Parallelize Code Processing Next Generation Sequencing Data

Anna Mengjie Yu

Department of Integrative Biology, University of Texas at Austin, Austin TX 78712, USA

Abstract

The advent of next generation sequencing technology has enabled high-throughput genome sequencing in a much shorter time at a substantially lower cost. However, processing millions of lines of genome raw read data takes substantially large amount of computation time. In this project, I used OpenMP to parallelize the code processing next generation sequencing genome data, including data filtering (filterDuplicateSeq.c) and assembly result assessment (contigStatistic.c). The parallelized data filtering code achieved almost linear speedup, however, assembly result assessment code did not achieve expected speed up, probably due to large amount of communication time. Results and possible future improvement will be discussed.

Introduction

In the past decade there has been a huge increase in the number of completely sequenced genomes due to the race of multibillion-dollar genome-sequencing projects. The enormous amount of biological sequence data flooding into the sequence databases necessitates the development of more efficient tools for comparative genome sequence analysis. The information deduced by such analysis has various applications, such as structural and functional annotation of novel genes and proteins, genome evolution studies, constructing metabolic pathways *etc*. Such study also proves invaluable for pharmaceutical industries, such as biomarker identification and new drug discovery.

With the advent of next generation sequencing technology, more genomes can be generated at a substantially lower cost in a much shorter time. With the increasing amount of genomic data, it seems impractical to analyze them on uniprocessor machines. Hence there is a need for improving the performance in analyzing genome sequences on parallel cluster computers. Duplicated sequences in genome assembly will result repeat structure making it hard for assembly software to handle. In this study we explored the possible ways to parallelize code to delete duplicated sequences in FASTA file. We also explored possibly efficient ways to assess genome assembly metrics for genome assembly output data.

Method

All codes were compiled on Stampede login node using the command:

```
icc –openmp –o executable code_name
```

All executables were run on compute node using:

```
idev –p development –m 120
```

Three different types of affinity scheduling (none, scatter, compact) were tested to compare the speedup. The following bash command file was used to generate the output.

```
#!/bin/bash
corecount=16

#Affinity Scheduling
for affinity in none scatter compact;

do
      #Thread Count
       for ((t=1; t<=corecount; t++));
       do echo "${t} ${affinity}";
          KMP_AFFINITY=${affinity} OMP_NUM_THREADS=${t} ./executable input_file output_file
       wait;
       done >${affinity}.Outfile
done
```

The parallelization strategies were discussed in detail in the result part. The speedup and efficiency were calculated and visualized in Excel.

Result and Discussion

1. Parallelize data filtering code

The input file try10000.fasta contains 10000 lines in FASTA format, with alternating header lines (containing sequence length and sequence platform information) and DNA sequence lines.

```
login3.stampede(112)$ wc -l try10000.fasta
10000 try10000.fasta
login3.stampede(114)$ head -2 try10000.fasta
>HQ66C0U02GILHX length=54 xy=2555_0195 region=2 run=R_2012_07_11_16_20_50_
GGAGGTGAGTGTGAGCCCTCACCTCTGAGTGCACAAGGAACCCTGGTCACCGTC
```

The algorithm used is shown as below:

Read Input FASTA File

⇩

Store header in header[]
Store Sequence in dna[]

⇩

Search for exactly the same or subset sequences
And store the line number in delete[]

⇩

Print out header and dna whose
line number not in delete[]

The work-sharing for construct for loop is used when detecting the same or subset sequences .

```
//store subset or duplicated DNA sequences in delete[]
    for (k = 0; k< j; k++){
    #pragma omp parallel  for
        for (l = k+1; l < j; l++)
            //if two sequences are exactly the same
            {
            if (strcmp(dna[k], dna[l]) ==0 && k != l){
                delete[k]=-1;
                delete[l]=-1;
            }
            //if string K is subset of string L
            else {
                char K[900];
                char L[900];
                strcpy(K,dna[k]);
                strcpy(L,dna[l]);
                if ( (strstr(strtok(L,"\n"),strtok(K,"\n")) != NULL) && (k != l) && (delete[k] != -1) ){
                    delete[k] = -1;
                }
            }
        }
    }
```

The reduction operation is used to combine all the count of unique sequences, using the operation add and variable count.

```
//print out the output file and the deleting message
    int count = 0;
    #pragma omp parallel for reduction (+:count)
    for (k = 0; k<j; k++){
    //      fprintf(outFile,header[k]);
        //if not in delete
        if( delete[k] != -1 ){
            count++;
            fprintf(outFile,header[k]);
            fprintf(outFile,dna[k]);
        }
    }
```

Three different loop schedule types (static, dynamic, guided) were tested, under none, scatter, compact affinity, respectively, to search for the best performance.

