Highly Scalable Genome Assembly on Campus Grids

Christopher Moretti, Michael Olson, Scott Emrich, and Douglas Thain
Department of Computer Science and Engineering
University of Notre Dame

ABSTRACT

Bioinformatics researchers need efficient means to process large collections of sequence data. One application of interest, genome assembly, has great potential for parallelization, however most previous attempts at parallelization require uncommon high-end hardware. This paper introduces a scalable modular genome assembler that can achieve significant speedup using large numbers of conventional desktop machines, such as those found in a campus computing grid. The system is based on the Celera open-source assembly toolkit, and replaces two independent sequential modules with scalable replacements: a scalable candidate selector exploits the distributed memory capacity of a campus grid, while the scalable aligner exploits the distributed computing capacity. For large problems, these modules provide robust task and data management while also achieving speedup with high efficiency on several scales of resources. We show results for several datasets ranging from 738 thousand to over 121 million alignments using campus grid resources ranging from a small cluster to more than a thousand nodes spanning three institutions. Our largest run so far achieves a 927x speedup with 71.3 percent efficiency.

1. INTRODUCTION

Scientists often are able to create applications that solve their domain problems on one core, or even a small cluster. Unfortunately, as they scale up their problem size these resources become insufficient due to the scale of the problem's CPU, memory, disk, or network requirements. Traditionally scientists have had to rework their solution into a custom high performance computing implementation that can overcome these requirements using larger and more complicated hardware. As an alternative to this, we consider the "many task computing" paradigm [18] as a way to solve problems at this scale on commodity hardware by coordinating thousands of sequential processes together into an atomic workload. We focus on two types of problems typically solved by high performance solutions at this scale: naturally parallel problems with memory, network, and disk limitations; and traditionally serial problems that have oppressive memory requirements.

Bioinformatics abounds with problems that fit these characteristics. Given the recent explosive growth of the amount of genomic data available, scientists are finding the need to perform genomic analysis at unprecedented scales. Many such analyses take place on sequences millions of bases long, but DNA sequencers are only capable of producing many short sequences.

Genome assembly is the process by which the thousands or millions of reads from a sequencing run are combined to produce large, contiguous sequences that represent actual chromosomes in the cell. This is generally done by trying to detect all the places where pairs of reads overlap each other, then running algorithms that use that overlap information to build larger sequences. Because the number of sequences in a sequencing run is very large, this is computationally intensive. Modern sequencers that produce a greater number of shorter sequences [17] further increase the amount of overlaps that need to be computed.

We demonstrate challenges and opportunities for many task computing on a campus grid with a distributed implementation of the memory-intensive candidate selection step and the computationally-intensive alignment step of genome assembly, which are discussed in detail in Section 2. Our implementations of these steps distribute the memory and processing loads across a campus grid to alleviate the specific limitations of the serial solutions.

The resulting products were run on several datasets whose sizes ranged from 100 thousand to 8 million sequences. The memory required for candidate selection was reduced from 18GB on a single core to less than 2GB per core across the cluster throughout the workload, ensuring consistent access to data without costly paging to disk. For the naturally parallel alignment step we were able to spread the computation of 121 million alignments across more than 1000 cores while maintaining 71.3% parallel efficiency.

In this paper, we provide an overview of genome assembly and its parallelizable aspects in Section 2. Our pipeline is integrated into a full genome assembly in Section 3. The distributed framework, data and benchmarking environment are detailed in Section 4. Sections 5 and 6 describe the design, implementation and performance of the first two stages of the assembly pipeline. We discuss our ability to scale to many cores at multiple institutions in Section 7.

2. OVERVIEW OF GENOME ASSEMBLY

Genome sequencing is the laboratory process of determining an organism's DNA string (A,G,T,C) from a biological sample. However, no current sequencing process is capable of producing an organism's entire string of millions or billions of bases. Instead, the physical process produces a large number of random substrings of the sequence known as *reads*. Depending on the exact process, these reads can vary in length from 25 to 1000 bases each [17]. Individually, these reads have limited scientific value. *Genome as-*

MTAGS '09 November 16th, 2009, Portland, Oregon, USA

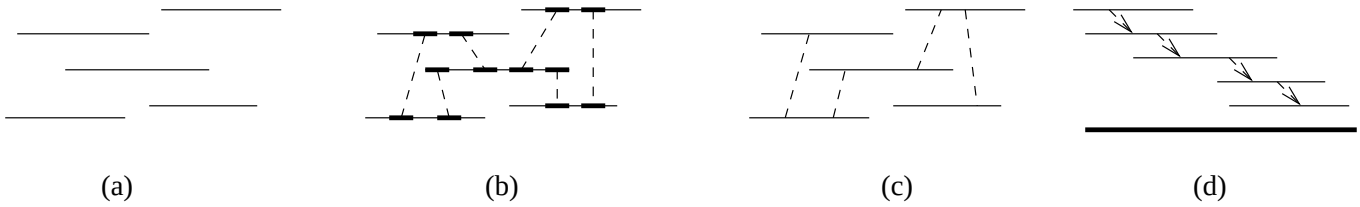


Figure 1: Logical Overview of the Assembly Process.
(a) Assembly begins with a set of unrelated reads. (b) Short exact matches are found that select candidates that may overlap. (c) Actual overlaps are computed. (d) A layout is generated based on the overlaps and a consensus sequence is generated.

Time to Assemble *A. Gambiae*

Assembly Framework	Alignment Algorithm	System Size	Candidate Selection	Alignment	Consensus
Celera	Celera	4 cores	4 hrs, 20 min		3 hrs, 11 min
Modular	Complete	1 core	* 1 hr, 35 min	* 12 days, 4 hrs	2 hrs, 33 min
Modular	Banded	1 core	* 1 hr, 35 min	* 7 hrs, 9 min	
Modular	Complete	campus grid	5 min	45 min	
Modular	Banded	campus grid	**5 min**	**11 min**	

Time to Assemble *S. Bicolor*

Assembly Framework	Alignment Algorithm	System Size	Candidate Selection	Alignment	Consensus
Celera	Celera	4 cores	crashed after 9 days		N/A
Modular	Complete	1 core	* 14 hr, 39 min	* 50 days, 15 hrs	17 hrs
Modular	Banded	1 core	* 14 hr, 39 min	* 46 hrs, 17 min	
Modular	Complete	campus grid	34 min	2 hrs, 40 min	
Modular	Banded	campus grid	**34 min**	**43 min**	

Table 1: Summary of Results
** indicates time estimated from campus grid run*

sembly is the computational process of arranging reads in the correct order to produce the largest possible contiguous strings known as *contigs*. There are many assemblers [10, 14, 25, 16] that solve the problem in a variety of ways. Here, we consider three steps shown in Figure 1: candidate selection, alignment, and consensus.

