CMPUT 481 Winter 2024

Assignment: Shared-Memory Programming

Name: Sanjeev Kotha
Student Number: 1622030
CCID: kotha

PSRS Algorithm

The parallel sorting by regular sampling algorithm is one sorting algorithm that tries to achieve good suitable performance over a diverse range of MIMD architectures. This algorithm combines sequential sort, regular sampling, and partitioning to support load balancing, exchange of these partitions (not much in shared-memory systems), and a merge phase at the end of it. I present an implementation of PSRS on a shared-memory system tested on a MacBook Pro M2 using C++ and pthreads.

We will look at how this implementation of PSRS performs on variable-sized input and concern ourselves with performance evaluation using speedup and phase analysis.

Amdahl’s Law

Since PSRS has a sequential portion that does sorting, even in the ideal case that we could achieve unit-linear speedup for all of the other parallelizable portions, as we increase the number of processors p, we would be bound by the sequential portion of PSRS to achieve maximum speedup.

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

Implementation

As mentioned earlier, I implemented PSRS using shared-memory pthreads on a 16-core MacBook Pro M2 using C++ std 17. I wanted to use barrier synchronization for my implementation but since Apple does not provide a default implementation of its related functions, I had to seek an external source (cited in code) and modify it to make use of it locally.

To make experimentation less cumbersome, we implemented command-line arguments to set the number of threads (p) and the number of keys (n) to be sorted. For test data generation, we generate random data within the program using **time** as the seed value. This ensures that different pseudo-random sequences of values are generated. This method for supplying test data is much faster than reading an input file.

Additionally, when we measure the duration of time between phases of PSRS, we make use of the UNIX **gettimeofday()**, ensuring accurate time measurements.

For each thread, we maintain a data structure called ThreadControlBlock that stores the ID, the local block of data to be sorted as well as other data needed for further processing.

```
struct ThreadControlBlock {
    int id;
    long *localBlock;
    long blockSize;
    vector<pair<long, long>> localSample;
    vector<pair<long, long>> pivots;
    vector<pair<long, long>> partitions;
    vector<pair<long, long>> truePartitions;
    vector<long> mergedPartition;
};
```

For phase one, we assign a disjoint contiguous block of (n/p) keys to each processor. Each processor in parallel sorts it locally using sequential quicksort. Then, each processor gathers local samples at 0, w, 2w …, (p-1)w (w = n/(p^2)) to form a representation of the local block.

```
void *sortAndSampleLocal(void * arg) {
    auto * myTCB = (struct ThreadControlBlock *) arg;
    long * block = myTCB->localBlock;
    long size = myTCB->blockSize;

    qsort(block, size, sizeof(long), comparator);

    long w = arraySize / (threadCount * threadCount);

    for (int i = 0; i < threadCount; i++) {
        myTCB->localSample.emplace_back(block[i * w], (i * floor(arraySize
/ threadCount)) + (i * w));
    }

    pthread_barrier_wait(&mybarrier);
    pthread_exit(nullptr);
}
```

Then on our main thread, we begin phase two by collecting all these local samples from different blocks and sorting them. We select pivots from this collective sorted sample at indexes p + rho - 1, 2p + rho - 1, …, (p-1)p + rho - 1 (rho = floor(p/2)). After broadcasting a copy of these pivots to every thread, each thread in parallel forms p different partitions from the p-1 pivots. To reduce computational complexity, we just keep track of the indexes of the partitions rather than the whole data.

```
void *createPartitions(void *arg) {
    auto *myTCB = (struct ThreadControlBlock *)arg;
    long *block = myTCB->localBlock;
    long start = 0;
    long end;
    for (auto pivot : myTCB->pivots) {
        end = binary_search(block, myTCB->blockSize, pivot.first);
        myTCB->partitions.emplace_back(start, end);
        myTCB->truePartitions.emplace_back(start + myTCB->id * floor(arraySize / threadCount), end + myTCB->id * floor(arraySize / threadCount));
        start = end + 1;
    }

    myTCB->partitions.emplace_back(start, myTCB->blockSize - 1);
    myTCB->truePartitions.emplace_back(start + myTCB->id * floor(arraySize / threadCount), myTCB->blockSize - 1 + myTCB->id * floor(arraySize / threadCount));

    pthread_barrier_wait(&mybarrier);
    pthread_exit(nullptr);
}
```

On shared-memory implementations like mine, complexity of phase 3 is negligible and partition indexes are exchanged using a global partition tracker.

```
void *exchangePartitions(void *arg) {
    auto *myTCB = (struct ThreadControlBlock *)arg;
    for (int i = 0; i < threadCount; i++) {
        globalPartitions[i].push_back(myTCB->truePartitions[i]);
    }

    pthread_barrier_wait(&mybarrier);
    pthread_exit(nullptr);
}
```

