Multithreading

Yeldos Balgabekov
yb2n17@soton.ac.uk

April 10, 2018

Abstract

In this report a comparative analysis of different scheduling techniques to enable multi-threaded computing was completed. The experiment stats with observation of the behavior of scheduling types such as "static", "auto", "dynamic", or "guided" adjusting number of chunks where possible. The second half of the report describes implementation of an affinity scheduling. Throughout the whole experiment two same experiments were conducted. To complete this work ARCHER's training materials [EPCb] was used and ARCHER machines were accessed [EPCa].

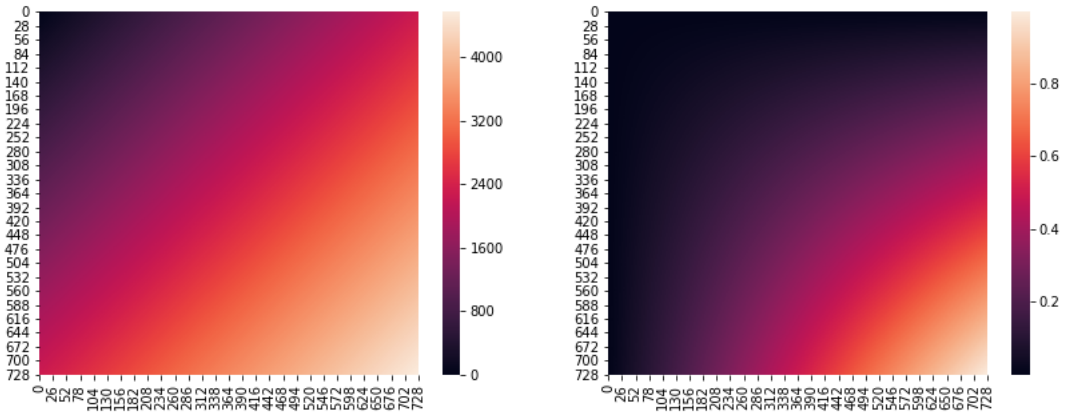

Figure 1: Initial NxN Matrices: for loop 1 (left) and loop 2 (right)

1 Introduction

In this report we can observe comparative analysis of several multi-threaded programming scenarios. We will see how parameters such as scheduling type, chunk size, and number of threads affect computation speed. The analysis will be conducted on two loops. On Figure 1 we can observe the initial matrices used for further computations. In the first loop we transform the upper triangle of the initial matrix and store the result in an NxN matrix that was initially filled with zeros:

```
for (i=0; i<N; i++){
    for (j=N-1; j>i; j--){
    a[i][j] += cos(b[i][j]);
    }
}
```

In the second loop we transform the initial matrix and store the result in an array of the length N that was initially filled with zeros as well:

```
for (i=0; i<N; i++){
    for (j=0; j < jmax[i]; j++){
```

```
            for (k=0; k<j; k++){
                c[i] += (k+1) * log (b[i][j]) * rN2;
            }
        }
}
```

where array $jmax$ is generated under the following conditions:

```
for (i=0; i<N; i++){
    expr =  i%( 3*(i/30) + 1);
    if ( expr == 0) {
      jmax[i] = N;
    }
    else {
      jmax[i] = 1;
    }
}
```

Looking at the both of the loops we may witness that the workload per iteration changes over the cycle what is depicted on the Figure 2. The loop workload per iteration is steadily falling for the Loop 1 and is "spiky" for Loop 2. The "spikes" are driven by the array $jmax$

