PREPRINT: A Framework for Scalable Genome Assembly on Clusters, Clouds, and Grids

Christopher Moretti, Andrew Thrasher, Li Yu, 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 genomic 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 the Scalable Assembler at Notre Dame (SAND) framework that can achieve significant speedup using large numbers of commodity machines harnessed from clusters, clouds, and grids. SAND interfaces with the Celera open-source assembly toolkit, replacing two independent sequential modules with scalable parallel alternatives: the candidate selector exploits distributed memory capacity, and the sequence aligner exploits distributed computing capacity. For large problems, these modules provide robust task and data management while also achieving speedup with high efficiency. We show results for several datasets ranging from 738 thousand to over 320 million alignments using resources ranging from a small cluster to more than a thousand nodes spanning three institutions.

Index Terms—C.2.4 Distributed Systems, Bioinformatics, Genome assembly.

1 Introduction

The landscape of genomics and bioinformatics research is undergoing a dramatic change. The cost of traditional genome sequencing was once in the range of millions of dollars and only accessible to national-scale centers focused on the study of model organisms. Today, second generation sequencing devices are operating at hundreds of modest institutions and typical sequencing costs of medium-sized genomes are on the order of tens of thousands of dollars, and will drop further with soon-to-be released third generation platforms [11]. Any ordinary academic or commercial lab will be able to generate gigabytes to terabytes of genomic data.

However, raw data of a newly sequenced organism is of little use until it is assembled. Genome assembly is a very computationally intensive task, so sequencing centers (and newer generation sequencing vendors) have historically incorporated high end computers into their facilities and installations. Accordingly, assembly software has been created, evaluated and deployed on parallel supercomputers. For example, a recent result in large scale assembly by Kalyanaraman et al. [22] employs the MPI framework on a dedicated BlueGene/L..

As sequencing becomes inexpensive and commonplace, there will be a greater need for computational power, and utilizing a high-end computer for each device is not practical. Instead, we argue that commodity sequencing devices can be well served by commodity computing systems. The typical researcher today has access to a wide variety of computing power in the form of clusters, clouds, grids. However, to harness these resources requires software that can run on large collections of non-dedicated, heterogenous machines.

To address this, we have created SAND - the Scalable Assembler at Notre Dame. SAND is an framework that can operate on clusters, clouds, and grids. It consists of two main stages that speed up assemblies: candidate selection and sequence alignment. The first stage depends on the aggregate memory of the distributed system, while the second stage relies on the aggregate computation power. SAND is both elastic and fault-tolerant, and can be run on any collection of machines, including high performance computers, dedicated clusters, and desktop workstations. SAND is modular, making it easy to change algorithms as the field of bioinformatics advances. Specifically, the core ideas in SAND are ideal for diverse and/or error-prone data characteristic of some second and early third generation sequencing platforms such as the PacBio instrument [11].

To evaluate the correctness of SAND, we assemble and verify an assembly of the malaria mosquito *Anopheles gambiae* derived from tens of diverse individuals [24]. To evaluate the performance, we measure the strong scalability of the system on mosquito and synthetic sorghum sequence data on a dedicated cluster of up to 512 cores using both complete prefix-suffix alignment and the common optimization of banded alignment. We compare the latter to the performance of the widely used Celera assembler, showing that SAND is faster than Celera in absolute terms, but also scalable to larger numbers of cores. Finally, we demonstrate scalability and robustness by assembling human sequence data on a system of over 1000 cores drawn from multiple institutions.

In this paper, we provide an overview of overlap-based genome assembly and its parallelizable aspects in Section 2. The overall architecture of SAND and the validation of results are presented in Sections 3 and 4. Sections 5 and 6 describe the design, implementation and performance of the candidate selection and sequence alignment stages, respectively. In Section 7, we discuss our ability to scale to many cores and multiple institutions Section 9 in the online supplement provides further details on candidate selection and scalability techniques.

This paper is an extension of an earlier workshop paper. [27]

	Dataset	Number Reads	Average Read Size	Candidate Pairs	Uncomp. Size	Task Data Size	Comp. Size	Task Data Comp. Size
Small	*A. gambiae scaffold*	101,617	764	738,838	80MB	684MB	22MB	188MB
Medium	*A. gambiae complete*	2,586,385	763	12,645,128	2.5GB	13GB	642MB	3.6GB
Large	*S. bicolor simulated*	7,915,277	747	121,321,821	5.7GB	127GB	1.7GB	35GB
Huge	*H. sapiens complete*	31,257,852	654	327,025,224	80GB	299GB	20GB	79GB

TABLE 1
Summary of Genome Data Used in this Paper

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 directly producing an organism's entire string of millions or billions of bases. Instead, the process produces a large number of random substrings of the sequence known as *reads*. Reads can vary in length from 25 to 1000 bases, depending on the sequencing technology [31]. *Genome assembly* 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 [7], [18], [19], [28], [30] that solve the problem in a variety of ways. Here, we consider three computational steps: candidate selection, alignment, and layout + consensus.

In the *candidate selection* step, we must find all potential overlaps between the suffix of one read and the prefix of another. To ensure that enough reads will overlap, it is necessary to oversample by a factor of 5 to 10. In principle, every single read should be compared to every other read, but this would be computationally infeasible. To reduce the problem, assemblers often use 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 considerably.

Next, overlap regions are refined using modified versions of *sequence alignment* [17] to find differences including sequencing errors. This step has a worse case complexity of O(mn) where m and n are the sizes of the two sequences considered. Because there can be millions of candidates, this step takes the most time in existing assemblers.

