontended	0.029	0.041	0.046	0.401	2.334	1.140
atomic	0.014	0.058	0.069	0.066	0.089	0.088
atomic_seqcst	0.014	0.058	0.073	0.074	0.076	0.133

Table 13: Syncbench execution by Intel

	Thread 1	Thread 2	Thread 4	Thread 8	Thread 16	Thread 32
parallel_task	0.079	0.113	0.113	0.113	0.249	0.213
parallel_task_deps	0.082	0.717	0.993	0.995	1.393	1.690
master_task	0.078	0.392	2.215	7.711	19.807	46.936
master_task_deps	0.101	1.328	2.839	8.884	20.463	63.679

Table 14: Taskbench execution by Intel

References

[1] Barlas, Gerassimos *Chapter 4 - Shared-Memory Programming: OpenMP.*Multicore and GPU Programming, Elsevier Inc, pp. 165–238, 2015.

[2] Chapman, Barbara, et al *OpenMP in a Heterogeneous World*. Springer Berlin / Heidelberg, 2012.

[3] D. Li, H. Mei *ECharts: A declarative framework for rapid construction of web-based visualization* 2018.

[4] D. Li, *A Parallel PCG Solver for Large-Scale Groundwater Flow Simulation Based on OpenMP* pp. 306–309, 2011.

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

[6] J.M. Bull and D. O'Neill, *A microbenchmark suite for OpenMP 2.0.*SIGARCH Comput. Archit. News, vol. 29, no. 5, pp. 41–48, 2001.

[7] J. M. Bull, F. Reid and N. McDonnell, *A microbenchmark suite for OpenMP tasks.* in Proceedings of the 8th international conference on OpenMP in a Heterogeneous World (IWOMP '12) pp. 271-274, 2012.

[8] J. Protze *Thread-local concurrency: a technique to handle data race detection at programming model abstraction* pp.144-155, 2018.

[9] J. Schuchart, Mathias Nachtmann, José Gracia *Patterns for OpenMP Task Data Dependency Overhead Measurements* pp.156-168, 2017.

[10] Massaioli, Federico *OpenMP Parallelization of Agent-Based Models* Parallel Computing, vol. 31, no. 10, pp. 1066–1081, 2005.

[11] Marowka, Ami. *TBBench: A Micro-Benchmark Suite for Intel Threading Building Blocks* Journal of Information Processing Systems, vol. 8, no. 2, pp. 331–346, 2012.

[12] OpenMP ARB, *OpenMP Application Programming Interface Version 5.0*. 2018

[13] OpenMP ARB, *OpenMP Application Programming Interface Version 4.5*. 2015

[14] OpenMP ARB, *OpenMP Application Programming Interface Version 4.0*. 2013

[15] S. Pino, L. Pollock, S. Chandrasekaran. *Exploring translation of OpenMP to OpenACC 2.5: Lessons Learned*, 2017

[16] Speziale *An Optimized Reduction Design to Minimize Atomic Operations in Shared Memory Multiprocessors* pp. 1300–1309, 2011

[17] W. Cheng , Chandrasekaran *Portable Mapping of OpenMP to Multicore Embedded Systems Using MCA APIs* CM SIGPLAN Notices, vol. 48, no. 5, pp. 153–162, 2013.