# Parallel Mapping Approaches for GNUMAP

Nathan L. Clement*, Mark J. Clement†, Quinn Snell† and W. Evan Johnson‡§
*Department of Computer Science
University of Texas at Austin, Austin, TX 78712
Email: nathanlclement@gmail.com
†Department of Computer Science
Brigham Young University, Provo, UT 84602
‡Department of Statistics
Brigham Young University, Provo, UT, 84602
§Department of Oncological Sciences, Huntsman Cancer Institute
University of Utah, Salt Lake City, UT, 84105

***Abstract*—Mapping short next-generation reads to reference genomes is an important element in SNP calling and expression studies. A major limitation to large-scale whole-genome mapping is the large memory requirements for the algorithm and the long run-time necessary for accurate studies. Several parallel implementations have been performed to distribute memory on different processors and to equally share the processing requirements. These approaches are compared with respect to their memory footprint, load balancing, and accuracy. When using MPI with multi-threading, linear speedup can be achieved for up to 256 processors.**

***Keywords*-next-generation sequencing; short-read mapping; sequence mappers; parallel computing; biology computing**

## I. Introduction

Over the past 30 years, Sanger-type sequencing [1] has been the standard technique for DNA sequencing. This approach has enabled, among other things, the sequencing of the first complete human genome sequence [2]. However, recent developments in sequencing technologies have led to a second generation of sequencing approaches, from Illumina/Solexa, ABI/SOLiD, 454/Roche, and Helicos, which currently produce gigabases sequence information during each instrument run.

Next generation sequencing technologies promise to revolutionize the field of biomedical research by producing large volumes of sequence data for a reasonable price. However, the size of the datasets generated are much larger and the diverse nature of the dataset produced novel statistical and computational challenges that must be overcome. One very important problem facing researchers today is the identification and characterization of single nucleotide polymorphisms (SNPs). Researchers are often interested in identifying SNPs that vary between one individual a reference genome, or in comparing the sequence composition of two individuals, strains or species at homologous or orthologous regions. Researchers are most often interested in associating these SNPs with disease or important phenotypes of interest, so the accurate identification of these SNPs is extremely important.

When identifying SNPs, short reads 35-100 base pairs in length are mapped to their best position in the human genome. See Figure 1 for an example mapping. A rigorous probabilistic approach to mapping repeat regions and reads with lower quality scores can result in a significantly larger number of mapped reads. This can lead to the identification of regions of interest on the genome that otherwise would have been overlooked—for example, mapping to the large number of repetitive genomic elements in mammalian genomes. This approach requires the algorithm to have the entire genome in memory so that a read can be proportionally mapped to all matching sites, and while it requires more memory and processing time, it results in a more accurate mapping [3].

In this paper, we explore several parallelization methods to the GNUMAP algorithm that use multiple processors and nodes to significantly decrease the overall time and memory requirements. Similar approaches have been taken with comparable mapping environments (sequences of longer length from the Roche 454 platform [4]) and mapping programs (BWA, SOAP2, and Bowtie [5]). Because of the increase attention to accuracy employed by the GNUMAP algorithm, there are additional challenges. In this paper we discuss several of these challenges, outlining those that have trivial solutions or minimal noticeable effects, and presenting some whose solutions are non-trivial or would require substantial negative changes to the algorithm.

The following section presents a detailed overview of the GNUMAP algorithm so that later descriptions of the parallelization will make sense.

## II. Algorithmic details

Many mapping algorithms discard reads from repeat regions and do not utilize quality scores once the base has been "called." GNUMAP provides a probabilistic approach that utilizes this additional information to provide more accurate results from fewer costly sequencing runs. Accurately mapping reads to repetitive genomic elements

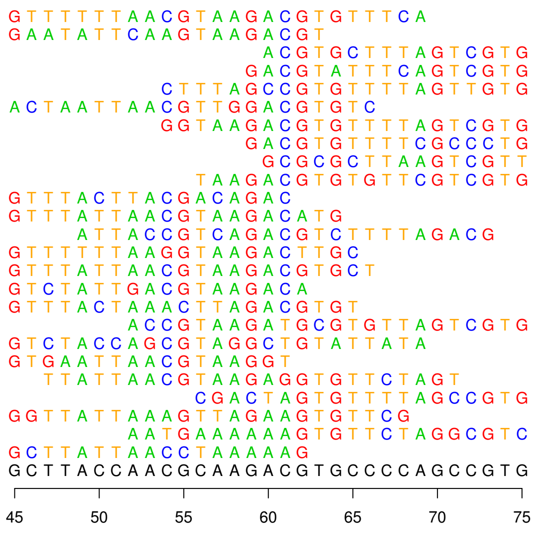

Figure 1. A representation of the final results from the sequence mapping process. Each of these 25 sequences were mapped to the reference genome, shown at the bottom. Notice the probable SNP at position 56, where most reads have a `T` instead of a `C`

is essential if next-generation sequencing is to be used to draw valid biological conclusions. For example, a ChIP-seq experiment attempts to accurately identify small DNA regions interacting with a protein of interest. Binding motifs often appear in or near the repeat regions [6] [7] that are ignored by some mapping algorithms. Other applications such as transcription mapping, alternative splicing analysis, and miRNA identification may also suffer from inaccuracies if repeat regions are ignored.