Finally, the assembler lays out reads in an estimated order, creates contiguous sequences, and then combines them together into larger structures called *scaffolds* in a process simply refered to as the *consensus* step. Although it is possible to run consensus in parallel, this step contains the least amount of parallelism and can often be computed in a few hours on a single commodity machine. Therefore, we do not address consensus further in this paper.

In a genome assembly the alignment step is the most naturally parallel, with no tasks requiring inter-computation communication or having dependencies on prior tasks. Most previous approaches to parallelizing assembly have focused on this step but still run candidate selection step using memory-intensive sequential programs discussed in Section 8. This paper presents solutions that solve both components on commodity distributed systems such as clusters, clouds, and grids.

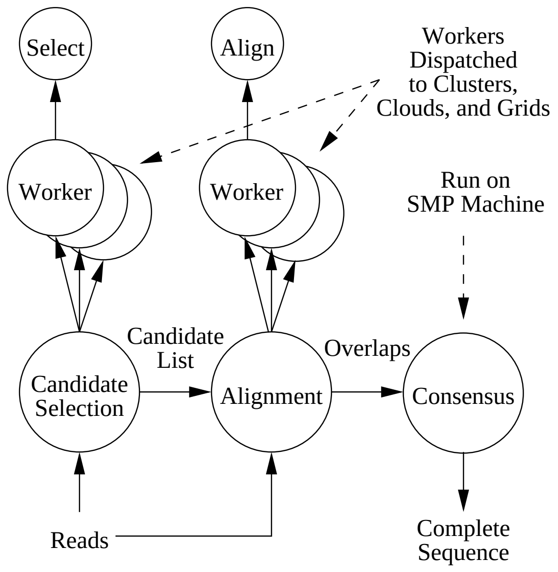


Fig. 1. Architecture of SAND

3 SAND

SAND is an open source scalable framework that replaces the first several stages of an overlap-based assembler similar to [34]. Specifically, SAND generates OVL alignment information for the Celera Assembler (CA) [28] but can be easily modified for other software. We chose CA to start because it is widely used for processing whole genome shotgun data (e.g., [39]) and is relatively modular: the top-level program is a script that invokes each of the stages discussed previously using files to communicate between steps. The candidate selection and alignment steps are woven into a single module, and can be run on a batch system such as SGE, relying on a shared filesystem to communicate the sequence data.

As shown in Figure 1, SAND replaces two stages in CA with scalable versions that exploit the memory capacity and computational power of clusters, clouds and grids, without requiring a shared filesystem. The new modules are compatible with the old implementations, so we can improve the assembly step by step. For example, a new overlap module can be developed and used for reads from the PacBio instrument [11] that can reach tens of thousands of nucleotides in length.

The algorithmic details of both candidate selection and alignment are an open topic of research in bioinformatics, so we allow the user to provide custom algorithms for each if desired. In this paper, We use two different alignment algorithms used in overlap-based assemblies: *Complete* is the full prefix-suffix alignment algorithm, which is simple but expensive and *Banded* is a simple heuristic improvement on

Dataset	Cores	SAND Selection	SAND Alignment	Celera Consensus
Small	1	220 sec	539 sec	127 sec
	128	10 sec	29 sec	
Medium	1	265 min	377 min	193 min
	128	5 min	7 min	
Large	1	104 hrs	40 hrs	17 hrs
	128	49 min	19 min	

TABLE 2
Summary of SAND Performance

prefix-suffix. Both are found in any bioinformatics textbook and easily implemented in an afternoon. Although these algorithms are less sophisticated than those found in CA, the structure of SAND results in both faster absolute performance and scalability to larger systems.

All experiments were run on the datasets shown in Table 1. *Small* consists of the all the reads from the largest scaffold of *Anopheles gambiae M*, *Medium* is the entire *A. gambiae M* genome, and *Large* is a set of simulated reads of the *Sorghum bicolor* genome [29]. 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. The *Huge* dataset is the Venter human genome [43], which we employ in a final demonstration of scalability.

Table 2 shows some typical results of using SAND on a single core and on a large cluster. As can be seen, candidate selection and alignment dominate the costs of assembly as the data size increases. SAND reduces these steps considerably.

Validation of SAND was performed on technical and biological levels. Technically, the goal of SAND is to provide a scalable framework for new and previously published assembly modules such as candidate selection. As expected, SAND and the original UMDOverlapper implementation in [34] produce the same set of candidates with the same parameters. Extensive biological testing of UMDOverlapper has shown that replacing early versions of the CA overlapper produced a significantly improved genome in Drosophila [34] and rat [35]. Our contribution is overlap detection such as UMDOverlapper can be run on multiple cores each with smaller amounts of RAM (Section 6). As with candidate selection, any alignment module can be scaled to larger, more heterogeneous systems using our SAND framework (Section 7).

Most validation was done with respect to *Anopheles gambiae* M, our medium dataset, whose genome we recently published [24]. There was no substantial difference in common measures of genome assembly quality (e.g., N50 size) or obvious structural differences when compared to the PEST reference genome [24]. This is expected given previous results of UMDOverlapper and CA that SAND is built on. Although the subject of future work, we are using now SAND for *Anopheles gambiae* and for other arthropod genomes. The scalable nature of SAND enables the parameter exploration often necessary for biologically optimal results.

