CP 331 Assignment 2: Parallel Merge
Group 2

Hilts, Vaughan
hilt2740@mylaurier.ca, 120892740

Morouney, Robert
moro1422@mylaurier.ca, 069001422

Rusu, David
rusu0260@mylaurier.ca, 131920260

February 2^{nd} 2016

Contents

0.1 Naive Implementation

This was our first approach at solving the problem. The goal was to get something working fast.

generate 2 sorted lists of size N_a, and N_b in two blocks of contiguous memory.

Each processor p calculates it's block of A to merge with the equations:

$$i_{a,start,p} = p \left\lfloor \frac{N_a}{P} \right\rfloor + \min(p, N_a \text{mod} P) \tag{1}$$

$$n_{a,p} = \left\lfloor \frac{N_a}{P} \right\rfloor + \begin{cases} 1 & p < N_a \text{mod} P \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Where $i_{a,\text{start},p}$ is the start index of it's block into list A, and $n_{a,p}$ is the length of the block.
Next process p calculates the block of B to merge with the block in A.

$$i_{b,\text{start},p} = \max\{x \in \mathbb{N} : B_x < A_{i_{a,\text{start},p}-1}\} + 1 \tag{3}$$

$$n_{b,p} = \max\{x \in \mathbb{N} : B_x < A_{i_{a,\text{start},p}+n_{a,p}-1}\} + 1 - i_{b,\text{start},p} \tag{4}$$

We then merge these two lists with a sequential merge

This works well for small lists of sizes smaller than 2^{32} elements but becomes unwieldily with larger lists, since allocating this much contiguous memory will quickly become not practical.

0.1.1 Results

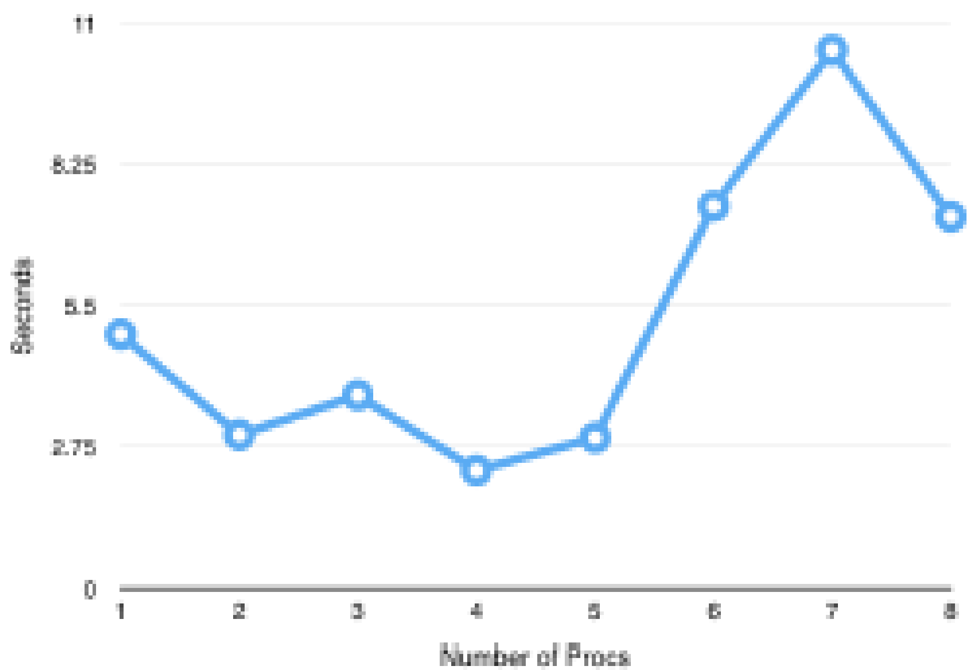

Figure 1. Performance on 10^8 size lists

0.2 Virtual Memory Implementation

To solve the problem of allocating contiguous memory, we instead allocate a few pages of memory and and build some abstractions on top of them to give the impression of contiguous memory.

This still allows us to have fast random access into memory but without the requirement of large contiguous blocks of memory. The rest of the algorithm is exactly the same as in our naive approach.

