Michael Spohn

Parallel HeapSort

Heap sorting is one of the popular sorting algorithms used, especially with priority queues and other data structures. The heapsort algorithm involves taking an array of values and rearranging the values in a tree-like structure, order where the maximum or the minimum is the root. Afterwards, the root is swapped to the end of the array, solidifying its sorted position. The array is heapified again and this process continues until the values are in ascending or descending order. This is a quick sorting algorithm compared to other comparison-based sorts, however the two stages it has can slow it down with large array sizes being used. Parallelizing the heapsort algorithm could possibly assist in decreasing the amount of time it takes to complete these two stages.

```
// Build the max heap from the middle point towards index 0
for (int i = n/2 - 1; i >= 0; i--) heapify(arr, n, i);

for (int i = n-1; i > 0; i--) {
    swap(arr[0], arr[i]);   // Swap root to end of heap
    heapify(arr, i, 0);     // Heapify on current index
}
```

The heapify function involves comparisons to determine the largest (or smallest) value and if values in the array need to be swapped around. The serial solution involves taking an array, building the heap, and then swapping values while rebuilding the heap until all values have been swapped to their correct position. Two for loops are used in this process. A node structure could also be used but wasn't in this serial solution. A node structure could simplify the procedure for determining the left and right child values for a given value, as this is done in the heapify function. The heapify function is also recursive, so values can be swapped around, and the heap can continue to be structure correctly starting at the midpoint. The heapify function is used as is for every parallel attempt, however the ideas discussed and attempted could possibly be implemented in the heapify function instead of around it.

```
void heapify(int arr[], int size, int i) {
    int max = i;                    // largest index is root
    int l = 2*i + 1;                // left child
    int r = 2*i + 2;                // right child

    // If left or right child is larger, update max
    if (l < size && arr[l] > arr[max]) max = l;
    if (r < size && arr[r] > arr[max]) max = r;

    // If child was larger than root, swap and heapify on the new max
    if (max != i) {
        swap(arr[i], arr[max]);
        heapify(arr, size, max);
    }
}
```

There are four different programs used for this project: serial, pthread, omp, and heap. The serial program is what was previously discussed and is the foundation of the other programs. The pthread program utilizes pthreads to parallelize the sorting stage of the heapsort algorithm. The omp program utilizes OpenMP to parallelize both stages of the heapsort algorithm. Finally, the heap program utilizes OpenMP to parallelize the initial heap building stage of the heapsort algorithm. The idea of these parallelizations is to reduce the amount of time during the swapping around of values, however due to restrictions on this simple heapsort foundation, values may get swapped incorrectly or multiple times over.

The pthreads program focuses on the sorting stage of the heapsort algorithm, where a for loop constantly swaps the first and last values of the heap. Then the array is heapified, disregarding the last value. The idea of using pthreads here is to do multiple swaps and heapifies at once, however the issue here is that values could be swapped incorrectly, and the array may be heapified with values in incorrect positions. Therefore, threads will wait until a loop is completed before doing their loop. This ends up becoming much like the serial program with a higher cost, especially since there's a condition to break away from the loop if the index is less than zero. This is not a parallel solution since there is no work being done in parallel, as threads are having to wait their turn.

```
void *Slave(void* rank) {
    int my_rank = (long) rank;

    for (; j > 0; j--) {
        pthread_mutex_lock(&mutex1);
        if (j <= 0) break;
        //cout << "Thread_" << my_rank << ": " << j << endl;
        if (arr[0] > arr[j]) {
            swap(arr[0], arr[j]);   // Swap root to end of heap
            heapify(arr, j, 0);     // Heapify on current index
        }
        pthread_mutex_unlock(&mutex1);
    }
    pthread_mutex_unlock(&mutex1);

    return NULL;
}
```

The OpenMP programs do work in parallel in comparison to the pthreads program. The omp program attempts to heapify parts of the array and slowly heapify them again in a merging fashion. While this is done in parallel, this method theoretically takes more time since more loops are having to be done for smaller portions of the given array. Then, all the sorted parts will be heapified with neighboring parts of the array until the entire thing is sorted. This is much more work than the serial program, but it utilizes OpenMP successfully in doing parallel work.

```
    #pragma omp parallel num_threads(thread_count)
    {
        int local_n = n/thread_count;
        int my_rank = omp_get_thread_num();

        int divisor = 2;        // Used in determining even/odd local array
        while (divisor <= thread_count) {
            #pragma omp barrier // Wait for all threads to finish
            if (my_rank % divisor == 0) {
                for (int i = local_n - 1; i >= 0; i--)
                    heapify(arr, local_n*2, i);
                for (int i = (local_n*2)-1; i > 0; i--) {
                    swap(arr[0], arr[i]);   // Swap root to end of heap
                    heapify(arr, i, 0);     // Heapify on current index
                }
                local_n*=2;
            }
            divisor *= 2;       // Double divisor
        }
    }
```

