# ASSIGNMENT 2: OPENMP PARALLELISM

Shuai Ke - 1609628

December 8, 2017

## 1 Introduction

This report will improve the serial code in three ways. There are OpenMP, New features in OpenMP v4 and MPI, and for each optimisation in the three aspects, it will contain a description of the implementation, a comparison of run-time and an analysis of the result. The *Figure-1* shows the summary of the best run-times in each section, comparing with original run-times. The left y-coordinate is the run-times in $N = 2000$ and the right is in $N = 4000$.

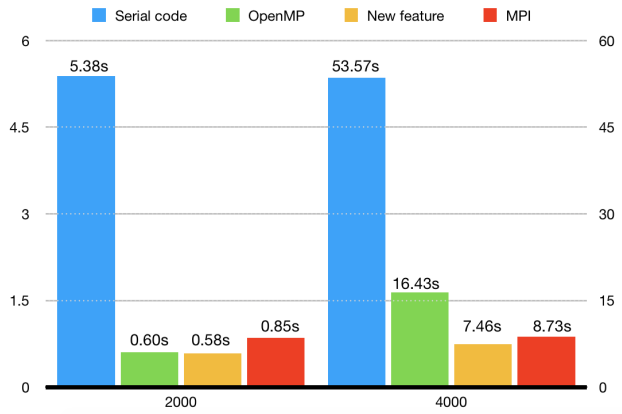


Figure 1: The Summary of the best Run-times

## 2 OpenMP

There are different performances by paralleling program in different ways. This section will compare the run-times in which different parallel methods and analyse the reason for which they have various performances. It has three layers of loops in the 'run' function. The first layer is the iteration, and for each iteration, the program has two layers of loops in the main part, denoted as the second and third layer loop respectively. All optimisations of parallel programming will focus on the last two layers.

### 2.1 Paralleling in the Third Layer Loop

First of all, the third layer loop can be paralleled by using **#pragma omp parallel for** statement, and since the threads share the accumulator named 'dot', the program uses **#pragma omp critical** statement to make sure no more than one thread modifies the accumulator at the same time. However, it spends more time than unoptimized code after paralleling in third layer loop, and the performance is worse when using more threads. The results of the original code and paralleling in different thread numbers are shown in the *Table-1* (the matrix size is $1000 * 1000$).

| | original | 1 thread | 2 threads | 4 threads |
|---|---|---|---|---|
| run-times | 0.57s | 42.52s | 203.58s | 276.34s |

Table 1: Run-times of threading in the third layer loop

There are three possible reasons for this phenomenon. The first one is CPU frequently create and destroy threads. Denoted the time of creating and destroying thread are $T_c$ and $T_d$, $Ntread$ is the number of threads and $Iters$ is the total number of iterations given data size in $N$, then the extra run-time equals to $Iters * N * Ntread * (T_c + T_d)$. Assuming $N$ is $1000$, the $Iters$ will be 2957, then $N * Iters$ is $3 * 1e^6$ approximately. So even $(T_c + T_d)$ is very small, the extra run-time cannot be ignored after multiplying a sufficiently large coefficient. The second possible reason is the inefficient rate of vectorisation. We can have compiler reports by using **-ftree-vectorizer-verbose=2** flag, and the serial code can vectorize the third loop, while it cannot be vectorized after paralleling. Because the vectorization analysis in compile-time cannot predict the behaviours of parallelization which is dynamically distributing tasks to each thread at the run-time. The last one is the barrier. Since all thread shares the accumulator, the program needs using **critical** statement which leads to block. In another word, when one thread operates the shared accumulator, the others need to wait for the first thread finish the operation.

### 2.2 Threading Overheads

As mentioned before, frequently creating and destroying teams will lead to extra consumption. So in this part, the program still parallels the third layer loop, but create the threads in the second layer loop, to reduce the threading overheads. Additionally, this way cannot solve the inefficient rate of vectorisation problem since it still parallels in the third layer loop. Consider the number of threads is 4, the threading overheads are only $0.01s$ in this way, compared with $8.176s$ in creating threads in the third layer loop.

