Assignment 1

Shared- Memory Programming

Name: Brandon Zukowski

Student Id: 1481598

CCID: bjzukows

PSRS Algorithm

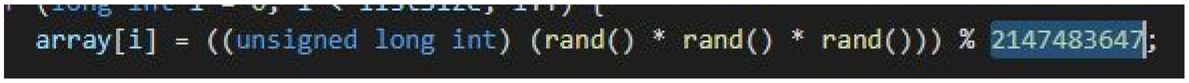
The PSRS algorithm involves 4 phases where the key idea is to perform most of the sorting through parallel processes in phase 1, phase 3, and phase 4, while phase 2 involves sequential pivot selection by one main processor. we know that according to Amdahl's law a parallel algorithm is largely affected by its sequential sections, therefore the PSRS algorithm looks to keep phase 2 as small as possible when selecting pivots for load balancing the sorting of the array. Amdahl's law is defined as the following, where (seq) is the time required to complete all sequential processes.

$$\lim_{p \to \infty} S(p) = \frac{1}{seq}$$

Ultimately the more sections of our algorithm that run in parallel, the better speed up gain we get from adding more processors. We will look to examine the performance of PSRS and output metrics such as speedup, execution times for each phase of the algorithm, and the effects of varying the input size.

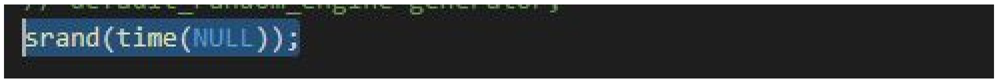
Implementations

Our test environment involved using the MinGW compiler for windows on a machine with 6 cores. The POSIX library functions srandom() and random() are not included with the MinGW compiler for windows. So we generated large random numbers by multiplying many smaller random numbers from srand() and rand(), followed by taking the modulus of the largest value for a long int data type (which is 2^31 - 1 = 2147483647).



```
  array[i] = ((unsigned long int) (rand() * rand() * rand())) % 2147483647;
```

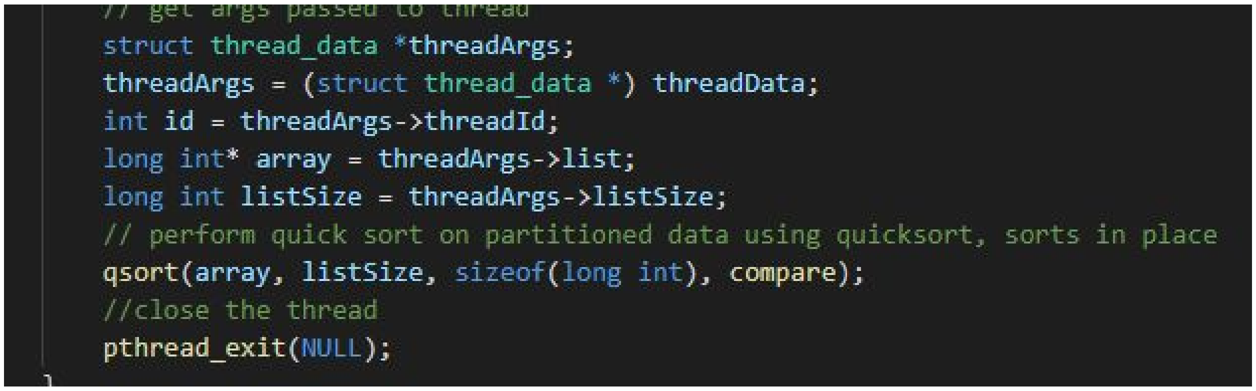
The random number was seeded with the current time, therefore guaranteeing we would always get new numbers with each program run.



```
srand(time(NULL));
```

