CS 267: Homework 3

Richard Barnes, Danny Broberg, Jiayuan Chen

1 Introduction

This report describes our implementation of a de novo genome DNA assembly program using Unified Parallel C (UPC). UPC is an extension of the C language, outfitted for a Partitioned Global Address Space (PGAS) programming model. This model associates memory physically with specific processes but enables shared access. This bears syntatic similarities to OpenMP and other shared memory programming models.

Our paper is organized as follows:

1. description of the computational resources
2. a section that describes the general algorithm
3. a section that describes the major design choices and optimizations
4. notes on MPI and its connection to UPC
5. 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 The Algorithm

Sequencing long strands of DNA is difficult and time-consuming. For strands beyond a certain length it may not even be technically feasible. Therefore, faster and more scalable methods have been developed. Shotgun de novo genome assembly breaks a long piece of DNA into many shorter segments. These segments can be easily sequenced.

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

The problem then is to reassemble the original DNA by fitting the pieces together. In the simplified data set given here, the data takes the form of contiguous sequences of 19 or 51 bases (henceforth, a "kmer") with known prefixes and suffixes. We are guaranteed that each kmer is unique. Together, the kmers a De Bruijn Graph.

Our program will construct this graph and traverse its connected components. To do this, a hash table will be used to store nodes of the graph with hashed kmers acting as keys. The values of the hash table will correspond to the prefix and suffixes and, thereby, serve as edges linking the nodes together.

Our algorithm begins by detecting how many kmers are in the input. It then allocates a shared memory hashtable three times larger than this number. Members of the hashtable are stored via open addressing with linear probing: this is wasteful of memory, but greatly simplifies the data structure, thereby reducing the potential for errors. Using a hashtable of size $3N$ ensures that occupancy stays well below 70%, a point at which performance has been empirically shown to degrade significantly.

Kmers are hashed using Dan Bernstein's `djb2` hash function. Tests using the small data set with Austin Appleby's `MurmurHash3` didn't show a significant reduction in collisions. Since kmer elements are drawn from a library of four symbols, only 2-bits of information are required to represent each symbol. Therefore, as kmers are read they are transformed to this more compact representation.

Additionally, a local array of pointers to the hash table is allocated. As the process reads kmers and adds them to the hash table, it makes a note of which kmers have prefixes indicating that they begin sequences and uses this local array to store that information. These starting-kmers (smers) form the starting points of the graph traversals which generated the contiguous (output) sequences. One might worry that storing smers locally would result in them being unevenly distributed; however, this is not a problem if we assume (a) that the smers are distributed evenly throughout the file and (b) that the smers, on average, are part of components of approximately equal size. These assumptions are reasonable because it is possible to randomize the order of the lines in the input file, which means that smers are evenly distributed and, therefore, component sizes should therefore be as well. For the purposes of the assignment, we do not randomize the input, but do verify that the assumption is at least nominally fulfilled by the test data provided as shown in Figures 1 and 2.

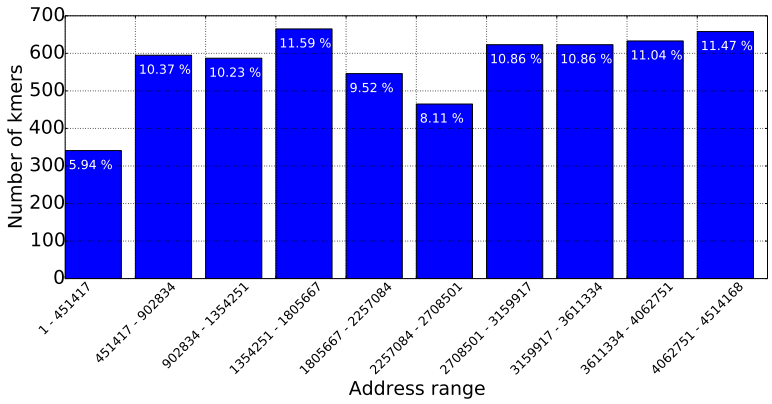

Figure 1: Distribution of start kmers to address space (Small data set)

