CS 267: Homework 3

Richard Barnes, Danny Broberg, Jiayuan Chen

April 1, 2016

1 Introduction

This report describes our implementation of Unified Parallel C (UPC) for de novo genome DNA assembly. UPC is an extension of the C language, outfitted for a Partitioned Global Address Space (PGAS) programming model. This model has the advantage of a shared address space for all the processors, but each variable is physically associated with a specific processor.

Our paper is organized as follows:

1. description of the computational resources
2. a section that describes the general serial algorithm approach
3. a section that describes the general extension to UPC
4. evaluation of the code's performance

2 Machine Description

For our codes we use NERSC's Edison machine. The machine has 5,576 compute nodes. Each node has 64GB DDR3 1866 MHz RAM and two sockets, each of which is populated with a 12-core Intel "Ivy Bridge" processor running at 2.4 GHz. Each core has one or two user threads, a 256 bit vector unit, and is nominally capable of 19.2 Gflops. Notably, for our purposes, each core has its own L1 and L2 cache. The L1 cache has 64 KB (32 KB instruction cache, 32 KB data) and the L2 cache has 256 KB. The 12 cores collectively share a 30 MB L3 cache. The caches have a bandwidth of 100, 40, and 23 Gbyte/s, respectively.[1] Both the L1 and L2 caches are 8-way and have a 64 byte line size.[2]

3 Generalized Serial Approach

Before coding up the genome assembly in parallel, we will first describe the approach for shotgun de nova genome assembly with a serial algorithm for comparison with setting up a UPC algorithm. Algorithm 1 shows a cleaned up version of the main bulk of the code while Algorithms 2- 4 show the code used to define packing routines and structures for the kmers. Overall one can see that there are three parts to the code: (1) Initialization of kmer values and hash table memory, (2) De Bruijn Graph Construction, (3) De Bruijn Graph Traversal.

[1] http://www.nersc.gov/users/computational-systems/edison/configuration/
[2] http://www.7-cpu.com/cpu/IvyBridge.html

The code takes the input to be a set of unique k-mers along with their corresponding forward and backward extensions. To denote start/terminating kmer for a contig, there is a possibility of an extension being a "guard" extension F which is incorporated into the hash table. We assume that the unique kmers have already been screened for errors so that the first step of the algorithm is to read in the unique values and construct a hash table with keys equal to the kmer and values equal to the forward and backward extensions. As shown in Algorithm 1 the hash table is initialized with pre-allocated memory. Within the graph construction step, the hash table is populated and a Start list is created to denote which Kmers will begin a contig.

After the hash table has been populated, the Graph Traversal step begins by looping through the kmer Start list, which provides a "seed" for the contigs that are created. Each traversal seed is added onto by appending the the forward extension to the last n-1 k-mer values (where n is the kmer length defined in 4) and referring to the hash table for the next forward extension. This process is continued until a "guard" extension is reached, at which point the contig is finished and printed to an external file.

Within the serial code, many external functions and variables are referenced which were defined in Algorithms 2- 4. These are displayed below the serial implementation code.

Algorithm 1: Serial Implementation (C)

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/time.h>
#include <math.h>
#include <time.h>
#include "packingDNAseq.h"
#include "kmer_hash.h"

int main(int argc, char **argv) {
   /* define current contig and extension variables, input kmer files */
   char cur_contig[MAXIMUM_CONTIG_SIZE]
   char unpackedKmer[KMER_LENGTH+1]
   char left_ext, right_ext, *input_UFX_name;
   int64_t contigID = 0, totBases = 0, ptr = 0;
   int64_t posInContig, nKmers, cur_chars_read, total_chars_to_read;
   kmer_t *cur_kmer_ptr; /* (defined in Algorithm 4) */
   start_kmer_t *startKmersList = NULL, *curStartNode; /* (defined in Algorithm 4) */
   unsigned char *working_buffer;

   /* ============================================== */
   /* ============== GRAPH CONSTRUCTION ============== */
   /* ============================================== */