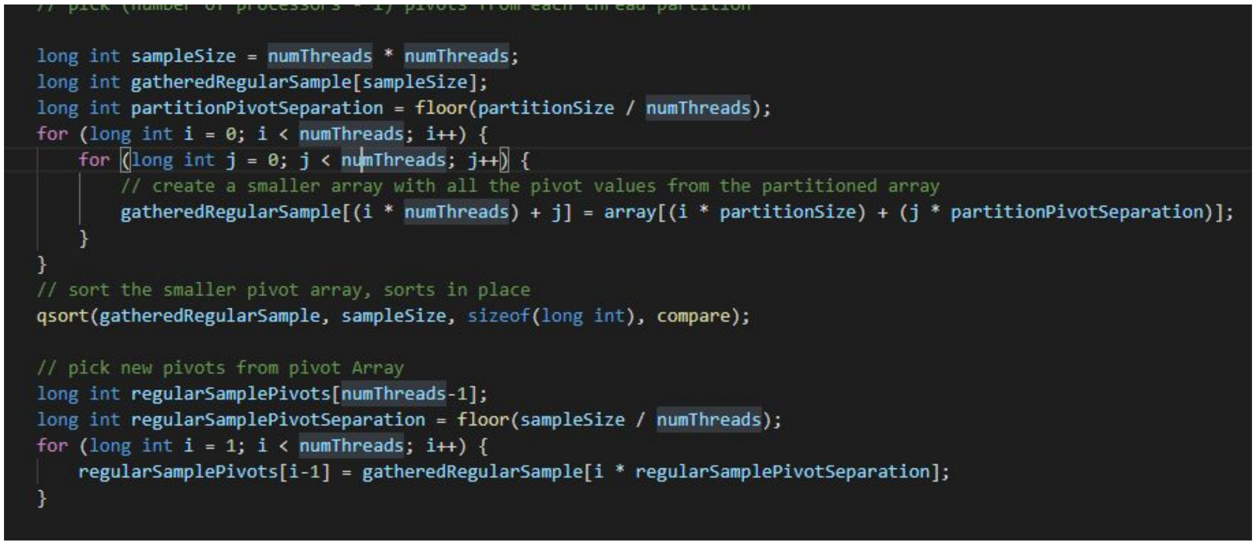
For every test run, we defined the number of processors and the array size to be sorted. We performed 7 runs for each input combination and averaged the last 5 runs in order to eliminate experimental errors due to process or startup overhead. We measured the execution time of each phase as well as the total time to complete the sort from beginning of phase 1 to end of phase 4. We tested the algorithm over a processor range of 1 to 6 and over three input sizes of 10 million, 50 million, and 100 million large integers.

For Phase 1 we simply partition the array passing almost equal parts of it to each processor which allows for a load balanced sort, followed by each processor performing quicksort on its segment. The sorting is performed in place since each processor is given a pointer of where to start sorting and the length of the segment it needs to sort. We do not need to use any mutex locks to control resource access in phase 1 since the segments to sort are evenly distributed amongst the processors and are exclusive from each other.



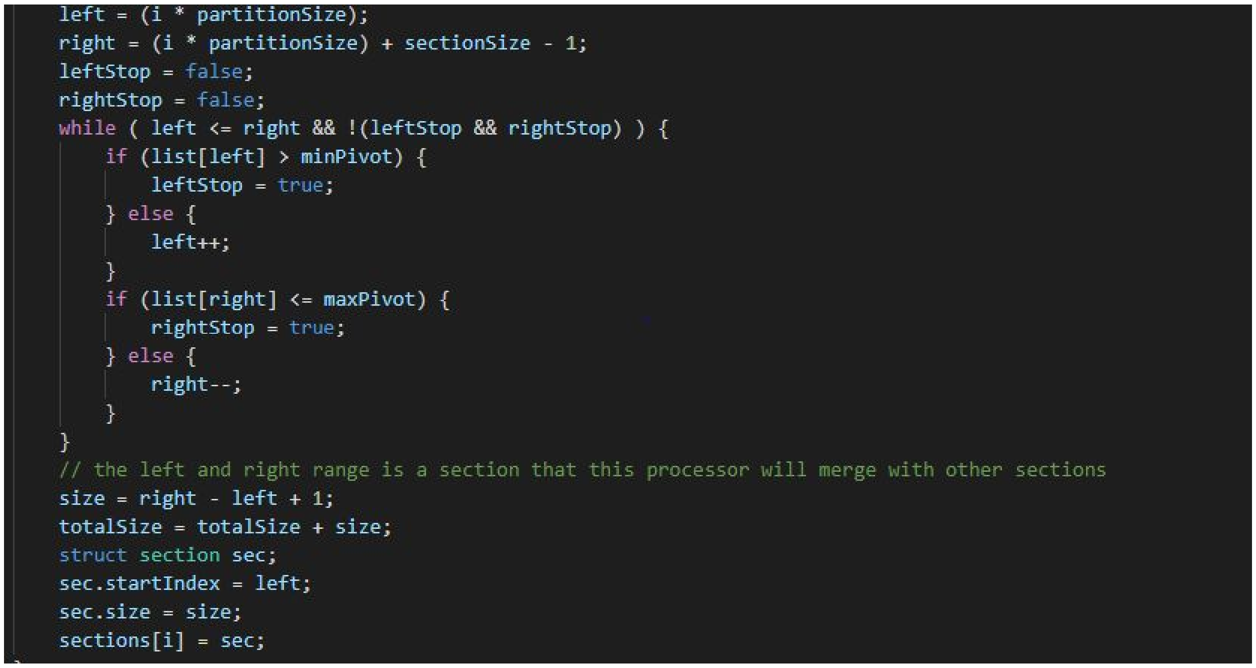
```
    // get args passed to thread
    struct thread_data *threadArgs;
    threadArgs = (struct thread_data *) threadData;
    int id = threadArgs->threadId;
    long int* array = threadArgs->list;
    long int listSize = threadArgs->listSize;
    // perform quick sort on partitioned data using quicksort, sorts in place
    qsort(array, listSize, sizeof(long int), compare);
    //close the thread
    pthread_exit(NULL);
}
```