Several programs (RMAP[8], SeqMap[9], and ELAND) have attempted to significantly speed up this mapping process through creating a hash map to efficiently map reads to the genome. Reads are broken into short, 9–15 base pair segments and assigned a numerical value in the hash map according to their sequence. The genome is then scanned and the hashing function is used to find corresponding locations for the genomic sequences in the read hash table. The reads at these locations are then aligned with the genome until either a match is found or the alignment is deemed too insignificant to continue.

The GNUMAP algorithm effectively incorporates the base uncertainty of the reads into mapping analysis using a *Probabilistic Needleman-Wunsch* algorithm. The Probabilistic Needleman-Wunsch was developed to improve upon the common dynamic programming algorithm used for sequence alignment to accurately use reads with lower confidence values.

Care must be taken to develop an algorithm that can accurately map millions of reads to the genome in a reasonable amount of time. In the GNUMAP algorithm, the genome is first hashed and then stored in a lookup table rather than hashing the reads. This allows reads to be accounted for in all of the duplicate genome sites. Next, the reads are efficiently stored as a position-weight matrix so that quality scores can be used when aligning the read with genomic data. A Needleman-Wunsch alignment algorithm is modified to use these matrices to score and probabilistically align a read with the reference genome. Figure II is a flowchart which shows the major steps of the algorithm.

### *Step 1: Hashing and Storing the Genome*

Hashing a large portion of the data allows for quick data retrieval while still maintaining a reasonable amount of memory use. GNUMAP creates a hash table from the genome instead of the reads, allowing for the computation of a probabilistic scoring scheme.

The entire genome is hashed based upon either a user-supplied hash size or the default hash size of nine. A larger hash size will tolerate fewer mismatches. For example, in a 30bp read, a hash size of 9bp will guarantee that the read is matched to every possible location while still allowing for three mismatches. Larger hashes will require more memory, but will also reduce the search space. The amount of memory, $B$, required based on the number of bases in the genome, $s$, and the mer-size of the hash, $k$, can be computed as follows:

$$B = 4 * (4^k + s). \tag{1}$$

For example, for a genome ($s$) of 200,000bp and a mer-size ($k$) of 9, the total memory used ($B$) will be $4 * (4^9 + 200{,}000) \approx$ 2Mb of RAM.

### *Step 2: Processing the Reads*

One of the novel approaches implemented by GNUMAP lies in the data structure used for storing the reads. Instead of storing the reads as simple sequences, or even sequences with an attached probability as in the *FASTQ* format, each sequence is stored as a position-weight matrix (PWM) (see Table I for an example).

Raw data from the Solexa/Illumina platform are obtained as either an intensity file or a probability file. From either of these files, it is possible to compute a likelihood score for any nucleotide of any position on any given read. There will often be a lack of distinction between the most probable and another base, such as the ambiguity between $G$ and $T$ seen in position 3 of the PWM in Table I. Since each read is stored in memory as a position-weight matrix, the information included in each base call allows for the correct mapping of a given sequence. Converting these bases to a single probability score will result in the loss of information.