4 The Work Queue Framework

A variety of systems today make it easy for the individual researcher to obtain large numbers of cores. An individual might have a small cluster of homogeneous machines in the lab, or have access to a large institutional cluster shared between many researchers using software such as Sun Grid Engine [16]. Many institutions scavenge idle cycles from workstations and clusters using software like Condor [42] to create a campus grid of thousands of non-dedicated machines. Multiple institutions can band together to create national grid like the Open Science Grid [2] or the TeraGrid [1]. Commercial providers such as Amazon EC2 or Windows Azure provide metered access to virtually unlimited resources.

Unfortunately, each one of these systems has its own user interface, programming model, and semantics of execution. Instead of accessing these systems directly, we use them to start another layer of software that provides a common execution environment. In SAND, we use Work Queue [44], a data intensive master-worker system. We use the native interface of the cluster, cloud, or grid to start hundreds of worker processes, which then contact a master process directly. The execution environment is managed entirely between the master and the workers, and no further interaction with the cluster, cloud, or grid is necessary.

Overlaying the master and worker on the existing system has several benefits. Because the worker persists across multiple task executions, commonly used input files, executables, and libraries can be cached at the execution site, speeding up later tasks. Dispatching a new task to an existing worker is a matter of a single network communication, which is much faster than the minutes necessary to allocate a new virtual machine or processor in an existing cluster, cloud, or grid. For workloads such as SAND that consist of many short running but data intensive jobs, these properties are critical.

In practice, the user runs the master program on a workstation or server as part of their normal activity. Worker processes can be started on clusters, clouds, and grids by simply logging into the desired machine and running the `worker` executable. To facilitate starting large numbers of workers through batch systems we provide scripts named `condor_submit_workers`, `sge_submit_workers`, etc. that simply submit workers as batch jobs. Workers may be started on multiple systems simultaneously, as we will demonstrate in Section 7.

Figure 1 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 is written back over the network to the master. The master receives and verifies the result, then writes it to permanent storage. Making the master responsible for result 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 the master can validate the correct form of results as they are produced.

To evaluate the performance of SAND in a controlled

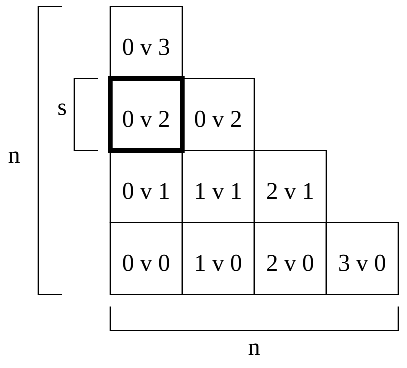


Fig. 2. Distributed Candidate Selection

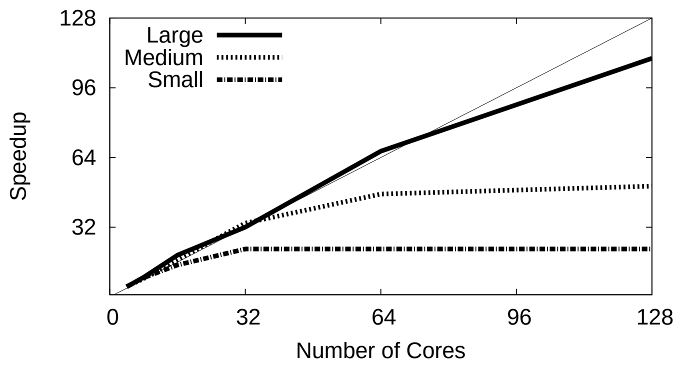


Fig. 3. Scalability of Candidate Selection

manner, we make use of a dedicated cluster consisting of AMD Opteron 2356 2.3GHz quad-core CPUs with 2GB of RAM allocated to each core. The results in Sections 5 and 6 employ up to 512 cores harnessed by submitting workers to these machines through the SGE batch system. Where sequential performance is indicated, we employ a single node of this cluster. In Section 7, we run SAND on larger multi-institutional systems with a wide variety of heterogeneous machines. Although such systems are uncontrolled, they are effective in exposing the challenges of the execution environment, and demonstrate that SAND can run effectively in such an environment.

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: 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 [34].

Typically k-mer counting is done by adding each k-mer in the input to a single hash table, then traversing the table to find all pairs of reads that share at least one k-mer. UMDOverlapper [34] 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 [7], [18], [28] and it has been adapted for newer generation sequencing [26].

The problem with both k-mer and minimizer counting methods is that they require memory proportional to the number of sequences. When physical memory is exhausted, the hash table will begin to swap, impacting performance by several orders of magnitude. An alternative is to use an out-of-core algorithm to compute subsets of the problem that fit in memory. However, this increases both complexity and computation time.

Parallelization does not affect the results of k-mer matching for valid candidates, but may affect what is considered repetitive (i.e., simple repeats or transposable elements ion the genome). We use meryl, Celera's native k-mer counter, to address this concern by "masking" likely repeats prior to candidate selection. This allows us to speed up candidate selection by distributing subsets of tasks across multiple processors while preserving the validity of the k-mer approach. Figure 2 shows the general strategy after masking. Each task will involve loading two subsets of the sequences into memory, and determining the candidates for that subset. The smaller we divide the tasks, the more parallelism can be exploited, but the the more total data must be transferred. In other words, the communication-to-computation ratio (CCR) [8] is a function of the task size.

In the online supplement to this paper, we derive the formula used to choose the optimal number of sequences per task (approximately $n/22$) Using current hardware, the distributed algorithm has a maximum possible speedup of 6.5x over a single large memory machine. However, when a large memory machine is not available, the distributed system is significantly better than an out-of-core implementation.

