Threaded Programming
Coursework Part 1

Chaolin Han

s1898201

1 Introduction

In this coursework several parallelised programming tests were performed to study the loop scheduling in OpenMP. Two different loops obtained their best performance at different schedule levels. Time consumptions were recorded and analysed to explain effects of different scheduling options in parallel loops.

2 Timed Tests

There are two sections in the testing. Section 1 is testing the time performance among five different schedule clause options, while three of them have a vary chunksize. The best scheduling option for each loop is found and used in the next section.

Section 2 focuses on the performance at different level of thread numbers, followed by further discussion in speedup and scheduling strategy.

3 Results and Explanation

The programme is run and timed on the back end of Cirrus. The kind and chunksize of the schedule clause are variables used to find the scheduling option that have the highest performance, i.e. gives the shortest running time.

3.1 Section 1

Five scheduling options are tested in this section and repeated 10 times, with an average execution time recorded for analysis. The last three scheduling options are tested on different chunksize ($n = 1, 2, 4, 8, 16, 32, 64$). The trend of execution times affected by chunksize are presented Figure 1 and Figure 2.

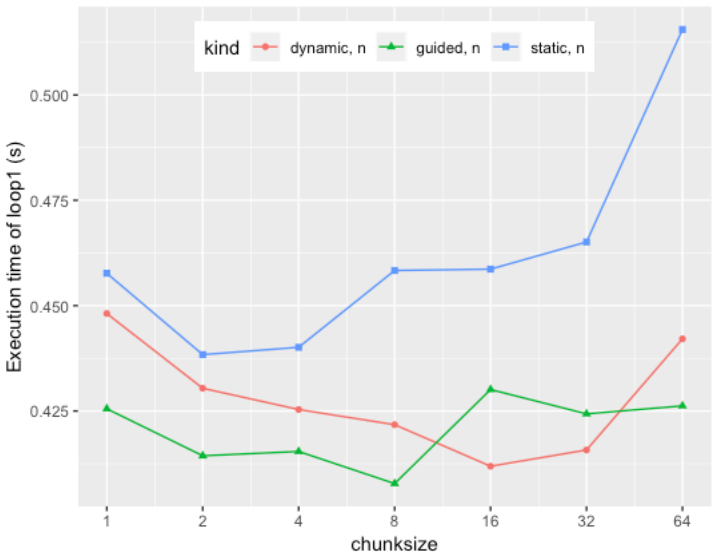

Figure 1: Execution time of different chunksize for loop1

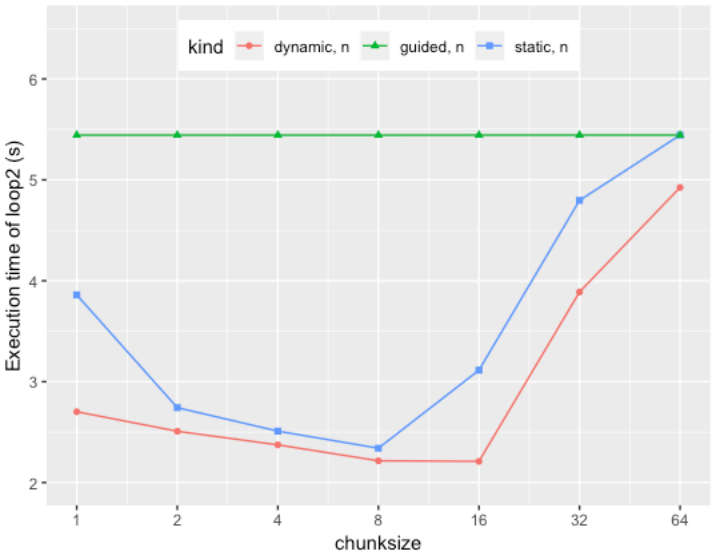

Figure 2: Execution time of different chunksize for loop2

For loop1, the execution time of DYNAMIC scheduling options declines as chunksize increases, but when chunksize is greater than 16 the downward trend stops, and execution time start to increase. The execution time of GUIDED fluctuate slightly between 0.407s and 0.431s. For STATIC, increasing chunksize from one to two results in a better performance but as chunksize increases form two on, the execution time becomes longer.