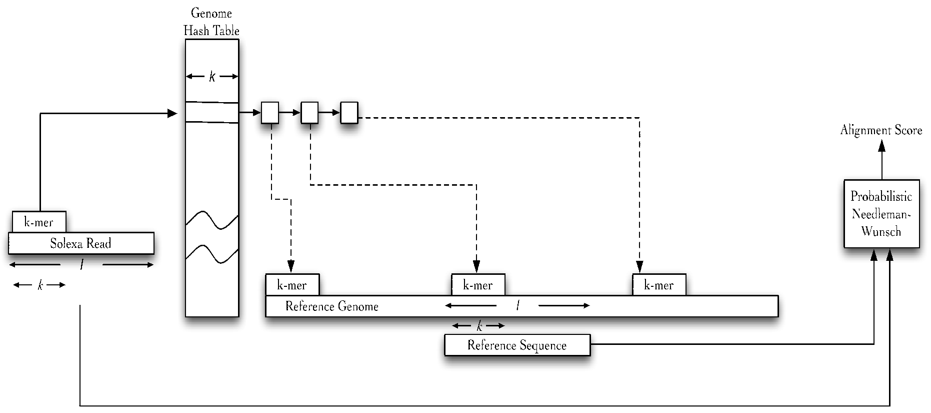

Figure 2. A flow-chart of the GNUMAP algorithm. First, the algorithm will incrementally find a $k$-mer piece in the consensus Solexa read. This $k$-mer is used as an index into the hash table, producing a list of positions in the genome with the exact $k$-mer sequence. These locations are expanded to align the same $l$ nucleotides from the read to the genomic location. If the alignment score passes the user-defined threshold, the location is considered a hit, and recorded on the genome for future output.

### *Step 3: Score Individual Matches*

In order to match the reads to the reference genome, the reads are first subjected to a quality filter, removing reads with too many unknown bases. In order to pass GNUMAP's quality filter, a sequence (stored as a position-weight matrix) must be able to obtain a positive score when aligned with its own consensus sequence (the sequence created from using only the most probable bases). Using this method, very few reads are discarded by the quality filter (usually only removing reads identified by the Solexa pipeline as having an intensity of zero at each base).

A sliding window of size $k$ is used to create a hash value which can be used to find matching positions in the reference genome. The matching genomic sequence is then aligned to the read using the probabilistic Needleman-Wunsch algorithm (see Table I).

The probabilistic Needleman-Wunsch score (PNWScore) for read $r$ and genomic sequence $S$ at position $i, j$ in the dynamic programming matrix $NW$ can be calculated as:

$$NW_{i,j} = max \begin{cases} NW_{i-1,j-1} + \sum_{k \in A,C,G,T} PWM_{k,j} * cost_{k,S_i} \\ NW_{i-1,j} + gapcost \\ NW_{i,j-1} + gapcost \end{cases} \tag{2}$$

given that $cost_{k,j}$ is the cost of aligning the character at position $r_j$ with the character $k$. For example, using the PWM in Table I, the calculation of the score for position 3,3 in the dynamic programming matrix would be:

$$max \begin{cases} 0.208 + 0.172 * -1 + 0.136 * 1 + 0.317 * -1 + 0.375 * -1 \\ -1.792 + -2 \\ -1.792 + -2 \end{cases} \tag{3}$$

(For this example, a match yields a cost of 1, a mismatch yields a cost of -1, and the cost for a gap is -2. This results in a cost of $-0.520$ which is stored at position 3,3 in the dynamic programming matrix $NW$.

Table I
DP Matrix for Probabilistic Needleman-Wunsch.

| j | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| PWM | $A$ | 0.059 | 0.000 | 0.172 | 0.271 | 0.300 |
| | $C$ | 0.108 | 0.320 | 0.136 | 0.209 | 0.330 |
| | $G$ | 0.305 | 0.317 | 0.317 | 0.164 | 0.045 |
| | $T$ | 0.526 | 0.578 | 0.375 | 0.356 | 0.325 |
| NW | | T | T | T | T | C |
| | 0 | -2 | -4 | -6 | -8 | -10 |
| T | -2 | **0.052** | -1.948 | -3.948 | -5.948 | –7.984 |
| T | -4 | -1.844 | **0.208** | -1.792 | -3.792 | -5.792 |
| C | -6 | -3.844 | -1.792 | $\underline{\textbf{-0.520}}$ | -2.448 | -4.448 |
| A | -8 | -5.844 | -3.792 | -2.374 | **-0.978** | -2.978 |
| C | -10 | -7.984 | -5.792 | -4.131 | -2.774 | **-1.318** |

Aligning the genomic sequence `TTCAC` and read `TTTTC`, with the optimal alignment shown in **bold**. Also notice the PWM for the sequence, with several fairly ambiguous positions (especially the final position, probably representing a `C`, even though the probabilities for the `C` and `T` are nearly equal).

### *Step 4: Processing Scores*

Once the read has been scored against all plausible matches in the genome, a proportional share of this read will be added to all the matching genomic locations . In order to compute the hit score at a position in the reference genome, a posterior probability for each read is computed. For a read $r$, the algorithm first finds the $n$ most plausible match locations on the genome, $M_1 \ldots M_n$. These matches are scored using the probabilistic Needleman-Wunch algorithm, to obtain the scores $Q_1 \ldots Q_n$. The value added to the genome $G$ for each read, $r$, obtained from each significant match location $M_j$, signified by $G_{M_j}$, will then be

$$G_{M_j} = \frac{Q_{M_j}}{n_{M_j} Q_{M_j} + \sum_{k \neq j}^{n} n_{M_k} Q_{M_k}}, \quad (4)$$

where $n_{M_k}$ is the number of times the sequence located at position $M_k$ appears in the genome.

