Assignment 1

Shared- Memory Programming

Name: Brandon Zukowski

Student Id: 1481598

CCID: bjzukows

PSRS Algorithm

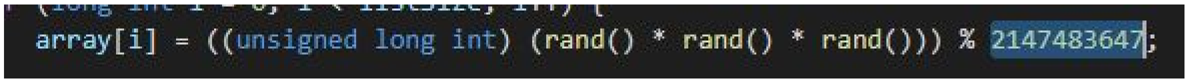
The PSRS algorithm involves 4 phases where the key idea is to perform most of the sorting through parallel processes in phase 1 and phase 4, while phases 2 and 3 involve sequential rearrange/exchange logic. we know that according to Amdahl's law a parallel algorithm is largely affected by its sequential sections, therefore the PSRS algorithm looks to keep phases 2 and 3 small in the work that has to be done to exchange sections of the arrays and pick the pivots, while all the heavy lifting of sorting is done in phases 1 and 4 by (p) processors. 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.

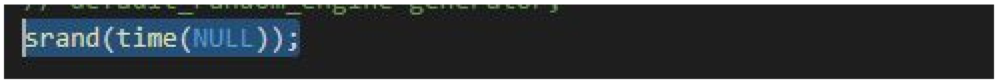
Implementations

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 16 and over three input sizes of 32 million, 64 million, and 128 million large integers.

Results

Figure 1 (32 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	10.3729	7.3672	0	0.4934	2.5146
2	7.8282	3.6043	0	0.5023	3.7243
3	4.6234	2.4186	0	0.5346	1.6734
4	3.9623	1.8214	0	0.5064	1.6751
6	3.3252	1.4283	0	0.5638	1.324
8	2.7719	1.1693	0	0.5263	1.0924
10	2.4734	0.9581	0	0.05372	0.9782
12	2.3091	0.8623	0	0.5424	0.8992
14	2.4493	0.9253	0	0.5492	0.9731
16	2.4823	0.9892	0	0.5134	0.9992

Figure 2 (64 million large integers as input array to sort)
** Note the time is measured in seconds

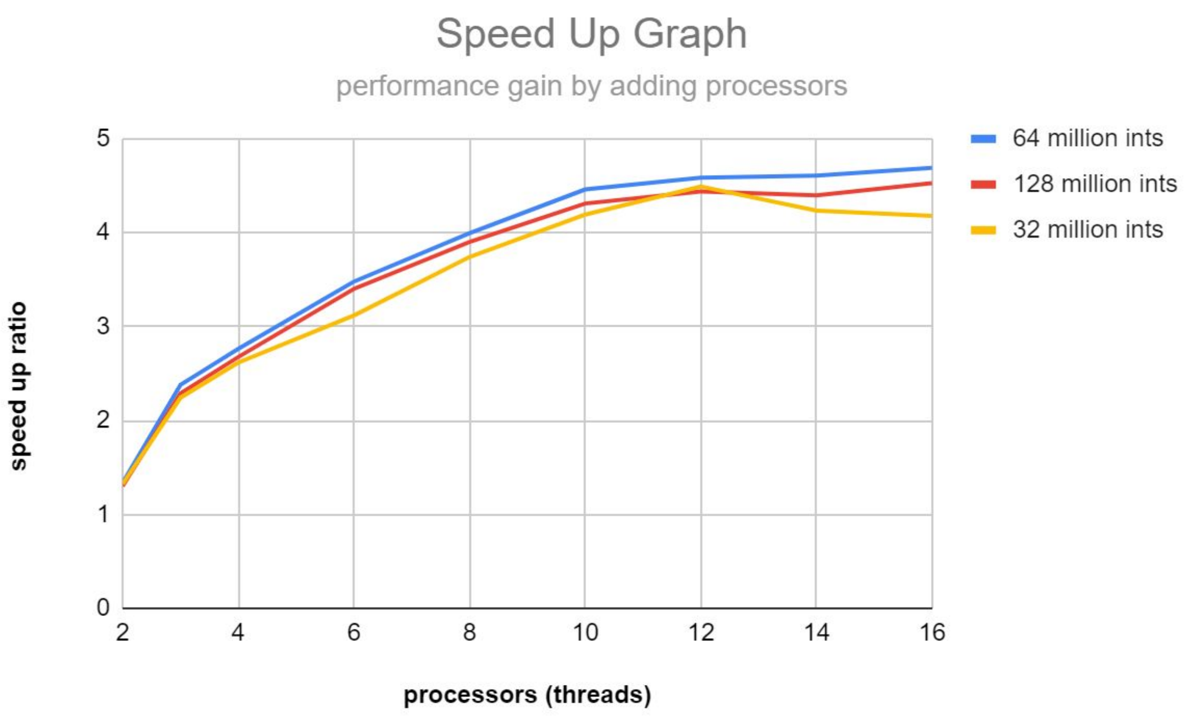
processors	Total time	Phase 1 time	Phase 2 time	Phase 3 time	Phase 4 time
1	22.6857	7.3846	0	1.0134	15.4637
2	16.9354	7.3843	0	1.01342	8.3823
3	9.5321	5.0193	0	1.097	3.416
4	8.2081	3.8471	0	1.0243	3.3372
6	6.5171	2.8932	0	1.0964	2.5281
8	5.6762	2.3291	0	1.0273	2.3245
10	5.0851	2.0521	0	1.524	1.9834
12	4.9471	1.9012	0	1.0943	1.9525
14	4.9234	1.9455	0	1.1203	1.8932
16	4.8345	1.8974	0	1.0345	1.8763