1.1 Parallelize code in default scheduling setting

Table 1: Computation time used with different numbers of threads in default scheduling

threads	none	scatter	compact
1	14.6814	14.677	14.6754
2	7.4986	7.403	7.402
3	4.9569	4.9672	4.9515
4	3.733	3.7398	3.731
5	3.0285	3.0346	2.9843
6	2.5417	2.6396	2.5013
7	2.2175	2.2151	2.146
8	1.9741	2.1547	1.8962
9	1.7825	1.9872	1.7186
10	1.859	1.849	1.5644
11	1.7982	1.8264	1.7124
12	1.7195	1.6708	1.7552
13	1.6392	1.6416	1.6644
14	3.0306	1.5606	1.5906
15	2.9193	1.5264	1.515
16	2.7487	1.4667	1.4639

From Table 1 and Figure 1, we can see that with the increasing number of threads, the computation time goes down. The sharpest decrease in computation times appears at increasing number of threads from 1 to 2, the computation time decreases much slower with further increasing of number of threads, probably due to increasing communication cost between threads.

An abnormal trend of increase in computation time was observed in none affinity when thread number exceeds 14, which also contributed to decrease in speedup and efficiency (Figure 1). The highest efficiency was achieved by both scatter and compact affinity setting (Figure 1 and Supplementary Table 1). The shortest computation time observed was 1.4639 seconds using compact affinity setting (Table 1).