When using this scoring method, the total score for each sequence at a particular site in the genome is weighted by its number of occurrences in the genome. If a given sequence occurs frequently, the value added to a particular matching site in the final output is down-weighted, removing the bias that would occur if the match was added to all repetitive regions in the genome. If, however, there are the same number of duplicate reads as the number of times the sequence is duplicated in the genome then a whole read will be added to each of the duplicate locations in the final output.

This scoring technique requires the hashing and storing of the genome instead of the set of reads. Because the score for a given read is not only calculated from its alignment score but also by the number of occurrences of *similar* regions in the genome, the genome must be scanned for each read to fairly allocate the read across all matching sites.

#### ***Step 5: Create Output***

After all the reads have been matched and scored on the genome, two output files are created. The first file is in SAM format and identifies the highest scoring match for each read. The second file contains the genome in .SGR or .SGREX (SGR-EXtended) format providing a genome-wide base-pair resolution overview of the mapping results, which can be viewed in the UCSC Genome Browser or Affymetrix's Integrated Genome Browser.

## III. Methods

In this section, two terms are used to identify different architectures. A *thread* refers to a lightweight process spawned within an instantiation of GNUMAP. A *node* executes a stand-alone instance of GNUMAP that has no shared memory with other instances.

Two major objects are routinely used by GNUMAP: the genome and the seed index table. The genome object includes both the compressed character reference genome and the final numerical mapping results at each genomic location. Once program setup has completed, the character representation and seed-index table are read-only, whereas the mapping results are read-write. On a single node, shared memory access to either of these objects reduces data duplication and memory requirements. Because of this, simultaneous genomic reads can be performed by any number of processors, but a locking mutex prevents more than one processor from performing writes.

### *A. Multi-Threading on a Single Machine*

When using multiple threads on a single node, GNUMAP employs a master-slave paradigm to efficiently handle a large number of reads. Each thread is assigned a constant number of reads from the sequence files. As a given thread finishes its current workload, it obtains additional reads from the sequence file until they have all been mapped. When matching genomic locations are found, the thread will access a shared genome object to store this information, and save the read for later writing. Periodically, a master thread will write this information to a file in SAM format.

Each step requires a certain amount of data sharing.

1) In the first step, care must be taken to avoid race conditions in mapping DNA sequences. To control this, one thread reads the data from the file and stores it to a global array. All other threads use a semaphore to determine which reads from the array they should map. This sequential component of the algorithm can limit the total speedup of the algorithm, but when each thread is assigned a large enough number of reads, this overhead is minimal.
2) In the second step of the computation, there is a race condition if two threads write to the same location in the genome object at the same time. If each read is assumed to come from an equally-likely random location in the genome, in theory there would not need to be any control structures surrounding genome writes. However, DNA from many next-generation sequencing experiments originate from only a few genomic locations, weakening this assumption. Moreover, as the genomic coverage in a specific location grows higher, there is a greater chance for multiple threads to be scoring reads at the same location. For these reasons, each write to the genome is surrounded by a locking mutex. Since the critical section for assigning a read score to the genome is small, threads do not spend a significant time waiting for genome availability.
3) Two different output types are produced by GNUMAP: an SGR (or SGREX—SGR-EXtended) file, providing a genome-wide base-pair resolution overview of the mapping results (such as SNP locations or regions with many matches), and a SAM file reporting matching locations for each read. The second step of this computation described above is used to create the SGR file, and the third to the SAM file, in the following manner. After positive matches are found, each thread stores information about this match to a global list. Periodically during the mapping process, a single thread will print all matches to an output file. (This is also critical when using MPI to

reduce the memory footprint, which will be explained in Section III-C.) To maintain data integrity, access to this list is limited to a single thread, requiring locking mutexes. Because it takes only a short amount of time to store these mapping results, the amount of time waiting for global list access is relatively short; however, as the number of threads increase, this access time can become a significant bottleneck (see Section IV-A for a discussion of how many threads is "too many").

### B. Employing MPI With Multi-Threading

When the entire Genome fits into the memory on a single node, a simplistic MPI approach can further take advantage of independence to significantly reduce the time of computation.[1]

With multi-threading enabled on a single machine, it is relatively simple to use MPI to split the work even further. GNUMAP uses the open source OpenMPI library to perform communication method calls, including starting individual processes and syncing information across nodes. In a similar manner to that taken by multi-threading on a single machine, the reads are split among nodes and mapped independently. To reduce the number of times synchronization must occur among nodes, each sequence file is split into equal portions, and the unique machine number assigned by OpenMPI is used to determine which section of the file should be mapped. In this manner, each node independently writes SAM records to a separate file, with no extra communication costs until the end. At the end of the mapping process, each node performs a global sum on the genome object, which is then printed out by a single node to a single file (along with any sort of additional analyses).

