Parallelizing Quick Sort Algorithm

Sheng Minkai (2022321799)

DaeHyeon Kang (2022321795)

Minhyuk An (2023321342)

연세대학교 YONSEI UNIVERSITY | SOFT COMPUTING LABORATORY

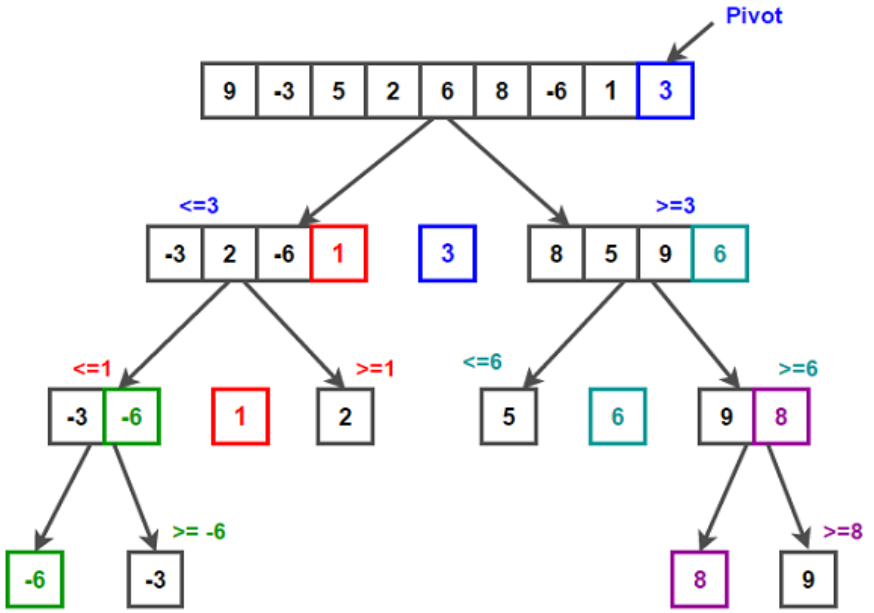
Contents

Problem Identification

- Implementation Hypothesis
  - How well will quick sort algorithm do with extremely large amounts of datasets

- Sensitive to choice of pivot
  - More relevant the larger the dataset is

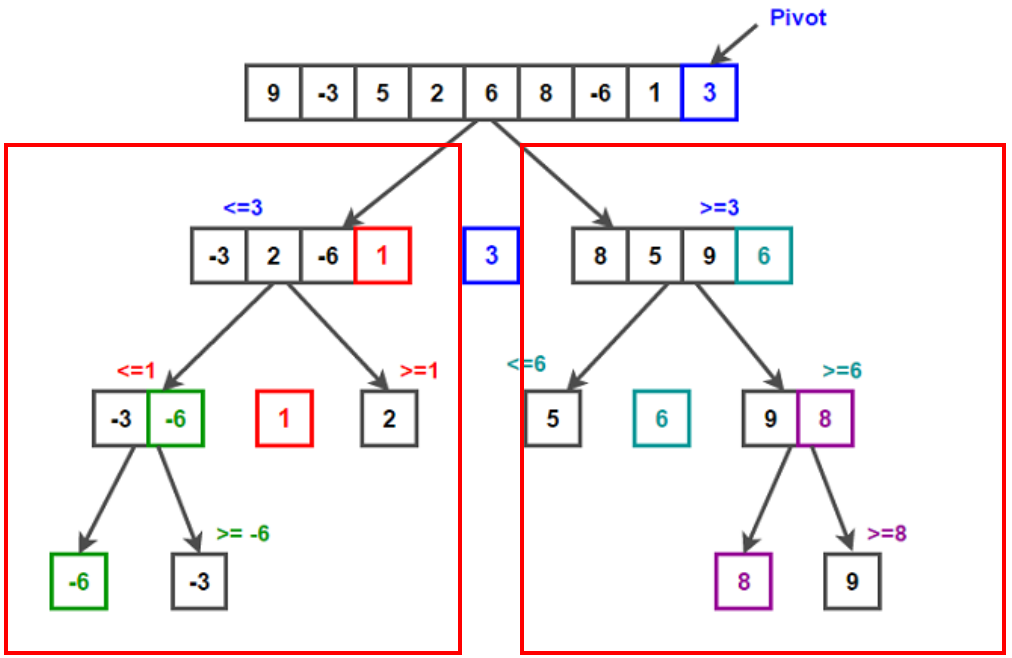
- Susceptible to stack overflow
  - Recursion depth is too high