The SAND candidate selection stage begins by computing the ideal task size, taking into account the properties of the input data and the given network bandwidth and computational speed of the hardware. The input is divided into subsets, and the corresponding tasks are generated. Each task is sent to a worker along with a sequential executable. When complete, the generated candidates are returned to the master.

The system is fault tolerant in several ways. In the event of a worker or task crash, the master will note the failure and retry the task elsewhere. As tasks are completed, they are noted in a checkpoint file, so restarting the master will allow the workload to continue where it left off. If a worker machine should have less memory than expected, an out-of-core algorithm is used to ensure adequate performance.

We measured the performance of candidate selection on the small, medium, and large datasets on a dedicated cluster, using 1-128 cores, each with a memory limit of 2GB per core. Figure 3 shows how speedup varies with the number of available cores. (Here, speedup is the performance relative to 1 core performing an out-of-core computation using 2GB of RAM.) As can be seen, larger datasets are capable of using more cores efficiently: small achieves a speedup of 22x, medium 50x, and large 109x. By trading computational time

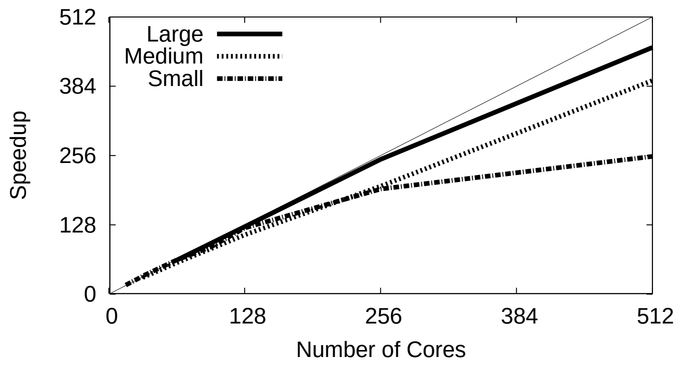


Fig. 4. Scalability of Complete Alignment

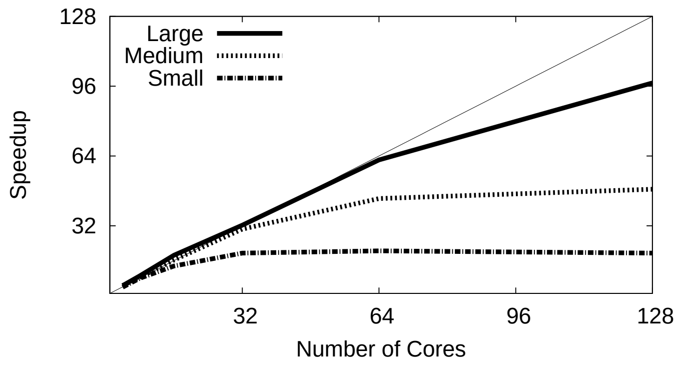


Fig. 5. Scalability of Banded Alignment

for distributed memory capacity, we can achieve significant speedups over sequential execution.

6 Alignment

The alignment stage of SAND takes a library of sequences and a list of candidate sequence pairs generated by the previous stage. The output is a set of overlap records indicating which sequences align well, which is used by the final stages of the assembler. The exact alignment algorithm to be used varies with the biological goals of the research.

The simplest is *complete prefix-suffix alignment* [17], which constructs a dynamic programming matrix, and runs in $O(nm)$ time, where n and m are the lengths of the sequences compared. Complete prefix-suffix alignment is in some sense the most "correct" method of assembly, because it will find the best overlap with the fewest assumptions. Until now, it has been computationally infeasible, so most assemblers like Celera apply heuristics such as *banded alignment*, in which only a portion of the dynamic programming matrix is constructed, given some prior knowledge of diversity in a sample. Below, we will show that SAND makes complete alignment feasible, and makes banded alignment more scalable.

Computing multiple alignments from a single set of reads is more naturally parallel than candidate selection, because each alignment is fully independent and does not involve a time-space tradeoff. On our dedicated cluster, complete prefix-suffix alignment has a CCR of 7KB/CPU-sec, while banded alignment has a CCR of 72KB/CPU-sec. Ignoring queueing effects, a 1Gbit/s network could support no more than 1800 simultaneous workers with banded alignment, and 18,000 with complete alignment. The implementation challenge lies in effectively managing the local state associated with so many tasks and workers.

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 a cluster [20], [28]. 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 1 shows the sequence library and required task data sizes for our four datasets. The task data is 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 resources might not be persistent or 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.

The SAND alignment master 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 sequence data and stores it in a hash table for fast lookup based on the sequence identifier. To prevent task submission latency from limiting effective parallelism, a large number of alignments are grouped together into a single task to be executed by the serial worker code. To decrease total data sent over the network, the candidate list is sorted, so that that pairs sharing a first sequence can easily be grouped together with the shared sequence copied only once. (In practice, we see an average of 1.15 sequences transmitted per alignment.) Once the tasks have been buffered, the alignment program and the input buffer are sent over the network to the worker.

Each worker computes the alignment between the indicated sequences. To minimize data transfer, only aligments better than a user-specified threshhold are returned to the master for persistent storage. Because the master may run for many hours or days, it also tracks the set of tasks completed, so that it can recover and continue after a failure.

While the master's design considerations save on disk space 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.