For Phase 2, if our program has (p) number of processors, we gather (p) pivots from each partition sorted from phase 1 and store in a variable called (gatheredRegularSample). We then sort the sample in ascending order and pick (p-1) pivots from this sample and store in (regularSamplePivots) which we will use to divide up and exchange partitions to each processor in phase 3.



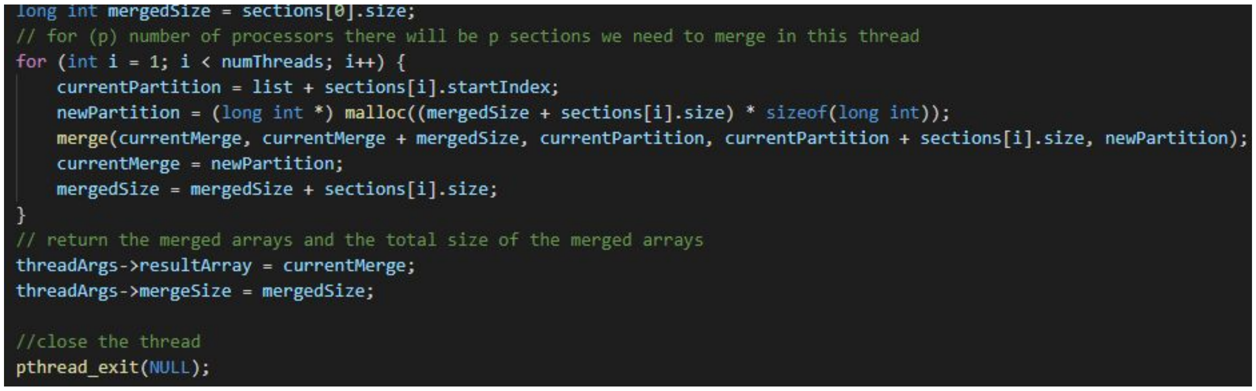
```
// pick (number of processors - 1) pivots from each thread partition

long int sampleSize = numThreads * numThreads;
long int gatheredRegularSample[sampleSize];
long int partitionPivotSeparation = floor(partitionSize / numThreads);
for (long int i = 0; i < numThreads; i++) {
    for (long int j = 0; j < numThreads; j++) {
        // create a smaller array with all the pivot values from the partitioned array
        gatheredRegularSample[(i * numThreads) + j] = array[(i * partitionSize) + (j * partitionPivotSeparation)];
    }
}
// sort the smaller pivot array, sorts in place
qsort(gatheredRegularSample, sampleSize, sizeof(long int), compare);

// pick new pivots from pivot Array
long int regularSamplePivots[numThreads-1];
long int regularSamplePivotSeparation = floor(sampleSize / numThreads);
for (long int i = 1; i < numThreads; i++) {
    regularSamplePivots[i-1] = gatheredRegularSample[i * regularSamplePivotSeparation];
}
```

For Phase 3, each processor is given a minimum pivot and a maximum pivot value which will act as a range in determining which sections it needs from the other processors. We use a sliding window approach to find the range of indices that are between its pivot values determined from phase 2. Each processor scans the list segments by having a left pointer at the beginning of the segment and right pointer at the end of the segment. We keep moving the left and right pointer closer to each other until all elements between the left and right pointer are within the range of the minimum and maximum pivot values. We store these section ranges in a struct (section), which can then be used by the processors in phase 4 for merging the exchanged partitions.