### C. Employing MPI to Reduce the Memory Footprint

The statistical rigor employed by GNUMAP limits the mechanisms that can be used to reduce the memory footprint. Each character in the genome can feasibly be reduced to 4 bits (four different nucleotides plus an ambiguity character), decreasing the memory requirement by half, and smart memory allocation can reduce the size of the $k$-mer lookup table to a reasonable size (around 12GB). Other algorithms have created structures that significantly reduce this memory footprint [10], [11], [12]; however, nothing can be done to compress the Genome object. To obtain the most accurate final result, the posterior likelihood score for each read mapped to the genome is added to the corresponding genomic location. This requires at least single floating-point precision, which is 4 bytes that cannot be compressed. Two methods to avoid this are either increasing the number of positions stored at a given memory cell ("binning"), or post-processing the reads in such a way that only a portion of the genome is in memory at a time. Distributing memory across MPI nodes can also reduce the memory footprint.

- **Binning** Internally, GNUMAP creates five floating point arrays to represent the four nucleotides—A, C, G, and T—and one ambiguity character. Determining locations that are different from the reference genome requires knowing what characters existed in the sequence data set at that location. If these are binned, then a sequence of A's, C's, G's, and T's would appear all as one nucleotide. In addition, GNUMAP uses a pairwise-Hidden Markov Model (pair-HMM) to determine the probability of each nucleotide matching to a specific location and being represented by a distinct character. Binning locations from these scores would discard the information obtained from a pair-HMM, resulting in an inability to call SNPs.
- **Post-Processing** Post-processing the reads requires only a portion of the genome to be in memory at a time, but this also has issues. GNUMAP's probabilistic algorithm assigns a score to all locations a given read could match. Repetitive genomic regions and reads with very similar alignment scores all need to be added to the genome, requiring many entries for a single read. Taking shortcuts to find an approximate answer would decrease the accuracy. Accuracy is one of the major motivating factors behind the creation of GNUMAP, so this option is also undesirable.
- **Memory Distribution** A third alternative is to only mildly compress the internal data structures, and instead spread the reference genome and corresponding memory accesses across independent nodes. While this effectively solves the problem of high memory requirements, maintaining consistency with the original GNUMAP algorithm requires additional communication costs. In order to compute a posterior alignment score consistent across the entire genome, the score from an individual genomic region must be combined with each other region. The specific algorithm GNUMAP uses is described below.

Utilizing MPI, GNUMAP uses the following large-memory algorithm for memory distribution (refer to Figure 3 for labels):

```
1     The MASTER_Node reads the genome and assigns
      each SLAVE_Node its portion of the Genome.
2     Each SLAVE_Node hashes and stores its portion
      of the Genome, allocating memory as if it were
      only mapping to this single piece of the Genome.
3     In order, each SLAVE_Node performs the mapping of $k$ reads (a subset of all those in a given
      run).
3.1   Each SLAVE_Node divides the $k$ reads among its
```

[1]The entire human genome together with lookup structures requires approximately 12GB RAM; when doing SNP analysis, this requirement increases to approximately 35GB.

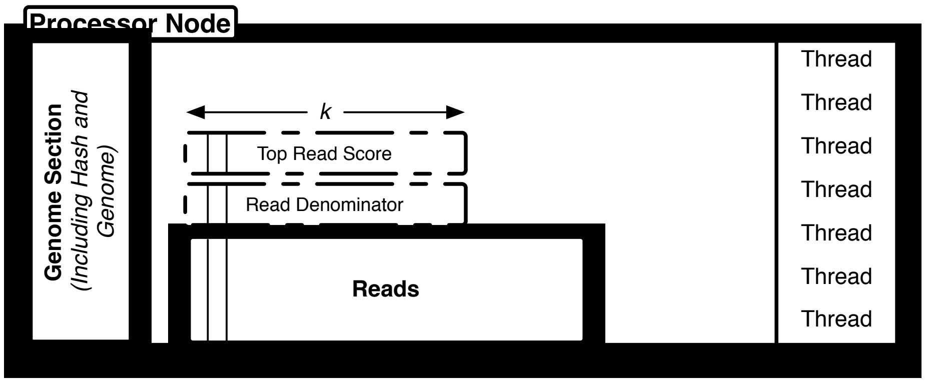

Figure 3. Pictorial representation of a single node of an MPI large-memory run. *k* reads are processed at a time, storing the needed information in the *Read Denominator* and *Top Read Score* arrays. Individual nodes will communicate information at synchronization points.