Figure 3 (128 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	44.6235	31.9673	0	2.0121	10.7162
2	34.2472	15.8345	0	2.0285	16.1372
3	19.4941	10.2784	0	2.1976	7.0193

4	16.6712	7.9953	0	2.0621	6.6146
6	13.1127	5.7563	0	2.1854	5.1738
8	11.4327	4.9736	0	2.0354	4.4256
10	10.3512	4.1337	0	2.1174	4.1034
12	10.0523	3.8782	0	2.1843	3.9883
14	10.1431	3.9783	0	2.2583	3.9612
16	9.8552	3.9287	0	2.0245	3.9505

Figure 4:

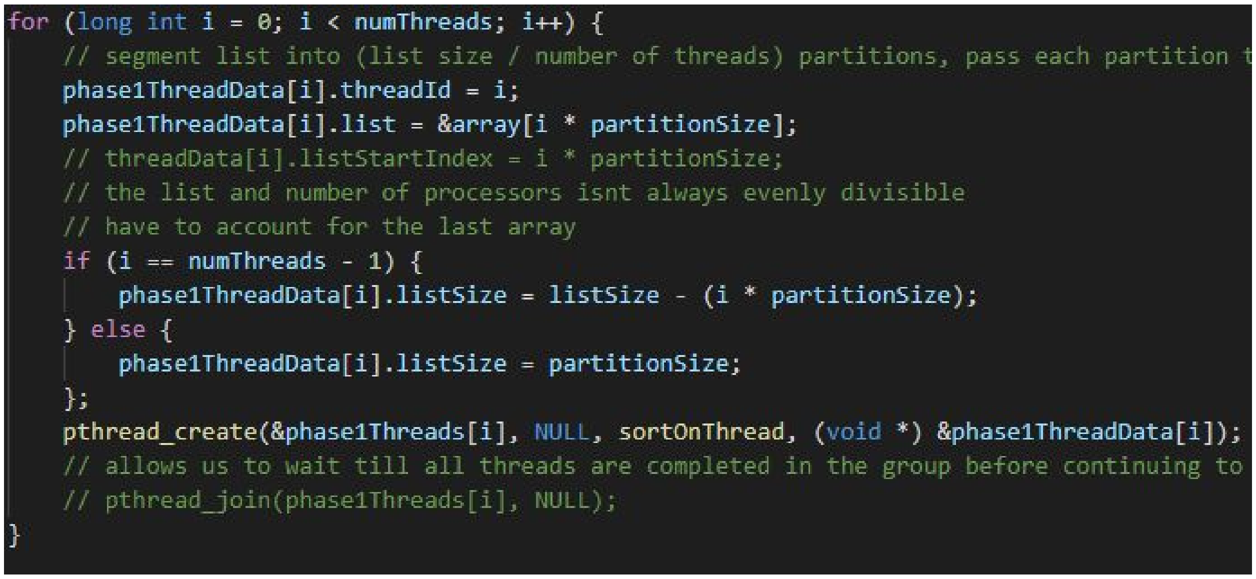


Discussion

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. We can see from figure (4) that the algorithm follows a sublinear performance curve as we add more resources and processors to sort the array. We start to lose the speedup gain roughly after 12 processors. As expected Phases 1 and 4 take up most of the total execution time of the program while phases 2 and 3 take up less than 5% of the total execution time for figure (1), figure (2), and figure (3), which is good since this will maximize our speedup results as we add more processors. If we were looking to improve the speed up of this algorithm further we would look at ways to reduce the execution time of phase 3 since it is a sequential section. Phase 2 is also sequential but barely contributes at all to the overall execution time so it is negligible. Moreover, figure (4) shows that the PSRS algorithm performs consistently across various input sizes of large random numbers.

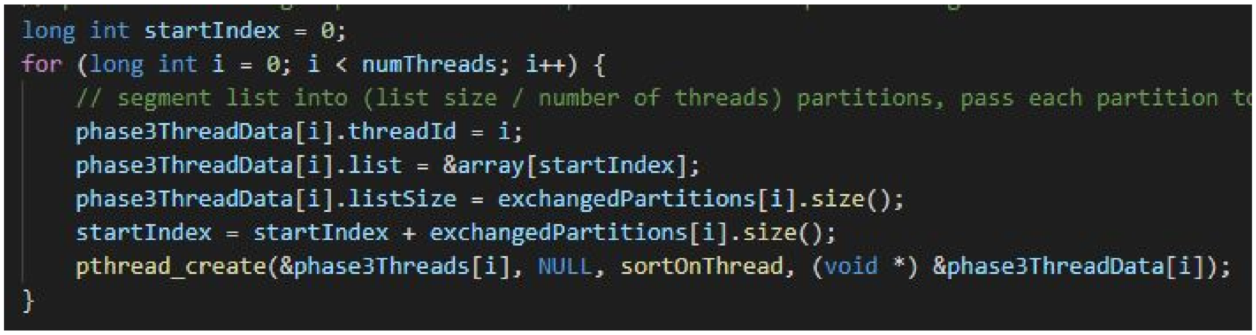
It was interesting to see that for figure (1) and figure (3) Phase 1 seemed to take up the majority of the execution time with 1 to 4 processors, but for figure (b) it was Phase 4 that took the most execution time for 1 to 4 processors. It seems strange that phase 1 and 4 change initial execution ratios where phase 1 takes up most of the execution time initially in figure (1), but then phase 4 dominates phase 1 in figure (2), where finally phase 1 dominates phase 4 again in figure (3). It is uncertain why this is the case and we would have to re-run the same tests in order to confirm this behaviour.

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.