Approach

- Independently sort each subarray
  - Merge results at the end

- Utilization of Open Multi-Processing API (OpenMP)
  - Creating multiple parallels between pivot point
  - Seeing the difference of # of threads

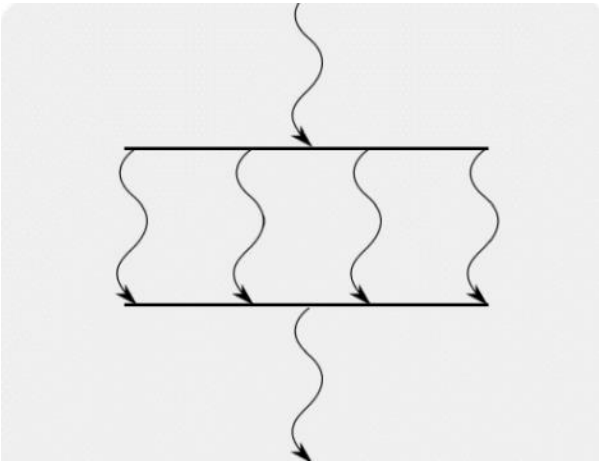


OpenMP

- Globally sequential & locally parallel

- Simplicity of sequential regions and parallel regions
  - Not N concurrently-executing threads

- Limitations
  - Requires compiler support
  - Does not consider dependencies
  - Dividing work optimally among threads not guaranteed



Implementation

- QuickSort_ Parallel function

```
void quickSort_parallel(int* array, int lenArray, int numThreads){

    int cutoff = 1000;

    #pragma omp parallel num_threads(numThreads)
    {
        #pragma omp single nowait
        {
            quickSort_parallel_internal(array, 0, lenArray-1, cutoff);
        }
    }

}
```

Implementation

- QuickSort_ Parallel_internal function

```
void quickSort_parallel_internal(int* array, int left, int right, int cutoff)
{

    int i = left, j = right;
    int tmp;
    int pivot = array[(left + right) / 2];


    {
        /* PARTITION PART */
        while (i <= j) {
            while (array[i] < pivot)
                i++;
            while (array[j] > pivot)
                j--;
            if (i <= j) {
                tmp = array[i];
                array[i] = array[j];
                array[j] = tmp;
                i++;
                j--;
            }
        }

    }


    if ( ((right-left)<cutoff) ){
        if (left < j){ quickSort_parallel_internal(array, left, j, cutoff); }
        if (i < right){ quickSort_parallel_internal(array, i, right, cutoff); }

    }else{
        #pragma omp task
        { quickSort_parallel_internal(array, left, j, cutoff); }
        #pragma omp task
        { quickSort_parallel_internal(array, i, right, cutoff); }
    }

}
```

Implementation

- OpenMP

```
#pragma omp task
{
    quickSort_parallel(arr1, lenArr, numthreads);
    quickSort_parallel(arr2, lenArr, numthreads);
    quickSort_parallel(arr3, lenArr, numthreads);
    quickSort_parallel(arr4, lenArr, numthreads);
}
#pragma omp taskwait
```

Attempt 1

- Proof of concept
  - Split into 2 threads
  - Test a small amount of numbers
  - 100 random numbers

- Results
  - 127.1 microsecond difference per instance
  - Multi thread is faster by 34%

Algorithm	# of Threads	Micro Seconds	Seconds
Single Thread	1	4,926	0.04926
Multi Thread	2	3,655	0.03655
Difference	1	1,271	0.01271