To merge all these new partitions for each thread in phase 4, I use a priority queue based k-way merge algorithm.

```
for (auto & i : globalPartitions[myTCB->id]) {
        if (i.first <= i.second) {
            long t = *(values + i.first);
            pq.emplace(t, i.first, i.second);
        }
    }

    while (!pq.empty()) {
        QueueObject smallest = pq.top();
        pq.pop();
        result.push_back(smallest.getValue());
        long start = smallest.start;
        long end = smallest.end;
        if (start < end) {
            long t = *(values + start + 1);
            pq.emplace(t, start + 1, end);
        }
    }
```

The time complexity of this k-way merge is O(Nlogk) where N is the total number of elements. After this, each thread's merged partition gets concatenated to produce the sorted array. To verify the correctness of the sequential and parallel portions of the algorithm, I perform a sequential sort using the C++ sort() and an element-wise equality check with the sorted output from PSRS.

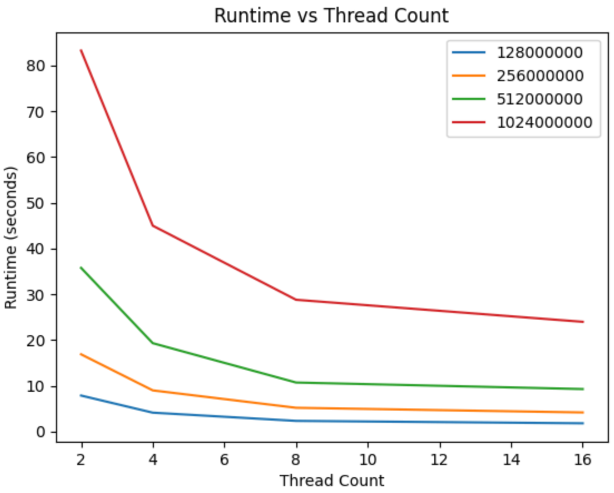
Experimentation

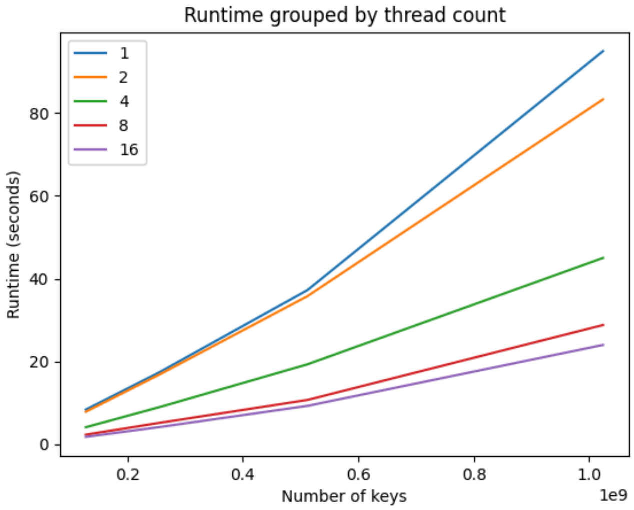
I tested this implementation locally on variable input sizes of 128M, 256M, 512M, and 1024M keys to be sorted along with thread counts of 2, 4, 8, and 16.

Sizes	Sorting Time (in seconds)				
	1 thread	2 threads	4 threads	8 threads	16 threads
128,000,000	8.4052	7.8875	4.1477	2.3503	1.8250
256,000,000	17.4556	16.8956	9.0181	5.2105	4.1943
512,000,000	37.2541	35.7921	19.3349	10.7340	9.3049
1,024,000,000	94.9410	83.2883	45.0136	28.8190	24.0024

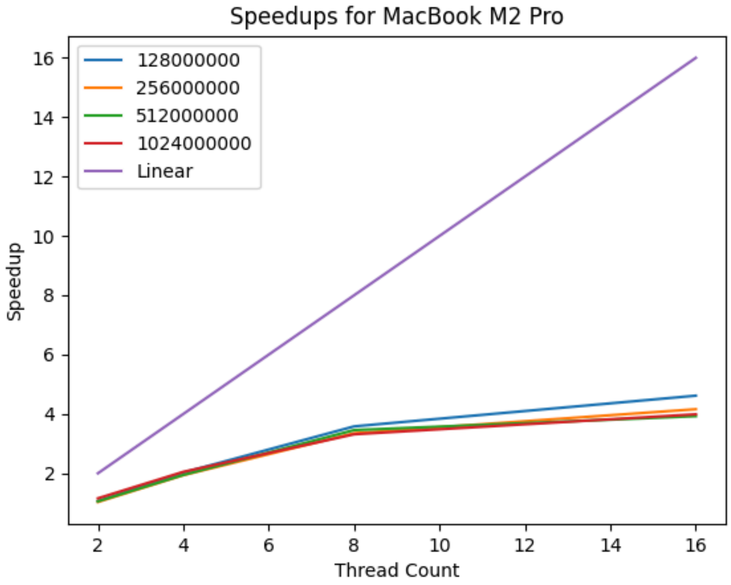
When performing the experiments locally, to collect data, I ran 7 runs for each datapoint and averaged the last 5 of those runs to avoid process or start-up overheads.