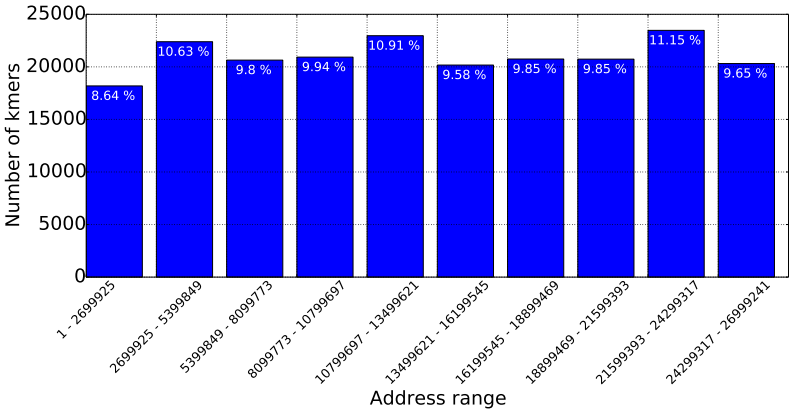

Figure 2: Distribution of start kmers to address space (Long data set)

Since the size of the file and the number of threads/processes can be determined at run-time, it is possible to assign threads to read separate, non-overlapping blocks of lines from the file. Each process begins reading and, as it does so, adding kmers.

In our algorithm, adding a kmer is a two-step process. If a kmer hashes to an address owned by the process which read it, then the process saves the kmer to that address immediately. If, however, the kmer hashes to an address which is not owned by the process, then the process adds the kmer to a linked list in its private local storage: the kmer will be added to the shared space later. This scheme guarantees that kmers being added to the hashtable do not encounter race conditions.

After all of the kmers have been read in this way, the kmers stored in the linked lists must be added to the hashtable. But race conditions must be avoided. One way to do this is to lock addresses or, more practically, blocks of addresses. But locks can create performance bottlenecks and introduce the possibility of deadlocks.

Therefore, the hashtable is divided into evenly-sized and sequential blocks based on the number of threads. Each thread is given exclusive access to a block and deposits that block's associated kmers from its linked list. A barrier synchronizes the threads and then each thread does the same operation with the next block until each thread has seen every block.

This round robin distribution must be performed twice because kmers may hash to filled slots in the table and, in the course of linear probing, be slotted for insertion into an address outside the block the kmer's depositing thread currently has access to. However, twice is guaranteed to be sufficient because the 33% maximum occupancy of the hash table coupled with the uniform distribution of the hash function guarantees that there will be sufficient slots in each block such that no kmer will move across more than one boundary via linear probing. UPC's default strict memory operation ordering guarantees that this round robin approach will not have race conditions since each thread has exclusive access to the memory it is manipulating and operations performed on that memory are guaranteed to be performed in sequential order.

As the kmers are inserted into the hashtable, they are removed from the linked list. It is therefore simple to check that all kmers have been added: the list will empty if this is the case. As the kmers are inserted into the hashtable, the smers are noted and their final addresses saved to their inserting thread's private local memory.

Once all of the kmers have been added to the hashtable, each thread iterates through its saved smers and traverses the table. No special synchronization is needed for this step because only read operations will take place.

Kmers are read from a striped file. Such a file is distributed into blocks across multiple hard drives, which can increase data throughput. We don't expect to see much benefit from this when running on a single node due to contention for that node's connection with memory; however, when running across several nodes there may be a large benefit.

For maximum benefit, each process should read memory from a separate part of the file, with no overlap. However, this is difficult to achieve if the number of processes might change. Therefore, the block size is chosen to be considerably smaller than the amount of data a process will read in. The effect is that the processes cycle between which drive they are getting data from. We do not attempt to synchronize this cycling and it is difficult to analyze theoretical (given potentially different read rates), but the ultimate effect is less contention than would be expected if everything were read from a single hard drive. For similar reasons, we write outputs to the scratch space for our tests.