`Threads`, and performs the *Multi-Threading on a Single Machine* as described in Section III-A. The numerator for the final score for a read is the alignment score at that location divided by the number of times it appeared in the genome. The denominator is computed by summing together the alignment scores across all the "good" matching locations. (See [3] for a further description.) The denominator is saved in the *Read Denominator* array and the highest matching score in the *Top Read Score* array.

3.2 Each `SLAVE_Node` performs an `MPI_REDUCE_ALL` with a `SUM` on the *Read Denominator* for $k$ reads.

3.3 Each `SLAVE_Node` performs an `MPI_REDUCE_ALL` with a `MAX` on the *Top Read Score* for $k$ reads.

3.4 Each `SLAVE_Node` uses the *Read Denominator* to obtain the posterior score, and writes the SAM output to a unique file for each read which is equal to the value in *Top Read Score*.

4 Each `SLAVE_Node` performs any needed genomic analysis (such as SNP identification), and prints the Genome to separate *sgr* or *sgrex* file, thus reducing the need for a semaphore.

## IV. Results

To evaluate performance, a subset of a 32,000,000 read Illumina lane (short-read archive number NA20828, found at http://trace.ncbi.nlm.nih.gov/Traces/sra/sra.cgi?cmd=viewer&m=data&s=viewer&run=ERR005645) was mapped to the human genome. Each sequence from this data set are 54 bases in length, and are in *fastq* format (each nucleotide in the sequence has an associated probability). It is difficult to plot results for different parallelization approaches due to the different problem sizes that are needed to expose parallel artifacts. A comparison of these different problem sizes was performed by plotting a relative speedup for each algorithm. For the Multiple Processors algorithm, all threads run on a single 32-processor node. The Multiple Nodes and Large Memory experiments run on up to 32 nodes with 8 processors each for a total of 256 processors.

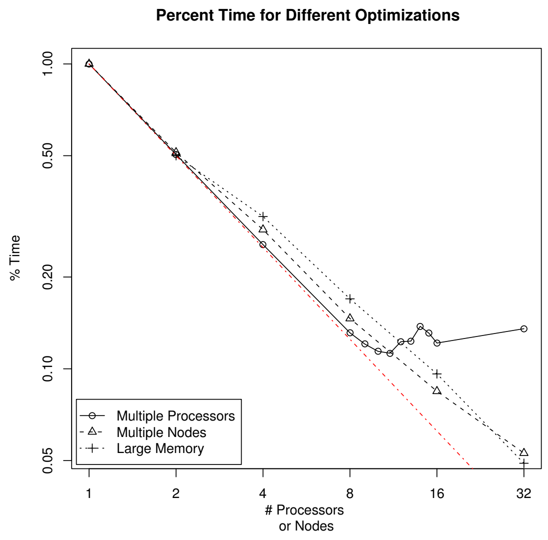

Figure 4. Plots for the relative speedups for different optimizations. Note that each axis is logged, so the straight line drawn in red shows perfect linear speedup. Splitting the reads across multiple nodes achieves near-linear speedup. Splitting the *memory* across multiple nodes can obtain nearly log-linear speedup. Because of the high cost of thread synchronization, linear speedup is not possible beyond 8 threads.

Figure 4 plots the relative percentage of time for different number of processors instead of the absolute time in order to compare the efficiency of all approaches fairly. Using Multiple Processors on a single node results in linear speedup as long as 8 or fewer processors are used. Beyond this, the synchronization costs overwhelm the parallel benefits.

When spreading the memory across multiple nodes—both with shared memory and without—nearly linear speedup can be achieved even with 32 nodes (256 processors). The Large Memory program seems to perform slightly worse, until the number of

### A. Multi-Threading on a Single Machine

To determine the cost of synchronizing multiple threads, we ran a test data set on a shared-memory machine with 32 processors[2]. Figure 4 shows an almost-perfectly linear speedup until GNUMAP tries to use more than 8 threads, after which the completion time does not decrease significantly. In this particular example, a single compute node using 8 threads mapped 600k reads to human chromosome 1 in 564 seconds. The time for 10 threads was 478 seconds.

These results can only be obtained when using the GNU `g++` compiler. When compiling with the `mpic++` compiler, using multiple threads in fact takes longer than a single thread. This is likely due to MPI communication libraries that prevent multiple threads from executing simultaneously. In Figure 5, the `mpic++` experiments were performed with a smaller problem size in order to diagnose the speedup limitation. This is why they appear to take a shorter amount of time.