In the *alignment* step, we must find all overlaps between the suffix of one read and the prefix of another. To ensure that enough reads will overlap, a genome sequencing project oversamples from the DNA in the cell by a factor of 5-10. In principle, every single read should be compared to every other read. However, this is an $O(n^2)$ problem that is computationally infeasible for larger problem sizes.

To avoid this problem, *candidate selection* is performed first to find candidate pairs or reads that are likely to overlap. This is usually done using a method known as k-mer counting, in which each subsequence of length k in the input is added to a hash table and any sequence pairs that share at least one exact match of length k are considered to be candidates. This is a linear time algorithm that reduces the work necessary in the alignment step considerably. Finally, the assembler lays out reads in the proper order from alignment, creates one or more combined sequences, and then forms them together into larger structures called *scaffolds*. Since this paper does not address these steps in detail, we simply refer to them jointly as the *consensus* step.

Because a typical genome sequencing project creates millions of reads, genome assembly is one of the most computationally intensive problems in bioinformatics, and would benefit from parallelization. The alignment step is the most naturally parallel, requiring millions of pairs of sequences to be compared using a self-contained alignment algorithm. No task requires inter-computation communication or has dependencies on prior tasks. Most previous approaches to parallelizing assembly have focused on programming models and hardware architectures for tightly-coupled parallelism, requiring dedicated high performance clusters or massively parallel supercomputers. Additionally, previous approaches have only focused on parallelizing the alignment step, still running the candidate selection step using slow, memory-intensive sequential programs. This paper presents solutions that solve the first two components of assembly in parallel using a master-worker approach with sequential programs.

3. A SCALABLE MODULAR ASSEMBLER

We use the Celera [14] open source assembler as a starting point. Celera is widely used for processing whole genome shotgun data, and was the tool employed in the original *Anopheles gambiae* assembly [23]. Celera is relatively modular; the top-level program is a script that invokes each of the stages discussed above, using local files to communicate between steps. The candidate selection and alignment steps are woven into a single module. Each stage is multithreaded and can achieve parallel speedup on a single machine.

Table 1 shows some typical results from the Celera assembler on a 4-core Opteron 2356 CPU with 32GB RAM. We repeated the previously-assembled *Anopheles gambiae* and attempted an assembly of *Sorghum bicolor*, which is an order of magnitude larger. The *A. gambiae* assembly completed in 7 hours and 31 minutes, but the *S. bicolor* assembly ran for 9 days before crashing. If we are to

	Dataset	Number Reads	Average Read Size	Candidate Pairs	Uncomp. Size	Task Data Size	Comp. Size	Task Data Comp. Size
Small	*A. gambiae scaffold*	101617	764.22	738838	80.2MB	684.2MB	21.9MB	187.6MB
Medium	*A. gambiae complete*	1801181	763.66	12645128	1.4GB	13.2GB	0.4GB	3.6GB
Large	*S. bicolor simulated*	7915277	747.57	121321821	5.7GB	127GB	1.7GB	34.6GB

Table 2: Genomes Used in this Paper

make genome assembly an everyday task in biological research, it must be sped up by several orders of magnitude.

As shown in Figure 2, we are replacing each stage in the assembler with a scalable version that exploits the memory capacity and computational power of a campus grid. The new modules are compatible with the old implementations, so we can improve the assembly step by step. In this paper, we will demonstrate improvements in the candidate selection and alignment stage, leaving consensus for future work. The algorithmic details of both candidate selection and alignment are an open topic of research in bioinformatics, so we allow the user to custom algorithms for each by providing a simple sequential program run in parallel on many workers.

We use three different methods of alignment: *Complete* is the full Smith-Waterman alignment algorithm, which is simple but expensive; *Banded* is a simple heuristic improvement on Smith-Waterman, and *Celera* is the complex, finely tuned heuristic built into Celera. *Complete* and *Banded* are found in any bioinformatics textbook and easily implemented in an afternoon. Both perform worse than *Celera* on a single node, but our framework can scale these simple algorithms up to massive scale.

Table 1 summarizes the performance of our modular assembler. The first two steps of the *A. gambiae* assembly were reduced from 4 hours, 20 minutes down to **16 minutes**. The first two steps of the *S. bicolor* assembly were reduced from over 9 days down to **77 minutes**. (Of course, the consensus step awaits parallelization and is still measured in hours.) By harnessing a campus grid, we can convert a long term batch job into a nearly-interactive activity. The remainder of this paper explains how this is accomplished.

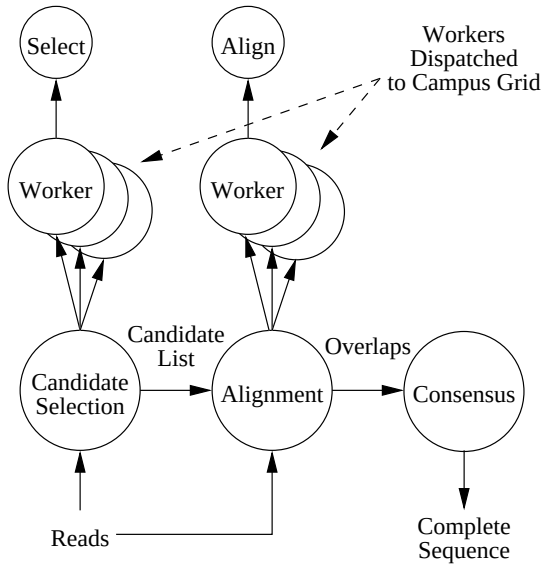


Figure 2: A Scalable Modular Assembler

4. ENVIRONMENT AND DATA

Many institutions manage hundreds or thousands of computation resources that are available for users via batch systems. These "campus grids" tend to be a mixture of desktop workstations, classroom and laboratory machines, and cluster nodes. Some resources may be dedicated to serving campus grid users, while many participate only when idle from their primary purpose. At our institution Condor [26] is used to manage most scavenged resources, while Sun Grid Engine [6] is used for dedicated high performance clusters. In both cases users submit jobs to the relevant queues and the distributed computing system facilitates execution of those jobs on available cores.

Middleware. The requirements of matching jobs to available resources means that even in the absence of contention for resources there may be a latency of 30 seconds or more before a submitted job begins remote execution, which is not unusual for these types of systems. This is especially detrimental for short-running jobs, in which the execution time is not much more than the latency. To combat this, we adapt a simple observation from Falkon [19] that we can dispatch as a batch job long-running middleware to execute many short-running tasks on a node without having to pay the overhead of submitting each task as a batch job.