4 Design Choices and Optimizations

We made three major design choices: (a) using open hashing with linear probing rather than separate chaining, (b) using round robin exclusive access rather than locks, (c) strip and memory blocking decisions.

The choice to use open hashing with linear probing (a) was motivated by a desire for a simple algorithmic design which avoided complex structures in shared memory. For good performance, this design requires a hash function which not only provides a uniform distribution, but also even spacing to minimize collisions. We did not verify that `djb2` provides this, but reduced fallout from a bad hash spacing by using a $3N$-sized hashtable.

Separate chaining guarantees immediate insertion and retrieval proportional to bin occupancy whereas open hashing provides insertion and retrievals which are both proportional to the number of adjacent occupied addresses. Therefore, separate chaining is likely to have faster insertion times. Again, using a sufficiently large hashtable, as we have, reduces this problem.

Therefore, we expect the only downside to our choice of open hashing to be the wasted memory. However, given the sizes of the datasets here, this is not problematic for us.

The choice to use round robin kmer insertion (b) rather than locks was also motivated by design simplicity. Shared memory structures and deadlock potential were entirely avoided by this design. We therefore expect that this choice should be both faster and safer than alternative designs. An earlier alternative design we tried was to use atomic conditional-updates to write to relevant memory locations; however, these accesses were implemented as simple locks and not available on all of the UPC compiles we used. Thus, they added complexity with few additional benefits.

Testing the foregoing assertions would require significant redesign of our algorithm, which we did not have time to do. Therefore, we rely on our arguments above as justification of the choice.

Our choice of file striping patterns and UPC memory blocking (c) provides another potential avenue for optimization. However, long queue times on Edison interfered with carrying out systematic tests.

5 What About MPI?

UPC was perhaps the largest design choice we made. (Of course, it wasn't really a choice because part of the assignment was to use it.) Using a two-sided communication paradigm, like MPI would

necessitate a few changes to our algorithm.

The simplest approach would be to have each thread allocate a portion of memory corresponding to part of the hashtable. As in our algorithm, each thread can then write to its portion of the hashtable and save kmers corresponding to other threads' address blocks in a list in its local memory. Since each thread knows which thread will ultimately own these saved kmers, the thread can send them directly to their owner using non-blocking I/O. Once they have all been sent, the thread can use blocking I/O to receive the additional kmers it owns from other threads. If this fills a message buffer, then the threads could alternate between send and receive modes several times if necessary.

Now each thread owns a part of the hashtable and a set of smers. Each thread will have approximately the same number of smers because the smers were evenly distributed in the input (see above). For each kmer, a thread sends a message to the node which owns that kmer's hash and receives back the value of the hash. Since each thread is doing this, threads must interleave work on their contigs with replies to incoming requests for values. Therefore, sends should be non-blocking and the thread should be able to receive messages whose tag corresponds to either to values being returned or values being requested.

Race conditions for data will not occur here because each thread always has exclusive access to its own memory. Separate chaining is useful in this instance because then thread which owns an address can always be accurate predicted (i.e. there is no need for value or hashtable-write requests to be forwarded on to another thread).

6 Results

Figure 3 shows the absolute run-times of our algorithm on the small dataset for varying numbers of threads. We also processed the small dataset across eight nodes with a much larger number of processors. As Figure 4 shows, using about eight threads on a single node is faster than the fastest time achieved running on many threads across many nodes. For this small dataset, the processing power of each thread is probably under-utilized and communication overhead likely dominates the time-to-completion. Therefore, we only analyze the small dataset in terms of running on a single node.

