Multi-core Programming
Assignment 3

Ali Gholami

Department of Computer Engineering & Information Technology
Amirkabir University of Technology

https://aligholamee.github.io
aligholami7596@gmail.com

Abstract

One common example of parallel processing is the implementation of the merge sort within a parallel processing environment. In the fully parallel model, you repeatedly split the sublists down to the point where you have single-element lists. You then merge these in parallel back up the processing tree until you obtain the fully merged list at the top of the tree. In this report, we'll analyze the possibility of Merge-sort parallelization using OpenMP tasks and sections.

Keywords. *Merge-sort, Parallel Recursive Sort, Parallel Sort, Heterogeneous Programming, OpenMP, C Programming, C++ Programming, Parallelization, Multi-thread Programming.*

1 Merge-sort Parallelization with Tasks

1.1 Problem Specification

This assignment focuses on the parallel sorting problem using OpenMP. The goal is to improve the speed of array sorting by implementing the parallelization in recursive section of the sorting algorithm. We'll conduct the two features of OpenMP called *Tasks* and *Sections* respectively. The initial code implemented for the Merge-sort is given below.

```
void mergeSort(int *a, int n) {
        int m;
        if (n < 2)
        return;
        m = n / 2;
        mergeSort(a, m);
        mergeSort(a + m, n - m);
        merge(a, n, m);
}
```

1.2 Task Parallelization

Considering the *Tasks* as units of work, we'll generate them by a single thread. These generated tasks are put into a *Task Queue*. The other threads are waiting for the first task to be available. Each thread takes a task from the queue and starts working on it. This idea is clearly described in the figure 1.1 which is brought from the official slides from Intel.

1.3 Task Execution

Tasks are queued and executed whenever possible at the so-called task scheduling points. Under some conditions, the runtime could be allowed to move task between threads, even in the mid of their lifetime. Such tasks are called untied and an untied task might start executing in one thread, then at some scheduling point it might be migrated by the runtime to another thread `[by Hristo Iliev on Stackoverflow]`.

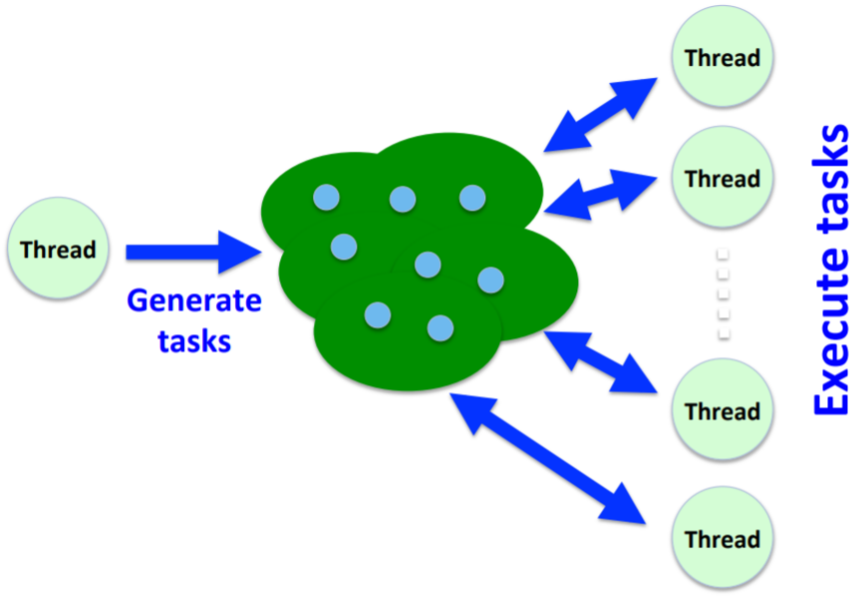

Figure 1.1: Illustration of Task Concept in OpenMP.

1.4 Task Parallelization in Recursive Loops

Tasks are useful in many scenarios as well as recursive loops. In a high-dimensional space, the Merge-sort algorithm generate large numbers of single numbers at the leaves of the computational tree. These leaves can be merged by different threads. Thus, Parallelization can be done in every depth of the tree. An important criteria that needs to be satisfied is that *depths of the tree are dependent on each other*. Thus, when some threads complete their job on some leaves, they have to wait until other threads are done with other leaves. This can be achieved using the *taskwait* directive.