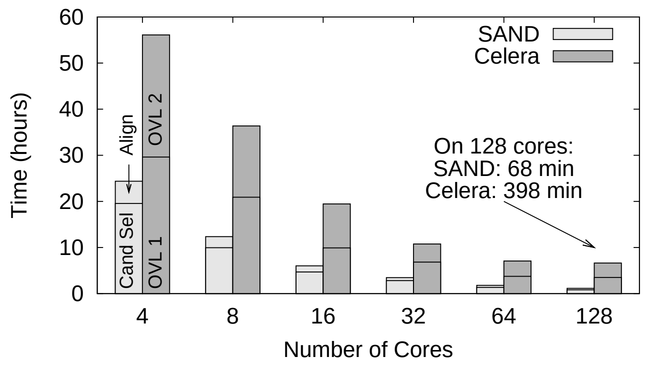


Fig. 6. SAND and Celera Compared

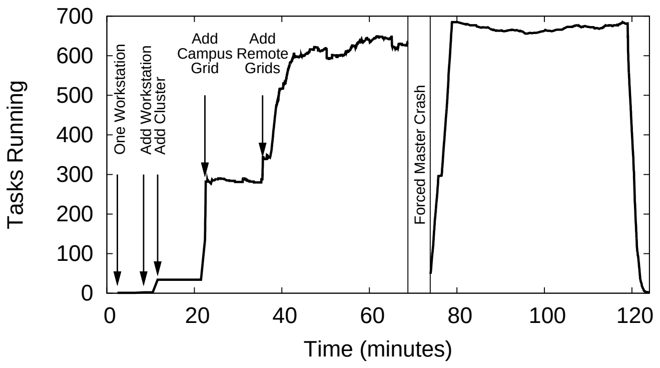


Fig. 7. Scaling SAND from a Workstation to the Grid

However, these memory requirements are much less than those of the hash table used in candidate selection.

Figure 4 shows the scalability of the alignment stage on our dedicated cluster, using complete prefix-suffix alignment. Speedup is measured relative to the time to complete all alignment using SAND in sequential mode. On the small dataset, SAND achieves close to linear speedup on 128 cores, but drops to about 50 percent efficiency by 512 cores. As the dataset sizes increase, scalability improves, showing 80% efficiency at 512 cores on the medium dataset, and 90% efficiency at 512 cores on the large dataset. This is relatively easy to achieve due to the high CCR of complete alignment.

Figure 5 shows the scalability of alignment using the banded alignment algorithm on the same dedicated cluster. The banded algorithm does less work than the complete algorithm, so it has a higher CCR. The banded algorithm peaks at 20x speedup on the small dataset, 50x on the medium, and 100x on the large.

Figure 6 shows the performance of SAND against the equivalent first two stages of the Celera Assembler on our dedicated cluster. The performance of Celera flattens out after 64 cores, due to the overhead of transmitting the entire sequence library to every node in the computation. At 128 cores, SAND is 5.8X faster than Celera, and does not require the use of a shared filesystem, making it easier to obtain resources for the computation. Similar results are obtained with the small and medium datasets.

As this article went to press, version 7.0 of the Celera assembler was released. To test the sensitivity of SAND to the Celera implementation we ran the small dataset through Celera 5.4, 6.1 and the newest 7.0 version. The assembly results were nearly identical using SAND-created overlaps; however, we noticed that newer versions of Celera (6.1 and 7.0) required roughly 1.6X more time for the consensus step when using SAND-created overlaps. The observed constant difference was consistent running the medium mosquito dataset (2,779 seconds vs. 1,578 seconds) with CA7.0. Therefore, SAND can easily be used with all three Celera versions with only a modest performance hit in the consensus stage.

7 SAND ON CLUSTERS, CLOUDS AND GRIDS

This far, we have evaluated SAND on a dedicated cluster to provide consistent performance results. However, SAND is designed to function in uncontrolled wide-area systems encompassing clusters, clouds, and grids. Section 9.2 in the online supplement details many of the techniques necessary in this environment.

Figure 7 demonstrates that SAND can transparently scale up from a single workstation to a wide-area distributed system, handling both failures and resource variations. One author started the SAND alignment master on his workstation, with one worker running. After a few minutes, he asked a co-worker to start a worker on her workstation, and then submitted some workers 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 Condor pool 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 cores in use at once. 7998 tasks ran at Notre Dame, 7760 at Purdue, and 1232 at Wisconsin.

Next, we demonstrate that SAND can effectively complete an assembly of the human genome [43] consisting of 31M sequences totalling 20 gigabases. To accomplish this, we submitted workers to our campus Condor pool, where approximately 1000 cores were available at any given time. The candidate selection stage generated 327M candidate pairs in 11 hours, using up to 413 workers at once, with an overall speedup of 274x. (This is very close to the runtime predicted in Figure 10 in the online supplement.) A complete prefix-suffix alignment completed in 2.5 hours using up to 1015 workers, for a speedup of 952x. For comparison, we also ran a banded alignment, which completed in one hour using up to 245 workers for a speedup of 175x. SAND is able to effectively harness large, unreliable systems to complete the assembly in about half a day.

To verify that SAND functions correctly in a cloud environment, we used the Amazon Elastic Computing Cloud service to allocate 100 cores using "large" instances each with 7.5 GB RAM. The virtual machines were brought online, and the

	Cores	SAND Selection	SAND Alignment
Cloud - WAN	100	1445 sec	565 sec
Cloud - LAN	100	35 sec	130 sec
Campus Grid	100	55 sec	176 sec

Fig. 8. SAND Performance on Cloud

SAND worker started on each. We tested the performance of SAND on the small dataset in three configurations: with workers in the cloud and the master at Notre Dame (Cloud - WAN), with workers in the cloud and the master in the same cloud (Cloud - LAN), and using our Condor-based campus grid (Campus Grid). Table 8 summarizes the results. The Cloud-LAN configuration achieves performance very similar to that of our campus grid, however, the Cloud-WAN configuration pays a significant penalty for using the wide area network, which is generally discouraged by cloud providers.