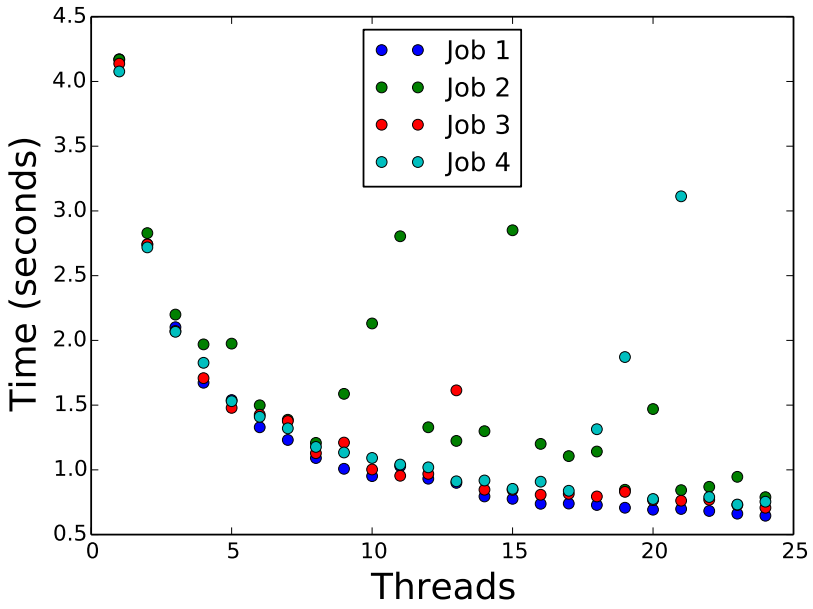

Figure 3: Absolute run-time versus threads (Small data set)

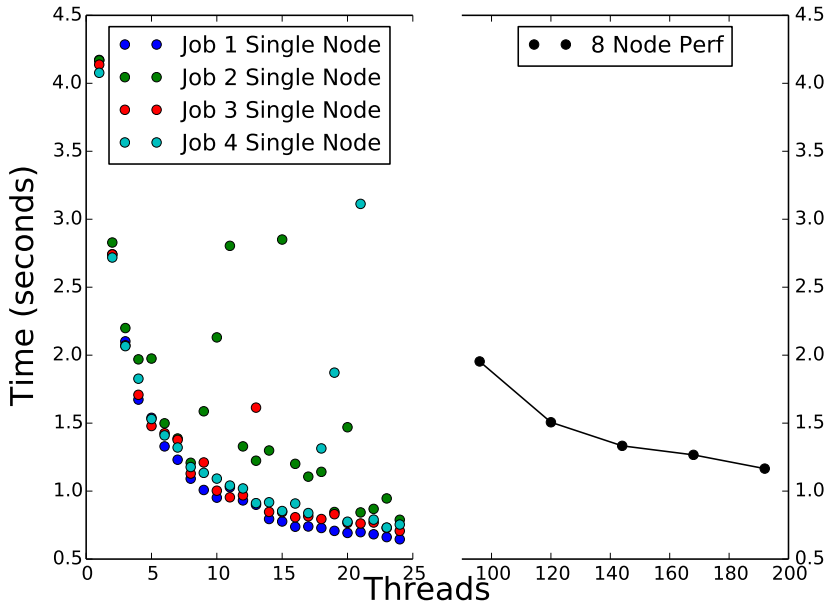

Figure 4: Distribution of start kmers to address space (Small data set)

Figure 5 shows how our algorithm scales on the small test verses idealized performance, as given by the formula $\frac{time_0/t}{time_t}$. The Ivy Bridge design links pairs of cores to shared memory buses. Given that our code is not cache-optimized, we expect minimal gains past 12 threads. Figure 3 supports this.