Attempt 2

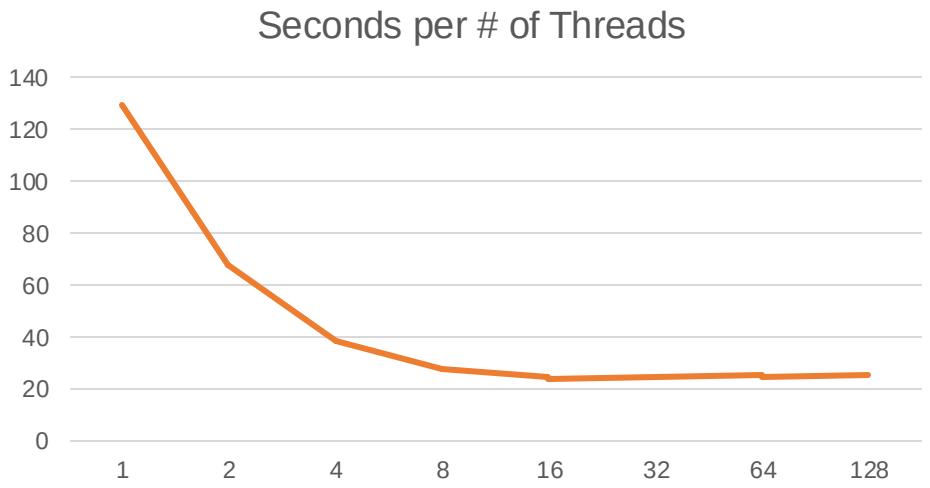
- Increasing size of threads & instances
  - Split into 32 threads
  - Test bigger # of instances
  - 20,000 random numbers

- Results
  - 32 multi thread is faster by 885%
  - 27.65% faster per thread showing decrease in efficiency per thread

Algorithm	**# of Threads**	**Micro Seconds**	**Seconds**
Single Thread	1	144,201,210	144.2
Multi Thread	32	14,630,483	14.63
Difference	31	127,570,727	129.57

Attempt 2

- With 32 threads being the fastest, higher thread counts start showing worse performance



Rank	Algorithm	# of threads	CPU cycles	Micro secs	Secs
4	Multi Thread	256	55431550139	25210600	25.21
3	Multi Thread	128	55253053022	25129418	25.13
2	Multi Thread	64	54463860517	24770486	24.77
1	**Multi Thread**	**32**	**53966704758**	**24544239**	**24.54**
5	Multi Thread	16	60881347852	27689197	27.69
6	Multi Thread	8	83819010981	38121381	38.12
7	Multi Thread	4	149498953304	67992995	67.99
8	Multi Thread	2	284104157817	129212201	129.2
10	Multi Thread	1	556500860191	253096787	253.1
9	Single Thread	1	548554831796	249485762	249.5

Attempt 2

- With 32 threads being the fastest, higher thread counts start showing worse performance

Rank	Algorithm	Optimization	# of threads	CPU cycles	Micro secs	Secs
	Multi Thread	O0	512	57372331508	26093416	26.09
	Multi Thread	O0	1024	62154942071	28268569	28.27
	Multi Thread	O0	2048	64195930763	29196811	29.20
	Multi Thread	O0	4096	64586982016	29374652	29.37
	Multi Thread	O0	8192	65554735505	29814782	29.81
	Multi Thread	O0	16384	failed	failed	failed
	Multi Thread	O0	32768	failed	failed	failed

Attempt 2

- O0: No optimization
- O1: Compiler basic code optimization
- O2: Deeper loop and register optimizations
- O3: Optimizations targeting multi-core and SIMD, like auto-vectorization and parallelization

Rank	Algorithm	Optimization	# of threads	CPU cycles	Micro secs	Secs
1	**Multi Thread**	**O3**	**32**	**32168720272**	**14630483**	**14.63**
2	Multi Thread	O2	32	32235606151	14660911	14.66
3	Multi Thread	O1	32	33850838505	15395528	15.40
4	Multi Thread	O0	32	53966704758	24544239	24.54

Future Improvements

- Dynamic Scheduling
  - The static scheduling currently used can be changed to guided or dynamic to make recursive segmentation more balanced

- Testing Different Algorithms
  - Compare performance of fast, merge, and heap sorting

- Exploring GPU parallelism
  - Implementing GPU versions with CUDA or OpenCL

Conclusion

- Improved performance by 885% compared to single thread
- Compared the linearity of thread counts
- Compared the performance difference between optimizations
- Provided ideas to improve future works

Q & A