For loop2, GUIDED stays at almost the same level (around 5.444s) regardless of the chunksize. Later graph shows that the results of AUTO is almost the same as that of GUIDED. For STATIC and DYNAMIC, the execution time drops when chunksize is at a lower level and increase at higher level of chunksize. DYNAMIC reaches its best performance at chunksize = 16 with execution time 2.210s, whilst for STATIC chunksize = 8 gives the best execution time 2.340s.

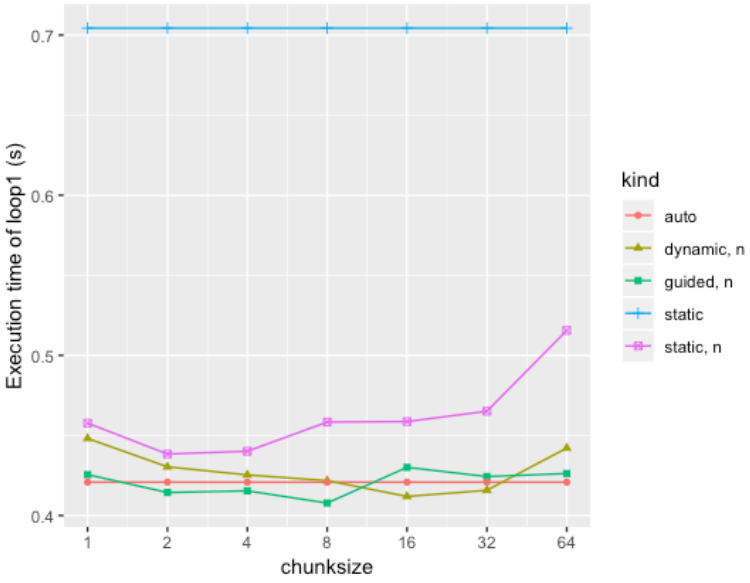

Figure 3: Execution time of different scheduling options for loop1

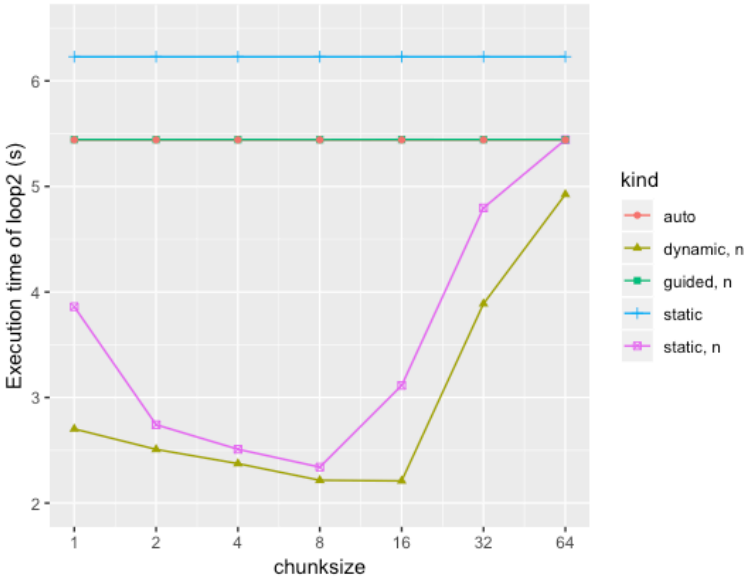

Figure 4: Execution time of different scheduling options for loop2

Together with the first two scheduling options without chunksize, the execution times are plotted in Figure 3 and Figure 4. Among these scheduling options, $(GUIDED, 8)$ and $(DYNAMIC, 16)$ give the best execution time to loop1 and loop2 respectively.

It is noticed that in the graph of loop2, the plotted line of GUIDED is mostly the same as that of AUTO, which indicates that when taking AUTO scheduling, the runtime evolves to behave like GUIDED when considering load balance and low overheads, but it is not as efficient as $(DYNAMIC, n)$ and $(STATIC, n)$.

3.2 Section 2

As the best option for each loop are decided in section 1, timing tests on different number of threads are carried out on the back end of Cirrus. Let P denote the number of threads, the speedup (S) and efficiency (E) can be calculated from the data obtained from the output, using

$$S(P) = T\_1/T\_P$$

And

$$E(P) = S(P)/P = T\_1/(P\ T\_P\ )$$

Where T_1 is the serial time running on one thread.