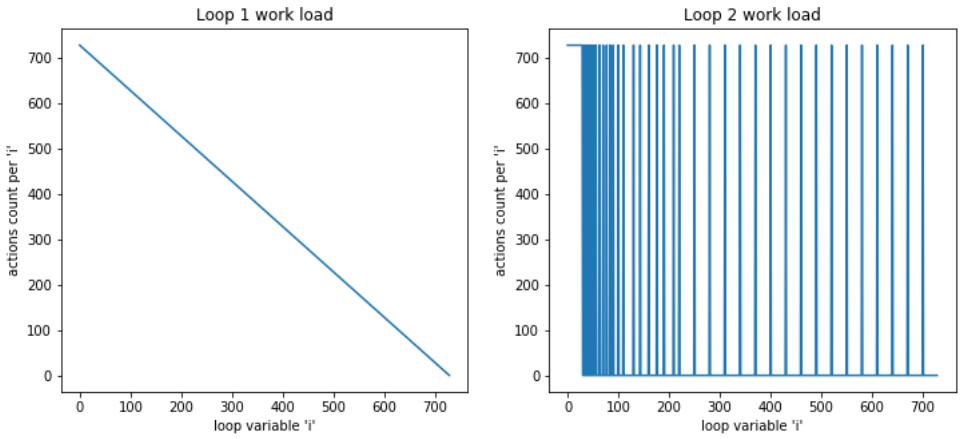

Figure 2: Work load per iteration

To obtain more demonstrative comparison of multi-threaded scenarios we will iterate the above mentioned loops 100 times each and measure the time.

2 Parallelization

2.1 Implementation

In our experiment, we will use OpenMP parallelization directives ("#pragma omp parallel") to wrap those sections of the code to be parallelized. Scope of the variables within the parallelization sections will be classified using shared or private. We will compile the code with 'cc -O3 filename', where $-O3$ flag is used to optimize the computation [NL]. To avoid recompilation of our programs we will use $SCHEDULE(RUNTIME)$ specifying the scheduling type through system environment variables such as $OMP\_SCHEDULE$ or $OMP\_NUM\_THREADS$. Finally, to run all of the scenarios we will use files *runall* and *Compact_runall* could be used. The latter file saves results in a compact way into the file *Compact_report.txt*:

```
#!/bin/bash

## declare an array of threads count
declare -a scheduleType=("static" "dynamic" "guided")
```

```
declare -a chunkSize=(1 2 4 8 16 32 64)
declare -a numThreads=(1 2 3 6 12 24)

cc -O3 compact_loops.c

echo "threads_num,sched_type,chunk_size,loop1_time,loop2_time,total_time,vaild1,valid2
## Run different scheduling types on 6 threads
export OMP_NUM_THREADS=6
export SCHED="static"
export CHUNK_SIZE=null
export OMP_SCHEDULE="static"
./a.out >> Compact_report.txt

export OMP_NUM_THREADS=6
export SCHED="auto"
export CHUNK_SIZE=null
export OMP_SCHEDULE="auto"
./a.out >> Compact_report.txt

export OMP_NUM_THREADS=6
for SCHED in "${scheduleType[@]}"
do
    for CHUNK_SIZE in "${chunkSize[@]}"
    do
        export SCHED=$SCHED
        export CHUNK_SIZE=$CHUNK_SIZE
        export OMP_SCHEDULE="$SCHED,$CHUNK_SIZE"
        ./a.out >> Compact_report.txt
    done
done

## Run the best schedule type on various number of threads (For Loop 1)
export SCHED="guided"
export CHUNK_SIZE=4
export OMP_SCHEDULE="$SCHED,$CHUNK_SIZE"

for threads in "${numThreads[@]}"
do
    export OMP_NUM_THREADS=$threads
    ./a.out >> Compact_report.txt
done

## Run the best schedule type on various number of threads (For Loop 2)
export SCHED="dynamic"
export CHUNK_SIZE=8
export OMP_SCHEDULE="$SCHED,$CHUNK_SIZE"

for threads in "${numThreads[@]}"
do
    export OMP_NUM_THREADS=$threads
    ./a.out >> Compact_report.txt
done

## Run affinity
cc -O3 compact_loops_affinity.c

for threads in "${numThreads[@]}"
do
```

```
    export OMP_NUM_THREADS=$threads
    ./a.out >> Compact_report.txt
done
```

