Final Project

Parallel Merge Sort Using Shared Memory

Carlos Marquez

U69320592

marquezc@mail.usf.edu

James Keith

Jame’s UID

Jame’s Email

Stephen Astfanous

Stephen’s UID

Stephen’s email

John Astfanous

John’s UID

John’s email

Other dude

Other dude

Other dude

***Abstract*— This project takes a look into the optimization of mergesort by providing parallelization. We take a look and compare several implementations of mergesort such as utilizing kernel threads, child processes, and recursion. We also observe the limits of mergesort and compare it’s limits with algorithms such as bubblesort.**

***Index Terms*— mergesort, threads, processes, recursion, analysis, bubblesort**

# INTRODUCTION

The purpose of this project was to gain a deeper understanding of how to use a multiprocessor programming approach to optimize the mergesort sorting algorithm.

In this project, we create three different implementations of mergesort. We implement mergesort using threads, children processes, and recursion. We analyze and compare the runtimes of these implementations against each other as well as against an O(n2) implementation of bubblesort.

We run these implementations of mergesort and bubblesort against a number of elements which are randomly generated. The number of elements to sort are supplied by the user at runtime.

# Methodology

Since mergesort is an extremely popular algorithm, there were a multitude of sources to reference. We started by simply implementing a merging algorithm. The following is our implementation of the merge algorithm.

**Merge( ):**

void merge (int low, int high){

//-------------------------------

// indexes needed for merge sort

//-------------------------------

int mid = (low+high)/2;

int left = low;

int right = mid + 1;

int temp[high - low + 1];

int i = 0;

int curr = 0;

//-------

// merge

//-------

while(left <= mid && right <= high){

if (target[left] > target[right])

temp[curr++] = target[right++];

else

temp[curr++] = target[left++];

}

while(left <= mid)

temp[curr++] = target[left++];

while(right <= high)

temp[curr++] = target[right++];

for (i = 0; i < (high - low + 1); ++i)

target[low + i] = temp[i];

}

This merge algorithm does not vary and is used by every implementation of our mergesort.

After implementing our merging algorithm, we continued our implementation by developing the recursive mergesort algorithm. The recursive mergesort algorithm we implemented can be found below.

**Recursive Mergesort:**

void \* mergesortRecursive(void \*a){

arrInf \*inputArr = (arrInf \*)a;

int mid = (inputArr -> low + inputArr -> high) / 2;

//----------------------

// Split array in half

// Set two threads to

// work on each half

//----------------------

arrInf arrInfo[2];

arrInfo[0].low = inputArr -> low;

arrInfo[0].high = mid;

arrInfo[1].low = mid + 1;

arrInfo[1].high = inputArr -> high;

if (inputArr -> low >= inputArr -> high)

return 0;

//----------------------------

// Create the recursion calls

//----------------------------

mergesortRecursive(&arrInfo[0]);

mergesortRecursive(&arrInfo[1]);

merge(inputArr -> low, inputArr -> high);

return 0;

}

This recursive implementation of mergesort had been done in previous courses such as Analysis of Algorithms, so recreating the algorithm was trivial. As with all recursive mergesort implementations, the runtime of this algorithm is O(n log(n)) at the cost of O(n) additional space.

**Process Mergesort:**

void mergesortProcesses(void \*a){

arrInf \*inputArr = (arrInf \*)a;

int mid = (inputArr -> low + inputArr -> high) / 2;

//----------------------

// Split array in half

// Set two threads to

// work on each half

//----------------------

arrInf arrInfo[2];

arrInfo[0].low = inputArr -> low;

arrInfo[0].high = mid;

arrInfo[1].low = mid + 1;

arrInfo[1].high = inputArr -> high;

if (inputArr -> low >= inputArr -> high)

return;

//----------------------------

// Create the process calls

//----------------------------

pid\_t pid1;

pid\_t pid2;

if ((pid1 = fork()) == 0){

mergesortProcesses(&arrInfo[0]);

}

else if ((pid1 != 0) && (pid2 = fork()) == 0){

mergesortProcesses(&arrInfo[1]);

}

else if ((pid1 != 0) && (pid2 != 0)){

wait(NULL);

mergeShm(inputArr -> low, inputArr -> high);

}

}

The next step was to create the threaded implementation of mergesort. We had two options for the type of threads to use. We could either use kernel threads or process threads. Ultimately, we decided on using kernel threads since we want to increase the degree of parallelism. If we were to have used process threads, the main process would have to decide which thread to assign to the CPU and it would be limited to one single thread. Thus, if using process threads, we would have no parallelism. We introduced parallelism by using kernel threads. By using kernel threads, the operating system could see all kernel threads created by the mergesort algorithm, allowing for all threads to run concurrently. The following is the implementation for our threaded mergesort algorithm.