Besides, there is a conditional statement **if(row != col)** in the third layer loop. Therefore, the program divides the loop in two parts, the first part iterates from $0$ to $row - 1$ and the second one is from $row+1$ to $N-1$, to reduce computation in conditional statement. The **nowait** statement can eliminate the waiting time between two loops since threads create in advance. The run-times of using **nowait** and without **nowait** are shown in *Table-2* (the matrix size is $1000 * 1000$).

Comparing with the third layer loop, the reduced time is the threading overheads. Although the program using **nowait** command has a better result than without using **nowait**, this optimisation still not improve the performance. There is

| thread number | 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| wait | 48.80s | 204.13s | 232.42s | 267.47s |
| nowait | 48.52s | 160.13s | 204.56s | 250.33s |

Table 2: wait vs nowait

a line-by-line profile by using VTune. If the thread number is $8$, the total run time will be $2164s$(the sum of all thread run-times), and the **critial** and **barrier** statements account for a large proportion, $58.09\%$ and $41.27\%$ respectively. Therefore, the program has an unexpected performance in this optimisation since it has $99.4\%$ time in the block.

## 2.3 Shared Accumulator

Because the blocking time is caused by the accumulator, this part will discuss and compare the efficiencies in four methods of threads accessing the accumulators below.

### 2.3.1 Shared One Accumulator

In this way, the program uses one shared accumulator and implements mutual exclusion by **#pragma omp critical**. Since the threads creating in the second layer loop, the procedure of initialization and calculating $xtmp[row]$ should execute in one thread rather than each thread. In addition, the other threads should wait the initialization process finished and the calculation of variable $xtmp[row]$ should wait for all threads finished. In order to achieve this, the program uses **#pragma omp master** and **#pragma omp barrier** statements. The *Figure-2* shows the procedure of this method. The blue lines are the dependencies which lead the blocking time (**barrier** statement). The block causing by **critical** is happened in paralleling section.

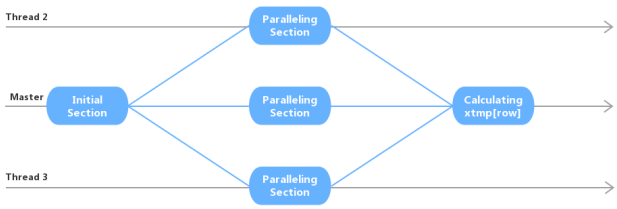


Figure 2: Shared Accumulator Procedure

### 2.3.2 Array of Accumulators

In order to reduce the blocking time, the program can create an array as the accumulators. Although all threads share the array, the private accumulator of each thread has their own position. In addition, it still needs **master** and **barrier** statements in summation and $xtmp[row]$ variable computations. The *Table-3* shows the results of this method below (the matrix size is $1000 * 1000$).

| thread number | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| run-times | 3.77s | 6.35s | 9.59s | 7.78s | 16.53s |

Table 3: Run-times of array of accumulators

It has a significant improvement, but the main reason for the effect of efficiency is still blocking. For example, the blocking time occupies $96\%$ under 8 threads. And it also has another drawback, thrashing cache. Thrashing occurs when a computer's virtual memory subsystem is in a constant state of paging, rapidly exchanging data in memory for data on disk, to the exclusion of most application-level processing (Wiki).

### 2.3.3 Private Accumulators

Using private accumulator, it can reduce the impact of thrashing cache. It means that each thread has their private accumulator. And after paralleling section, all private accumulators should add to a shared accumulator. However, this process needs **critical** command to ensure access to only one thread at a time. It not only spends additional blocking time in **critical** command, but also increases the waiting time of **barrier** command. Because the final computation should strictly follow the summation. The blocking time comparison under $8$ threads and $N = 1000$ is shown in *Table-4*. It clear that the blocking time causing by **barrier** statement is increased and the increment approximately equals to the blocking time causing by **critical** statement.

| | barrier | critical | others | total |
|---|---|---|---|---|
| array | 62.682 | 0s | 2.61s | 65.292s |
| private | 125.033 | 49.149s | 5.778s | 179.960s |

Table 4: Blocking time comparison under 8 threads

### 2.3.4 Reduction

OpenMP will make a copy of the reduction variable per thread, and after the loop, the local variable will be combined into the global variable using the reduction operator. Although the **reduction** command has a better load balance through a tree structure, it still cannot avoid the block causing by $xtmp[row]$ variable computations. So the run-times in direct proportion to the thread number, like the run-time in 1 thread are 1.76s, 2 threads are 5.63s and 8 threads 17.70s (the matrix size is $1000$).