To accomplish this, we use Work Queue [27], a general purpose master-worker system in which the batch job (a "worker" process) connects over the network to a process on a central node (the "master") that dispatches the smaller tasks to run. This dispatch is much faster than the dispatch latency in the campus grid queue. Additionally, the workers retain state between tasks, so files needed by many tasks only need to be transferred from the master to a given worker once. Another advantage on some systems without preemption is that later tasks are less influenced by the submitter's degraded priority (which falls as the submitter uses campus grid resources over time), because unlike separately submitted batch jobs, they are not being evaluated individually for execution on the batch system.

In practice, the user runs the master programs normally on his or her workstation. The worker processes can be submitted to the campus grid, or run individually from the command line on nodes where the user has login access. Combination of these methods is possible, and will be demonstrated in Section 7.

Figure 2 shows how the pieces work together. In general, the master streams the executable and input files to the worker, which writes them to local disk. The worker invokes the executable, storing the output locally. When the task is finished, the output written back over the network to the master. The master receives and verifies the results data, then writes it to permanent storage. Making the master responsible for results storage allows several advantages over having the application or the worker store the results: no globally available shared filesystem is required, worker processes are completely independent of the application, and master processes can interchange methods of verification based on available resources or application-specific workload-level considerations.

Genomic Data. The primary experiments were run on three genomic datasets shown in Table 2 The smallest dataset consisted of the all the reads from the largest scaffold of *Anopheles gambiae S*, the next was the entire *A. gambiae S* genome (unpublished, manuscript in preparation), and the largest was a set of simulated reads of the *Sorghum bicolor* genome [15]. The size and number

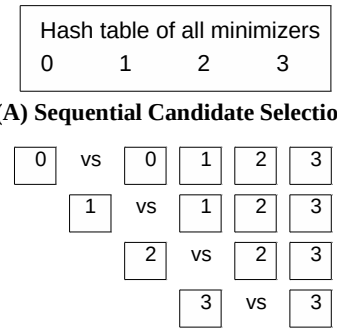


Figure 3: Memory Needed for Candidate Selection

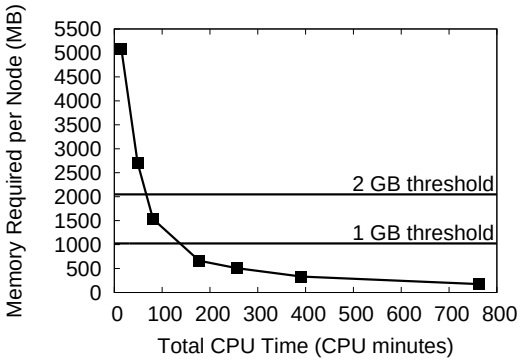


Figure 4: Candidate Selection Memory Usage Per Node

of candidate alignments for each dataset is summarized in Table 2. The *A. gambiae* genome was sequenced using traditional Sanger sequencing, which has longer read lengths, but is more expensive and time consuming. The simulated *S. bicolor* dataset was generated by extracting reads of 500-1000 bases from the finished *S. bicolor* genome with randomized starting positions.

For benchmarking the candidate selection and alignment implementations, each data set was run multiple times, varying the number of workers from 16 to 512. When possible, jobs were run on nodes in the Notre Dame Condor pool, although for larger numbers of workers we harnessed machines from other institutions.

5. CANDIDATE SELECTION

The candidate selection step suggests pairs of reads that may overlap. It takes as its input a set of sequences and outputs a set of candidate pairs for the alignment stage. It is based off of the idea of k-mer counting. A k-mer counter starts with the assumption that if two sequences share at least one short subsequence that matches exactly, then they are more likely to have significant overlap. Hence the goal is to find all pairs that share at least one subsequence of length k (a k-mer) that match exactly. In the experiments below, k was chosen to be 22, based on results from [20].

Conventional Approaches. Typically k-mer counting is done by adding each k-mer in the input to a large hash table, then traversing the table to find all pairs of reads that share at least one k-mer. UMDOverlapper [20] introduced minimizers, which are a subset of all possible k-mers that reduce the number of k-mers one needs to keep track of without losing specificity. Many assemblers use some variation on this method [3, 8, 14].

The problem with both the k-mer and minimizer counting methods is their memory usage. To add millions of minimizers to a hash table along with metadata used 16GB of memory for the large dataset. A regular k-mer counter would take even more. Most k-mer counters solve this problem by storing intermediate data to the disk, which is not only slow, but also limits the amount of available parallelism. To ameliorate this issue, we consider a method in which memory usage is distributed along with computation.

Minimizer counting is not a "naturally parallel" problem, and the conventional approach of breaking the sequences into n/l subsets of size l and giving each subset to a worker does not work. Because every sequence's minimizers must be compared against every other sequence's minimizers, every sequence must have been in a hash table with every other sequence at the same time.

Parallel Algorithm. Consider that if the sequences in each subset are added to a hash table with the sequences in every other sub-

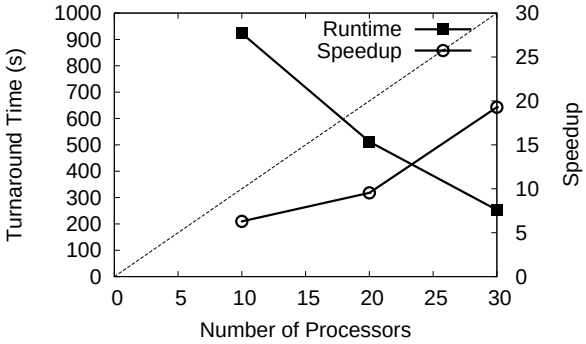


Figure 5: Scalability of Cand. Selection on Medium Genome

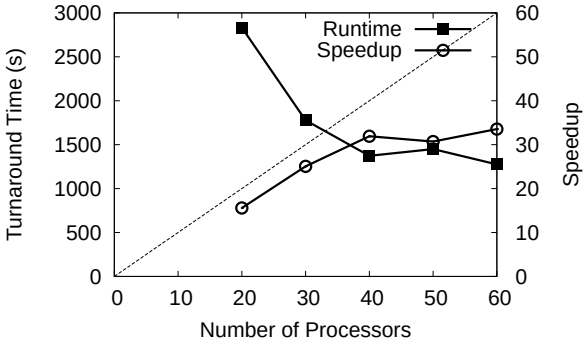


Figure 6: Scalability of Cand. Selection on Large Genome

set, then the above criterion will have been met, and every possible combination of sequences will have been made. To parallelize this, instead of adding all sequences to a large hash table, every possible pair of subsets is given as a task to the worker, which computes and returns the candidates. Figure 3(B) illustrates how the subsets are grouped and distributed to the workers. At first glance it may seem counterproductive to increase the amount of computation in this manner. A genome has nm k-mers if there are n sequences with an average length of m bases, so it takes $O(nm)$ time to add them to the hash table. Once the parallel method divides them into n/l subsets, there will be $(n/l)(n/l+1)/2 = O(n^2/l^2)$ tasks that need to be completed, each of which will take $O(lm)$ time to complete. This means that the overall amount of computation that must now be done is $O(n^2m/l)$.

