POLITECNICO
MILANO 1863

A.Y. 2024-2025 Parallel Computing Challenge1: Report

WangYanlong, 11005879, yanlong.wang@mail.polimi.it

Submitted Files: *mergesort.cpp, Makefile, Script.bat(for windows), Script.sh(for linux); GoogleColabFile(In case can't run locally, eventhough there're just two threads can be used)*

October 23, 2024

1 Environment Setup And Run CPP File

There are two files called Makefile and mergesort.cpp. We compile the cpp file by *make* and then run the *exe* with inputing size. Like below: *(If using the similar Shell Script in linux like WSL, there may be memory limitation for large data, which will result in killing the running progress before finishing.)*

- So for window we can use mingw32-make by *Script.bat* file and run directly without limitation.

```
@echo off
if not exist ..\bin ( mkdir ..\bin )   #make directory
make                        #compile the cpp file to exe
..\bin\mergesort-co size   #run the exe with size like (1000000000 we use)
```

2 Parallel The Merge Sort Algorithm With Depth And Cutoffs

2.1 Parallel The Recursive Procedure With Just Depth

We can use OpenMP to accelerate the recursive process, because they are independent halves. So here we can create shared tasks separately for each half and add a Synchronize check before merging.

```
void MsSerial(int *array, int *tmp, const size_t size) {
    int maxDepth = 4; // Add a maxDepth to control the parallel scale here
    #pragma omp parallel //parallel region
    {
        #pragma omp single //single thread
        MsSequential(array, tmp, true, 0, size, 0, maxDepth);
    }
}
void MsSequential(int *array, int *tmp, bool inplace, long begin, long end, int depth){
    if (begin < (end - 1)) {
        const long half = (begin + end) / 2;
        if (depth > 0) { //shared tasks
            #pragma omp task shared(array, tmp)
            MsSequential(array, tmp, !inplace, begin, half, depth - 1);
            #pragma omp task shared(array, tmp)
            MsSequential(array, tmp, !inplace, half, end, depth - 1);
            #pragma omp taskwait //sychronization
        } else { ... }
}
```

- Firstly, at the function `MsSerial()`, we create pool of threads and start with one of them.
- At the function `MsSequential()`,decide if parallel the recursive process according the depth level.
- Finally, we use *#pragma omp taskwait* to ensure sychronization.

Compare paralleled algorithm with original sequential one without using -O3 command for both:

$$\text{SpeedUp1} = \frac{121.46}{26.55} \approx 4.58 \quad \text{(with about } 10^9 \text{ data size; without using -O3 when running)}$$

```
PS D:\StudyMaterial\Autumn Semester\Parallel Computing\Project\mergesort> g++ -fopenmp mergesortS.cpp -o mergeS
PS D:\StudyMaterial\Autumn Semester\Parallel Computing\Project\mergesort> ./mergeS 1000000000
Initialization...
Sorting 1000000000 elements of type int (3814.000000 MiB)...
done, took 121.463000 sec. Verification... successful.
```

Figure 1: Sequential Merge Sort

```
PS D:\StudyMaterial\Autumn Semester\Parallel Computing\Project\mergesort> g++ -fopenmp MergeDepth.cpp -o mergedepth
PS D:\StudyMaterial\Autumn Semester\Parallel Computing\Project\mergesort> ./mergedepth 1000000000
Initialization...
Sorting 1000000000 elements of type int (3814.000000 MiB)...
done, took 26.554000 sec. Verification... successful.
```

Figure 2: Paralleled Merge Sort With Just Depth

2.2 Further Optimization By Adding Cutoff For Switching Sort Types

At the function `void MsSequential()`, we can also add a cutoff for checking the subarray's size. If the subarray is small enough, we don't want to dig deeper anymore, instead just using the easier sequential sort algorithm like quick sort.

```
if ((end - begin) <= cutOffSort) { // If below cut-off, perform a sequential sort
    if (inplace) {
        std::sort(array + begin, array + end); //default sort
    } else {
        std::copy(array + begin, array + end, tmp + begin);
        std::sort(tmp + begin, tmp + end);  //noinplace
    }   return;
}
```

- Because the sequential algorithm like quick sort works also very well for small size data.
- If we still do recursive procedure or even with parallel tasks, there will be so many small pieces to deal with, which will decrease the performance.
- Here we just use the default sort() algorithm from std library when subarray's size is less that cutoff.

2.3 Further Optimization By Adding Cutoff For Paralleling Merging 2 Halves

Based on the principle of merge sort algorithm, the two halves waiting for merging are already in order which means natural parallelization. So we can create two tasks to do parallel work from different direction to middle for further speeding up. (*the details of parallel function are in cpp file, too long to put here*)

```
bool mergeByParallel = (end - begin) >= cutOffMerge; // > cutoff? parallel merge
if (mergeByParallel)  {
    MsMergeParallel(array, tmp, begin, half, half, end, begin);   }  //parallel version
else {
    MsMergeSequential(array, tmp, begin, half, half, end, begin);    } //sequential one
```

- We only do this parallel version for large array (otherwise, there will be too many tasks to manage which in turn will drop down the performance). This means we need another cutoff to switch between the sequential and parallel merge algorithm.