## 2.4 Paralleling in the Second Layer Loop

This optimisation will parallel the second layer loop. Since each iteration does not need sharing the accumulator, the threads are completely paralleling, which reduces the blocking time caused by **critical** and **barrier** statements. In addition, the third layer loop can be vectorised, showing by the compiler report, because the whole loop will do in one thread. Therefore, the result has a significant improvement. The line chart *Figure-3* shows the relationship between thread number and program run-time. The x-coordinate is the thread number and the y-coordinate is the value of run-times. In the beginning, the computing power is the bottleneck, thus it has a remarkable increase after adding some threads. While the memory bandwidth will be the new bottleneck when the threads are plenty, so the run-times will tend to steady. Considering $N = 2000$, if the program uses one thread to process data, the spin time is equal to zero, which means the rate of data loading is bigger than the rate of CPU processing data so that the CPU with no need for waiting data. However, after increasing thread number to $16$, there is $1.159s$ spin time when $N = 2000$ and $75.37s$ when $N = 4000$. Therefore, the memory bandwidth is the main factor which affects the program performance. Besides, there is an unexpected phenomenon when thread number is bigger than

16, the CPU usage is lower than $100\%$ in each thread. For example, $46.56\%$ in $N = 2000$ and $68.99\%$ in $N = 4000$ when thread number is 17, notice that the CPU usage is $100\%$ when threads are no more than 16. It also has low spin time since the core will switch thread when the current thread waits for data. In $N = 4000$, the spin time offsets the effect of low usage, by contrast, the performance under 17 threads is worse than 16 threads since the spin time is small in $N = 2000$.

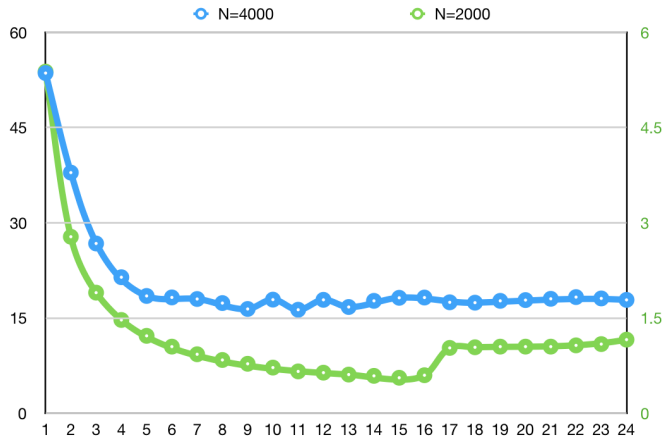


Figure 3: Relationship between Thread and Run-time

### 2.5 Load Balance

Load imbalance is another possible factor which will affect the results. The **schedule** statement can control the distribution of tasks, and the default setting is **static** dividing the loop into equal-sized chunks or as equal as possible. There are also other options, **dynamic** using a work queue to give a chunk-sized block of tasks to each thread; **guided** same to dynamic scheduling, but the chunk size is decreasing. The result of **static**, **dynamic** and **guided** without setting chunk size are $0.60s$, $1.36s$ and $1.71s$ respectively under $16$ threads when $N = 2000$. And if we set chunk size equals to $125$ by using **dynamic** model, the program has the same run-time with **static** model. Therefore, the best way is distributing tasks averagely since the calculated quantity in each iteration is equal $(o(N))$.

## 3 New Features in OpenMP v4

OpenMP v4 provides some new features, like CPU affinity and SIMD directive. This section tries to use those features to improve the program.

### 3.1 NUMA

Under NUMA, a processor can access its own local memory faster than non-local memory, which also provides higher overall memory bandwidth. The *Figure-4* shows the structures of UMA and NUMA.