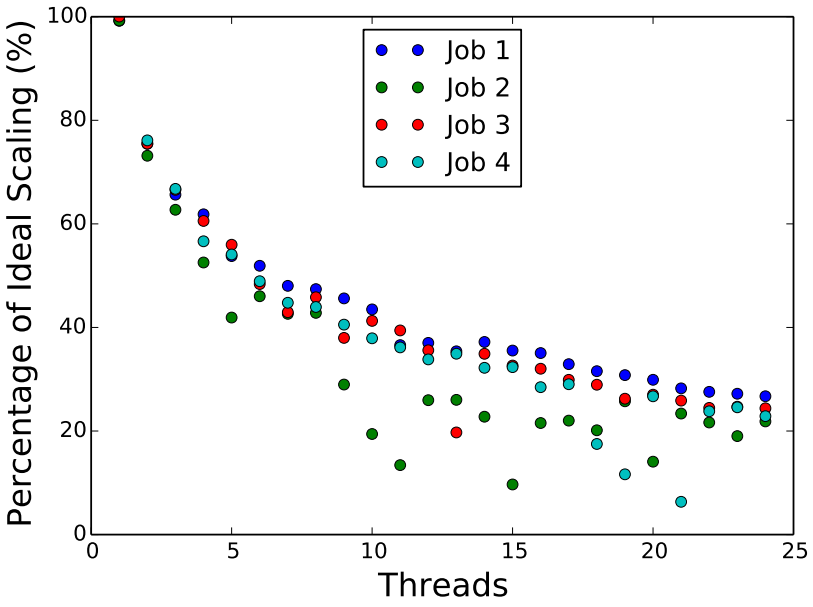

Figure 5: Scaling versus threads (Small data set)

Figure 6 shows absolute run-times for varying numbers of threads for the large dataset. Our algorithm took 30 seconds to perform this operation using a single thread, so we achieve a 4x speed-up for the largest numbers of threads used. However, Figure 7 shows that this represents abysmal scaling.

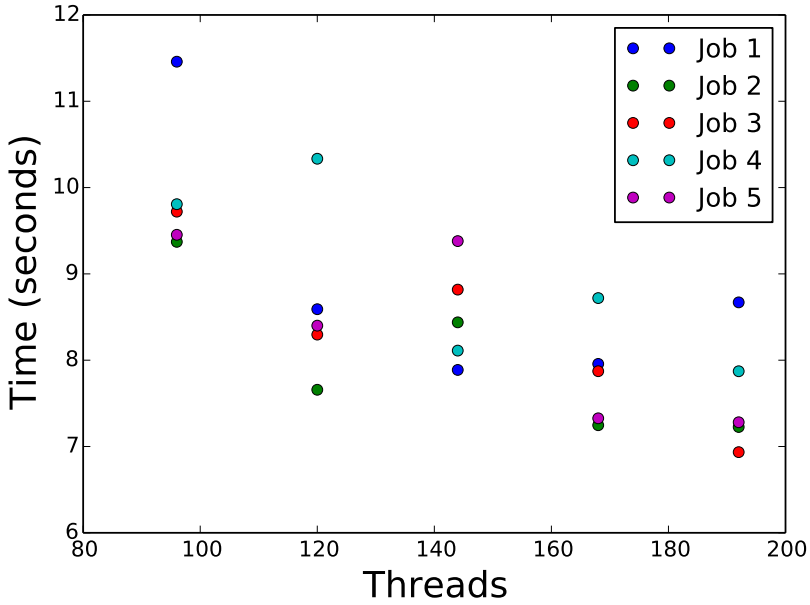

Figure 6: Absolute run-times versus threads (Large data set)