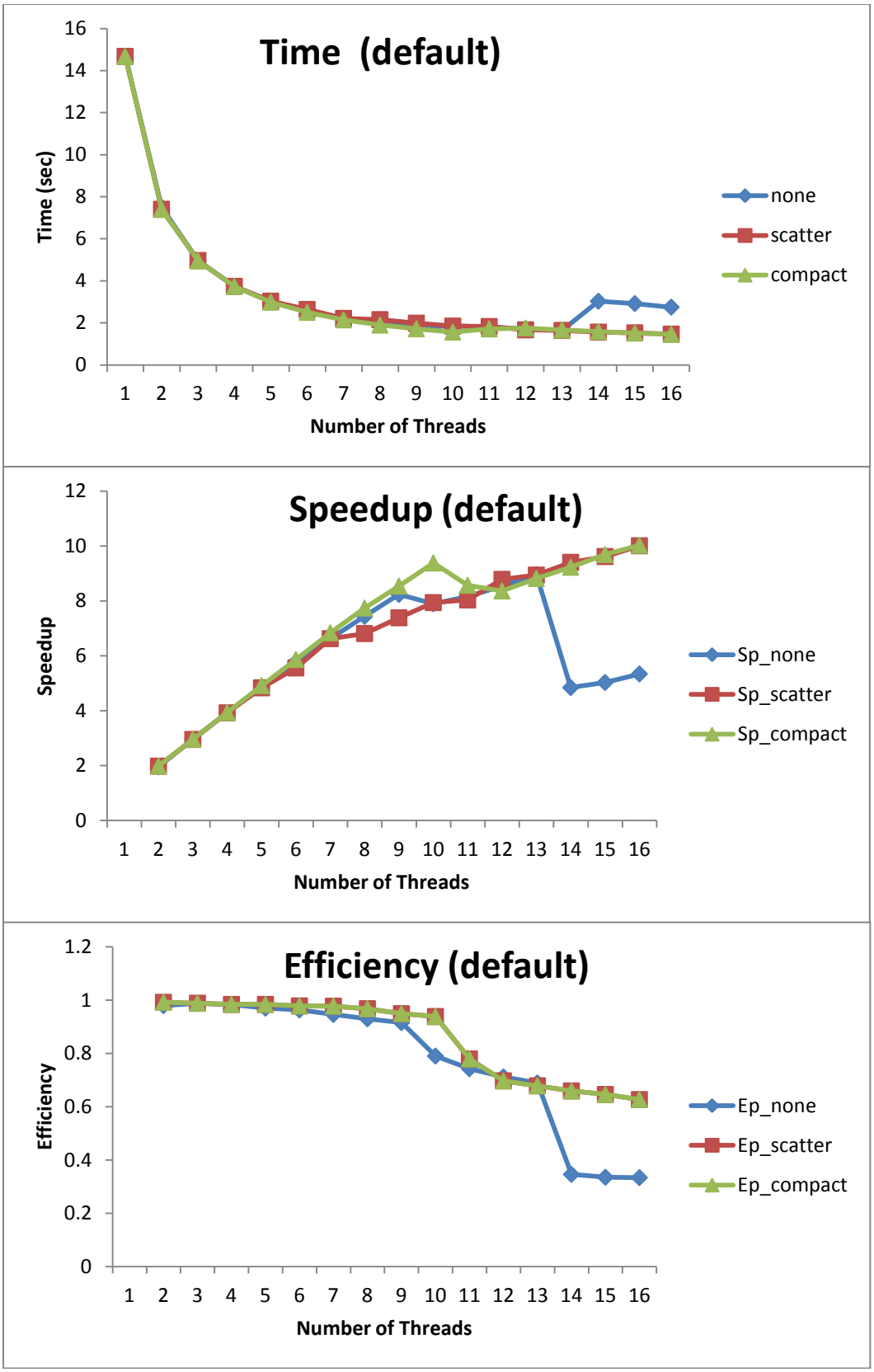

Figure 1: Time, Speedup and Efficiency with different numbers of threads in default schedule

1.2 Parallelize code using static scheduling

Code snippet:

```
for (k = 0; k< j; k++){
      #pragma omp parallel  for schedule(static)
            for (l = k+1; l < j; l++)

#pragma omp parallel for reduction (+:count) schedule(static)
      for (k = 0; k<j; k++){
                  //if not in delete
            if( delete[k] != -1 ){
                  count++;
                  fprintf(outFile,header[k]);
                  fprintf(outFile,dna[k]);
            }
      }
```

Table 2: Computation time used with different numbers of threads in static scheduling

threads	none	scatter	compact
1	14.6624	14.6893	14.8
2	7.3846	7.406	7.3901
3	4.955	4.9472	4.9395
4	3.729	3.7362	3.7143
5	2.9959	3.0029	2.9811
6	2.5029	2.5339	2.493
7	2.2065	2.2447	2.1468
8	1.9611	2.1706	1.8725
9	1.7837	2.0056	1.6903
10	1.8901	1.8532	1.5501
11	1.8151	1.7612	1.7157
12	1.7122	1.6697	1.7422
13	1.6252	1.6319	1.6541
14	2.9653	1.5695	1.5861
15	2.8673	1.5185	1.5146
16	2.8509	1.4742	1.465

From Table 2 and Figure 2, we observed a general trend of decreasing computation time with increasing thread number. The highest efficiency was observed at thread number 2 in both scatter and compact affinity setting (Figure 2 and Supplementary Table 1). We also observed the abnormal trend of computation time increase in none affinity setting with number of threads reaches 14. And the shortest computation time was achieved by using 16 threads in compact affinity setting. The default schedule setting depends on the compiler, the Intel compiler uses static as default schedule setting (Cyrus, oral communication). The computation time, speedup and efficiency from default setting looks similar to the results in static setting.