Figure 4 describes the tradeoff of increased total computation for reduced memory usage per node. Despite the additional computational complexity, the advantages to this method are twofold. First, each subset of size l can be computed entirely in memory. Second, because the subsets do not rely on other subsets, they can be computed in parallel on workers in the cluster.

Implementation. The distributed candidate selection begins by dividing the input into n/l subsets of length l. Tasks are generated by making all possible pairs of subsets, including one where it is compared with itself, resulting in $(n/l)(n/l+1)/2$ total tasks.

Once tasks are generated, each one is sent to the worker along with an executable that performs the sequential candidate selection step on that pair. When a worker completes a task it sends the list of candidates back to the master, which outputs them to a file and assigns the worker a new task.

A potential problem is that even though the smaller task sizes cause the candidate selection to require much less memory per task, the workers on which the it runs will often have less memory than the powerful machines running the master. The candidate selection executable will check the amount of memory available on the worker and further subdivide its task into subsets small enough to be executed in core and operate on them sequentially.

Lastly, candidate selection has a recovery mechanism in case the master is terminated. Each time results for a task are received and written to permanent storage, a second file is updated that contains a list of each pair of subsets that was completed. If the master is interrupted, it resumes by only submitting tasks that are not listed in the recovery file.

Results. Distributing candidate selection only helps when the dataset does not fit in the memory of a single node. So, we only evaluate the medium and large datasets. To ensure that all tasks would be able to run on the machines available in our campus grid, we chose a subset size of 2GB for the large dataset and 1GB for the medium dataset. We determined empirically that the candidate selection requires space in memory roughly equal to 2.3KB/sequence. Thus, to keep the large dataset's tasks under our memory threshold, we need 22 subsets, each of size 360000. For the medium dataset the subset size was set at 250000, making 8 subsets. Speedup was calculated by assuming that the memory must be kept below the threshold and finding how long it takes to run the subsets sequentially in a low-memory environment.

The results for the medium dataset are summarized in Figure 5. The medium dataset scales up to 30 nodes, maintaining about 50% parallel efficiency. When the subset size is reduced even further more nodes can be used, however it does not significantly reduce the overall runtime, only the memory used per worker. Similar results can be observed for the large dataset in Figure 6, which scales almost linearly up to 40 nodes.

One of the biggest challenges in the candidate selection is the data transfer time. Each subset of sequences must be compared with every other subset, so the sequences must be transferred to a worker once for every subset. In the context of Figure 3(B), this means that a subset needs to be transferred once for each subset in its row and once again for each row in which it appears. In other words, given n/l subsets, each subset of size l must be transferred n/l times, which means a total of $O(n^2/l)$ data must be transferred from the worker overall. In the future, performance might be improved by implementing a caching scheme in which the master prefers to send tasks to workers that already have one or both of the subsets.

6. ALIGNMENT

Computing multiple alignments from a single set of reads is a naturally parallel problem, composed of hundreds of millions of self-contained computationally-intensive tasks, that can be solved using the many-tasks paradigm. The input for our aligner is a library of sequences and a list of candidate sequence pairs generated by the candidate selection step discussed in Section 5. The output is a list of the sequence pairs that overlap and data about where the alignments occur in the sequences.

Alignment Algorithms. An important choice in any assembler is the algorithm used for alignment of the reads. For the experiments here, we use a simple Smith-Waterman (SW) alignment commonly taught in bioinformatics textbooks [7]. This algorithm computes alignments in time proportional to the lengths of the sequences by computing progressive overlap scores in a dynamic programming matrix. The reasons we chose SW are twofold. First, it can be implemented very easily, highlighting the ability of our framework to be reused by domain experts not familiar with distributed systems programming. Second, its increased sensitivity may be required in certain cases, such as in SNP discovery programs like MOSAIK [9] and in short-read sequence assemblers.

Conventional Approach. Given a naturally parallel problem, the intuitive approach is to split the problem up into as many tasks as there are resources, and submit those tasks as batch jobs to the campus grid [11, 14]. The simplest way to do this is to prestage the work locally and require the batch system to transfer the task input data with the batch job. An issue with this solution, however, is its voracious consumption of local state. As most batch systems require all files to be in place on submission and remain in place (because of the likelihood of latency, out-of-order execution, or eviction) the framework would have to prestage locally a file corresponding to every task. For workloads in which sequences appear in many different candidates this means that the master must have enough disk space for many times the total data set size. As an example, Table 2 shows the sequence library and required task data sizes for our three workloads. The task data corresponds to the amount of data that must be sent over the network.

A related alternative to the conventional approach is similar, but the data are prestaged onto the resources where the computation will take place. The tasks would then be run on resources with the appropriate task input. A complication with this method is that the input data are quite large and the target campus grid resources are neither persistent nor reliable. The former limits our ability to prestage all the tasks' data to every compute node. The latter limits our ability to carefully craft exactly which tasks will run on which resources and prestage the appropriate task input files accordingly.

Note that variants of these two approaches can ameliorate some of their major drawbacks, but at the cost of requiring additional high-capacity, reliable resources. Moreover, in any of these cases, each task runs as a separate batch job incurring the full overhead associated with the batch system. This becomes worse as workloads

get bigger due to degraded batch system priority for later tasks.

Design and Implementation. The alignment master's system architecture is designed to avoid the disk space, network latency, and bandwidth bottlenecks encountered in the conventional approach. To prevent excessive consumption of disk space and slow filesystem access to many small files, the master process reads in the input library (genetic sequences in this case) and stores the sequences in a hash table for fast lookup based on the sequence identifier. To prevent task submission latency from limiting effective parallelism, the input data (the sequence ID, the sequence metadata, and the sequence data for each candidate pair) for many separate instances of the serial program are grouped together into task buffers. To decrease total data sent over the network, the candidate list is sorted; what this means for the alignment application is that pairs sharing a first sequence can easily be grouped together with the shared sequence copied only once in a task buffer rather than once for every pair that includes it. Once the tasks have been buffered together, the sequential program and the task buffer are sent over the network to the worker.

Because the master may run for many hours or days, it includes a recovery mechanism for starting back up a workload during which the master has crashed. The recovery mechanism in the master surveys completed pairs from the results and mimics starting a new workload for those not yet completed. Once the recovery mechanism has discerned all the completed pairs, the remainder of the workload continues unhindered.

While the master's design considerations save on disk space as shown in Table 2 and conserve network bandwidth, this comes at the cost of requiring all the sequences in memory on the master throughout the workload, rather than just during task construction.

Results. We measure our ability to scale using both strong scaling and weak scaling. A workload that indicates good strong-scaling efficiency will, for a constant workload problem size, see its speedup scale by the same factor as the increase in number of processors. A workload that indicates good weak-scaling efficiency will keep a constant turnaround time if both the problem size and the number of nodes are increased by the same scaling factor.