Finally, the heap program continues the use of OpenMP to try and reduce the amount of time the initial heap-building process takes. Unlike the sorting stage, the heap building can be divided up into separate parts if one subtree doesn't overlap another that's being addressed by another thread. Each subtree must be dealt with individually or this process won't work. The heap program does not implement this entirely, but it uses a function that calculates what level an item is in the heap. Conditional statements can be used to determine how many threads should be used at a time to avoid overlapping. Unfortunately, the implementation done in the heap program doesn't have the subtrees addressed by different threads, meaning all threads visit all subtrees. A method of thread hopping should be used to accomplish this idea.

```
#pragma omp parallel num_threads(thread_count)
{
        for (int i = mid; i >= 0; i--) {
            #pragma omp barrier
            r = revel(n, i, 1);
            if (r > 1 && my_rank < 2) {             // 2 threads max
                heapify(arr, n, i);
            } else if (r > 2 && my_rank < 4) {      // 4 threads max
                heapify(arr, n, i);
            } else if (r > 3 && my_rank < 8) {     // 8 threads max
                heapify(arr, n, i);
            } else if (r > 4) {                    // 16 threads max
                heapify(arr, n, i);
            } else if (my_rank == 0) {             // 1 thread only
                heapify(arr, n, i);
            }
        }
}
```

When comparing these functions in terms of execution time, as all programs have a form of time calculation, it should be noted that cost plays a factor in the different of times. Also, not all implementations are fully functional and work as expected. Some programs can only sort to a certain limit of array size, likely due to how they're implemented and/or the amount of space threads are able to use with recursive functions. Below is a time comparison of the four programs that have been discussed in this report.

Michael Spohn

Array size = 100:

Threads:	1	2	4	8	16
Serial	0.000701904s	N/A	N/A	N/A	N/A
PThread	0.00199986s	0.00242591s	0.00229287s	0.00248885s	0.00302887s
OMP	*0.0010232s*	0.0009735s	0.0011054s	*0.0014792s*	*0.0023544s*
Heap	0.001008s	0.0014623s	0.0021415s	0.0034575s	0.0050847s

Array size = 1,000:

Threads:	1	2	4	8	16
Serial	0.000132799s	N/A	N/A	N/A	N/A
PThread	0.000313044s	0.001724s	*0.00200796s*	*0.0024929s*	0.00282383s
OMP	*2.85e-05s*	0.0002806s	0.0007054s	0.0009227s	*0.001723s*
Heap	0.0002013s	0.0066317s	0.0110323s	0.0216601s	0.0419657s

Array size = 10,000:

Threads:	1	2	4	8	16
Serial	0.00174499s	N/A	N/A	N/A	N/A
PThread	0.00284815s	0.00355506s	*0.0365801s*	*0.0363059s*	0.037926s
OMP	*7.97e-05s*	0.0018752s	0.0026946s	0.0042836s	0.0052502s
Heap	0.0022557s	0.0664761s	0.108876s	0.212206s	0.327494s

Array size = 100,000:

Threads:	1	2	4	8	16
Serial	0.022192s	N/A	N/A	N/A	N/A
PThread	0.0265841s	*0.250867s*	*0.253922s*	*0.261906s*	*0.402133s*
OMP	*0.0005953s*	0.0220212s	0.0323025s	0.0356308s	0.0439449s
Heap	0.0272312s	0.657644s	*1.09251s*	*2.11977s*	*4.10632s*

Michael Spohn

Array size = 1,000,000:

Threads:	1	2	4	8	16
Serial	0.271324s	N/A	N/A	N/A	N/A
PThread	0.321778s	*2.62971s*	*2.72129s*	*2.76726s*	*2.82026s*
OMP	*0.0051182s*	0.272322s	0.342145s	0.388048s	0.428749s
Heap	0.332086s	6.61951s	*11.0592s*	*16.3572s*	*41.0962s*

Array size = 10,000,000:

Threads:	1	2	4	8	16
Serial	3.98446s	N/A	N/A	N/A	N/A
PThread	4.43895s	*41.059s*	*42.9531s*	*43.9207s*	*44.7354s*
OMP	*0.0475216s*	3.9651s	4.52558s	5.03522s	5.29879s
Heap	4.61243s	*52.6675s*	*> 1m*	*> 1m*	*> 1m*

Based off the ideas that have been implemented, the heap program should've been the most optimal despite the cost increase. However, as stated before, it's not set up in a way to do thread hopping, so all subtrees in the heap are being accessed based on the conditional. Therefore, from what was implemented, the omp program showed the most promising results. It should be noted that italicized entries in the above table mean that the program ran successfully, however the check_sorted function returned false. Also, with array size 10,000,000, execution times for the heap program were growing with the number of threads being used. If the implementation was done correctly, this would not occur, leading to execution times over one minute long.

In conclusion, there is a possibility to utilize parallelization with the heapsort algorithm, with or without a node structure. However, there will be an increase in cost, and the actual sorting isn't as easy to parallelize since it's a process that needs to be done one at a time. While

execution times in these implementations did not decrease, the ideas on how to use threads to do parallel work is present and further implementation should be doable based off of these programs.