From the data collected above, we can tell that there are performance gains from PSRS by sorting large lists in parallel but this is not the whole picture. Since we are adding on more computational resources in the form of threads, we have to also consider granularity i.e. the ratio of computation to communication. As we increase the number of threads, we should increase the number of keys to be sorted as well to maintain good granularity.



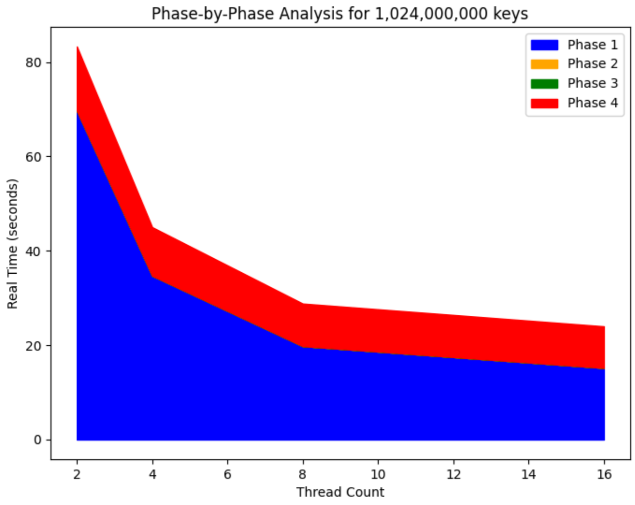


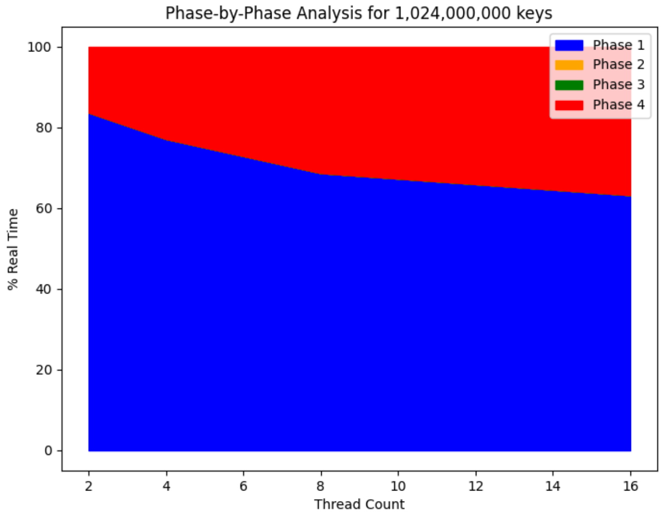
We see that as we vary the number of keys to be sorted, for larger sizes, a jump to a higher number of threads performs significantly better having a much shorter runtime. You may have also noticed that as we increase the number of keys, the run time increases at a much faster rate for a lower number of threads as compared to a higher number of threads. Both of these observations are related to how maintaining good granularity is important.



Going back to the topic of speedup and Amdahl's law, we know that speedup for a fix problem size n is defined as $S(p) = t_1/t_p$ where t1 is time to solve sequentially and tp is time to solve in parallel using p threads/processors.

Looking at the speedup graph, as the thread count increases, the lines flatten after 8 threads indicating diminishing returns for data sizes of 128M to 1024M. This sublinear speedup is not too surprising as communication overheads, especially with a higher number of threads cause a hit to performance. Due to real-world limitations of parallelism, we see that none of the lines approach the ideal linear speedup curve. We can also see that the PSRS implementation is very consistent concerning speedup across varying input sizes and number of threads.





Looking at the phase-by-phase analysis graphs, phase 1 and phase 4 take up most of the total execution time and both of these are parallel work. Phase 2 is barely noticeable because the process of gathering pivots is very trivial and negligible as it gets zeroed out from microseconds when calculating execution time.

Since this is a shared-memory implementation, phase 3 is also negligible as partitions are being exchanged within memory. As we increase the thread count, the percentage of real time spent in phase 1 and 4 decreases slowly and this indicates speedup although it is sublinear.

Ending Remarks

The implementation of the shared-memory PSRS showed increased speedups while demonstrating Amdahl's law and the real-world limits of parallelism due to various factors. By varying the input sizes and the number of threads, we could see how the concept of granularity comes into play. Lastly, we could see that phases 1 and 4 were the most time-consuming portions of the PSRS algorithm, and optimizing these phases would be a huge benefit as both of these phases are parallel in nature.