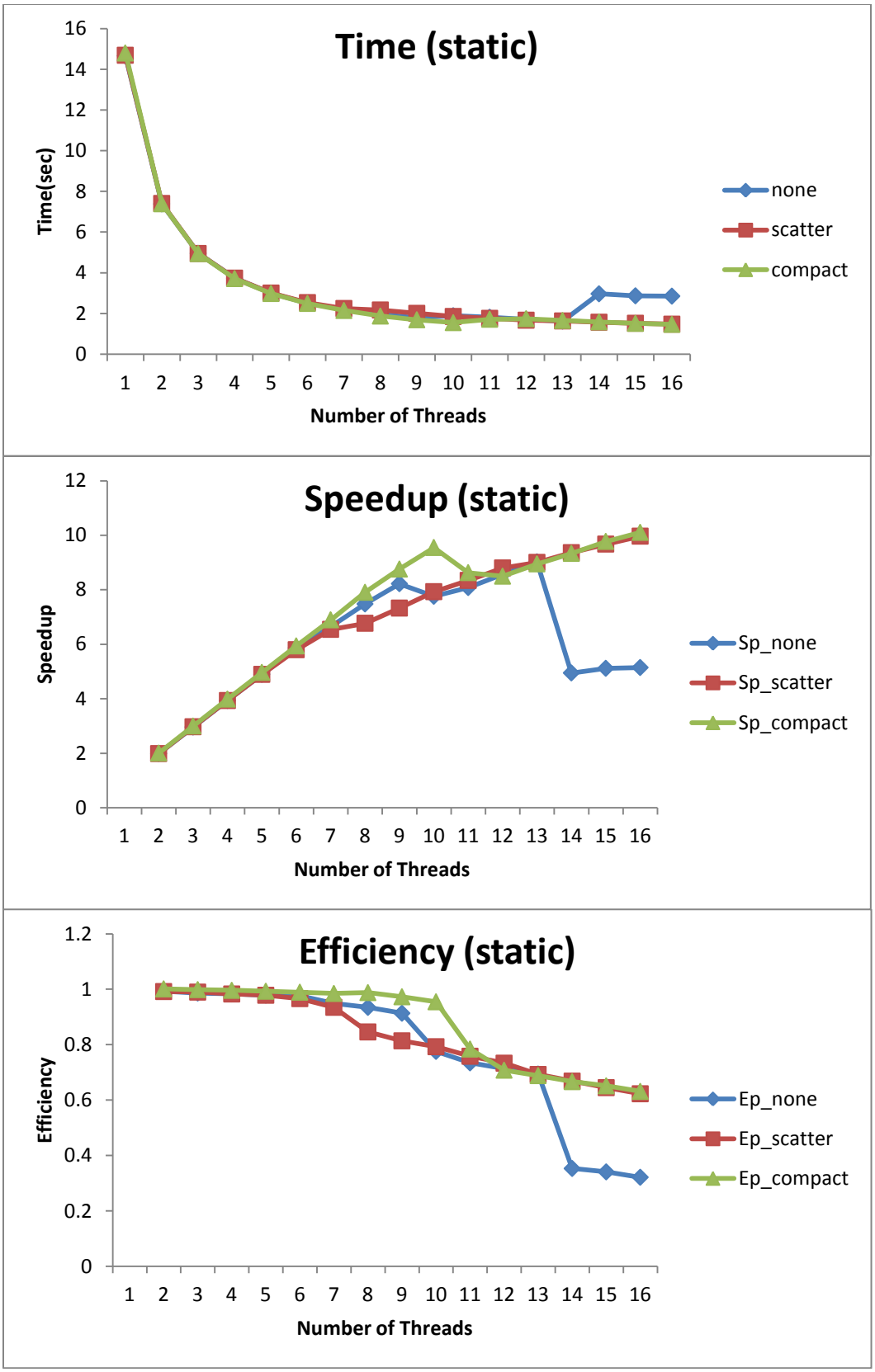

Figure 2: Time, Speedup and Efficiency with different numbers of threads in static schedule

1.3 Parallelize code using dynamic scheduling

Code snippet:

```
 for (k = 0; k< j; k++){
     #pragma omp parallel  for schedule(dynamic)
          for (l = k+1; l < j; l++)

#pragma omp parallel for reduction (+:count) schedule(dynamic)
     for (k = 0; k<j; k++){
               //if not in delete
          if( delete[k] != -1 ){
               count++;
               fprintf(outFile,header[k]);
               fprintf(outFile,dna[k]);
          }
     }
```

Table 3: Computation time used with different numbers of threads in dynamic scheduling

threads	none	scatter	compact
1	14.6963	14.7729	14.8067
2	8.2124	8.4266	7.771
3	5.4391	5.4382	5.1817
4	4.0902	4.1048	3.9098
5	3.2851	3.3003	3.1253
6	2.7409	2.7523	2.6152
7	2.3785	2.4043	2.243
8	2.0967	2.1486	1.968
9	1.8717	2.0034	1.837
10	1.8751	1.9318	1.6834
11	1.7704	1.854	1.6155
12	1.7178	1.7625	1.7493
13	1.702	1.6958	1.7013
14	1.9631	1.6421	1.6481
15	2.0049	1.6012	1.5806
16	2.0937	1.5545	1.5402

