Technical Report for Second Project in Parallel Computing

Yasheng Sun
117020910076
sunyasheng123@gmail.com

Abstract

This report concludes the second project in parallel computing. This project involves the basic usage of OpenMP function and application of OpenMP in real world. We achieve the approximation of π following the previous project but through integration under OpenMP framework. Furthermore, we explore the usage of OpenMP in some advanced data structure such as link list instead of naive array data structure. All my code is publicly available on my github https://github.com/sunyasheng/Parallel-Computing.

1. Introduction

Project involves the understanding and application of OpenMP. In this project, we first put forward a more elegant way to compute π through integration than the method in project 1. Then we focus on the application of OpenMP to a more complicated data structure such as link list instead of naive array form[1]. A simple but tedious method is first proposed and then we refine the code based on some advanced characteristic such as Task in OpenMP.

2. Estimate π through integration

2.1. Methodology

In previous project we estimate π by Mento Carto which is not very accurate. Here we estimate π based on integration.

$$\int_0^1 \frac{1}{1+x^2} dx = \frac{\pi}{4}$$

This integration is computed by summing up all the areas of rectangles below the function curve. More rectangles are used, more precise the estiamted integration is.

From Figure 1 we can find that usage of OpenMP is very elegant and concise. We only need to note where we want to parallel and what we want to reduce and everything will be finished by the complier. It is pretty convenient that less code is needed to do reduction in a neat fashion under the OpenMP framework.

```
#pragma omp parallel
{
    double id = omp_get_thread_num();
    double x;

#pragma omp for private(x) reduction(+:sum)
    for(i=0;i<num_steps;i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step*sum;
}
```

Figure 1. Clip of the program in computation of π under OpenMP framework.

2.2. Result

Here we present the computation result under different number of threads when we divide the integration interval to 1000000000 segments in the Table 1. The estimated value is pretty close to the ground truth value. This table shows us the power of parallelization in efficiency improvement. By using two threads, we almost obtain double productivity compared with the traditional serial programing pattern. In the meantime, the complier have done the reduction automatically for us, which simplifies the programing significantly.

3. OpenMP Parallization in Link List

So far, the application of OpenMP we illustrated is limited in simple data structure such as array or matrix. More complicated data structure is required in real world application to describe and store data.

3.1. Problem Statement

We formulate this problem as shown in Figure 2. Many tasks is stored by a link list, which brings difficulty to direct parallelization under OpenMP framework.

number of threads	estimated value	consumed time
1	3.14159	5.59387s
2	3.14159	2.69163s

Table 1. Estimation Performance under different number of involved threads.

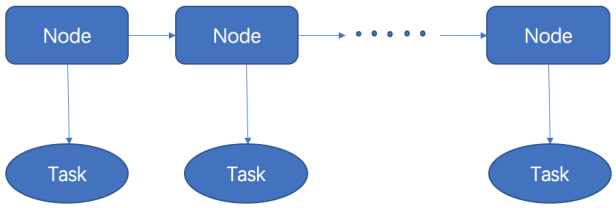

Figure 2. The toy problem is formulated to a linked list of tasks.

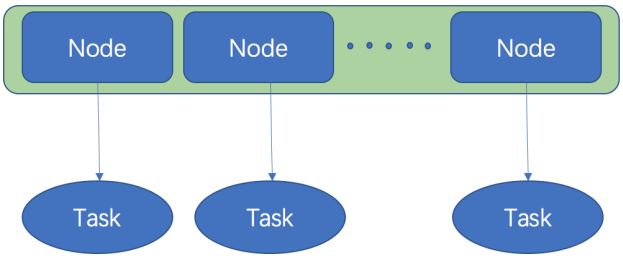

Figure 3. Implementation of naive conversion of the link list task.

3.2. Naive Reduction

A intuitive solution is to represent the orignal link list structure with array we are familiar with as shown in Figure 3. This strategy requires us to visit all the elements in link list and reorgnize those elements to the array form. Once we obtain the array form, we can solve this problem with the approach above.

3.3. Result of Naive Implementation

Table 2 shows the result under the naive implementation of reduction and pure serial pattern. Obviously, this manual parallelization perform better than the pure serial operation. But this strategy is not scalable to other scenario since it requires specified nasty manual array structure building operation in a certain problem. In next section, we will introduce a new characteristic in OpenMP to tackle this problem.

3.4. Reduction by Task

When we use the task command, we can solve this problem in an elegant fashion as shown in Figure 4. Tasks are independent units of work, which are composed of code to execuate, data environment and internel control variable. If we assign a segment of code as Task, the system runtime will automatically create multiple tasks one for each thread.

Pattern	consumed time
Serial Pattern	10.840s
Navie Reduction	7.009s
Task Reduction	6.840s

Table 2. Comparsion between different execuation patterns.

```
#pragma omp parallel
{
#pragma omp single
{
    printf(" %d threads \n", omp_get_num_threads());
    p = head;
    while(p){
#pragma omp task firstprivate(p)
{
        processwork(p);
}
        p = p->next;
    }
}
}
```

Figure 4. Clip of code to parallel the link list task with the task techinque.

4. Conclusion

In this project, we explore the basic usage of OpenMP function and application of OpenMP in more complicated scenario. We achieve the approximation of π following the previous project but through integration under OpenMP framework in a neat fashion. Furthermore, we explore the usage of OpenMP in some advanced data structure such as link list instead of naive array data structure. Experiment shows that we can achieve parallelization over more complicated data structure with task techinque under OpenMP framework.

References

[1] https://www.openmp.org/wp-content/uploads/Intro_To_OpenMP_Mattson.pdf