8 Related Work

Parallel approaches for deBrujn-based assembly ([21] and [23]) use a different assembly graph that is ideal for short reads, but relies heavily on error correction techniques [21]. Here, we focus on the different problem of overlap-based assembly ideal for long reads that will remain important; third-generation instruments will soon generate error-prone reads on the order of tens of thousands of characters (see [11]).

Because determining overlaps between candidates has been the most time intensive step of an overlap-based assembly, it was 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 [20]. 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 [22]. More recent work has explored usiong FPGAs [40] and the Cell processor [37] to speed up alignment, which would provide up to a 100X speedup. Systems such as CloudBurst [38] have applied the Map-Reduce [14] data-parallel computation model to similar bioinformatics problems on similar distributed systems.

The parallel solutions to genome assembly have relied on on batch processing, complex programming paradigms or specialized hardware. In contrast, we are interested in a growing trend to develop modular genome assembly components such as the UMDOverlapper [34] but to enable them to run on more heterogeneous systems composed of cores from clusters, clouds, grids. Here, we extend this modular design concept [31] to facilitate candidate selection and alignment modules that are highly adaptable to many types of distributed resources. Note that although we have used Celera Assembler as the basis for these results our scalable framework can be used in other assembly systems as done by related frameworks [34]. We believe improvements in our framework can accomodate custom assembly algorithms along with state-of-the-art updates in Celera reported in [26].

The general concept of executing bag-of-tasks applications on a master-worker framework is well established. A number of software systems have been designed for harnessing the idle cycles of computers to execute bag-of-tasks applications, including Condor [42], SETI@Home [41] and its generalized descendant BOINC [4], Piranha [15], and Entropia [10], to name a few. We have demonstrated SAND running on Condor and SGE, but there is no fundamental barrier to running on top of any of these other systems.

There exist a number of application level frameworks for bag-of-tasks applications. Perhaps the earliest is Linda [3], which offered three primitives that could be added to any general purpose language. With the addition of a conditional-swap operator, Linda programs can be made fault-tolerant [5]. More recently, we have seen the development of Condor-MW [25], a C++ application framework which has been used to solve large scale optimization problems. APST [9] and MyGrid [12] are designed to attack large parameter sweep problems using existing executables, such as monte carlo studies on biological simultations. Falkon [32] is also a master-worker framework specialized for very rapid task execution, and applies the techinque of data diffusion [33] to improve input performance on data intensive tasks. SAND shares the same underlying architecture as these systems, but plays a greater role in the decomposition of the problem into suitable tasks. To our knowledge SAND is the first application of bag-of-tasks concepts to the problem of production-quality genome assembly at scale on a diverse collection of resources.

In recent years, the theory of master-worker computing has placed much attention on the scheduling of tasks to processors. For example, Rosenberg [36] considers a system of homogeneous performance but varying failure probability, and shows how to generate schedules arbitrarily close to optimal. Banino et al [6] consider arbitrary networked systems with heterogeneous CPU and network performance: the scheduling problem is NP-hard in the general case, but can be made tractable by considering the steady state. To avoid the limitations of a single master (such as we have shown above), Beaumont et al [8] consider how to schedule tasks using hierarchical masters. This problem is also NP-hard when buffer sizes are limited, so several heuristics of varying quality are presented and evaluated.

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 National Science Foundation grants CNS06-43229 and CNS08-55047.

We thank the staff at the Purdue Rosen Center for Advanced Computing and the Wisconsin Condor Team for sharing their computing resources to make this work possible. We thank Dinesh Rajan for assisting with the testing on the Amazon cloud. We thank the anonymous reviewers for their comments and suggestions.

The SAND software is licensed under the GNU General Public License and is available for download at `http://www.nd.edu/~ccl/software/sand`, along with the datasets used in this paper.

REFERENCES

[1] TeraGrid. http://www.teragrid.org.

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

[3] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.

[4] D. Anderson. BOINC: A system for public-resource computing and storage. In *Proc. IEEE/ACM Workshop on Grid Computing*, 2004.

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

[6] C. Banino, O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Scheduling strategies for master-slave tasking on heterogeneous processor platforms. *IEEE Trans. on Parallel and Distributed Systems*, 15:319–330, April 2004.

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

[8] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, L. Marchal, and Y. Robert. Centralized versus distributed schedulers for bag-of-tasks applications. *IEEE Trans. on Parallel and Distributed Systems*, 19(5):698–709, May 2008.

[9] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The AppLeS parameter sweep template: user-level middleware for the grid. In *Proc. of ACM/IEEE Supercomputing*, 2000.

[10] A. Chien, B. Calder, S. Elber, and K. Bhatia. Entropia: Architecture and performance of an enterprise desktop grid system. *Journal of Parallel and Distributed Computing*, 63:597–610, May 2003.

[11] C.-S. S. Chin, J. Sorenson, J. B. Harris, W. P. Robins, R. C. Charles, R. R. Jean-Charles, J. Bullard, D. R. Webster, A. Kasarskis, P. Peluso, E. E. Paxinos, Y. Yamaichi, S. B. Calderwood, J. J. Mekalanos, E. E. Schadt, and M. K. Waldor. The origin of the Haitian cholera outbreak strain. *The New England Journal of Medicine*, 364(1):33–42, Jan. 2011.