In dynamic schedule, the default chunk size is 1, and the tasks are dynamically scheduled among the threads. From Table 3 and Figure 3, the general trend of increasing thread number and decreasing computation time was observed. The abnormal increase in computation time in none affinity setting in high number of threads is less obvious than the previous default and static affinity setting. The compact affinity setting outcompetes none and scatter affinity setting achieving the highest efficiency (Figure 3 and Supplementary Table 1).

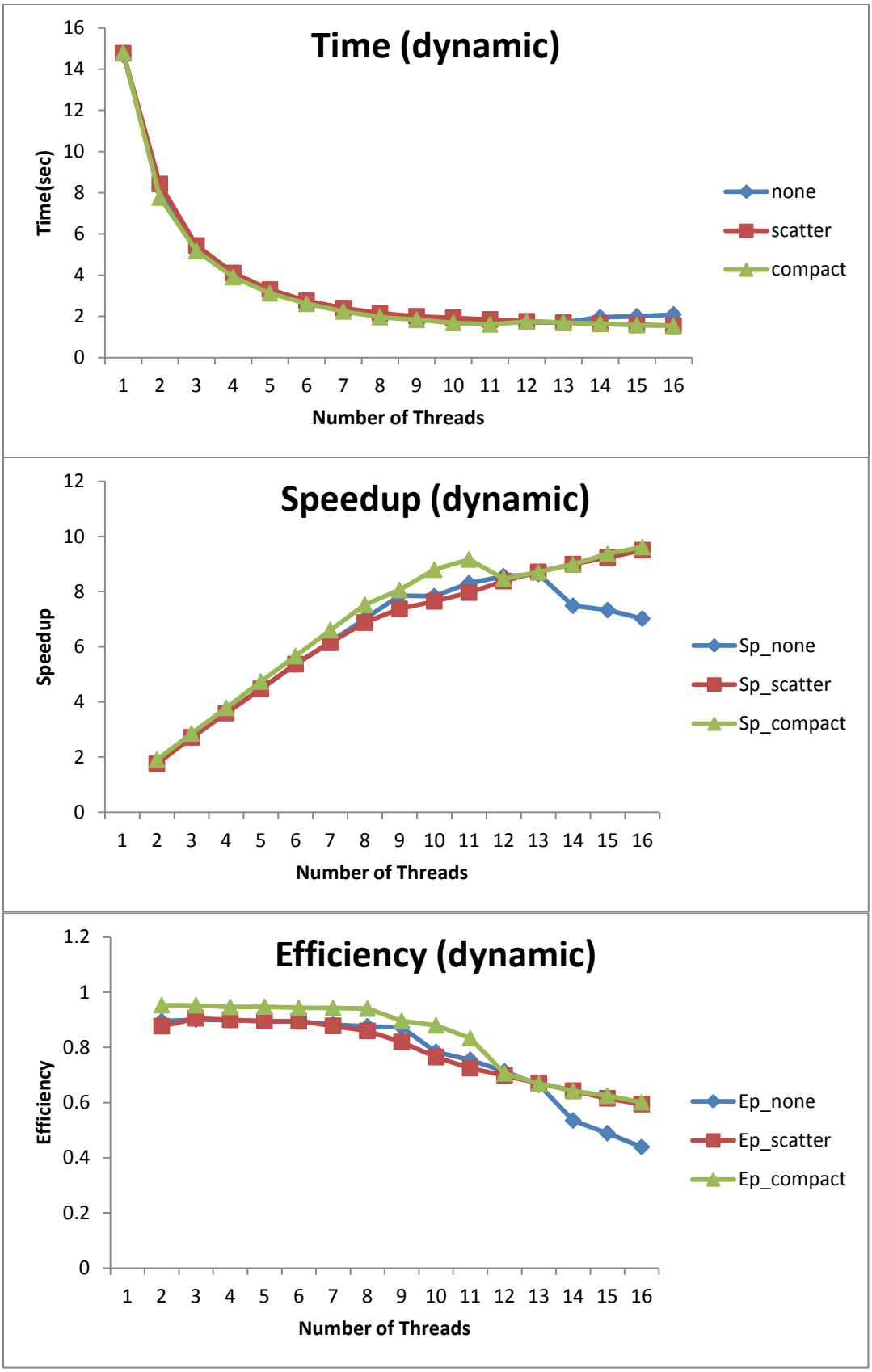

Figure 3: Time, Speedup and Efficiency with different numbers of threads in dynamic schedule