0.2.1 Results

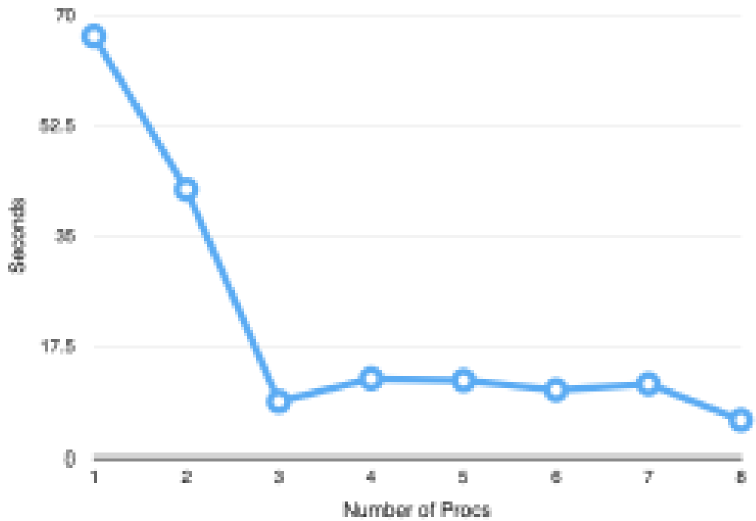

Figure 2. Performance on 10^8 size lists

0.3 Binary Search

The linear search in the naive approach of the algorithm is suitable for most array sizes when the set is quite small. However, there are some cases where replacing it with a binary search can be more effective. Specifically, a linear search is used to find the last index of an element that is smaller (when creating partitions of the lists). The results of this are summarized below:

0.3.1 Distances are fairly equal between gaps

When the number of partitions and the gaps between the partitions are fairly equal, the binary search performs at an acceptable level. There is a small, but not significant speed up (since only about half the time is spent in this search ? and this is serially, so the less the better). However, this is one of the few cases where the speed up is able to make a difference. When we use Linear Search, the search will remain fairly constant and the CPU cache is able to accurately predict what to store. When we do the same thing with binary search with consistent chucks, we see results that are similar. This would seem reasonable, considering linear search is on the order of and binary search is logarithmic in comparison. However, in reality there are some other factors that make this not always true?

0.3.2 When gaps are not equal

If your data source gaps are not neatly compacted or the data source has suffered from abnormal randomly, you may have runs where the binary search is significantly slower. Binary search incurs some overhead from its use which is often neglected when studying from a purely Big-O standpoint. There are two things to consider when comparing it with linear search

- In Linear Search, the CPU cache is on your side. The processor has already decided to keep the memory chunks ahead of you in memory and that means access to them will be quick. In contrast, Binary Search does not have this advantage as it may be jumping around quite a few points before settling down on a range.
- There is purely computational overhead in Binary Search. Branching is often neglected and other minor things in Big-O notation which become relevant here.

So, these two things help us draw a conclusion after many benchmarks.

1. If the number of iterations required for linear search is relatively low (10,000 comparisons is a low-figure that is fair from testing), then it can actually be faster to use it instead of Binary Search, despite Binary Search being able to do this theoretically quicker. We have the CPU Cache to blame for this.
2. The other interesting thing to note however, is the implication of a parallelization this would have if it was extended to be parallelized somehow. As the processor count goes up, the effectiveness of a binary search decreases, as the range clearly shrinks. To combat this problem, further heuristics which could select between the two algorithms would be ideal.

0.4 Regarding heuristics for algorithms

A good technique for doing this often deployed in High Performance Computing which allows code to be portable across many platforms is pre-benchmarking. In a pre-benchmarking situation, the software will run a set of tests on various input sizes on the test data and measure the time required for each. Each test is an algorithm or combination of algorithms ? which together make up a strategy. Then, the real execution uses the best strategy to execute on the full set of data. Since certain strategies work on different sets of data and hardware differently, having a few different ones and automatic selection is a big boon. This applies with the linear vs. binary situation as well, in these cases they are both strategies.

To maximize gains in these situations, it would be possible to setup a pre benchmark for these two different strategies and at the start of the launch, pick the best one for the current situation

0.5 Pooled Memory Implementation

Merging 2 ordered lists is a fairly simple task with contiguous lists as you can see above. The difficulty comes when trying to merge 2 lists so large that the resulting list is larger than the heap size of the system they are accessed on. This problem has implications in almost every field that uses “Big Data” and finding a solution to this problem was a vital step in completing this assignment.

To solve this we implemented a linked list of buckets where each bucket held a slice of lists A and B which will be merged. With this we can control the the size of each allocated block of contiguous memory. To take this a step further we aligned memory on a 32byte boundary each time it was allocated. Because we targeted the Intel Xeon processors on the ORCA cluster of SHARCNET we knew that each processor has a 64 byte cache line size. By investigating further we found that 32 bytes of this are reserved for instructions. By aligning allocated memory on a 32 byte boundary and controlling the size of each bucket to make sure it is divisible by 32 we limited potential L1 cache misses.