Calculating conventional parallel speedup for a heterogeneous and dynamic set of resources is not meaningful. Further, because the benchmarks were so large and contained so many alignments it was not feasible to simply run all the alignments sequentially. We use the workload's average execution time across all tasks, multiplied by the number of tasks completed as the sequential runtime for the parallel speedup computation. Note that later, in Figures 12, 13 and 14 where we graph the speedup as a function of time for both problematic and corrected instances, the average run time from the corrected version is used.

In the benchmarks below, each task contained 5000 alignments. Our benchmarks showed that when running on a sufficiently fast network, such as a local cluster, task size did not have a significant effect on performance, which can be seen in Figure 11.

Task size becomes more important when many nodes are further away in the network, as the transfer time for each task does not scale linearly with the size of the task. Larger task sizes pay the same overhead while sending more data, and utilize the workers better, resulting in faster run times and better speedup. However, there are two major downsides to increased task size. First, if the system is especially volatile, more work is lost when a worker is evicted. Second, the master queues a large amount of tasks to ensure that the master never runs out of tasks to assign. A larger task size will take up more memory per task, increasing the memory consumption. The effects of excessive memory consumption are discussed in more detail in Section 7.

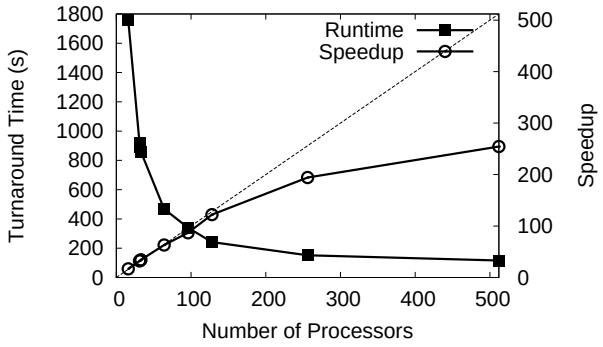


Figure 7: Scalability of Alignment on Small Genome

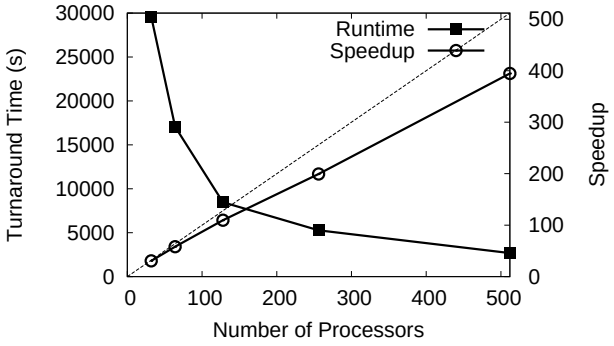


Figure 8: Scalability of Alignment on Medium Genome

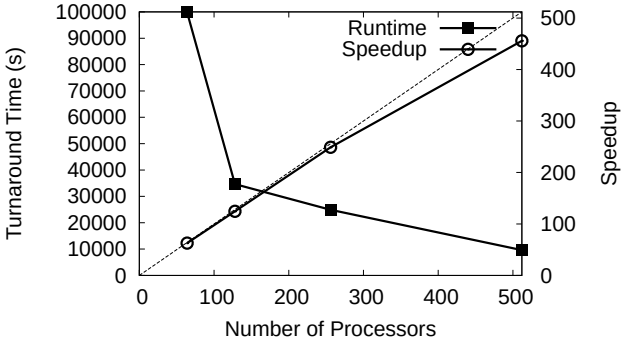


Figure 9: Scalability of Alignment on Large Genome

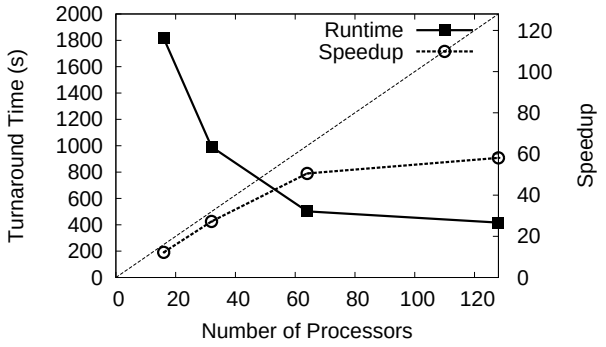


Figure 10: Effect of Faster Alignment

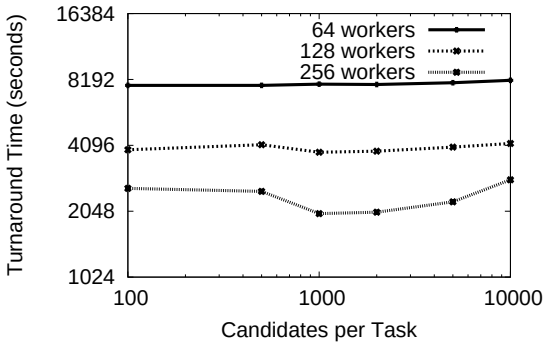


Figure 11: Alignment Candidates Per Task

We observed scaling speedup for almost all of the benchmarks. However, each benchmark has features that shed light on the strengths and weaknesses of the system. For our smallest dataset we achieved near linear speedup until about 128 workers (Figure 7). Because this is the smallest dataset, with too many nodes all the work is completed before some nodes receive a task.

The medium dataset (Figure 8) yielded better results; the dropoff in speedup did not occur until 512 nodes were used. The large dataset displayed similar scalability to the previous dataset. It was able to run on 512 cores in only 9595 seconds, for a speedup of 455.74. This dataset did highlight some of the challenges of the assembly problem and of distributed computing in general. These are discussed in detail in Section 7.

Banded Alignment. One of the primary advantages of our framework is its ability to substitute any alignment algorithm for the one used in our benchmarks. So, in addition to these benchmarks, we have also considered how our framework adapts to alignment programs that are considerably faster than the simple solution we use above. We tested this by implementing a simple banded alignment, in which only a narrow band of the SW dynamic programming matrix is computed [7]. In this case, the amount of data remains the same while the execution time of each task decreases significantly. As a result of the increased relative overhead, we would expect decreased scalability. The results are summarized in Figure 10. We achieve increasing speedup up to 64 workers, at which point we begin to experience diminishing returns.

Pipelining. In the previous sections we discuss the candidate selection and alignment steps separately. However, in practice the two steps can be pipelined, because the aligner can begin constructing and submitting tasks as soon as the candidate selection begins generating candidates. Once the candidate selection completes, its workers can be redirected to work for the aligner's master process. For example, in one instance the candidate selection on the medium dataset ran in 423 seconds on 60 workers. The aligner ran in 502 seconds for a combined runtime of 925 seconds. However, the pipeline running 30 workers on candidate selection and 40 on alignment finished in only 654 seconds.