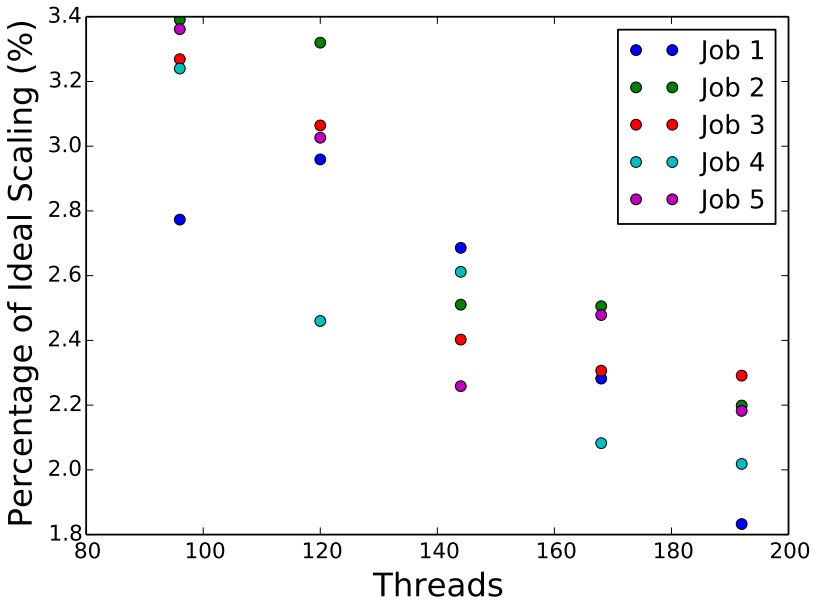

Figure 7: Scaling versus threads (Large data set)

We hypothesized that the poor scaling was due to the high volumes of internode communication necessary to explore the hashtable. To test this, we ran the big dataset on a single node with various numbers of threads. As shown in Figure 8 this achieves faster performance than dividing the work between eight nodes.

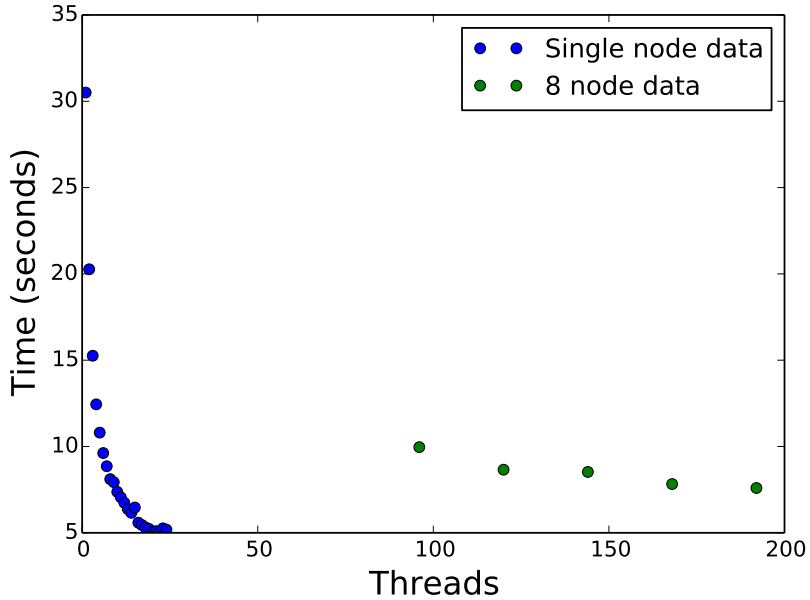

Figure 8: Absolute time versus threads including a single node (Large data set)

As Figure 9 shows, the run-times we see running on eight nodes seem to be part of the same scaling curve the single node experiences.