This allowed us to achieve the maximum speed when generating the lists. Since the list generation is non-trivial and only needed to be done by one processor we decided for the work division to use $2^n + 1$ cores. This allowed the master processor to spend time generating the lists then sending them out then receiving them and doing the final compilation. Using this scheme we were able to maintain a processor utilization of above 98% at all times during the run.

Since we used a linked list of buckets, dividing the tasks to the remaining processors was a trivial task. Each processor receives buckets in order and splits their lists recursively into individual elements then merges up to a final list. This final list is then sent back to the master processor which outputs the final result to a file. The result is not as fast as the naive solution however it is far more scaleable when using large sets of data.

0.5.1 The Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <mpi.h>
#include <time.h>

///DEFINITIONS
#define NUM_ELEMENTS     (1<<32) // 2^32
#define MASTER           0

/// If the size of a pointer is 4 bytes (32bit arch)
/// then to align the sizeof(bucket) on a 2^n boundry
/// we must reserve enough space for the pointer to the
/// next bucket.  If it is 8 we have to do a bit more work

#if __SIZEOF_POINTER__ == 4
#define BUCKETSIZE (1 << 4)
#elif   __SIZEOF_POINTER__ == 8
#define BUCKETSIZE (1 << 5)
#else
#error  " [ - ] sizeof(void*) is FUCKED"
#endif

///TYPE DEFININTIONS
typedef uint32_t    u32;
typedef uint64_t    u64;

//
//EXPLINATION:
//    u32 = 4 bytes
//    struct bucket = 4 or 8 bytes
```

```
//    data[BUCKETSIZE].a = 4 * BUCKETSIZE BYTES
//    "               " .b = "                     "
//
//    to align on 2^n boundry to maximize storage
//    u32 _ must be added when there is an 8 byte
//    pointer (64 bit arch) this adds an extra 4
//    bytes of padding to keep the sturct aligned in RAM
//

typedef struct bucket {
    struct       bucket    *next;
    u32          count;
#if __SIZEOF_POINTER__ == 8
    u32          _;
#endif
    struct {
                 u32 a;  // list a
                 u32 b;  // list b
    }data[BUCKETSIZE];

} bucket;

#define newbucket() do { uintptr_t *start; bucket *b; \
                         b = get_mem((u32)sizeof(bucket)); \
                         b->next = master_list; b->count = 0; \
                         master_list = b; \
                         total_buckets++; }while(0)

void *get_mem(u32 amt);
u32 *merge(u32 *sub_list, u32 block_size, u32 *sub_list2, u32 block_size2);
void split_and_merge(u32 *sub_list, u32 min, u32 max);

int main(int argc, char **argv)
{

    if (MPI_Init(&argc,&argv) != MPI_SUCCESS) {
        printf(" [ - ]  Filled to initialize MPI...Exiting");
        exit(-1);
    };

    int id, procs;
    int TAG_A = 1;
    int TAG_B = 2;
    int TAG_C = 3;
    MPI_Status  status;

    MPI_Comm_rank(MPI_COMM_WORLD,&id);
    MPI_Comm_size(MPI_COMM_WORLD,&procs);

    double wall_time1 = MPI_Wtime();

    // count the total numbers generated and placd in buckets as well as the
    u64 count = 0;

    // total number of buckets which have been created.
    u32 total_buckets = 0;

    u32   buckets_per_proc = NUM_ELEMENTS/BUCKETSIZE/procs + 1;
```

```
    if (id == 0)// MASTER
    {
        bucket              *master_list;
        uintptr_t           *start_addr;

        master_list = NULL;
        newbucket();

        printf(" [ + ] NOTE - sizeof(void*) = %d\n", __SIZEOF_POINTER__);
        printf(" [ + ] BUCKETSIZE = %'d\n",BUCKETSIZE);
        printf(" [ ? ] NUM_ELEMENTS = %u\n", NUM_ELEMENTS);
        srandom(time(NULL));


        double start = clock();
        while(count < NUM_ELEMENTS)
        {
            if(master_list->count >= BUCKETSIZE)
                newbucket();

            u32 i = master_list->count++;