2.2 Comparison of the Scenarios

We will time the speed of Loop 1 and Loop 2, implementing the above mentioned actions. As the first step, we will use 6 threads and adjust the schedules types "static", "auto", "static, n", "dynamic, n", "guided, n" with the chunk size $n = x^2$ where $x \subset [0, 6]$. Running the experiment, we may observe that the fastest result was achieved using the schedule time "guided,4" for Loop 1 (Figure 3) and "dynamic,8" for Loop 2 (Figure 4): around 0.033 secs and 0.098 for Loop 1 and Loop 2 correspondingly.

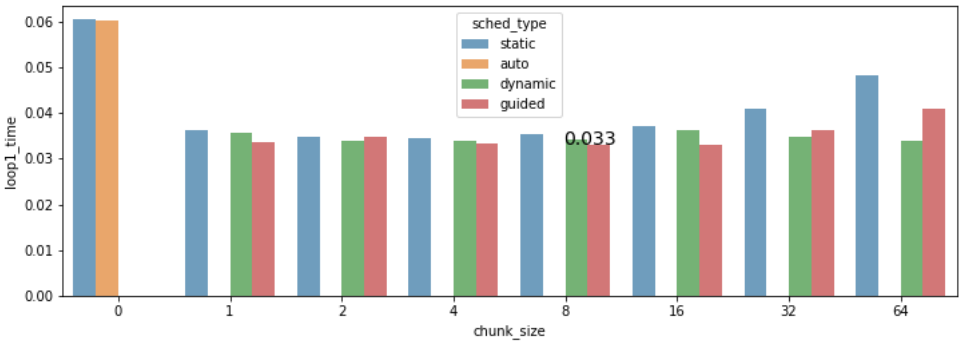

Figure 3: Schedule type comparison on 6 threads, Loop1

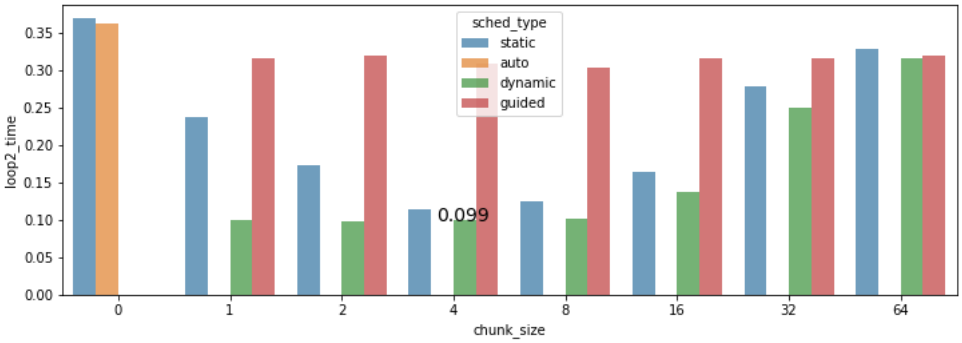

Figure 4: Schedule type comparison on 6 threads, Loop2

All in all the Figures 5 and 6 represent the speed up ratios that we achieved exploiting multi-threaded computing vs computing on one thread only.

To improve the computational speed we have also conducted an experiment increasing number of threads (1, 2, 3, 6, 12, 24) for "guided, 4" (for Loop 1) and "dynamic,8" (for Loop 2). A visual representation could be seen on the Figure 7, where a bigger optimization was achieved using 12 threads instead of 24.