The speedup is plotted in Figure 5. The ideal linear speedup is showed in dotted black line. The speedup of loop1 grows nearly linearly as thread number increases, but loop2 would encounter an “elbow point” at four threads. When the number of threads is less than 4 speedup grows nearly linearly but when it is greater than 4 the speedup

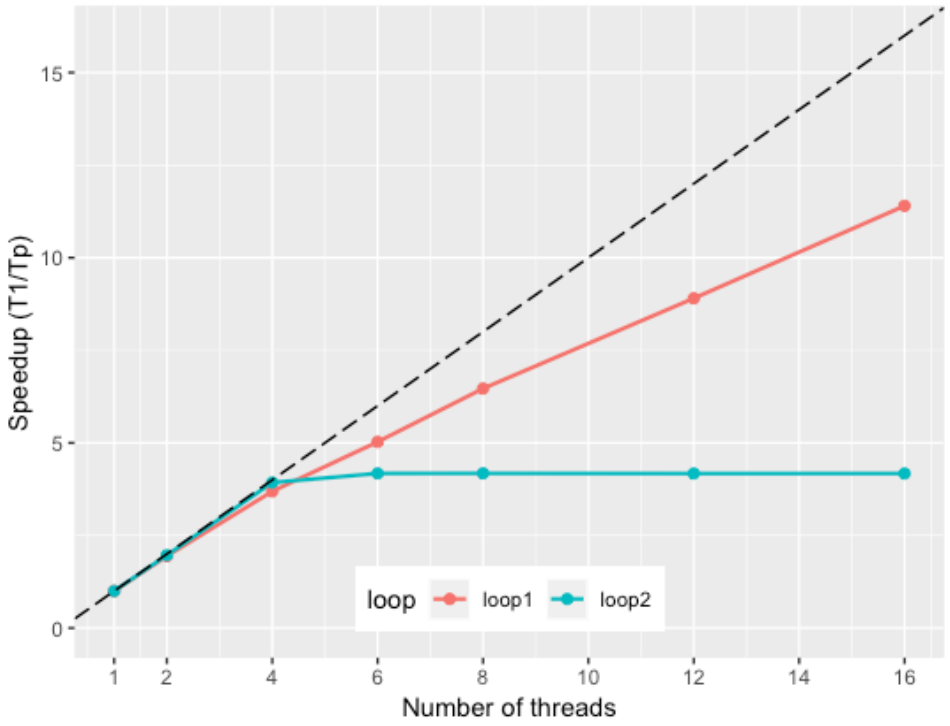

Figure 5: Speedup over different number of threads

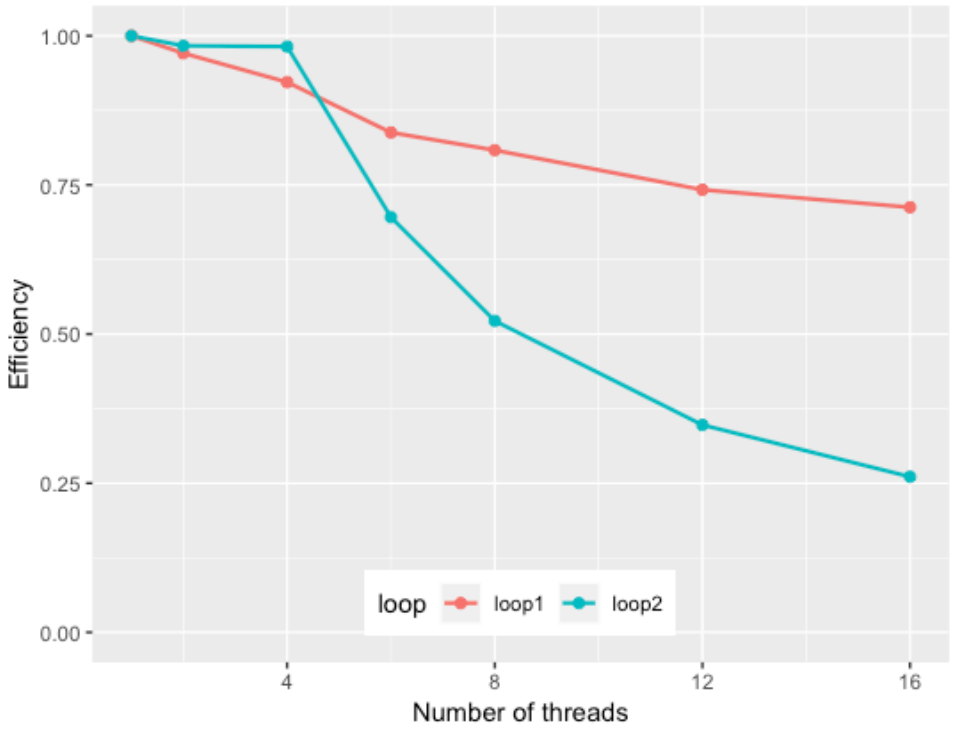

Figure 6: Efficiency over different number of threads

stays steady at around 4.17 regardless of the number of threads. The scheduling laws behind these two loops are discussed in section 3.