   /* Read in input file input_UFX_name */

   /* Initialize lookup table that will be used for the DNA packing routines */
```

```
28      init_LookupTable();  /* (defined in Algorithm 2) */
29
30      /* Extract the number of k-mers in the input file */
31      nKmers = getNumKmersInUFX(input_UFX_name); /* (defined in Algorithm 4) */
32      hash_table_t *hashtable; /* (defined in Algorithm 4) */
33      memory_heap_t memory_heap; /* (defined in Algorithm 4) */
34
35      /* Create a hash table */
36      hashtable = create_hash_table(nKmers, &memory_heap);  /* (defined in Algorithm 3) */
37      /* Read the kmers from the input file and store them in the working_buffer */
38      total_chars_to_read = nKmers * LINE_SIZE;
39      working_buffer = (unsigned char*) malloc(total_chars_to_read * sizeof(unsigned char));
40      cur_chars_read = fread(working_buffer, sizeof(unsigned char),
41                                                  total_chars_to_read , inputFile);
42
43      /* Process the working_buffer and store the k-mers in the hash table */
44      while (ptr < cur_chars_read) {
45         /* Add k-mer to hash table */
46         add_kmer(hashtable, &memory_heap, &working_buffer[ptr],
47                                    left_ext, right_ext);   /* (defined in Algorithm 3) */
48
49         /* Create a list with the "guard" kmers as left (backward) extension */
50         if (left_ext == 'F')
51            addKmerToStartList(&memory_heap, &startKmersList);/*(defined in Algorithm 3)*/
52
53         ptr += LINE_SIZE; /*Iterate to next start k-mer in working buffer */
54      }
55
56
57      /* ============================================= */
58      /* ============== GRAPH TRAVERSAL ============== */
59      /* ============================================= */
60
61      curStartNode = startKmersList;
62      while (curStartNode != NULL ) {
63         /* unpack the current seed kmer*/
64         cur_kmer_ptr = curStartNode->kmerPtr;
65         unpackSequence((unsigned char*) cur_kmer_ptr->kmer,
66                                    (unsigned char*) unpackedKmer, KMER_LENGTH);
67
68         /* Initialize current contig with the seed content */
69         memcpy(cur_contig ,unpackedKmer, KMER_LENGTH * sizeof(char));
70         posInContig = KMER_LENGTH;
71         right_ext = cur_kmer_ptr->r_ext;
72         /* Keep adding bases while not finding a terminal node */
73         while (right_ext != 'F') {
74            cur_contig[posInContig] = right_ext;
```

```
          posInContig++;
          cur_kmer_ptr = lookup_kmer(hashtable, (const unsigned char *)
&cur_contig[posInContig-KMER_LENGTH]); /* (defined in Algorithm 3) */
          right_ext = cur_kmer_ptr->r_ext;
       }

       /* Print the contig since we have found the corresponding terminal node */
       cur_contig[posInContig] = '\0';
       fprintf(serialOutputFile,"%s\n", cur_contig);
       contigID++;
       totBases += strlen(cur_contig);
       /* Move to the next start node in the list */
       curStartNode = curStartNode->next;
    }

    return 0;
}
```

Algorithm 1: Serial Implementation (C)

Algorithm 2: DNA packing routine (Fortran)

```
#ifndef PACKING_DNA_SEQ_H
#define PACKING_DNA_SEQ_H

#include <math.h>
#include <assert.h>
#include <string.h>

unsigned int packedCodeToFourMer[256];

#define pow4(a) (1<<((a)<<1))

void init_LookupTable()
{
        // Work with 4-mers for the moment to have small lookup tables
        int merLen = 4, i, slot, valInSlot;
        unsigned char mer[4];
        for ( i = 0; i < 256; i++ ) {
                // convert a packedcode to a 4-mer
                int remainder = i;
                int pos = 0;
                for( slot = merLen-1; slot >= 0; slot-- ) {
                        valInSlot = remainder / pow4(slot);
                        char base;
```

```
                        if( valInSlot == 0 ) { base = 'A'; }
                        else if( valInSlot == 1 ) { base = 'C'; }
                        else if( valInSlot == 2 ) { base = 'G'; }
                        else if( valInSlot == 3 ) { base = 'T'; }
                        else{ assert( 0 ); }
                        mer[pos] = base;
                        pos++;
         remainder -= valInSlot * pow4(slot);
                }
                unsigned int *merAsUInt = (unsigned int*) mer;
                packedCodeToFourMer[i] = (unsigned int) (*merAsUInt);
        }
}

unsigned char convertFourMerToPackedCode(unsigned char *fourMer)
{
        int retval = 0;
        int code, i;
        int pow = 64;
        for ( i=0; i < 4; i++) {
                char base = fourMer[i];
                switch ( base ) {
                        case 'A':
                                code = 0;
                                break;
                        case 'C':
                                code = 1;
                                break;
                        case 'G':
                                code = 2;
                                break;
                        case 'T':
                                code = 3;
                                break;
                }
                retval += code * pow;
                pow /= 4;
        }
        return ((unsigned char) retval);
}

void packSequence(const unsigned char *seq_to_pack, unsigned char *m_data, int m_len)
{
    /* The pointer to m_data points to the result of the packing */
    int ind, j = 0;     // coordinate along unpacked string ( matches with m_len )
    int i = 0;     // coordinate along packed string
```