[12] W. Cirne, D. Paranhos, L. Costa, E. Santos-Neto, F. Brasileiro, J. Sauve, F. Silva, C. Barros, and C. Silveira. Running bag-of-tasks applications on computational grids: the MyGrid approach. In *Proc. of Intl. Conf. on Parallel Processing (ICPP)*, October 2003.

[13] 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.

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

[15] D. Gelernter and D. Kaminsky. Supercomputing out of recycled garbage: preliminary experience with Piranha. In *Proc. International Conference on Supercomputing*, 1992.

[16] 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.

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

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

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

[20] 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.

[21] B. Jackson, P. Schnable, and S. Aluru. Parallel short sequence assembly of transcriptomes. *BMC Bioinformatics*, 10(Suppl 1):S14, 2009.

[22] 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).

[23] V. K. Kundeti, S. Rajasekaran, H. Dinh, M. Vaughn, and V. Thapar. Efficient parallel and out of core algorithms for constructing large bi-directed de Bruijn graphs. *BMC Bioinformatics*, 11:560+, 2010.

[24] M. K. N. Lawniczak, S. J. Emrich, A. K. Holloway, A. P. Regier, M. Olson, B. White, S. Redmond, L. Fulton, E. Appelbaum, J. Godfrey, C. Farmer, A. Chinwalla, S.-P. Yang, P. Minx, J. Nelson, K. Kyung, B. P. Walenz, E. Garcia-Hernandez, M. Aguiar, L. D. Viswanathan, Y.-H. Rogers, R. L. Strausberg, C. A. Saski, D. Lawson, F. H. Collins, F. C. Kafatos, G. K. Christophides, S. W. Clifton, E. F. Kirkness, and N. J. Besansky. Widespread Divergence Between Incipient Anopheles gambiae Species Revealed by Whole Genome Sequences. *Science*, 330(6003):512–514, 2010.

[25] 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.

[26] J. R. Miller, A. L. Delcher, S. Koren, E. Venter, B. P. Walenz, A. Brownley, J. Johnson, K. Li, C. Mobarry, and G. Sutton. Aggressive assembly of pyroseqencing reads with mates. *Bioinformatics*, 24(24):2818–2824, 2008.

[27] C. Moretti, M. Olson, S. Emrich, and D. Thain. Highly Scalable Genome Assembly on Campus Grids. In *IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS09)*, 2009.

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

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

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

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

[32] 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.

[33] I. Raicu, Y. Zhao, I. Foster, and A. Szalay. Accelerating large-scale data exploration through data diffusion. In *Proc. Workshop on Data Aware Distributed Computing*, 2008.

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

[35] M. Roberts, A. V. Zimin, W. Hayes, B. R. Hunt, C. Ustun, J. R. White, P. Havlak, and J. Yorke. Improving phrap-based assembly of the rat using reliable overlaps. *PLoS ONE*, 3:e1836, 2008.

[36] A. L. Rosenberg. Optimal schedules for cycle-stealing in a network of workstations with a bag-of-tasks workload. *IEEE Transactions on Parallel and Distributed Systems*, 13(2):179–191, February 2002.

[37] A. Sarje and S. Aluru. Parallel biological sequence alignments on the cell broadband engine. In *Parallel and Distributed Processing, 2008. IPDPS 2008.*, pages 1–11, April 2008.

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

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

[40] O. Storaasli and D. Strenski. Exploring accelerating science applications with FPGAs. In *Third Annual Reconfigurable Systems Summer Institute (RSSI)*, July 2007.

[41] W. T. Sullivan, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye, and D. Anderson. A new major SETI project based on project serendip data and 100,000 personal computers. In *5th International Conference on Bioastronomy*, 1997.

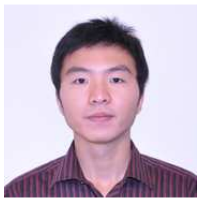
[42] D. Thain, T. Tannenbaum, and M. Livny. Condor and the grid. In F. Berman, G. Fox, and T. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley, 2003.

[43] J. C. Venter et al. The sequence of the human genome. *Science*, 291(5507):1304–1351, February 2001.

[44] 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.

Christopher Moretti graduated from the College of William and Mary in 2004 with a B.S. in Computer Science. He received the Ph.D. in Computer Science and Engineering from the University of Notre Dame in 2010. He is currently a lecturer in the Department of Computer Science at Princeton University.

Andrew Thrasher graduated from Anderson University in Indiana in 2009 with a B.A. in Computer Science, Mathematics and Physics. He is currently working towards a Master's degree in Computer Science at the University of Notre Dame.



Li Yu graduated from Huazhong University of Science and Technology in 2006 with a B.S. in Computer Science and in 2008 with a M.S. in Computer Science. He is currently a Ph.D. student at the University of Notre Dame, researching distributed computing systems.

Michael Olson graduated from St. Olaf College in 2004 with a B.A. in Mathematics.He received a Master's of Science in Computer Science and Engineering from the University of Notre Dame in 2010. He is currently the chief software architect at CenterX, a healthcare IT startup.

Scott Emrich received the B.S. in Biology and Computer Science from Loyola College in Maryland and the Ph.D. in Bioinformatics and Computational Biology from Iowa State University. His research interests include computational biology, bioinformatics and parallel computing, including arthropod genome analysis with applications to global health and ecology.