Figure 6 reveals that the efficiency becomes worse when increasing thread numbers. After the "elbow point" efficiency of loop2 drops sharply while the efficiency of loop1 gradually drops to 0.75 at the maximum thread number 16.

3.3 Discuss on Scheduling Options

The best scheduling option for loop1 is $(GUIDED, 8)$, for loop2 it is $(DYNAMIC, 16)$. These two schedules are chosen as a result of unbalanced workload in different iterations.

3.3.1 Workload of iterations in loop1

According to the code, the inner iterations on j create unbalanced workload. When the outmost iterations on i is separated by $omp\ for$, for larger i, the workload are less heavy. The inner iterations on j are plotted in Figure 7.

If using STASTIC option on four threads, then the first thread will always have the heaviest work while the fourth thread takes a light work over the iterations. GUIDED and DYNAMIC use the rule of first-come-first-served and result in a relatively balanced workload assignment on different threads. Among these options $(GUIDED, 8)$ has the best performance (0.408s) but all other GUIDED and DYNAMIC options are not much slower (vary from 0.407s to 0.450s), while $(DYNAMIC, 16)$ has a close performance (0.412s).

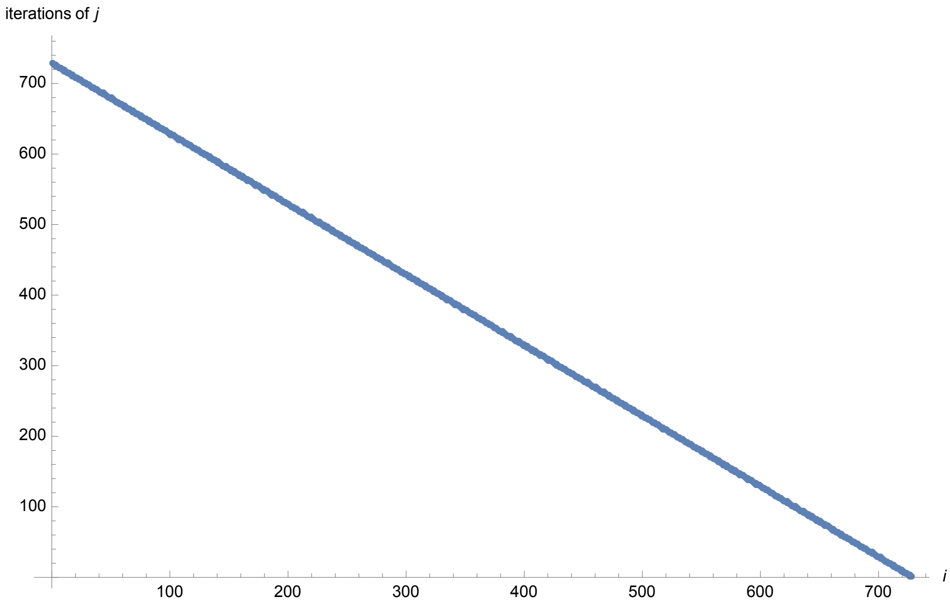

Figure 7: The times of iterations on j performed for each i in loop1

```
729 729 729 729 729 729 729 729 729 729 729 729 729 729 729 729 729 729 729 729 729 729 729 729 729 729 729 729 729 729
  1   1 729   1   1   1 729   1   1   1 729   1   1   1 729   1   1   1 729   1   1   1 729   1   1   1 729   1   1   1
  1   1   1 729   1   1   1   1   1   1 729   1   1   1   1   1   1 729   1   1   1   1   1   1 729   1   1   1   1   1
729   1   1   1   1   1   1   1   1   1 729   1   1   1   1   1   1   1   1   1 729   1   1   1   1   1   1   1   1   1
  1   1   1   1   1   1   1   1   1   1 729   1   1   1   1   1   1   1   1   1   1   1   1 729   1   1   1   1   1   1
  1   1   1   1   1   1   1   1   1   1 729   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1 729   1   1   1
  1   1   1   1   1   1   1   1   1   1 729   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1 729
  1   1   1   1   1   1   1   1   1   1 729   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
  1   1   1   1   1   1   1   1   1   1 729   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
  1   1   1   1   1   1   1   1   1   1 729   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
  1   1   1   1   1   1   1   1   1   1 729   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
  1   1   1   1   1   1   1   1   1   1 729   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
  1   1   1   1   1   1   1   1   1   1 729   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
  1   1   1   1   1   1   1   1   1   1 729   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
  1   1   1   1   1   1   1   1   1   1 729   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
  1   1   1   1   1   1   1   1   1   1 729   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
  1   1   1   1   1   1   1   1   1   1 729   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
  1   1   1   1   1   1   1   1   1   1 729   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
  1   1   1   1   1   1   1   1   1   1 729   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
  1   1   1   1   1   1   1   1   1   1 729   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
  1   1   1   1   1   1   1   1   1   1 729   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
  1   1   1   1   1   1   1   1   1   1 729   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
  1   1   1   1   1   1   1   1   1   1 729   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
  1   1   1   1   1   1   1   1   1   1 729   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
  1   1   1   1   1   1   1   1   1
```