Since O/S uses 'first touch' policy to control placement of items across the banks of memory, the program should rearrange the array $A$ in each thread. In order to let the data which will be processed later arrange to the local memory,

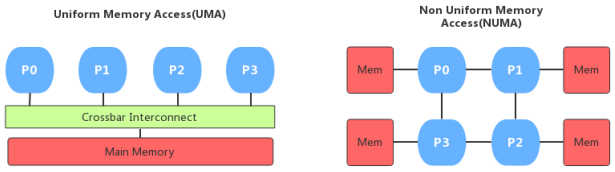


Figure 4: The structures of UMA and NUMA

the rearrange section and the calculating section should use the same paralleling schedule. It means that the thread rearranges the data to the local memory, then in the calculating section, it will have a faster rate of data loading. Additionally, the environment variable **OMP_PLACES** control the binding policy, it has three values, threads(corresponding to a single hardware thread), cores(corresponding to a single core) and sockets(corresponding to a single socket). The results of three policies in different thread numbers show in *Table-6* (the matrix size is 4000 * 4000).

| | 1 | 4 | 8 | 12 | 16 |
|---|---|---|---|---|---|
| threads | 53.69s | 16.95s | 14.64s | 9.61s | 7.54s |
| cores | 53.64s | 16.92s | 14.61s | 9.46s | 7.46s |
| socket | 53.84s | 14.87s | 9.47s | 9.24s | 7.55s |

Table 5: Run-times of three binding policies

The **threads** and **cores** models have similar run-times because the device not open hyper-threading. And it clear that it has a huge improvement by using the features of NUMA, because it enhances the memory bandwidth. The CPU spin time reduces to $9.17s$ under 16 threads (the total run-time is $7.46 * 16$), compared with the original spin time $75.37s$.

### 3.2 SIMD Directive

SIMD instructions can operator multiple data with a single instruction. Therefore, the program adds **#pragma omp simd** in the third layer loop to improve the computational efficiency. However, the run-time does not change since the compiler already vectorises this loop after using flag -**O3**.

### 3.3 Analysis

According to the **Amdahl's Law**, we can calculate the ideal parallel speed-up $(\frac{1}{P/N+S})$ and the real parallel speed-up $(\frac{T_1}{T_p})$. Notice that the parallelisable fraction in the original code is $99.4\%$ when $N = 4000$, denoted as $P$. Additionally, the program uses **cores** model to correspond thread and core, which can limit the core number by changing thread number. The line chart (*Figure-5*) shows the comparison, the x-coordinate is the thread number and the y-coordinate is the value of speedup.

It clear that the real speed-up increase slower than the ideal speed-up. Analysing this situation by **Roofline Model**, the operational intensity equals to $3/12 = 1/4$ (FLOPS/byte) in **jacobi.c** since the main statement **dot += A[ridx + col] * x[col]**. Considering the core number is $8$, the single

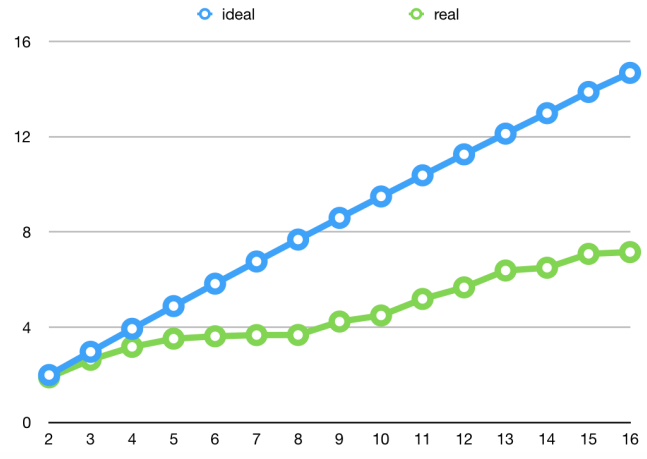


Figure 5: The comparison of speed-up in real and ideal

precisions equal to $5.10$ GFLOPS/s in $N = 2000$ and $2.55$ GFLOPS/s in $N = 4000$. The *Figure-6* shows that the L1 is the bottleneck when $N = 2000$ and the L3 is the bottleneck when $N = 4000$. In other words, when the program uses more cores, the memory bound and the compute bound are both increasing, but the compute power grows faster than memory bound. Therefore, the unexpected real speed-up caused by the memory bandwidth bound.