2.4 Speedup Compared With Just Using Depth Without Cutoff

Here is the further speedup by adding these two cutoffs, which are respectively 16 and 2048 We can find we have dropped the time again to **19.15**: Speedup2 $= \frac{26.55}{19.15} \approx 1.39$.

- **We just use -fopenmp without using -O3 for executing command at terminal.**

```
PS D:\StudyMaterial\Autumn Semester\Parallel Computing\Project\mergesort> ./merge 1000000000
Initialization...
Sorting 1000000000 elements of type int (3814.000000 MiB)...
done, took 19.153000 sec. Verification... successful.
With parameter depth: 4, cutOffSort: 16, cutOffMerge: 2048
```

Figure 3: Parallel With Both Depth And Cutoffs

3 Configuration Test For Different Parameters Value

We don't use g++ -O3 -fopenmp at previous steps and 4.1 as well when adjusting depth value to see the influence. Because this flag will helpful to automatically optimize loops(recursive procedure) and memory access, better use of CPU resources, minimize threading and function call overhead, etc.
So if using this aggressive optimizations strategy, we won't find the behind principle of the balance between the parallel scale and the tasks management overhead very well by ourselves. Actually we can do a test to see have it can improve the performance as below:

```
PS D:\StudyMaterial\Autumn Semester\Parallel Computing\Project\mergesort> g++ -O3 -fopenmp MergeDepth.cpp -o mergedepth
PS D:\StudyMaterial\Autumn Semester\Parallel Computing\Project\mergesort> ./mergedepth 1000000000
Initialization...
Sorting 1000000000 elements of type int (3814.000000 MiB)...
done, took 13.443000 sec. Verification...
```

Figure 4: Just With Depth But By Using O3 At Terminal When Executing

Compared with previous 26.55 without using O3, we almost achieve a double speedup by **13.44**. This value is even better than our previous 19.15 which has both kinds of cutoffs added.
And we should also avoid influences from other factors that using threads as well, such as opening browsers, videos, etc. (to keep the same condition for testing)

3.1 Change Depth (Still not using -O3 strategy at terminal)

Through the function `int omp_get_num_threads (void)` we can get the number of current pc threads which is ***16*** for mine. Therefore, the optimal deepth should be $\log_2(threads) = 4$. With this depth level we can evenly distribute our 16 tasks to 16 threads.

- If the depth value is too large, there will be so many small tasks, increasing the total overhead. And the tasks may also not be uniformly distributed across the threads.
- The below table shows how the executing time changes by adjusting this value. (data size is $10^9 \approx 3.7GB$ and without adding any cutoff here!)

	depth=1	depth=2	depth=4	depth=8	depth=16	depth=24	depth=28
Time	64.19	37.97	24.71	26.00	20.42	26.54	**292.75**

If depth is small than 4 we haven't fully used the threads and parallel scale is very low. But if the depth is too large like 28 which is near the bottom recursive level. There are a huge amount of small tasks about 2^{28}, which leads to a very bad performance (even worse than just by sequential sorting).

3.2 Change Cutoff For Sort (Using O3)

From my testing experience, this cutoff value that managing the threshold of switching the sort algorithm will have influence on performance as well.

- If cutoff is small, the recursive level will be too deep and there are many small subarray to manage.
- If this curoff is too large, the sequential algorithm will have longer time to sort each bigger subarray.
- Testing results with datasize $= 10^9$ depth $= 4$ cutOffMerge $= 2048$ unchanged.

	cutoff=2	cutoff=16	cutoff=32	cutoff=256	cutoff=1024	cutoff=2048	cutoff=4096
Time	**26.67**	11.76	13.15	16.20	14.51	14.13	17.03

From the testing result, we can find that the optimal value is around cutoff=16. Either the value is too small like 2 or too large like 4096, the total execution time will become longer.
And when cutOffSort is equal to cutOffMerge, the performance is slightly better than other near values.

3.3 Change Cutoff For Merge (Using O3)

With this cutoff we can decide when to use the parallel version merging. With the same reason like before, this value shouldn't be too small or too large as well.

- Usually it's bigger that the cutoff for sorting. Because the complexity of sequential sorting like quick sort is about $O(n\log(n))$ and for merging is just $O(n)$.
- Testing results with datasize $= 10^9$ depth $= 4$ cutOffSort $= 16$ unchanged.

	cutoff=128	cutoff=512	cutoff=1024	cutoff=2048	cutoff=4096	cutoff=8192
Time	**20.51**	16.88	14.97	14.25	14.43	18.55

We can find the optimal cutoff shouldn't be too small, otherwise too many small tasks will effect performance. And it actually just improves our performance slightly compared with previous two.

3.4 Change The Setup Of Threads (Using O3)

We can also strict the thread number that can be used for parallel in advance to see how it can influence the total performance with data size $= 10^9$ and same parameters as before (2 16 2028).

	threads=2	threads=4	threads=6	threads=8	threads=12	threads=16
Time	**48.59**	35.63	25.28	14.37	14.08	10.00

If we have just 2-4 threads, it will achieve a bad performance. But when we arrive 8 threads, the performance is already good enough that we actually can't improve it too significantly compared with before.