1.5 Parallelized Recursive Loop

Here is the final parallelized code for th recursive loops we just talked about. Note that the *parallel region* is implemented **outside** of the recursive function.

```
void mergeSort(int *a, int n) {

        int m;
        if(n < 2)
        return;
```

```
                // else...
                m = n / 2;

                #pragma omp task
                mergeSort(a, m);

                #pragma omp task
                mergeSort(a + m, n - m);

                #pragma omp taskwait
                merge(a, n, m);
        }
```

1.6 Testing & Evaluation

Assuming each *integer* as 4 bytes, we'll be filling the table 1.1 using the average time computed after 6 times of running the program. According to this assumption, each dimension can be computed as below:

- **100 MB**: $d = \frac{10^8}{4} = 25 * 10^6$
- **200 MB**: $d = \frac{2*10^8}{4} = 50 * 10^6$
- **300 MB**: $d = \frac{3*10^8}{4} = 75 * 10^6$
- **500 MB**: $d = \frac{5*10^8}{4} = 125 * 10^6$

Total Array Size					
Num of Threads	100 MB	200 MB	300 MB	500 MB	Average Speedup
1	6.120653	12.599889	19.009687	31.613370	-
2	8.140178	16.292684	24.851261	41.500230	0.85
4	4.665155	9.426776	14.257916	23.927245	1.9
8	3.540851	7.224303	10.782175	19.272577	1.2

Table 1.1: Results of Recursive Loop Parallelization using Tasks.

2 Merge-sort Parallelization with Sections

Sections are very similar to the tasks in the concept point of view. Unfortunately, sections are not efficient compared to the tasks. The main reason is related to the *time frame* they execute the code.

2.0.1 The Problem with Sections Block

Here is a simple but great explanation of the *Sections* bottleneck. Assume we have n threads to run the code ($n > 2$) and there are only 2 sections defined by the developer. Only 2 threads start executing the code in each of the *section* block and the other $n - 2$ threads should wait at the end of the *sections* parallel block. Recall that the *sections* parallel block contains an *implicit* barrier(shown with *) at the end. Here is the illustration of this phenomenon thanks to the *Hristo Iliev* explanation on the *Stack-overflow*.

```
                                                    [    sections     ]
                                          Thread 0: -------< section 1 >---->*------
                                          Thread 1: -------< section 2      >*------
                                          Thread 2: ------------------------>*------
                                          ...                                *
                                          Thread N-1: ---------------------->*------
```

2.1 Parallelized Recursive Loop

Here is the final parallelized code for the recursive loops using *sections*.

```
void mergeSort(int *a, int n) {

        int m;
        if(n < 2)
        return;

        // else...
        m = n / 2;

        #pragma omp parallel sections
        {

                #pragma omp section
                mergeSort(a, m);

                #pragma omp section
                mergeSort(a + m, n - m);
        }
        merge(a, n, m);
}
```

2.2 Testing & Evaluation

Assuming each *integer* as 4 bytes, we'll be filling the table 1.1 using the average time computed after 6 times of running the program. According to this assumption, each dimension can be computed as below:

- **100 MB**: $d = \frac{10^8}{4} = 25 * 10^6$
- **200 MB**: $d = \frac{2*10^8}{4} = 50 * 10^6$
- **300 MB**: $d = \frac{3*10^8}{4} = 75 * 10^6$
- **500 MB**: $d = \frac{5*10^8}{4} = 125 * 10^6$

Total Array Size					
Num of Threads	100 MB	200 MB	300 MB	500 MB	Average Speedup
1	6.639830	13.757173	20.468429	34.843211	-
2	3.523107	7.219967	10.830306	18.208799	1.8
4	3.519742	7.189632	10.897979	18.089861	1
8	3.590844	7.235916	10.896604	18.313119	1

Table 2.1: Results of Recursive Loop Parallelization using Sections.

3 System Specifications

Please refer to this link to see the complete system specification. These information are extracted using *CPU-Z*.