2.3 Analysis of the Results

We have seen that the scheduling type could be adjusted based on the workload structure. From Figure 2 we have seen that Loop 1 workload is gradually and steadily decreasing over time. Thus, when we tested on 6 threads we have seen quite long computation for "STATIC" and "AUTO" scheduling types (no chunks explicitly declared). They were even worse comparing to a one thread scenario due to additional actions driven by parallelization routines. Looking at the "Static" type

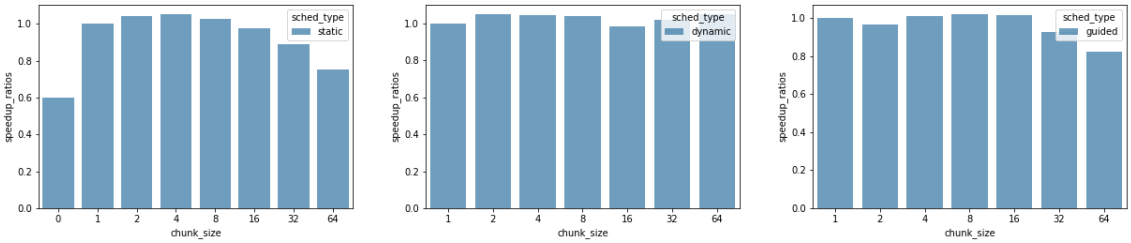

Figure 5: Loop 1 Speed Up Ratios

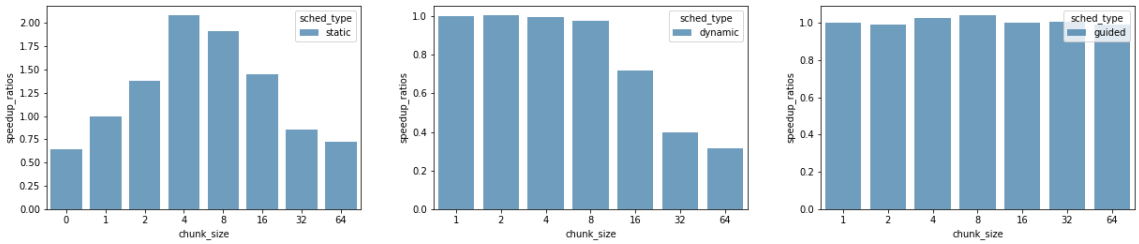

Figure 6: Loop 2 Speed Up Ratios

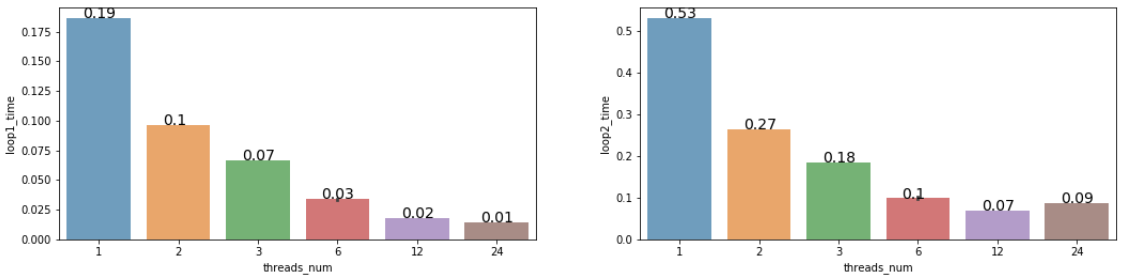

Figure 7: Thread count adjustment performance (left: Loop 1, Right: Loop 2)

we can see a lower optimization utilizing a higher number of threads (32, 64 threads). This is because the workload was as not as big to effectively use all the threads.

"Dynamic" and "Guided" scheduling types on the other had "balanced" the workload between all the threads as work using first-come-first-served principle. "Guided" scenarios were slightly better as the chunks decreased over time while for "dynamic" the chunk size is the same. The speed-up ratios are the logical consequences of the above summary. Going further we have seen that a bigger number of threads allowed us to optimize the work even more: to 0.01 sec with 24 threads vs 0.03 with 6 threads.