7. SCALING UP TO THE GRID

For very large problems, the computational resources required exceed the capacity of the clusters comprising Notre Dame's campus grid. At this point, we explore the ramifications of running on multi-institutional resources such as remote Condor pools or the Open Science Grid [1]. Our primary experiments in the section run on the large dataset using Condor's flocking mechanism, as an example of using remote grids. We start off with a discussion of managing workers efficiently at the grid scale, and how several obstacles can result in idle workers waiting to be assigned tasks.

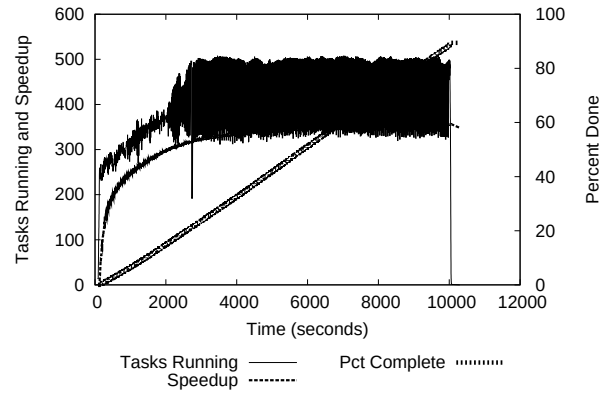
Waiting for Out-of-Core Task Data. Complete alignment on the large dataset scales at nearly linear speedup up to 256 workers, but saw a marked decrease in performance when using 512 workers. The biggest problem with running such a large dataset was memory. Although we were running the master on a machine with 8GB of memory, the large dataset was 5.7GB. This is loaded into memory to achieve the best retrieval times when building tasks. Additionally, the master buffers tasks in memory.

With 512 workers, the additional buffered tasks caused the master to exceed physical memory. When the master began to need paging for its task management, performance began to degrade. The effect of this can be seen in Figure 12(A). Because it takes significantly longer to create the number of tasks required, workers must wait longer to receive their task. When running with many workers, the amount of time necessary to give tasks to all the workers is longer than the amount of time it takes a worker to complete this task. This creates a convoy effect, where workers are spending more time waiting to be processed by the master than they spend actually working. This explains the large variation in the number of tasks working.

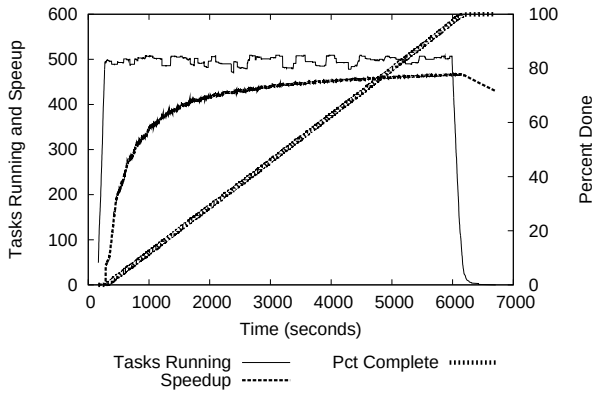
To combat this issue, we took a rather straightforward approach. Because DNA consists of only 4 letters, it is possible to represent a single base of DNA as a 2-bit number rather than a character to achieve nearly 75% compression. Once the amount of memory needed can be kept within the physical memory, the master is easily able to keep up with the workers requesting tasks. In this case, the number of workers running at any time remains relatively constant, subject only to minor fluctuations, mostly caused by changes in the number of workers active. Figure 12(B) shows how the same job ran on 512 workers with compression enabled.

Waiting for Network Transfers. When a master has too many workers connected to it, it takes the master longer to assign tasks to all the workers than it takes for an individual worker to finish its task. The same symptoms appear as in the memory case above: workers spend more time waiting to be given new tasks than they spend working, and efficiency suffers. Further, some workers can experience starvation, triggering idleness timeouts and exiting as the number of connections to the master gets too large. In this case, the main problem is waiting for the master to transfer task data to every worker. There are just over 650 machines in Notre Dame's Condor pool; to exceed this number we are forced to use machines from other institutions' Condor pools, particularly Purdue University and the University of Wisconsin.

While we could transmit data to machines at Notre Dame at an average speed of 42.29 MB/s (meaning data for a task could be



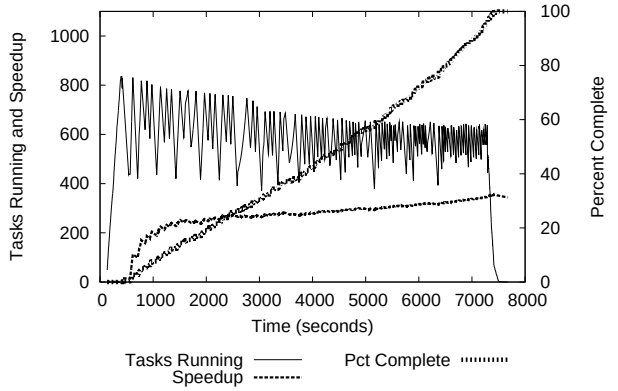
(A) No Compression



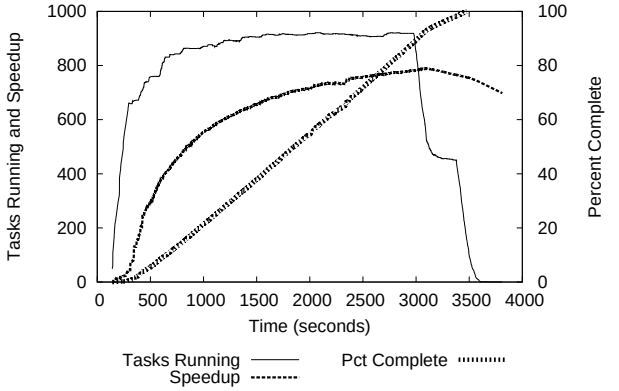
(B) Compression

Figure 12: The Effect of Data Compression.

These graphs show the effect of data compression on the master's ability to dispatch tasks using the large dataset. Each shows a timeline of a single run, with the number of tasks running, the cumulative speedup, and the percent complete over time. Figure 12(A) does not use data compression, and oscillates between 300 and 400 tasks running at once, reaching a speedup of slightly better than 300x. Figure 12(B) uses compression, and stabilizes at about 500 workers with a speedup of about 500x.



(A) Single Master



(B) Dual Masters

Figure 13: The Effect of Splitting Masters.