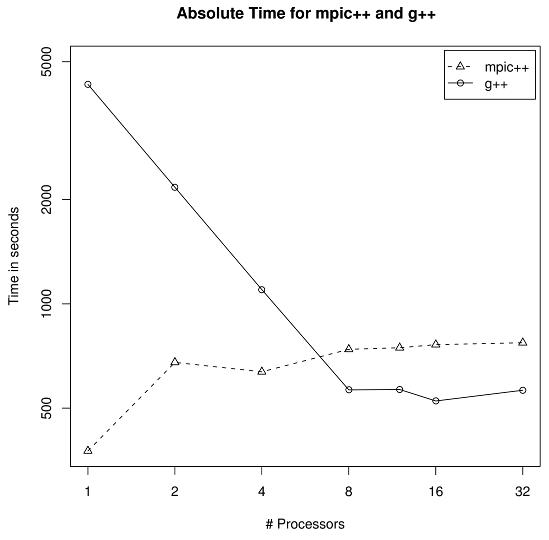

Figure 5. Comparison between mpic++ and g++. mpic++ does not allow more than one thread to be used, taking much longer if it does.

### B. MPI With Multi-Threading

Because of the minimal number of synchronizations required when spreading the reads across multiple nodes, approximately linear speedup can be seen with even 32 nodes (see Figure 6). For each of these runs, 1.2M sequences were aligned to the entire human genome. This took just under an hour on only one node, and only 178 seconds on 32 nodes (256 processors)[3]. Clearly, when the memory and hardware are available, this is the preferred option.

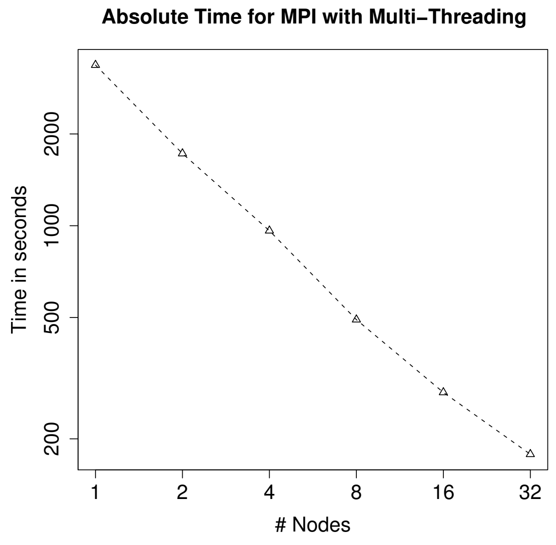

Figure 6. Absolute time for MPI with Multi-Threading. Linear speedup is achieved even with 32 nodes (256 processors).

*1) Memory Requirements:* With the default settings, GNUMAP requires 56 GB of RAM to map any number of sequences to the human genome. To perform SNP comparison, this increases to 100GB of RAM. Since many machines do not have this much memory, spreading the program across different nodes provides an obvious alternative.

### C. MPI to Reduce the Memory Footprint

Some installations may not have enough memory to fit the entire genome on a single node. If there are multiple nodes available, the genome can be divided into smaller sections and processed independently. With only a few points of communication, a global alignment score can be computed for each read, completing the mapping process without any reduction in accuracy.

In Figure 7, MPI was used to reduce the memory footprint with as many as 32 nodes. What took two processors nearly three days to complete could be finished in 6.6 hours with 32 nodes[4]. This was a much larger data set than used previously, and the entire genome wouldn't fit onto a single machine. As can be seen in Figure 7, the program achieves nearly-linear speedup as the workload is balanced.

The biggest cost for this type of optimization is in the synchronization after each portion of reads. One of the obvious

[2]Intel Nehalem EX clocked at 1.86 GHz with 256 GB RAM

[3]Two Quad-core Intel Nehalem processors per node, clocked at 2.8 Ghz with 24GB RAM and 4x DDR Infiniband connection

[4]Two Quad-core Intel Nehalem processors per node, clocked at 2.8 Ghz with 24GB RAM and 4x DDR Infiniband connection

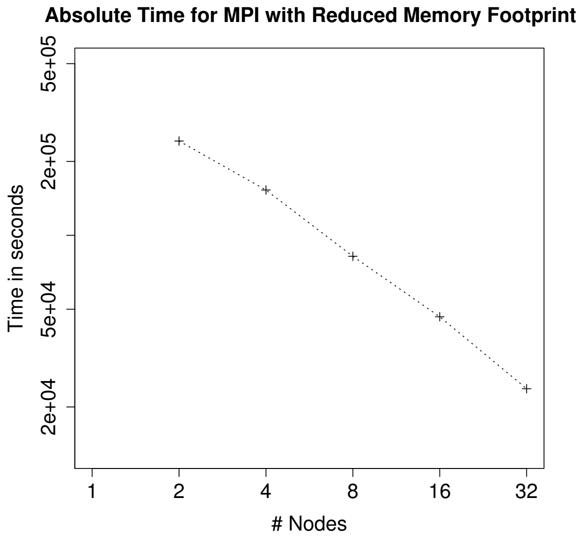

Figure 7. Absolute time for MPI with Reduced Memory Footprint. Note that a comparable time for only one node could not be obtained because of insufficient memory.