            /// ALL THE REQUIREMENTS WERE FOR RANDOM!!! AHAHAHA
            u32 a = NUM_ELEMENTS - (count + (random() % 2)),
                b = NUM_ELEMENTS - count;

            master_list->data[i].a = a;
            master_list->data[i].b = b;

            count++;
        }

        double serial_time = clock() - start;

        printf(" [ + ]  Generated %lu sorted random numbers in 2 lists\n", count*2);

        printf(" [ + ]  This took a total of %.5fseconds\n",serial_time/
CLOCKS_PER_SEC);

        printf(" [ + ] Total number of buckets generated = %u\n",total_buckets );
        buckets_per_proc = total_buckets/(procs - 1);

        printf(" [ + ] Total number of buckets per processor = %u\n",
buckets_per_proc);

        /// SEND BUCKETS TO MPI COMM WORLD
        MPI_Bcast(&buckets_per_proc, 1, MPI_INT, MASTER, MPI_COMM_WORLD);

        u32 p;
        u32 *temp_mem_a, *temp_mem_b;

        temp_mem_a = malloc(BUCKETSIZE * sizeof(u32));
        temp_mem_b = malloc(BUCKETSIZE * sizeof(u32));
        for( p = 1; p<procs; p++)
        {

            if (temp_mem_a == NULL || temp_mem_b == NULL){
```

```
                fprintf(stderr, " [ -  ] Error getting memory for temp_lists\n");
                exit(-1);
            }
            u32 j;

            for ( j = 0; j < buckets_per_proc; j++ ) {
                u32 i = 0;

                for ( i = 0; i < BUCKETSIZE; i++){
                    temp_mem_a[i] = master_list->data[i].a;
                    temp_mem_b[i] = master_list->data[i].b;

                }

                // SEND THE LISTS TO PROC P
                MPI_Send(temp_mem_a,BUCKETSIZE,MPI_INT,p,TAG_A,MPI_COMM_WORLD);
                MPI_Send(temp_mem_b,BUCKETSIZE,MPI_INT,p,TAG_B,MPI_COMM_WORLD);

                if(master_list->next != NULL)
                    master_list = master_list->next;

            }

            printf(" [ + ]  Buckets sent to Proc %d\n", p);

        }

        free(temp_mem_a);
        free(temp_mem_b);
        free(master_list);


    } else// SLAVE
    {
        u32     *list_a, *list_b, *list_c, *list_t;

        // THIS IS CUTE... IT PAUSES THE SLAVES UNTIL THEY HEAR FROM THEIR MASTER
        MPI_Bcast(&buckets_per_proc, 1, MPI_INT, MASTER, MPI_COMM_WORLD);

        u32 min = NUM_ELEMENTS, max = 0;

        list_a = malloc(BUCKETSIZE*buckets_per_proc * sizeof(u32));
        list_b = malloc(BUCKETSIZE*buckets_per_proc * sizeof(u32));
        list_c = malloc(2 * buckets_per_proc * BUCKETSIZE * sizeof(u32));
        u32 i;
        u32 a=0, b=0;
        for(i = 0; i < buckets_per_proc; i++)
        {
            list_t = malloc(BUCKETSIZE * sizeof(u32));
            u32 m;
            MPI_Recv(list_t,BUCKETSIZE,MPI_INT,MASTER,TAG_A,MPI_COMM_WORLD,&status);
            split_and_merge(list_t,0,BUCKETSIZE-1);
            for (m=0;m<BUCKETSIZE;m++){
                list_a[a++] = list_t[m];
            }

            MPI_Recv(list_t,BUCKETSIZE,MPI_INT,MASTER,TAG_B,MPI_COMM_WORLD,&status);
```

```
            split_and_merge(list_t,0,BUCKETSIZE-1);
            for(m=0;m<BUCKETSIZE;m++){
                list_b[b++] = list_t[m];
            }

            free(list_t);
        }

        ////////////////////////////////////////////////////
        //                                                //
                list_c = merge(list_a, a, list_b, b);
        //                                                //
        ////////////////////////////////////////////////////
        // IN THE MEAN TIME HERE'S A PICTURE OF SEXY A DUCK
        //                    ,-.
        //                ,--' ~.).
        //              ,'         `.
        //             ; (((__   __)))
        //             ;  ( (#) ( (#)
        //             |   \_/___ \_/|
        //             ,"  ,-'    `__".
        //             (   ( ._   ____`.)--._        _
        //              `._ `-.`-' \(`-'  _  `-. _,-' `-/`.
        //               ,')   `.`._))  ,' `.   `.  ,','  ;
        //              .'  .     `--'  /     ).   `.      ;
        //             ;     `-        /     '  )         ;
        //             \                       ')       ,'
        //             \                     ,'       ;
        //             \        .      `~~~'       ,'
        //              `.       `-                _,'
        //                `.                   ,--'
        //                  `-.___________,--'
        //
        //
        //     QUACK!
        ////////////////////////////////////////////////////


