Final Project

Parallel Merge Sort Using Shared Memory

Carlos Marquez

U69320592

marquezc@mail.usf.edu

James Keith

U01960154

keithj@mail.usf.edu

Stephen Astfanous

U82991394

sastfanous@mail.usf.edu

John Astfanous

U10888449

jastfanous@mail.usf.edu

Daniel Burkholder

U74377338

burkholderd@mail.usf.edu

***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:**

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

// Name: mergesortRecursive

// Purpose: Sorts items using the recursive

// mergesort approach.

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

void \* mergesortRecursive(void \*a){

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

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

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

// Split array in 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 O(n log n)

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

mergesortRecursive(&arrInfo[0]);

mergesortRecursive(&arrInfo[1]);

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

// Merge array O(n)

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

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:**

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

// Name: mergesortProcesses

// Purpose: Completes mergesort by continuously

// forking children.

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

void mergesortProcesses(void \*a){

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

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

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

// Split array in 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;

if (inputArr -> high - inputArr -> low > 5000000)

printf("Too many items to sort using processes.\n");

if ((inputArr -> high - inputArr -> low + 1) <= 10000)

mergesortRecursive(a);

else{

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

// Create the process calls

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

pid\_t pid1;

pid\_t pid2;

pid1 = fork();

if (pid1 < 0){

exit(-1);

}

else if (pid1 == 0){

mergesortProcesses(&arrInfo[0]);

exit(0);

}

else{

pid2 = fork();

if (pid2 < 0){

exit(-1);

}

else if (pid2 == 0){

mergesortProcesses(&arrInfo[1]);

exit(0);

}

}

int status;

waitpid(pid1, &status, 0);

waitpid(pid2, &status, 0);

merge(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:**

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

// Name: mergesortThreaded

// Purpose: uses parallelism to sort items

// each recursive call is implemented

// on a seperate kernel thread.

// If there are less than 10,000 items,

// switches to recursive approach

// to decrease overhead

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

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;

if ((inputArr -> high - inputArr -> low + 1) <= 10000 || threads >= 8){

mergesortRecursive(a);

}

else{

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

// Create the threads

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

pthread\_attr\_t attr;

pthread\_attr\_init (&attr);

pthread\_attr\_setscope(&attr, PTHREAD\_SCOPE\_SYSTEM);

int thr1 = pthread\_create(&thread[0], &attr, 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], &attr, 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);

}

threads += 2;

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

// Wait for threads to finish

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

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

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

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

}

}

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:**

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

// Bubble sort analysis

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

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

gettimeofday(&st, NULL);

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;

}

}

}

gettimeofday(&et, NULL);

isSorted(numOfElem);

printf("\nTime taken: %lu seconds, %lu microseconds\n\n", (et.tv\_sec - st.tv\_sec), (et.tv\_usec - st.tv\_usec));

# 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, 100000, 1000000, and 10000000 random items.

When we initially created our threaded and multiprocess mergesort implementations, we did not limit the creation of threads and processes. Thus, we would be creating N threads to sort N items, but the processor could only use 8 of these threads at a time. Thus, we limited the creation of threads to 8, increasing the work done by each thread.

Another thing we discovered was that the overhead for creating threads does not make sense if the thread will be working on small data sets, such as arrays of less than 10000 items. Thus, if a thread has 10000 items, it does not create threads to sort it, but instead uses the recursive algorithm. This increases the work done by each thread.

The following are the results. As stated, all datasets of less than 10000 items uses recursive mergesort:

100 Items:

Recursive MergeSort: 11 microseconds

Processes MergeSort: 11 microseconds

Threaded MergeSort: 11 microseconds

BubbleSort: 22 microseconds

1000 Items:

Recursive MergeSort: 128 microseconds

Processes MergeSort: 128 microseconds

Threaded MergeSort: 128 microseconds

BubbleSort: 1772 microseconds

10000 Items:

Recursive MergeSort: 1602 microseconds

Processes MergeSort: 1722 microseconds

Threaded MergeSort: 1607 microseconds

BubbleSort: 230310 microseconds

100000 Items:

Recursive MergeSort: 19195 microseconds

Processes MergeSort: 9687 microseconds

Threaded MergeSort: 6344 microseconds

BubbleSort: 26 seconds, 130822 microseconds

1000000 Items:

Recursive MergeSort: 221112 microseconds

Processes MergeSort: 64516 microseconds

Threaded MergeSort: 63553 microseconds

BubbleSort: N/A

10000000 Items:

Recursive MergeSort: 3 seconds, 1.8 x 1019 microseconds

Processes MergeSort: Not enough processes

Threaded MergeSort: 1 second, 1.8 x 1019 microseconds

BubbleSort: N/A

# CONCLUSION

When sorting 100, 1000, 100000, 1000000, and 10000000 items, we observe that threaded and multiprocess mergesorts run continuously better than the recursive implementation of mergesort.

These results were exactly as we expected. Mergesort is a divide and conquer approach to sorting items. However, only one CPU core is utilized with a recursive divide and conquer approach. This is because there is only ever one process or thread running at a time. Thus, the recursive algorithm will sort all of the items on the left half of the array iteratively and then sort the items on the right half. Once the sort is done, it will merge the items together, keeping the items sorted. However, when we introduce processes and threads, we are no longer limited to working on one half at a time.

When using processes or threads, each half of the array is sorted concurrently by a thread or process. This allows for the sorting of both the left and the right halves to occur at the same time, reducing runtimes.

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 or processes. Unfortunately, we could not increase the limit without sudo commands. Since we would quickly run out of processes when doing a multiprocess approach, the multiprocess mergesort could only sort an array of 5000000 items. This can be seen in the results.

Overall, the additional overhead cost incurred by the creating of threads and processes are almost entirely offset by the speed increase of the parallel mergesort. As can be seen in the test, the threaded and child process mergesort run in almost the same runtime. This is due to the fact that Linux employs copy-on-write and one-to-one threading. Thus, creation of processes and threads are essentially the same since the child processes are not writing to anything but shared memory.