1.4 Parallelize code using guided scheduling

Code snippet:

```
for (k = 0; k< j; k++){
      #pragma omp parallel  for schedule(guided)
            for (l = k+1; l < j; l++)

#pragma omp parallel for reduction (+:count) schedule (guided)
      for (k = 0; k<j; k++){
            //if not in delete
            if( delete[k] != -1 ){
                  count++;
                  fprintf(outFile,header[k]);
                  fprintf(outFile,dna[k]);
            }
      }
```

Table 4: Computation time used with different numbers of threads in guided scheduling

threads	none	scatter	compact
1	14.6951	14.7296	14.7131
2	7.37	7.3929	7.4305
3	4.9274	4.9467	4.9197
4	3.7124	3.7419	3.6893
5	2.9876	3.0015	2.9762
6	2.4996	2.5395	2.4831
7	2.1965	2.1898	2.1352
8	1.9644	2.0893	1.8639
9	1.7724	1.9477	1.683
10	1.8425	1.8379	1.5471
11	1.7328	1.77	1.6537
12	1.6836	1.6512	1.6933
13	1.6138	1.585	1.5978
14	1.9939	1.5297	1.5379
15	2.1535	1.5026	1.491
16	1.8831	1.4438	1.4426

In guided scheduling, the tasks are dynamically assigned to threads in blocks as threads request them until no blocks remain to be assigned. The block size decreases as more tasks completed. In guided scheduling, we also observe the general trend of decreasing computation time with increasing thread number. None, scatter and compact affinity setting achieved highest efficiency at thread number 2, being 0.9969, 0.9962, and 0.9900 respectively (Supplementary Table 1). The shorted computation time is also observed in compact affinity setting with 16 threads (Table 4).