Figure 8: The printed values of $jmax[i]$ $(0 \leq i \leq 728)$, which indicates the iterations performed in the inner loop for each i for loop2

3.3.2 Workload of iterations in loop2

loop2 have an extremely unbalanced workload for each i. Figure 8 gives the values of $jmax[i]$ which indicate the inner iteration times. When $jmax[i] = 1$ the inner iteration would do nothing, as the conditions in the for loops are not satisfied. This is referred as idle jobs. When $jmax[i] = 729$ the inner iteration would be heavily loaded with 729^2 times of calculations of $c[i]$ and referred as non-idle jobs. Thus, for most i $(0 \leq i \leq 728)$ there is no work to do but for the rest of them there is a heavy load.

Figure 9: The work distribution over different i $(0 \leq i \leq 728)$. The red points mean at these values of i, the inner loop is not idle (729^2 times of calculations to perform). Except these red points, all other points on the integer axis mean that there is no calculation to perform at those values of i.

Figure 9 shows the values of i that will have a workload (plotted in red points). The other points in the axis have no work to do. There are 67 red points in total, thus the optimal situation for the parallel task is to assign $67 \div 4 = 16.75$ non-idle jobs to each thread. It can also be noticed that the red points concentrate on the first half of the axis (Table 5 in Appendix) and on the second half of the axis they are sparse, thus using a chunksize of 16 would see that first two threads are assigned with most of the non-idle jobs (red points) but the last two threads with less non-idle jobs. As the schedule is DYNAMIC, the last two threads would finish their jobs quickly and receive the new chunks of non-idle jobs. Finally, all threads are expected to have done almost equal amount (16.75) of non-idle jobs, thus a chunksize that ensures this (i.e. load balance of the non-idle jobs) would be the optimal chunksize, in this case 16 is the best. The number of non-idle jobs in each chunk are calculated as Table 5 in Appendix, where using a chunksize=16 allows 45 chunks at most ($729/16 = 45.5625$)

3.3.3 Speedup

From Figure 5, it can be observed that increasing the number of threads from one to two will result in a speedup that is almost linear, where the line plots of the two loops are nearly coincides with the dotted line. For larger thread numbers, loop1 retains a speedup that are bit of slower than linear, with its efficiency declining to 71.25%. The performance degradation might be caused by the overhead in the scheduling and assignment.

The execution time of loop2 remains at the same level (around 2.07s) while increasing the thread number from 6 to 16, results in a “elbow point” in the line plot. Before the thread number reaches the “elbow point” the speedup of loop2 is almost linear but after that the speedup stops at around 4.10.

More threads do not improve the execution time of loop2. This can be explained by the unbalanced workload of loop2. Figure 9 and Table 5 tells that the non-idle iterations (red points) are concentrated on the first half of the axis, especially on the first 64 iterations that are contained in the first four chunk. While spreading chunks to threads, first four threads would always receive the heaviest load at first, the rest threads however only receive chunks that containing few jobs (one, two or zero) that are non-idle. DYNAMIC ensures that the rest threads keep doing these “easy” chunks over and over (first-come-first-served), whilst the first four threads are busy running

their heavy jobs. As a result, the execution time of the first four threads is the dominant part of the overall execution time. The later threads would always have a small amount of work to do, their execution times are small compared to that of the first four threads, thus from four on, increasing the thread number would not improve the execution time widely.