When using a sufficiently large number of workers on the large dataset, the master does not have enough network bandwidth to keep all of them busy. These figures show a timeline of a single run with approximately 950 workers using one master (A) and two masters (B). With a single master, workers complete faster than the master can dispatch new work, so not all nodes can be kept busy processing at once, and the speedup reaches less than 400x. With dual masters, peak speedup reaches 790x before settling out about 700x. Note that the unequal distribution of completing work in (B) causes the dropoff beyond 3000s.

transfered in only a few hundredths of a second), data to Purdue took an average of .36s, and data to Wisconsin was even slower, at .53s per transfer. In a job we ran with 900 submitted workers for a single master with 5000 candidates per task, the average transfer time was 0.27s. 835 workers completed tasks, with the others failing to find an available campus grid resource or exiting after starvation. This means the average time to transfer files to all 835 workers was 225s, which is greater than the typical task completion time.

In order to solve this efficiency problem, we split the list of candidate pairs in half and run the master on two separate machines. When using two masters on the above workload, sending data to 450 workers each averaging 0.27s per task takes only 121s, so both masters were able to work efficiently.

Figure 13(A) shows a timeline of workers waiting rather than actively computing associated with this problem for a similar job with 950 submitted workers, while Figure 13(B) shows the smoother two-master version of the same workload. The maximum number of workers running tasks at a time was 921 with two masters.

From Desktop to Grid. Now we give an example of a large production workload scaled up to run on a multi-institutional grid. We construct a scenario that serves to demonstrate all of the features of our framework, and illustrates a typical use performing a complete alignment of the large dataset, the simulated *S. bicolor*. The scenario presents several of the key components of our framework: adaptability to many types of resources (local execution, execution as a cluster job, execution on a campus batch system, execution as part of a multi-institutional resource pool); fault-tolerance to failures on the worker nodes; and fault tolerance to failures on the master node.

As in many fields, research in bioinformatics is highly exploratory. An active researcher may test many slight variations upon an algo-

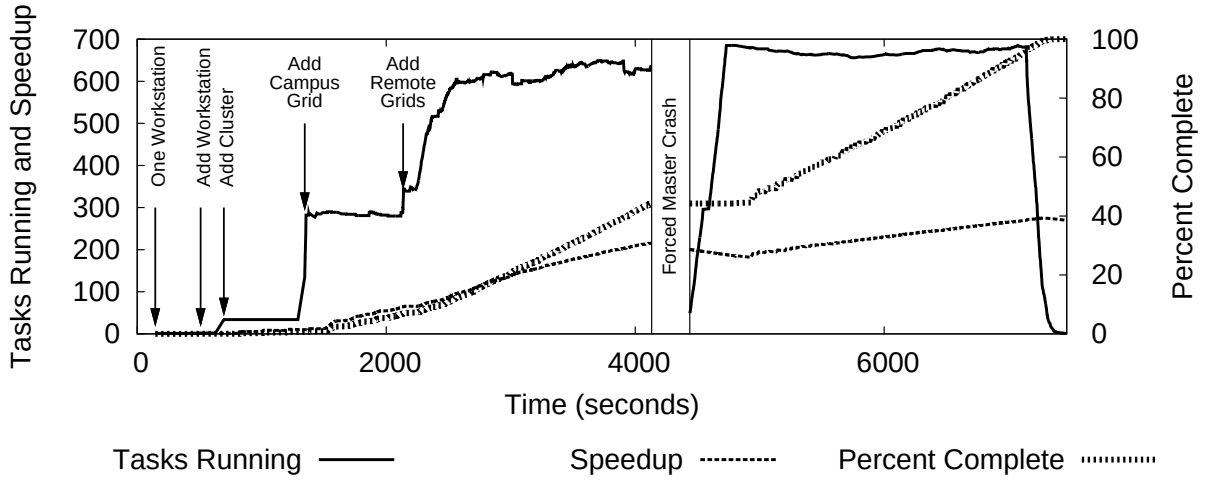


Figure 14: Scaling Up to the Grid

This figure shows the timeline of a large assembly run on a system grown progressively from a single workstation up to a large scale grid including resources at the University of Notre Dame, Purdue University, and the University of Wisconsin. The master is forcibly killed halfway through to demonstrate failure recovery.

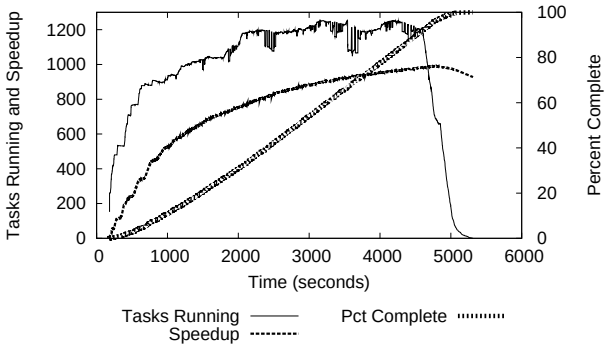


Figure 15: Sustaining Grid Scale

This figure shows the timeline of a 121M candidate run on the large sequence set using approximately 1300 workers at two institutions separated by a WAN. Two masters support almost 1300 at peak and use 1000 or more workers consistently for most of the 90-minute runtime, totaling a speedup of 927x.

rithm, generating a number of tests of various sizes before proceeding to analyze an entire dataset. Because our framework runs with an arbitrary number of workers, a user may slowly generate small results, then progressively add resources as confidence is gained.

Figure 14 gives a real example of this progressive growth. Using our framework, one author started a worker process on his workstation. After a few minutes, he surveyed the progress and determined that serial execution would not be sufficient, so he asked a coworker to start a worker on her own machine, and also prepared and submitted some batch jobs to his research group's 32-node cluster. As these jobs started running, speedup increased accordingly. Hoping to finish the alignments that afternoon, he submitted jobs to the campus computing grid at Notre Dame, followed by submissions to Condor-based grids at Purdue University and the University of Wisconsin. About halfway through the complete assembly, however, he accidentally powered off his workstation, causing the computation to halt. Fortunately, when the master was restarted, it loaded all of the complete results, accepted connections from the still-running workers, and continued where it left off. The entire assembly completed in just over two hours, with a speedup of 269x and a maximum of 680 CPUs in use at once. Note that the low speedup should not be alarming, because of the gradual nature in which the workers were added, and because of the crash in the middle of the job.

Table 3 summarizes the work distribution across sites. The tasks running at home were slower and exhibited more runtime outliers, because the local campus grid includes a large number of scavenged resources compared with more homogeneous dedicated grid resources at the other sites.

	Tasks	Average Runtime (s)
Total	16936	184.1 ± 53.8
Notre Dame	7998	215.3 ± 46.4
Purdue	7760	154.0 ± 40.8
Wisconsin	1232	170.1 ± 56.2

Table 3: Summary of Workload

Even making many connections over the WAN, the master was still able to maintain a steady task throughput with machines at three different institutions. The scalability is strong – taking into account that the final speedup is not reflective of the final state of the workload – and with an improved wide area network connection even more resources at remote institutions could be harnessed. Additionally the multiple-masters technique used before to demonstrate a solution to insufficient network bandwidth will still be advantageous.