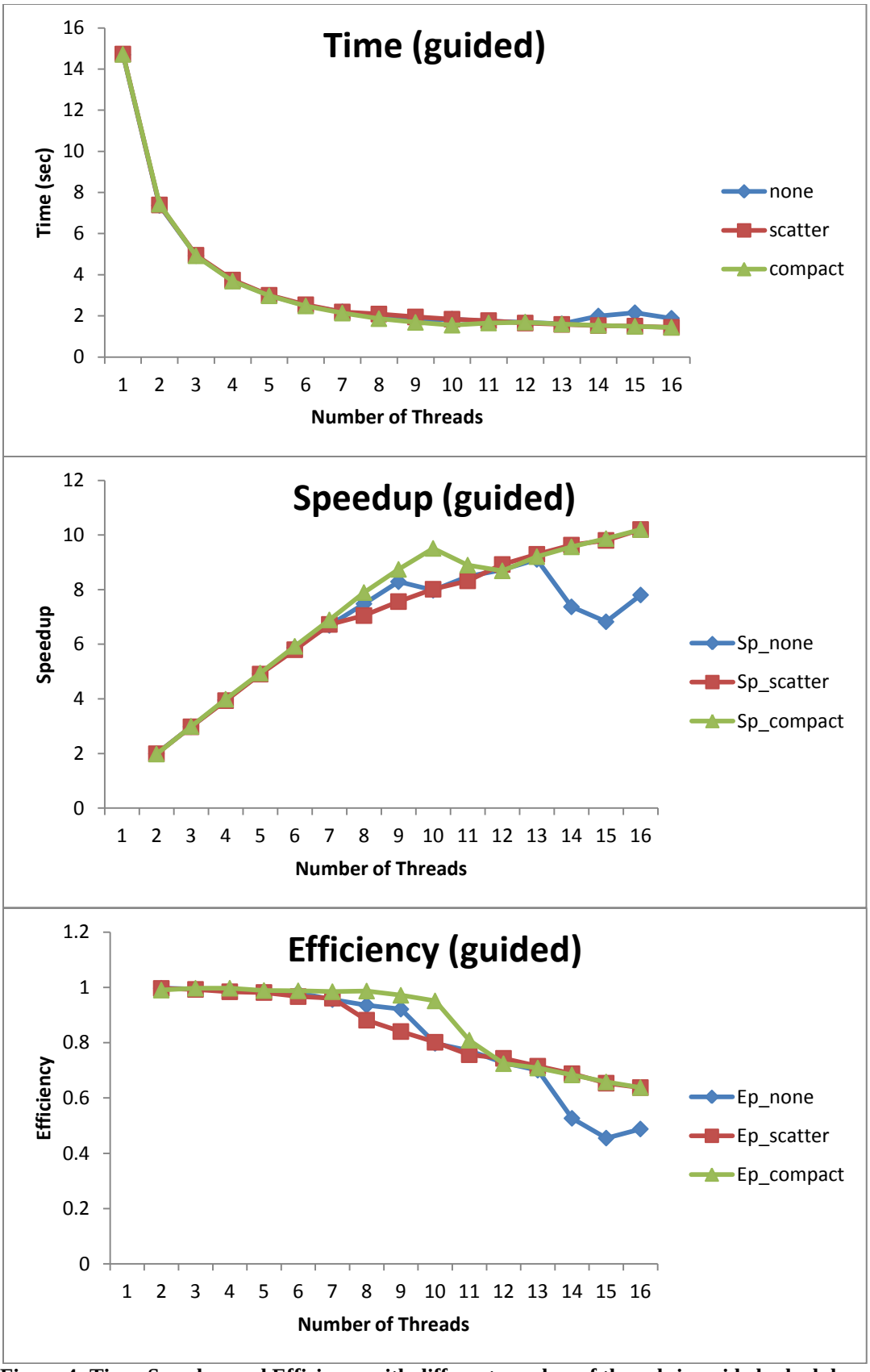

Figure 4: Time, Speedup and Efficiency with different number of threads in guided schedule

1.5 Effect of schedule type on performance

The Schedule directive admits three types of iteration distribution: static, dynamic and guided. The static is the strategy applied by the compiler, the loop iterations divide into pieces of certain sized chunk, and statically assign to threads. In dynamic distribution, the thread obtains as new set of iterations as it finished the previously assigned set. In the guided type, the size of each block is reduced in an exponentially decreasing way.

The slight difference in computation time was observed using different scheduling types. In this study, the shortest computation time was observed using guided scheduling. The abnormal behavior of increasing computation time with large number of threads was also the least obvious in guided scheduling setting (Supplementary Table 1). However, whether static, dynamic or guided is more efficient also depends on the input data, *e.g.* whether each dna sequence has similar or varying length, *etc*.

1.6 Effect of affinity types on performance

The KMP_AFFINITY environment variable assigns OpenMP threads to the processors based upon their physical location in the machine. The none affinity type is the default setting, which means not binding OpenMp threads to particular thread context. Scatter affinity distributes the threads as evenly as possible across the entire system. Compact affinity is the opposite to scatter affinity, which assigns the next thread as close as possible to the thread context where the previous thread was placed.

The shortest computation time observed was using compact affinity setting in all different for loop scheduling types (Table 1-4). It is possible that the communication overhead in compact affinity is smaller, since we are only using a small number of threads (1-16). When large number of threads are used, it is possible scatter affinity is faster than compact affinity.

2. Parallelize assembly result assessment code

The input file is the assembly result file from Velvet genome assembly. Velvet is a *de novo* genomic assembler developed by Daniel Zerbino and Ewan Birney, designed for short read sequencing technologies, such as 454 and Illumina. The velvet.fas input file contains 276247 lines, with alternating header line starting with a '>' sign, and DNA sequence line. The header line stores the information of the length of the contig (contiguous linear stretch of DNA consensus sequence) and the coverage of the contig.

```
login2.stampede(27)$ wc -l velvet.fas
276247 velvet.fas
login2.stampede(25)$ tail -2 velvet.fas
>NODE_59824_length_83_cov_6.771084
GTCATCTTGTGGTACTCTTTTTGTTGTTGAACAAACTTGTTTCGATGTTCATTAGAGAAGGCGTTAAAG
AAAATCAGGGTTCTTTCCTTTTCGCCAATTGTTAATTCAAAATCAGCCTTGAGGTCATCATACAACTTG
AAGAGAACCTCGAAGCGAGCCTT
```