The extremely unbalanced workload of loop2 can also account for the plot of GUIDED in Figure 4, where the execution time remains almost unchanged whatever the chunksize is, as GUIDED tend to firstly assign large chunks of heavy load to all the four threads, and the execution time of the large chunks are dominant part of the overall time while later small chunks cost a small amount of time in later cycles, thus changing the chunksize would not affect the overall time widely.

4 Conclusions

On four threads, the best scheduling option for loop1 is $(GUIDED, 8)$ and for loop2 it is $(DYNAMIC, 16)$. The unbalanced workload separated in $omp\ for$ determines that using DYMAINC or GUIDED options will give the programme best performance.

As the programme goes on the workload in each chunk are getting smaller, where DYNAMIC and GUIDED have close performance. Overhead could be an reason that causes a slowed speedup efficiency.

After the separation in $omp\ for$ in loop2, certain iterations on j will do nothing and result in some idle work formed, thus the workload for each i is extremely unbalanced. This situation leads to the unbalanced assignment of chunks of works when more threads are added, where most jobs are done by the first few threads and more threads can hardly improve the running time. To achieve high performance, we may consider a chunksize that give each thread a nearly equal amount of non-idle jobs.

There are few limitations in the tests remained to be considered in future works. The time records of loop1 is relatively small (some of them are less than one second), which may not be suitable for performance analysis. To deal with this we may increase the value of $reps$ in the source code to obtain a more persuasive result. The test is preformed ten times and averaged to analyse, where ten times might not be enough to reveal a statistical pattern that implies the fastest scheduling option. Future woks may focus on the more precise answer to the fastest schedule and in-depth analysis of the inner mechanism of scheduling options over chunksize and thread numbers.

5 Appendix

Kind	Chunksize	Average time
static	NULL	0.704
auto	NULL	0.421
static	1	0.458
static	2	0.438
static	4	0.440
static	8	0.458
static	16	0.459
static	32	0.465
static	64	0.516
dynamic	1	0.448
dynamic	2	0.430
dynamic	4	0.425
dynamic	8	0.422
dynamic	16	0.412
dynamic	32	0.416
dynamic	64	0.442
guided	1	0.426
guided	2	0.414
guided	4	0.415
guided	8	0.408
guided	16	0.430
guided	32	0.424
guided	64	0.426

Table 1: Average execution time for loop1

Kind	Chunksize	Average time
static	NULL	6.229
auto	NULL	5.441
static	1	3.860
static	2	2.742
static	4	2.510
static	8	2.340
static	16	3.114
static	32	4.796
static	64	5.443
dynamic	1	2.702
dynamic	2	2.509
dynamic	4	2.374
dynamic	8	2.216
dynamic	16	2.210
dynamic	32	3.889
dynamic	64	4.925
guided	1	5.443
guided	2	5.443
guided	4	5.443
guided	8	5.443
guided	16	5.444
guided	32	5.443
guided	64	5.444

Table 2: Average execution time for loop2

No. Thread	Speedup	Efficiency
1	1.000	1.000
2	1.942	0.971
4	3.689	0.922
6	5.028	0.838
8	6.466	0.808
12	8.903	0.742
16	11.401	0.713

Table 3: Speedup and efficiency of loop1

No. Thread	Speedup	Efficiency
1	1.000	1.000
2	1.967	0.983
4	3.928	0.982
6	4.176	0.696
8	4.177	0.522
12	4.172	0.348
16	4.173	0.261

Table 4: Speedup and efficiency of loop2

Rank of Chunk	No. Non-idle Jobs	Rank of Chunk	No. Non-idle Jobs
1	15	24	1
2	13	25	0
3	3	26	0
4	3	27	1
5	2	28	0
6	2	29	1
7	2	30	0
8	0	31	1
9	2	32	0
10	0	33	1
11	0	34	0
12	1	35	1
13	0	36	0
14	2	37	1
15	0	38	0
16	1	39	1
17	0	40	0
18	1	41	0
19	0	42	1
20	1	43	0
21	0	44	1
22	1	45	0
23	0		

Table 5: Number of non-idle jobs in separated chunksize of loop2, using DYMAMIC scheduling option.