Many-Node Run. Finally, we take advantage of using two masters on a single workload to show scalability beyond that shown in the scenarios above. Using multi-institutional resources, we completed Smith-Waterman alignments of the largest dataset – 121 million candidate pairs from a set of 8 millions sequences – in under one and a half hours. For comparison, the same workload serially would take over 57 days on an average resource from our pool. Figure 15 shows a peak of almost 1300 resources harnessed, sustained levels above 1000 for an hour during the workload, and a final speedup of 927x at 71.3% parallel efficiency.

8. RELATED WORK

Because determining overlaps between candidates is the most time intensive step of an assembly, it is the step most often parallelized. For example, to assemble the mouse genome the PCAP program was developed to use 24 compute nodes and a shared file system [11]. PCAP generated a total of 273 million overlaps that were processed in 80 distinct batch jobs, each of which took 7 days to compute on a Compaq ES40. Kalyanaraman et al. later reported an approach that could process 47 million maize candidate alignments in under 2 hours using 1024 processors of an IBM BlueGene/L [12]. More recent work has explored using FPGAs [24] and the Cell processor [21] to speed up alignment, which would provide up to a 100X speedup.

These parallel solutions to genome assembly have relied on on batch processing, complex MPI programming or specialized hardware. In contrast, we are interested in a growing trend to develop modular genome assembly components such as the UMDOverlapper [20], which can reliably work with phrap, the Celera assembler, and Atlas. Another example is the AMOS consortium [17], which is actively developing an open source, modular assembly pipeline. Here, we extend this modular design concept to facilitate custom parallel genome assembly. Rather than rely on specialized hardware and/or programming, we use custom candidate selection and alignment modules that are highly adaptable to many types of distributed resources.

Our assembly components use the well-known master-worker (MW) paradigm for distributed computing. The independent-failure nature of MW lends itself to fault tolerance [2] and other performance enhancements [4]. The Condor-MW framework has been used to scale up CPU-intensive applications such as optimization problems to nearly 2000 nodes [13]. Our use of conventional Unix programs as modular units is inspired by a similar many-tasks technique demonstrated by Falkon [19]. Similar systems such as CloudBurst [22] have applied the Map-Reduce [5] data-parallel computation model to a similar bioinformatics problem.

9. ACKNOWLEDGEMENTS

This work was supported in part by a University of Notre Dame strategic initiative for Global Health, by the National Institutes of Health (NIAID contract HHSN266200400039C) and the National Science Foundation (grant CNS06-43229).

10. REFERENCES

[1] The Open Science Grid. http://www.opensciencegrid.org.

[2] D. Bakken and R. Schlichting. Tolerating failures in the bag-of-tasks programming paradigm. In *IEEE International Symposium on Fault Tolerant Computing*, June 1991.

[3] S. Batzoglou et al. ARACHNE: A whole-genome shotgun assembler. *Genome Res.*, 12(1):177–189, January 2002.

[4] D. da Silva, W. Cirne, and F. Brasilero. Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids. In *Euro-Par*, 2003.

[5] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large cluster. In *Operating Systems Design and Implementation*, 2004.

[6] W. Gentzsch. Sun grid engine: Towards creating a compute power grid. In *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 35, Washington, DC, USA, 2001. IEEE Computer Society.

[7] D. Gusfield. *Algorithms on strings, trees, and sequences : computer science and computational biology*. Cambridge Univ. Press, January 2007.

[8] P. Havlak et al. The Atlas genome assembly system. *Genome Res*, 14(4):721–732, April 2004.

[9] L. W. W. Hillier et al. Whole-genome sequencing and variant discovery in *C. elegans*. *Nat Methods*, January 2008.

[10] X. Huang and A. Madan. CAP3: A DNA sequence assembly program. *Genome Res.*, 9(9):868–877, September 1999.

[11] X. Huang, J. Wang, S. Aluru, S.-P. Yang, and L. Hillier. PCAP: A whole-genome assembly program. *Genome Res.*, 13(9):2164–2170, September 2003.

[12] A. Kalyanaraman, S. Emrich, P. Schnable, and S. Aluru. Assembling genomes on large-scale parallel computers. *Journal of Parallel and Distributed Computing*, 67(12):1240 – 1255, 2007. Best Paper Awards: 20th International Parallel and Distributed Processing Symposium (IPDPS 2006).

[13] J. Linderoth et al. An enabling framework for master-worker applications on the computational grid. In *IEEE High Performance Distributed Computing*, pages 43–50, Pittsburgh, Pennsylvania, August 2000.

[14] E. W. Myers et al. A whole-genome assembly of Drosophila. *Science*, 287(5461):2196–2204, March 2000.

[15] A. H. Paterson et al. The Sorghum bicolor genome and the diversification of grasses. *Nature*, 457(7229):551–556, January 2009.

[16] M. Pop et al. Genome sequence assembly: Algorithms and issues. *Computer*, 35(7):47–54, 2002.

[17] M. Pop and S. L. Salzberg. Bioinformatics challenges of new sequencing technology. *Trends in Genetics*, 24(3):142–149, March 2008.

[18] I. Raicu, I. Foster, and Y. Zhao. Many-Task Computing for Grids and Supercomputers. In *IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS08)*, 2008.

[19] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde. Falkon: a Fast and Light-weight tasK executiON framework. In *IEEE/ACM Supercomputing*, 2007.

[20] M. Roberts et al. A preprocessor for shotgun assembly of large genomes. *Journal of Computational Biology*, 11(4):734–752, 2004.

[21] A. Sarje and S. Aluru. Parallel biological sequence alignments on the cell broadband engine. pages 1–11, April 2008.

[22] M. Schatz. CloudBurst: Highly sensitive read mapping with MapReduce. *Bioinformatics (Online Advance Access)*, April 2009.

[23] M. V. Sharakhova et al. Update of the Anopheles gambiae PEST genome assembly. *Genome Biology*, 8:R5+, January 2007.

[24] O. Storaasli and D. Strenski. Exploring accelerating science applications with FPGAs. July 2007.

[25] K. A. Swan et al. High-throughput gene mapping in caenorhabditis elegans. *Genome Res*, 12(7):1100–1105, July 2002.

[26] D. Thain, T. Tannenbaum, and M. Livny. Condor and the grid. In F. Berman, G. Fox, and T. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley, 2003.

[27] L. Yu, C. Moretti, S. Emrich, K. Judd, and D. Thain. Harnessing Parallelism in Multicore Clusters with the All-Pairs and Wavefront Abstractions. In *IEEE High Performance Distributed Computing*, pages 1–10, 2009.