Report of Assignment 3 — OpenMP Programming

Multiprocessor Systems, DV2544

Student: Ye Cai *Personal number:9503101967* *Email address: yeca16@student.bth.se*

1 Introduction

OpenMP is a portable standard for the programming of shared memory systems. The OpenMP API (application program interface) provides a collection of compiler directives, library routines, and environmental variables [1]

2 Quicksort

2.1 Implementation

This OpenMP version implement of quicksort is adopted Kil and others' algorithm from their article [2] seen as figure 1 which is impletated as figure 2.

```
void main(){
#pragma omp parallel
#pragma omp single
        void quick_sort(A, left, right)
}
void quick_sort(A, left, right){
        determination_pivot();  //determine 3-median pivot
        swap();                 //exchange elements
if(right-left>0)
    #pragma omp task
        seq_quick_sort(A, first, pivot-1)
    #pragma omp task
        seq_quick_sort(A, pivot+1, end)
}
void seq_quick_sort(A, left, right){
        determination_pivot();  //determine 3-median pivot
        swap();                 //exchange elements
        seq_quick_sort(A, first, pivot-1)
        seq_quick_sort(A, pivot+1, end)
}
```

Fig. 2 Quick sort algorithm using task decomposition

Figure.1 Quick sort algorithm from[2].

```
static void quick_sort(int *v, unsigned low, uns
{
    {
    #pragma omp task
    if (low < pivot_index)
        quick_sort(v, low, pivot_index - 1);

    #pragma omp task
    if (pivot_index < high)
        quick_sort(v, pivot_index + 1, high);
}

int main(int argc, char **argv)
{

    init_array();
    Read_Options(argc, argv);   /* Read argument
    if(PRINT==1)
        print_array();
    double start = omp_get_wtime();

    #pragma omp parallel
     {
      #pragma omp single
        quick_sort(v, 0, N - 1);
     }
```

Figure.2 Implement of algorithm within figure 1.

It is used task decomposition which is OpenMP task directives (#pragram omp task) to do the calculation of sub-arrays. The data distribution is shown as figure 3 with 4 threads. The root of recursion only use one threads and second time it will use two if subarrays is more than threads. Else subarrays will handled by thread which is idle dynamically.

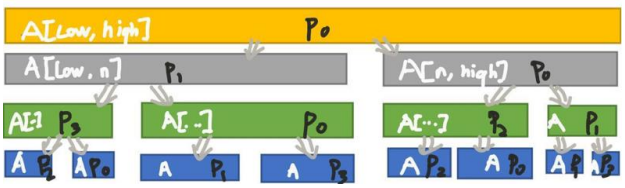

Figure.3 data distribution of 4 threads.

2.2 Measurements

Threads	Seq	2	4	8
Runtime	68.63s	39.01s	28.90s	40.7s
Speedup	-	1.76	2.37	1.56

Table.1 runtime and speedup of different threads

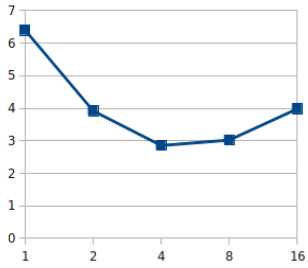

Figure.4 runtime of different threads.

Here speedup= Sp(n) =T*(n) /Tp(n) [1]. From table 1 we can see with the decreased number of threads the performance in runtime doesn't get a linear optimization. It may because a larger part of execution is not executed currently such as first iteration and second is done by one thread and two threads.

3 Gauss Elimination

3.1 Implementation

Due to the data dependency, we can't parallel the most outer loop, so the parallelism has happened at the elimination steps. It is used the directive "#pragma omp for schedule(static)" to parallelize the inner nested two for loops. Because its've chosen the schedule way as static, the data structure would be row-wise and block-wise statically. The matrix divided in chunks and assigned to the 8 threads just like figure 5. Then 8 threads would work currently. The algorithm referred S.F. McGinn's pseudocode in *part 2. OpenMP Parallel Version* of his article [3] which is implemented as shown in figure 6.

Chunk1	*a [0-7][0-63]*	**P0**
Chunk2	*a [8-15] [0-63]*	**P1**
Chunk3	*a [16-23] [0-63]*	**P2**
Chunk4	*a [24-31] [0-63]*	**P3**
Chunk5	*a [32-39] [0-63]*	**P4**
Chunk6	*a [40-47] [0-63]*	**P5**
Chunk7	*a [48-55] [0-63]*	**P6**
Chunnk8	*a [55-63] [0-63]*	**P7**

*Figure.5 data struct example with 8 threads and 64*64 matrix a.*

```
for (k = 0; k < N; k++) {
 #pragma omp parallel for num_threads(THREADS) schedule(static, chunksize)
 for (j = k+1; j < N; j++)
     A[k][j] = A[k][j] / A[k][k];

 y[k] = b[k] / A[k][k];
 A[k][k] = 1.0;

 #pragma omp parallel for num_threads(THREADS) schedule(static, chunksize) collapse(2)
 for (i = k+1; i < N; i++)
     for (j = k+1; j < N; j++)
         A[i][j] = A[i][j] - A[i][k]*A[k][j];              /* Elimination step */

 #pragma omp parallel for num_threads(THREADS) schedule(static, chunksize)
 for (i = k+1; i < N; i++) {
     b[i] = b[i] - A[i][k]*y[k];
     A[i][k] = 0.0;
 }
}//end of k loop
```

Figure.6 core code of implementation

3.2 Measurements

The runtime result and speedup of OpenMP version gaussian elimination implementation which executed with 8 threads shown in table 2.

Gaussian_seq	Gaussian_omp	Speedup[2]
68.754s	40.234s	1.709

Table.2 the speedup of OpenMP implement of gaussian elimination.

Note: speedup= Sp(n) =T*(n) /Tp(n)= 68.754s/40.234s=1.709[1].

Reference

1. Rauber, Thomas, and Gudula Rünger. *Parallel programming: For multicore and cluster systems*. Springer Science & Business Media, 2013.

2. Kim, Kil Jae, Seong Jin Cho, and Jae-Wook Jeon. "Parallel quick sort algorithms analysis using OpenMP 3.0 in embedded system." *Control, Automation and Systems (ICCAS), 2011 11th International Conference on*. IEEE, 2011.

3. McGinn, S. F., and Ruth E. Shaw. "Parallel Gaussian elimination using OpenMP and MPI." *High Performance Computing Systems and Applications, 2002. Proceedings. 16th Annual International Symposium on*. IEEE, 2002.