        MPI_Send(list_c,a+b,MPI_INT,MASTER,TAG_C,MPI_COMM_WORLD);
        printf(" [ + ] Proccessor %d is sending its merged list to MASTER\n",id);
        free(list_a);
        free(list_b);
        free(list_c);

    }

    if (id == 0) // MASTER
    {
        FILE* f;
        f = fopen("c.list","w");

        u32 p;
        for( p=1; p<procs; p++ )
        {
            u32 *list_c;
            u32 c_size = 2 * buckets_per_proc * BUCKETSIZE;
            list_c = malloc(c_size * sizeof(u32));
```

```c

            MPI_Recv(list_c, c_size, MPI_INT,p,TAG_C,MPI_COMM_WORLD, &status);
            printf(" [ + ]  MASTER recieved Buckets sent by Proc %d\n", p);
            u32 i;

            if(list_c == NULL){
                fprintf(stderr, " [ - ] Error receiving list!\n");
                exit(-1);
            }

            printf(" [ + ] PRINTING TO FILE!\n");
            for ( i = 0; i < c_size; i++){
                fprintf(f,"%u\n",list_c[i]);
            }
            free(list_c);

        }
        fclose(f);
        double wall_time2 = MPI_Wtime();

        printf(" [ + ] Program took %.5f seconds to merge %lu elements\n",
                wall_time2 - wall_time1,(u64) NUM_ELEMENTS*2);
        printf(" [ + ] Program finished successfully!\n");
    }

    MPI_Finalize();

    return 0;
}

/// get memory aligned to 32byte boundry.  Intel Xeon processors on the ORCA cluster
/// have a cache-line size of 64bytes.  32 bytes are for program memory and 32 bytes
/// are for instructions.  By aligning the memory on a 32byte boundry it makes it
/// easier for the processor to split and process the chunks of allocated memory.
void *get_mem(u32 amt)
{
    uintptr_t   mem_start;

    mem_start = (uintptr_t)malloc((size_t)(amt + 31));
    if((void *)mem_start == NULL)
    {
        fprintf(stderr," [ - ]  Failed to Allocated memory in get_mem function after
    \
                allocating %fMB on the heap\n",
                (double)total_buckets*sizeof(bucket)/1024.0/1024.0);
        exit(-1);
    }
    // accidentally had this aligned on a 33 byte boundry and it had a 2 - 3x slow
    // down depending on the node
    mem_start = (mem_start + 31) & ~(uintptr_t)31;
    return (void *)mem_start;
}

 /// merge the two lists while keeping the integrety of the order
u32 *merge(u32 *sub_list, u32 block_size, u32 *sub_list2, u32 block_size2)
{
    u32 i1, i2, merged_index;

```

```c
    u32 *merged_list;
    u32 merged_size = block_size + block_size2;

    i1 = 0;
    i2 = 0;
    merged_index = 0;

    merged_list = (u32 *)malloc(merged_size*sizeof(u32));

    while ((i1 < block_size) && (i2 < block_size2)) {
        if (sub_list[i1] <= sub_list2[i2]) {
            merged_list[merged_index] = sub_list[i1];
            merged_index++; i1++;
        } else {
            merged_list[merged_index] = sub_list2[i2];
            merged_index++; i2++;
        }
    }

    u32 i;
    if (i1 >= block_size)
        for (i = merged_index; i < merged_size; i++, i2++)
            merged_list[i] = sub_list2[i2];
    else if (i2 >= block_size2)
        for (i = merged_index; i < merged_size; i++, i1++)
            merged_list[i] = sub_list[i1];

    for (i = 0; i < block_size; i++)
        sub_list[i] = merged_list[i];
    for (i = 0; i < block_size2; i++)
        sub_list2[i] = merged_list[block_size+i];

    return merged_list;
}


void split_and_merge(u32 *sub_list, u32 min, u32 max)
{
    u32 *_; // stupid recursion

    u32 middle = (min+max)/2,
        lowerCount = middle - min + 1,
        upperCount = max - middle;

  // check if sub_list is already sorted
  if (max == min) {
    return;
  } else {
    //sort first half of sub_list
    split_and_merge(sub_list, min, middle);
    //sort second half of sub_list
    split_and_merge(sub_list, middle+1, max);
    /* Now merge the two halves */
    _ = merge(sub_list + min, lowerCount, sub_list + middle + 1, upperCount);
  }
}
```

Listing 1: Parallel Merge