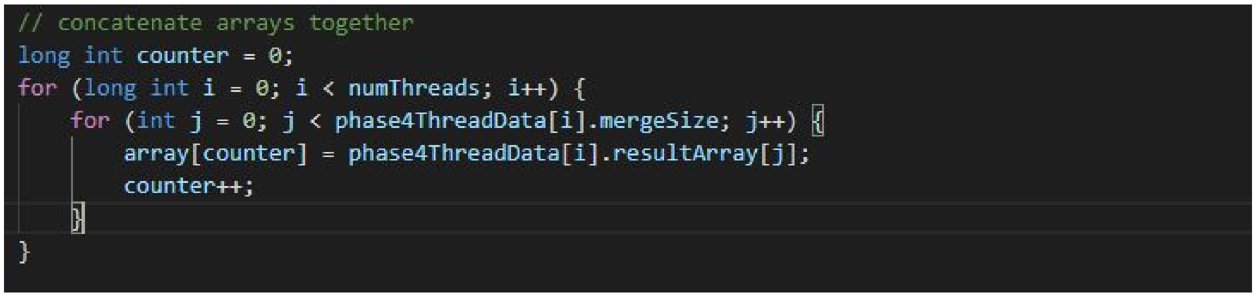
```
    left = (i * partitionSize);
    right = (i * partitionSize) + sectionSize - 1;
    leftStop = false;
    rightStop = false;
    while ( left <= right && !(leftStop && rightStop) ) {
        if (list[left] > minPivot) {
            leftStop = true;
        } else {
            left++;
        }
        if (list[right] <= maxPivot) {
            rightStop = true;
        } else {
            right--;
        }
    }
    // the left and right range is a section that this processor will merge with other sections
    size = right - left + 1;
    totalSize = totalSize + size;
    struct section sec;
    sec.startIndex = left;
    sec.size = size;
    sections[i] = sec;
```

For Phase 4, each processor loops and merges the sections created from Phase 3 to create a final sorted list for that processor.



```
long int mergedSize = sections[0].size;
// for (p) number of processors there will be p sections we need to merge in this thread
for (int i = 1; i < numThreads; i++) {
    currentPartition = list + sections[i].startIndex;
    newPartition = (long int *) malloc((mergedSize + sections[i].size) * sizeof(long int));
    merge(currentMerge, currentMerge + mergedSize, currentPartition, currentPartition + sections[i].size, newPartition);
    currentMerge = newPartition;
    mergedSize = mergedSize + sections[i].size;
}
// return the merged arrays and the total size of the merged arrays
threadArgs->resultArray = currentMerge;
threadArgs->mergeSize = mergedSize;

//close the thread
pthread_exit(NULL);
```

After the processors finish merging their exchanged partitions and each return a final sorted list, we concatenate the result arrays in order which gives our final sorted array.



```
// concatenate arrays together
long int counter = 0;
for (long int i = 0; i < numThreads; i++) {
    for (int j = 0; j < phase4ThreadData[i].mergeSize; j++) {
        array[counter] = phase4ThreadData[i].resultArray[j];
        counter++;
    }
}
```

Results

Figure 1 (10 million large integers as input array to sort)
** Note the time is measured in seconds