```
    // do the leading seq in blocks of 4
    for ( ; j <= m_len - 4; i++, j+=4 )
                m_data[i] = convertFourMerToPackedCode( ( unsigned char * ) ( seq_to_pack + j )) ;
    // last block is special case if m_len % 4 != 0: append "A"s as filler
    int remainder = m_len % 4;
    unsigned char blockSeq[5] = "AAAA";
    for(ind = 0; ind < remainder; ind++)
                blockSeq[ind] = seq_to_pick[j + ind];
    m_data[i] = convertFourMerToPackedCode(blockSeq);
}

void unpackSequence(const unsigned char *seq_to_unpack, unsigned char *unpacked_seq, int kmer_len)
{
        /* Result string is pointer unpacked_seq */
        int i = 0, j = 0;
           int packed_len = (kmer_len+3)/4;
        for( ; i < packed_len ; i++, j += 4 )
                *( ( unsigned int * )( unpacked_seq + j ) ) = packedCodeToFourMer[ seq_to_unpack[i] ];
        *(unpacked_seq + kmer_len) = '\0';
}

int comparePackedSeq(const unsigned char *seq1, const unsigned char *seq2, int seq_len)
{
        return memcmp(seq1, seq2, seq_len);
}

#endif // PACKING_DNA_SEQ_H
```

Algorithm 2: DNA packing routine (Fortran)

Algorithm 3: k-mer hash table routine (Fortran)

```
#ifndef KMER_HASH_H
#define KMER_HASH_H

#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <math.h>
#include <string.h>
#include "contig_generation.h"

/* Creates a hash table and (pre)allocates memory for the memory heap */
hash_table_t* create_hash_table(int64_t nEntries, memory_heap_t *memory_heap)
```

```
{
   hash_table_t *result;
   int64_t n_buckets = nEntries * LOAD_FACTOR;

   result = (hash_table_t*) malloc(sizeof(hash_table_t));
   result->size = n_buckets;
   result->table = (bucket_t*) calloc(n_buckets , sizeof(bucket_t));

   memory_heap->heap = (kmer_t *) malloc(nEntries * sizeof(kmer_t));
   memory_heap->posInHeap = 0;
   return result;
}

/* Auxiliary function for computing hash values */
int64_t hashseq(int64_t  hashtable_size, char *seq, int size)
{
   unsigned long hashval;
   hashval = 5381;
   for(int i = 0; i < size; i++)
      hashval = seq[i] +  (hashval << 5) + hashval;

   return hashval % hashtable_size;
}

/* Returns the hash value of a kmer */
int64_t hashkmer(int64_t  hashtable_size, char *seq)
   return hashseq(hashtable_size, seq, KMER_PACKED_LENGTH);

/* Looks up a kmer in the hash table and returns a pointer to that entry */
kmer_t* lookup_kmer(hash_table_t *hashtable, const unsigned char *kmer)
{
   char packedKmer[KMER_PACKED_LENGTH];
   packSequence(kmer, (unsigned char*) packedKmer, KMER_LENGTH);
   int64_t hashval = hashkmer(hashtable->size, (char*) packedKmer);
   bucket_t cur_bucket;
   kmer_t *result;

   cur_bucket = hashtable->table[hashval];
   result = cur_bucket.head;

   for (; result!=NULL; ) {
      if ( memcmp(packedKmer, result->kmer, KMER_PACKED_LENGTH * sizeof(char)) == 0 )
         return result;
      result = result->next;
   }
   return NULL;
```

```
}

/* Adds a kmer and its extensions in the hash table (note that a memory heap should be preallocated. ) */
int add_kmer(hash_table_t *hashtable, memory_heap_t *memory_heap, const unsigned char *kmer, char left_ext,
{
    /* Pack a k-mer sequence appropriately */
    char packedKmer[KMER_PACKED_LENGTH];
    packSequence(kmer, (unsigned char*) packedKmer, KMER_LENGTH);
    int64_t hashval = hashkmer(hashtable->size, (char*) packedKmer);
    int64_t pos = memory_heap->posInHeap;

    /* Add the contents to the appropriate kmer struct in the heap */
    memcpy((memory_heap->heap[pos]).kmer, packedKmer, KMER_PACKED_LENGTH * sizeof(char));
    (memory_heap->heap[pos]).l_ext = left_ext;
    (memory_heap->heap[pos]).r_ext = right_ext;

    /* Fix the next pointer to point to the appropriate kmer struct */
    (memory_heap->heap[pos]).next = hashtable->table[hashval].head;
    /* Fix the head pointer of the appropriate bucket to point to the current kmer */
    hashtable->table[hashval].head = &(memory_heap->heap[pos]);

    /* Increase the heap pointer */
    memory_heap->posInHeap++;

    return 0;
}

/* Adds a k-mer in the start list by using the memory heap (the k-mer was "just added" in the memory heap at
void addKmerToStartList(memory_heap_t *memory_heap, start_kmer_t **startKmersList)
{
    start_kmer_t *new_entry;
    kmer_t *ptrToKmer;

    int64_t prevPosInHeap = memory_heap->posInHeap - 1;
    ptrToKmer = &(memory_heap->heap[prevPosInHeap]);
    new_entry = (start_kmer_t*) malloc(sizeof(start_kmer_t));
    new_entry->next = (*startKmersList);
    new_entry->kmerPtr = ptrToKmer;
    (*startKmersList) = new_entry;
}