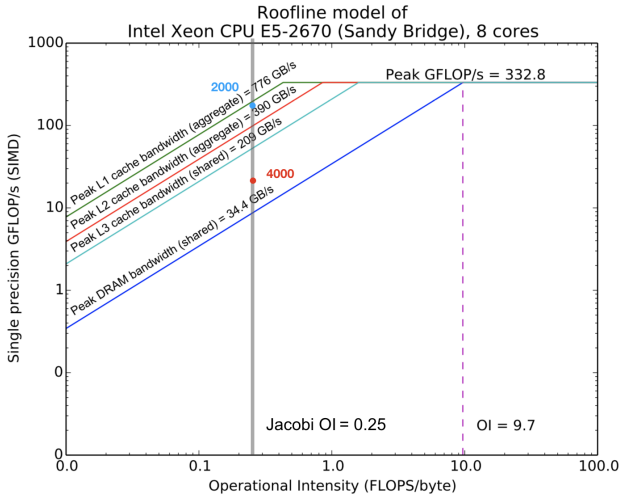


Figure 6: Roofline Model

# 4 MPI

In this section, the program will optimise from serial code by using MPI. Message Passing Interface (MPI) is a standardized and portable message-passing standard.

## 4.1 Paralleling in the Third Layer Loop

Like optimising serial code by OpenMP, it also can parallel in the third layer loop. Firstly, each process calculates their own region of this loop and stores the value in the variable named $local\_dot$; then using **MPI_Send** function to send the value to master process; master process uses **MPI_Recv** function to receive the values, figures out the summation $dot$ and the variable $xtmp[row]$; after that, master process will send the $xtmp[row]$ back to each process. Although it implements the paralleling, the result is not well. The run-times under different process numbers when $N = 2000$ are shown in *Table-6*. The process number is control by the parameter $ppn$ in the **jacobi.job** file.

| processes | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| run-times | 6.68s | 13.56s | 17.66s | 37.44s | 119.37s |

Table 6: Paralleling in the third layer loop by MPI

There are two main reasons for the worse run-times. Firstly, the **MPI_Send** and **MPI_Recv** functions will cause blocking, which means the call will wait for the parameters are safe for re-use in the program. Although this implement ensures the answer is correct, it wastes too many CPU resources. Secondly, the third loop can not vectorise effectually because it divides into many pieces and calculates in different process numbers.

## 4.2 Paralleling in the Second Layer Loop

The program also can parallel in the Second Layer Loop. It divides the loop into $nprocs$ chunks, $nprocs$ is the process number, and each process calculates a continuous set of variables $xtmp[row]$. Then, the master process receives those chunks from each process by using **MPI_Recv** function, splices them to one chunk and sends it back to each process because the $xtmp$ will be used to calculate the values in the next iteration. In this optimisation, the program not only has a simple implementation, but also reduces the blocking time caused by data exchange. For each iteration, it only needs once data exchange between the master process and the others, while paralleling in the second layer loop needs $N$ times. In addition, the third layer loop can be vectorised effectively. The results of this optimisation under different processes are shown in *Table-7*.

| processes | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| N=2000 | 5.44s | 2.83s | 1.55s | 1.01s | 0.85s |
| N=4000 | 54.08s | 26.97s | 20.21s | 13.21s | 8.73s |

Table 7: Paralleling in the third layer loop by MPI

# 5 Conclusion

In this report, the optimisations are distributed to three classes. The first one is using OpenMP to enhance computing power through multithreading. Many problems appearing in the experiments, like threading overheads, blocking and nonvectorisation, are solved one by one. The second section is based on the first part, which using the new features in OpenMPv4. It aims to address the memory bandwidth bound, and by using UNMA, the result has a remarkable effect, which is $0.58s$ in $N = 2000$ and $7.46s$ in $N = 4000$. The last part focuses on MPI, comparing with OpenMP, it needs to modify the code structure since it parallels in process. But MPi provides a method that communicating in the process, it can horizontal expansion when CPU or physical machine is sufficient.