adjustments that can be made is to look at load balancing, especially with fewer synchronizations. Even with a large grain size where millions of reads are processed before a synchronization event, there is always one machine with a larger proportion of difficult reads to map that becomes the bottleneck of the computation. This is largely due to the lack of randomness in DNA: certain regions are harder to map to, contain highly repetitive elements, or are naturally enriched during the sequencing process. Future work will try to identify a way to load balance more correctly to alleviate this problem.

## V. Conclusions

Mapping short next-generation reads to reference genomes is an important element in SNP calling and expression studies. A major limitation to large-scale whole-genome mapping is the large memory requirements for the algorithm and the long run-time necessary for accurate studies. Several parallel implementations have been performed to distribute memory on different processors and to equally share the processing requirements. These approaches are compared with respect to their memory footprint, load balancing, and accuracy.

When using multiple threads (*pthreads*) on a shared memory machine, linear speedup can be achieved until more than 8 threads are used. After this time, the mutual exclusion costs overwhelm the benefits of more processors. This approach can be combined with distributed-memory parallelization using MPI and *pthreads*. In this case, machines with a sufficient amount of memory can achieve nearly-linear speedup. When calling SNPs, 100GB of memory is required, with 56GB required for a normal mapping. If large memory machines are not available, the genome can be divided among multiple processors to reduce the memory footprint. This is also shown to obtain nearly-linear speedup, and is also a suitable option for SNP calling, which requires significantly more memory.

This research has shown that the mapping problem can be effectively parallelized in several different environments without reducing accuracy. Future work will focus on reducing the memory requirements overall and load balancing.

## References

[1] F. Sanger, S. Nicklen, and A. Coulson, "DNA sequencing with chain-terminating inhibitors." *PNAS*, vol. 74, no. 12, p. 54635467, 1977.

[2] I. H. G. S. Consortium, "Finishing the euchromatic sequence of the human genome." *Nature*, vol. 431, no. 7011, p. 931945, 2004.

[3] N. L. Clement, Q. Snell, M. J. Clement, P. C. Hollenhorst, J. Purwar, B. J. Graves, B. R. Cairns, and W. E. Johnson, "The GNUMAP algorithm: unbiased probabilistic mapping of oligonucleotides from next-generation sequencing," *Bioinformatics*, vol. 26, no. 1, pp. 38–45, 2010. [Online]. Available: http://bioinformatics.oxfordjournals.org/content/26/1/38.abstract

[4] D. Bozdag, C. Barbacioru, and U. Catalyurek, "Parallel short sequence mapping for high throughput genome sequencing," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, May 2009, pp. 1–10.

[5] D. Bozdag, A. Hatem, and U. Catalyurek, "Exploring parallelism in short sequence mapping using burrows-wheeler transform," in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, April 2010, pp. 1–8.

[6] J. van Helden, "Metrics for comparing regulatory sequences on the basis of pattern counts." *Bioinformatics*, vol. 20, pp. 399–406, 2004.

[7] P. Park, A. Butte, and I. Kohane, "Comparing expression profiles of genes with similar promoter regions," *Bioinformatics*, vol. 18, pp. 1576–1584, 2002.

[8] A. Smith, Z. Xuan, and M. Zhang, "Using quality scores and longer reads improves accuracy of solexa read mapping," *BMC Bioinformatics*, vol. 9, no. 1, p. 128, 2008.

[9] H. Jiang and W. H. Wong, "SeqMap: mapping massive amount of oligonucleotides to the genome," *Bioinformatics*, vol. 24, no. 20, pp. 2395–2396, 2008. [Online]. Available: http://bioinformatics.oxfordjournals.org/content/24/20/2395.abstract

[10] R. Li, C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, and J. Wang, "SOAP2: an improved ultrafast tool for short read alignment," *Bioinformatics*, vol. 25, pp. 1966–1967, 2009.

[11] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg, “Ultrafast and memory-efficient alignment of short DNA sequences to the human genome,” *Genome Biology*, vol. 10, p. R25, 2009.

[12] H. Li and R. Durbin, “Fast and accurate short read alignment with Burrows-Wheeler transform,” *Bioinformatics*, vol. 25, pp. 1754–1760, 2009.