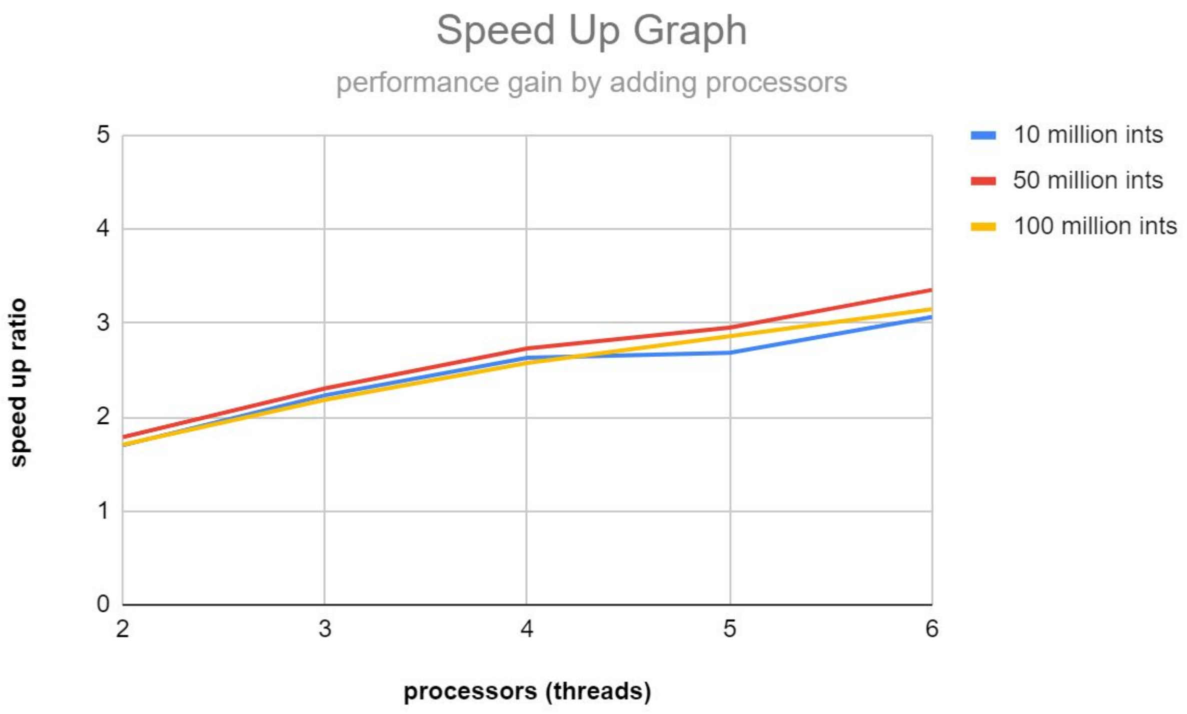
processors	Total time	Phase 1 time	Phase 2 time	Phase 3 time	Phase 4 time
1	3.439	3.398	0	0.04	0.001
2	2.025	1.833	0	0.017	0.162
3	1.541	1.375	0	0.026	0.14
4	1.307	1.114	0	0.039	0.154
5	1.281	0.987	0	0.048	0.246
6	1.122	0.883	0	0.048	0.191

Figure 2 (50 million large integers as input array to sort)
** Note the time is measured in seconds

processors	Total time	Phase 1 time	Phase 2 time	Phase 3 time	Phase 4 time
1	19.542	19.373	0	0.167	0.002
2	10.949	10.207	0	0.082	0.66
3	8.477	7.604	0	0.125	0.748
4	7.155	6.195	0	0.159	0.801
5	6.618	5.559	0	0.198	0.861
6	5.826	4.759	0	0.228	0.839

Figure 3:
Figure 2 (100 million large integers as input array to sort)
** Note the time is measured in seconds

processors	Total time	Phase 1 time	Phase 2 time	Phase 3 time	Phase 4 time
1	37.42	37.112	0	0.276	0.032
2	21.939	20.481	0	0.172	1.286
3	17.144	15.328	0	0.253	1.563
4	14.533	12.638	0	0.328	1.567
5	13.077	11.003	0	0.422	1.652
6	11.884	9.79	0	0.426	1.668



Discussion

Speed up is defined as the execution time of the algorithm on a single processor over the execution time of the algorithm with (p) number of processors.

$$S_p = T_1 / T_p$$

We can see from figure (4) that the algorithm follows a linear performance curve with a low increasing gradient as we add more resources and processors to sort the array. Our machine had a maximum of 6 cores so we could not determine when the linear performance would fall off past this range. Figure (4) shows that the PSRS algorithm performs consistently for speedup across various input sizes of large random numbers. We can see that the total execution time with (p) number of processors roughly increases linearly with the input size from figures (1), (2), and (3). For example, the input size between Figure (2) and Figure (1) differs by a factor 5 and is reflected in the total execution time between the two tables for any processor size (p) we pick. This same relationship applies between any of the three tables we pick.

As expected phases 1, 3, and 4 take up most of the total execution time, which is good since this will maximize our speedup results as we add more processors because all these phases perform parallel work. Phase 2 barely contributes at all to the overall execution time so it is negligible. One thing to note about phase 2 of the algorithm is that it picks pivots with a naive and simple approach, which means they might not be the most optimal pivots for load balancing in phase 3. This naive approach allows us to keep the sequential portions of the algorithm to a minimum which improves our ability for speedup. We could potentially gain a performance boost in phase 2 by choosing a method that picks more optimal pivots, however this improvement in phase 3 execution time due to better load balancing could be undermined by phase 2 performing more work in order to pick these optimal pivots.