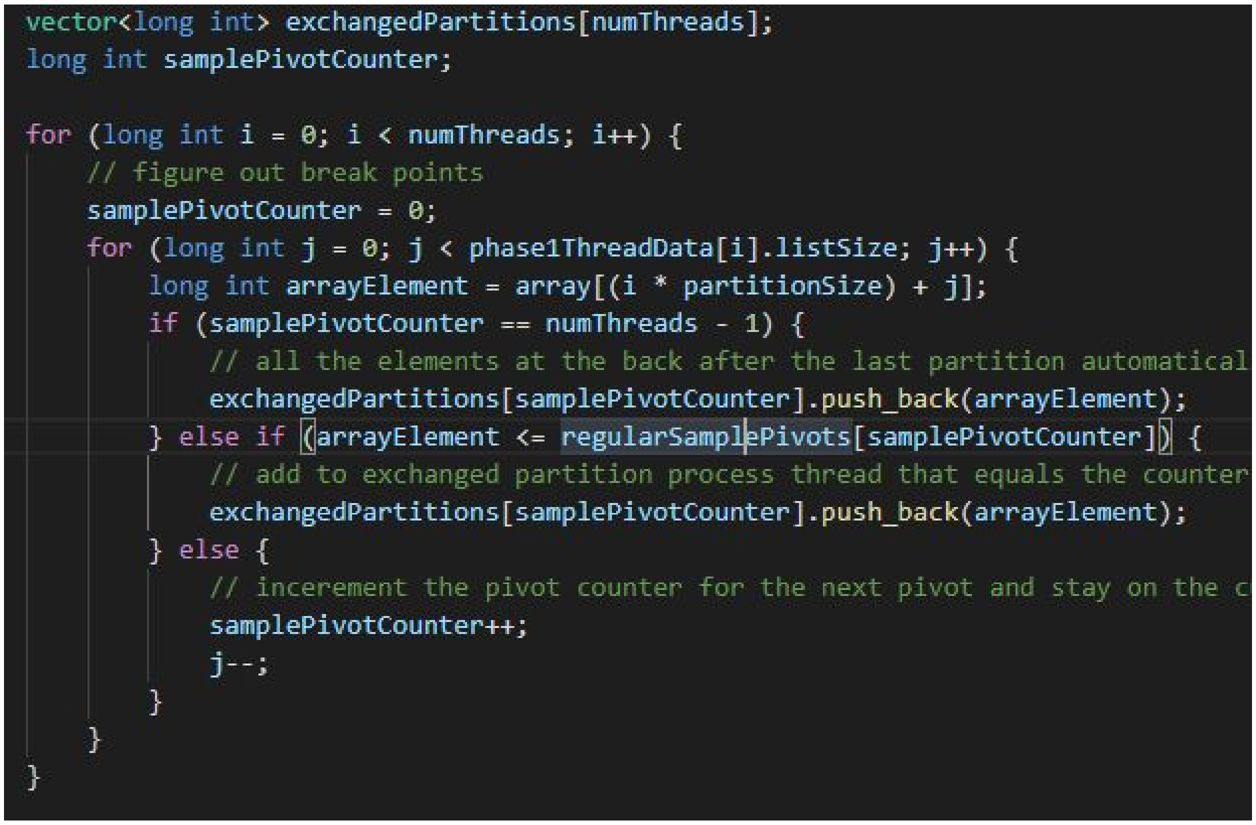
```
for (long int i = 0; i < numThreads; i++) {
    // segment list into (list size / number of threads) partitions, pass each partition t
    phase1ThreadData[i].threadId = i;
    phase1ThreadData[i].list = &array[i * partitionSize];
    // threadData[i].listStartIndex = i * partitionSize;
    // the list and number of processors isnt always evenly divisible
    // have to account for the last array
    if (i == numThreads - 1) {
        phase1ThreadData[i].listSize = listSize - (i * partitionSize);
    } else {
        phase1ThreadData[i].listSize = partitionSize;
    };
    pthread_create(&phase1Threads[i], NULL, sortOnThread, (void *) &phase1ThreadData[i]);
    // allows us to wait till all threads are completed in the group before continuing to
    // pthread_join(phase1Threads[i], NULL);
}
```

For Phase 4 each processor's collected parts from the exchange process are lined up to be right after the process before it and right before the processor ahead of it, so this means the entire array will be sorted once each processor is done merging their respective segments into sorted order.



```
long int startIndex = 0;
for (long int i = 0; i < numThreads; i++) {
    // segment list into (list size / number of threads) partitions, pass each partition t
    phase3ThreadData[i].threadId = i;
    phase3ThreadData[i].list = &array[startIndex];
    phase3ThreadData[i].listSize = exchangedPartitions[i].size();
    startIndex = startIndex + exchangedPartitions[i].size();
    pthread_create(&phase3Threads[i], NULL, sortOnThread, (void *) &phase3ThreadData[i]);
}
```

We use a series of vectors to help exchange the partitioned parts to each processor using the pivots in Phase 3
.



```
vector<long int> exchangedPartitions[numThreads];
long int samplePivotCounter;

for (long int i = 0; i < numThreads; i++) {
    // figure out break points
    samplePivotCounter = 0;
    for (long int j = 0; j < phase1ThreadData[i].listSize; j++) {
        long int arrayElement = array[(i * partitionSize) + j];
        if (samplePivotCounter == numThreads - 1) {
            // all the elements at the back after the last partition automatical
            exchangedPartitions[samplePivotCounter].push_back(arrayElement);
        } else if ((arrayElement <= regularSamplePivots[samplePivotCounter])) {
            // add to exchanged partition process thread that equals the counter
            exchangedPartitions[samplePivotCounter].push_back(arrayElement);
        } else {
            // incerement the pivot counter for the next pivot and stay on the c
            samplePivotCounter++;
            j--;
        }
    }
}
```