N50 is an important metric in evaluating *de novo* assembly performance, it is the length of the contig in the collection where the contig of same length or longer contains at least half of the total length of all the contigs. In this contigStatictics.c script, we extract the contig length information from the header line, and store them in the contig[ ] array. The length of assembled contig comes in random order, we use merge sort to sort the the contig [ ] array from small to large based on the contig length. The output of the contigStatistics.c script is shown as below. The contig length is written out to velvet.out file.

```
c557-304.stampede(18)$ contigStatistic_Velvet velvet.fas velvet.out
The number of contig is 138122
The maximum contig length is 2327874
The N50 is 24199
The average contig length is 1396
```

Instead of using serial merge sort, a parallelized version of merge sort was implemented. Code snippet of serial version of merge sort:

```
void mergesort_serial(int a[],int low, int high){
	int mid;
	int size = high +1;
	if (size == 2){
		if (a[0] <= a[1])
			return;
		else{
			SWAP(a[0],a[1]);
			return;
		}
	}
	if (low <high) {
		mid = (low+high)/2;
		mergesort_serial(a,low, mid);
		mergesort_serial(a,mid+1,high);
		merge(a,low,mid,high);
	}
}
```

Code snippet of openMP version of merge sort:

```
void mergesort_parallel_omp(int a[],int low, int high ,int threads){
	int mid = (low+high)/2;
	if (threads == 1)
		mergesort_serial(a,low,high);
	else if (threads>1){
		#pragma omp parallel sections
		{
			#pragma omp section
			mergesort_parallel_omp(a,low,mid,threads/2);
			#pragma omp section
			mergesort_parallel_omp(a,mid+1,high,threads-threads/2);
		}
		merge(a,low,mid,high);
	}
}
```

Table 5: Computation time used with different number of threads for parallel mergesort

threads	none	scatter	compact
1	0.1183	0.1171	0.1171
2	0.1182	0.1168	0.1169
3	0.117	0.117	0.117
4	0.1186	0.118	0.1169
5	0.1169	0.117	0.1169
6	0.117	0.118	0.1182
7	0.1169	0.117	0.117
8	0.1168	0.1183	0.1169
9	0.117	0.1169	0.1184
10	0.1168	0.118	0.1174
11	0.1171	0.1172	0.1168
12	0.1168	0.1182	0.117
13	0.117	0.1176	0.117
14	0.118	0.1172	0.1169
15	0.1176	0.117	0.1182
16	0.118	0.117	0.1171

Merge sort is an efficient divide-and-conquer sorting algorithm, with the average complexity being O(n log n). Even using 1 thread, the computation time is relatively short around 0.11 sec. To my surprise, the parallel merge sort version did not show speed up. Several reasons might be possible: 1) The merge sort part is only part of the main program, the main part of the computation time is spent in reading and writing files; 2) The communication time in scheduling sections between threads is equal to or outweigh the time speedup by parallel merge sort. It is possible that if the input file is larger, we might be able to see speedup in the computation time. Parallel I/O might be more suitable for future application of parallelizing genome files with lots of read and write.

This project explores some possible ways to speed up code used in analyzing genomic sequences in parallel cluster machines using openMP. So far, parallel computing is not widely used by the bioinformatic community, partly due to the difficulty in guarantee exact same output as serial implementation and not losing any useful biological information, also some syntax like 'break' within a loop not supported by parallel construct, *etc*. In the age of personalized medicine, faster and cheaper sequencing technologies are still emerging, scientists will be overwhelmed by the ocean of genomic data. The challenges ahead is how to analyze those genomic data both effectively and efficiently, parallel computing offers a possible solution to extract useful information from the ocean of genomic sequences in a much more time-efficient way.

Acknowledgements

I would like to acknowledge the Texas Advanced Computing Center (TACC) at the University of Texas at Austin for providing access to stampede supercomputers. I would also like to acknowledge the course instructors Victor Eijkhout and Cyrus Proctor for teaching us those parallel computing skills that will be very useful in the era of exploding genomic data.