**Threaded Mergesort:**

void \* mergesortThreaded(void \*a){

arrInf \*inputArr = (arrInf \*)a;

int mid = (inputArr -> low + inputArr -> high) / 2;

//----------------------

// Split array in half

// Set two threads to

// work on each half

//----------------------

arrInf arrInfo[2];

pthread\_t thread[2];

arrInfo[0].low = inputArr -> low;

arrInfo[0].high = mid;

arrInfo[1].low = mid + 1;

arrInfo[1].high = inputArr -> high;

if (inputArr -> low >= inputArr -> high)

return 0;

//-------------------

// Create the threads

//-------------------

int thr1 = pthread\_create(&thread[0], NULL, mergesortThreaded, &arrInfo[0]);

if (thr1 > 0){

pid\_t tid1 = syscall(\_\_NR\_gettid);

printf("Error in thread %d\n", tid1);

printf("OS has run out of threads to allocate.\n");

printf("Please try less items.\n\n");

exit(0);

}

int thr2 = pthread\_create(&thread[1], NULL, mergesortThreaded, &arrInfo[1]);

if (thr2 > 0){

pid\_t tid2 = syscall(\_\_NR\_gettid);

printf("Error in thread %d\n", tid2);

printf("OS has run out of threads to allocate.\n");

printf("Please try less items.\n\n");

exit(0);

}

//---------------------------

// Wait for threads to finish

//---------------------------

pthread\_join(thread[0], NULL);

pthread\_join(thread[1], NULL);

merge(inputArr -> low, inputArr -> high);

pthread\_exit(NULL);

}

Lastly, we implemented a simple bubblesort algorithm to compare our mergesort implementations against. As always, the bubblesort algorithm implementation runs in O(n2) runtime complexity. The following was the implementation for bubblesort.

**Bubblesort:**

for (i = 0; i < numOfElem; ++i){

sharedMemArray -> sharedArray[i] = rand() % 100000;

}

printf("Testing bubble sort...");

start = clock();

for (i = 0; i < numOfElem; ++i){

for (j = 0; j < numOfElem - i - 1; ++j){

if (sharedMemArray->sharedArray [j] > sharedMemArray->sharedArray[j + 1]){

int swap = sharedMemArray->sharedArray[j];

sharedMemArray->sharedArray[j] = sharedMemArray->sharedArray[j + 1];

sharedMemArray->sharedArray[j + 1] = swap;

}

}

}

# RESULTS

The results were quite surprising. The following are the average runtimes for each mergesort implementation as well as our bubblesort implementation. The average runtimes were calculated by running three trials for every implementation against 100, 1000, 10000, and 100000 random items.

When running our analysis, the threaded mergesort implementation could not obtain the necessary threads needed to execute when the array went to a size larger than 1000 items.

The following are the results:

100 Items:

Recursive MergeSort: 0MS

Processes MergeSort: 1MS

Threaded MergeSort: 60MS

BubbleSort: 0MS

1000 Items:

Recursive MergeSort: 0MS

Processes MergeSort: 0MS

Threaded MergeSort: 290MS

BubbleSort: 10MS

10000 Items:

Recursive MergeSort: 0MS

Processes MergeSort: 0MS

Threaded MergeSort: N/A

BubbleSort: 650MS

100000 Items:

Recursive MergeSort: 5MS

Processes MergeSort: 0MS

Threaded MergeSort: N/A

BubbleSort: 41190MS

# CONCLUSION

When sorting 100, 1000, and 10000 random items, we observed that the threaded implementations of mergesort continuously run worse than the recursive method of mergesort. However, the multiprocessed mergesort implementation ran better than the recursive implementation.

These results were quite surprising but it makes sense. Here are the reasons why. The creating of threads incurs an operating system overhead due to traps being issued and interrupts being serviced. If we were to ignore the incurred overheads, the threaded implementations of mergesort would run faster than the multiprocesses and recursive implementation. However, when taking into account the overheads, threaded parallelism does not make sense for mergesort. This changes when looking at the multiprocess implementation.

The multiprocess mergesort’s runtime ran considerable faster than the recursive mergesort for all except once case. This case was the test of 100 items. The reason that the multiprocess implementation ran worse than the recursive implementation was due to the overhead mentioned above. The creation of processes incurs an overhead cost. For test cases of 1000 and 100000 items, the sorting process takes sufficient time to make the cost of creating children processes worth it. However, when there is a sufficiently small amount of items, the program spends more time creating processes than it would take to solve the problem recursively, thus the reason for a higher runtime.

Another interesting problem we encountered was a limit in threads. Since we were running our program on the C4 Linux machines, we had a limit of 1000 threads. Unfortunately, we could not increase the limit without sudo commands. Since we were running out of threads, we were limited to sorting 1000 items by our threaded mergesort implementation. As a result, our threaded mergesort implementation could not be run against a large enough dataset to make the overhead of threads worth it.

Overall, the additional overhead costs incurred by the creating of threads makes a parallel approach of mergesort through the creation of kernel threads impractical. However, with a sufficiently large number of items to store (roughly 1000 items from our tests), the creation of children process for parallel mergesort is not only practical but offers a small increase in performance.