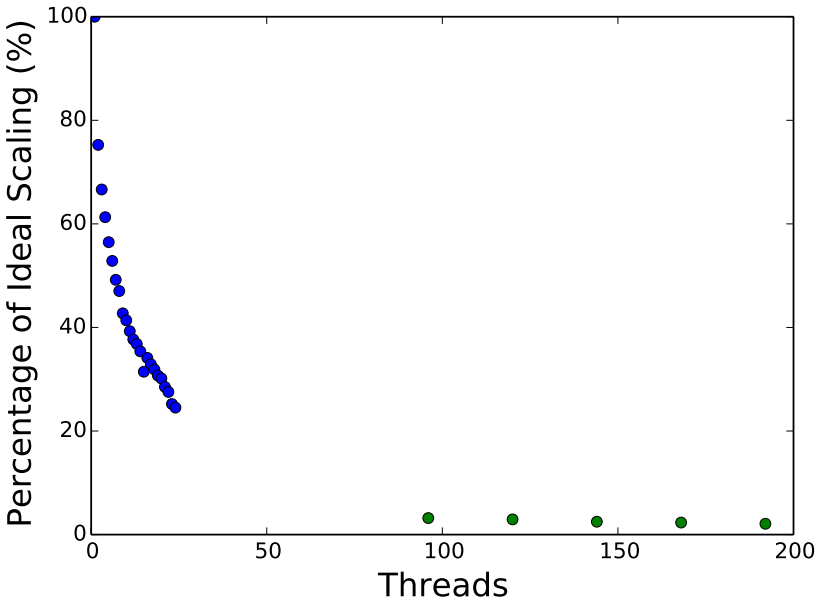

Figure 9: Scaling versus threads including a single node (Large data set)

This suggests that the scaling is somehow inherent to the algorithm. The computational motif most strongly associated with our algorithm is "graph traversal". The Berkeley View Dwarf Mine states that many current computer architectures fail to efficiently handle this motif because "most searching algorithms end up depending almost entirely on random memory access latency". In our case, accessing random memory means utilizing relatively expensive communication channels (especially expensive when these channels are between nodes), so it should not be surprising that additional processing power does not translate to significantly better runtimes: many of the threads are competing for access to memory! This is true even with striped file access due to L1 and L2 caches. UPC still seems like a good choice for this motif since the computation requires large memories and every other way we've covered for distributed memory would have even higher communication costs. Perhaps further progress would be made by having exceptionally light-weight processes run on all but the first node: these processes would do no computation, but allow access to their shared memory.

Figure 10 shows a break-down of the run-times of our algorithm for various numbers of threads showing the costs of both graph construction and traversal. As might be expected, construction takes longer than traversal, but the two follow similar curves. Therefore, optimizing construction is likely to bring gains regardless of communication details or other factors.

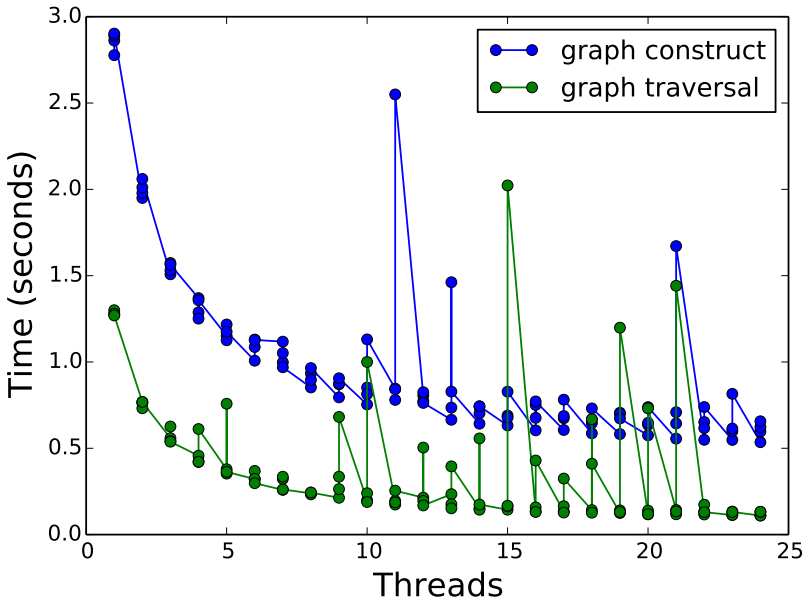

Figure 10: Absolute run-times versus threads for a single node showing break down by part of algorithm (Large data set)

Likewise, it's possible that optimizing for our algorithm's communication motif could be helpful. Our algorithm uses round robin node-to-node communication for building the hashtable and limited scatter operations for during memory allocation. Traversal is accomplished with essentially random node-to-node communication. Thus, a fully-connected (or close to) architecture is useful for us. Given the small numbers of nodes involved, this may be technically and monetarily feasible.