/* Deallocation functions */
int dealloc_heap(memory_heap_t *memory_heap)
{
    free(memory_heap->heap);
    return 0;
```

```
}

int dealloc_hashtable(hash_table_t *hashtable)
{
   free(hashtable->table);
   return 0;
}


#endif // KMER_HASH_H
```

Algorithm 3: k-mer hash table routine (Fortran)

Algorithm 4: Generation of contigs routine (Fortran)

```
#ifndef CONTIG_GENERATION_H
#define CONTIG_GENERATION_H

#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/stat.h>
#include <math.h>
#include <string.h>

#ifndef MAXIMUM_CONTIG_SIZE
#define MAXIMUM_CONTIG_SIZE 100000
#endif

#ifndef KMER_LENGTH
#define KMER_LENGTH 19
#endif

#ifndef LOAD_FACTOR
#define LOAD_FACTOR 1
#endif

#ifndef LINE_SIZE
#define LINE_SIZE (KMER_LENGTH+4)
#endif

static double gettime(void) {
    struct timeval tv;
    if (gettimeofday(&tv, NULL)) {
```

```
30          perror("gettimeofday");
31          abort();
32      }
33      return ((double)tv.tv_sec) + tv.tv_usec/1000000.0;
34  }
35
36  /* K-mer data structure */
37  typedef struct kmer_t kmer_t;
38  struct kmer_t{
39      char kmer[KMER_PACKED_LENGTH];
40      char l_ext;
41      char r_ext;
42      kmer_t *next;
43  };
44
45  /* Start k-mer data structure */
46  typedef struct start_kmer_t start_kmer_t;
47  struct start_kmer_t{
48      kmer_t *kmerPtr;
49      start_kmer_t *next;
50  };
51
52  /* Bucket data structure */
53  typedef struct bucket_t bucket_t;
54  struct bucket_t{
55      kmer_t *head;          // Pointer to the first entry of that bucket
56  };
57
58  /* Hash table data structure */
59  typedef struct hash_table_t hash_table_t;
60  struct hash_table_t {
61      int64_t size;           // Size of the hash table
62      bucket_t *table;                        // Entries of the hash table are pointers to buckets
63  };
64
65  /* Memory heap data structure */
66  typedef struct memory_heap_t memory_heap_t;
67  struct memory_heap_t {
68      kmer_t *heap;
69      int64_t posInHeap;
70  };
71
72  /* Returns the number of UFX kmers in a file */
73  int64_t getNumKmersInUFX(const char *filename) {
74      FILE *f = fopen(filename, "r");
75      if (f == NULL) {
76          fprintf(stderr, "Could not open %s for reading!\n", filename);
```

```
77          return -1;
78       }
79       char firstLine[ LINE_SIZE+1 ];
80       firstLine[LINE_SIZE] = '\0';
81       if (fread(firstLine, sizeof(char), LINE_SIZE, f) != LINE_SIZE) {
82          fprintf(stderr, "Could not read %d bytes!\n", LINE_SIZE);
83          return -2;
84       }
85       // check structure and size of kmer is correct!
86       if (firstLine[LINE_SIZE] != '\0') {
87          fprintf(stderr, "UFX text file is an unexpected line length for kmer length %d\n", KMER_LENGTH);
88          return -3;
89       }
90       if (firstLine[KMER_LENGTH] != ' ' && firstLine[KMER_LENGTH] != '\t') {
91          fprintf(stderr, "Unexpected format for firstLine '%s'\n", firstLine);
92          return -4;
93       }
94
95       struct stat buf;
96       int fd = fileno(f);
97       if (fstat(fd, &buf) != 0) {
98          fprintf(stderr, "Could not stat %s\n", filename);
99          return -5;
100      }
101      int64_t totalSize = buf.st_size;
102      if (totalSize % LINE_SIZE != 0) {
103         fprintf(stderr, "UFX file is not a multiple of %d bytes for kmer length %d\n", LINE_SIZE, KMER_LENGTH)
104         return -6;
105      }
106      fclose(f);
107      int64_t numKmers = totalSize / LINE_SIZE;
108      printf("Detected %lld kmers in text UFX file: %s\n", numKmers, filename);
109      return numKmers;
110   }
111
112   #endif // CONTIG_GENERATION_H
```

Algorithm 4: Generation of contigs routine (Fortran)

4 UPC approach

Our extension to the UPC langauge follows the general outline of the serial code...

5 Performance

6 Summary