Structurally Loop 2 (Figure 2) is different and has a "spiky" nature. In this case we have seen the fastest results on a "dynamic" scheduling type. "Guided" scheduling type did not give us good results in this case as the first threads are significantly overwhelmed with the tasks (Figure 2) and the later ones end up with idle waiting. If Loop 1 speed up was more or less stable, Loop 2 shows quite interesting results for Static type. We can observe highest improvement with 4 and 8 threads. One of the reasons was a high base: it was around 0.25 secs with 1 thread. Advancement of the "dynamic" scheduling type optimization, which was the best for Loop 2, by adjusting threads count showed us the topmost improvement at 12 threads: 0.07 secs.

3 Affinity scheduling

In this part we will observe affinity scheduling where we will manually allocate the workload (number of iterations a thread has) between the threads. As before we will use '#pragma omp parallel' OpenMP directives, compiling the code with 'cc -O3 filename'. In the code, we will allocate the work based on the concept where a thread that finished its initial workload will search for a thread that have most workload to do. The algorithm in a nutshell looks as follows:

- Allocate the initial workload between the threads (T)
- Once Thread A has finished its own tasks, it will find another thread with the highest workload (Thread B) and take a part of the Thread B's workload proportionally to (1/T).
- Note 1. To enable cache reuse for Thread B we will reallocate the "farthest"/"rightmost") tasks from the origin.
- Note 2. A thread should complete its own tasks first to allow cache reuse.
- Note 3. To make this algorithm working it is important to allocate the initial workload (N/T) gradually, so that those threads that will finish their works will have a chance to work with the unallocated workload.
- Keep doing the above steps until the all computation is completed.

Doing so we guaranteed that all threads will be engaged into computing till the end.

Figure 8 represents the results achieved for Loop 1 and Loop 2 using different number of threads. Comparing to the results in the Section 2, affinity scheduling type in general demonstrated better results, which minimizes idle time. From these figures we can see that using 12 threads is the most optimal scenario for 'Loop 1', while for 'Loop 2' it is 24 threads.

Finally, we have achieved 1.73 times improvement for Loop 1 (0.033027 secs with "guided,4" vs. 0.019062 secs with "affinity"), and 2.26 times improvement for Loop 2 (0.09886 with "dynamic,8" secs vs. 0.043776 secs with "affinity"), making "affinity" scheduling type the most optimal in our experiment.

4 Conclusion

In this work we have analyzed different scheduling techniques for parallelization and how they are workload balance dependent. We have experimented with "static", "auto", "dynamic", "guided", and "affinity" scheduling types as well as with chunk sizes and number of threads where it was possible. Adjusting number of threads we have witnessed the limitations when the workload is not big enough for a large number of threads. "Guided" scheduling type was the quickest for Loop 1, while "affinity" scheduling type on 24 threads gave us better results for the "spiky" workload (Loop 2).

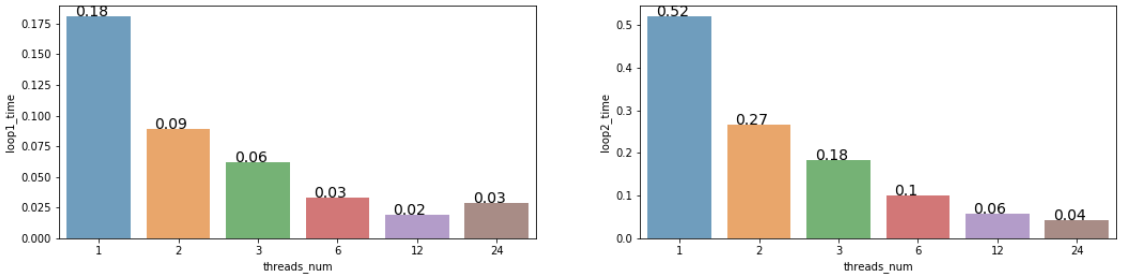

Figure 8: Affinity: thread count adjustment performance (left: Loop 1, Right: Loop 2)

References

[EPCa] EPCC. Archer hpc system.

[EPCb] EPCC. Shared-memory programming with openmp.

[NL] NERSC and Berkeley Lab. Cray compilers (fortran, c, c++).