Douglas Thain received the B.S. in Physics from the University of Minnesota and the Ph.D. in Computer Sciences from the University of Wisconsin - Madison. He is currently an Associate Professor of Computer Science and Engineering at the University of Notre Dame, where his research focuses on scientific applications of distributed computing systems.

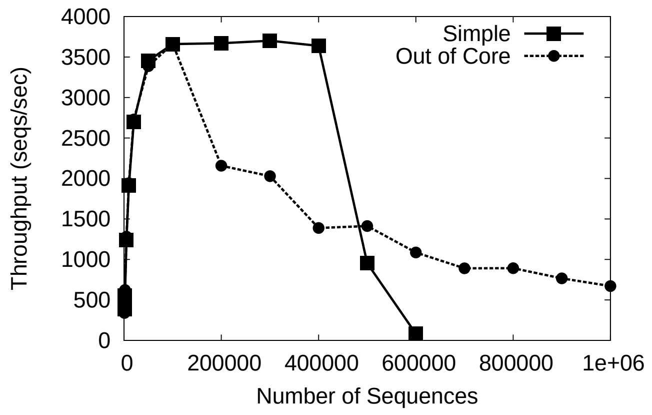


Fig. 9. Performance of Sequential Candidate Selection

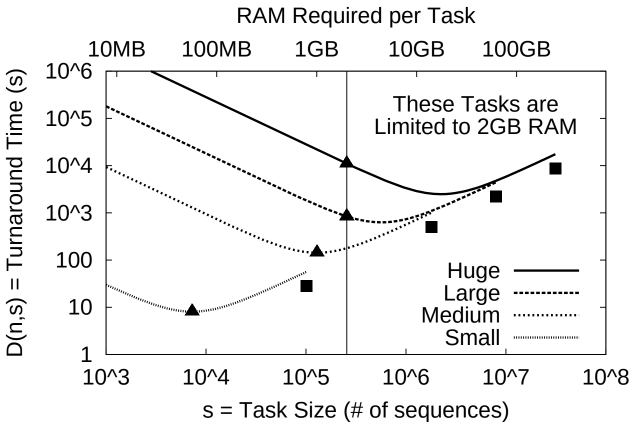


Fig. 10. Performance Tradeoffs in Candidate Selection

9 Online Supplement

9.1 Candidate Selection

Figure 9 shows the performance of our sequential candidate selection module. The simple implementation just creates a hash table and loads sequences as described above. When memory is exhausted, performance drops essentially to zero. The out-of-core implementation begins to drop off earlier, but maintains degraded performance as the number of sequences increases. In both cases, the peak performance is 3600 sequences per second when 100,000 sequences are held in memory.

To derive the optimal number of sequences per task for the distributed implementation:

- n is the total number of sequences in a genome.
- L is the average length of a sequence in bases.
- C is the number of sequences per second that can be processed when everything fits in memory.
- B is the network bandwidth in bytes per second.
- s is the number of sequences per task.

If the entire problem can fit in memory, than the time to process a complete genome sequentially is n/C. If we split the problem up into pieces of size s on each side, as shown in Figure 2, then each task will require $2s$ sequences as input and complete in $2s/C$ time. However, there are now n^2/s^2 tasks to be completed. Four bases can be packed into a byte, so each sequence will take $L/4B$ time to transfer. Ideally, it will take $(n^2/s^2) * (2s) * (L/4B)$ to transfer all input data to every task, and (assuming more nodes than tasks) all tasks will complete in parallel in $2s/C$ time. Putting it together, the turnaround time for the distributed implementation is:

$$D(n,s) = \frac{Ln^2}{2Bs} + 2s/C \tag{1}$$

Figure 10 shows the ideal execution time $D(n,s)$ for each each of our four datasets, as s varies over several orders of magnitude. (We take n and L from Table 1 and assume $C = 3600$ seqs/s and $B = 1$ Gigabit/sec.) The square at the end of each curve shows the sequential execution time, assuming a single host has enough memory for the entire task. A triangle is placed on each curve to indicate the best execution time, within the constraints of available memory. The minimum of each curve occurs when:

$$s_{min} = n\sqrt{\frac{LC}{4B}} \approx 0.04418n \approx n/22 \tag{2}$$

For example, the medium dataset achieves best performance when s is 117,562, yielding 484 total tasks. However, if we consider the huge dataset, the situation is somewhat different. Running on a single node would require over 119GB of RAM. The minimum of the curve is at about $s = 1,420,000$, but this would require 12GB of RAM per node. If we assume a maximum of 2GB RAM per node, then s can be no greater than $256,000$.

In all cases, the best possible speedup of the distributed algorithm at $s = s_{min}$ compared to the sequential algorithm on a large memory machine is:

$$\text{Speedup} = \frac{n/C}{D(n,s_{min})} = \frac{2B}{LC + \sqrt{4BLC}} \approx \mathbf{6.5} \tag{3}$$

Of course, a large memory machine is not always available, so the distributed algorithm is much better than an out-of-core alternative.

9.2 Scaling Techniques

The results presented in Sections 5 and 6 are performed on a single dedicated cluster to provide consistent results. However, running very large assemblies requires that we scale up to the uncontrolled environment of multiple clusters, clouds, and grids. In this section, we demonstrate those challenges and solutions by harnessing workers from multiple Condor pools located at the University of Notre Dame, Purdue University, and the University of Wisconsin. Such a wide area system consists of multiple routed networks, changes on a daily basis, and is shared with multiple users, so it is by nature dynamic and uncontrolled. The results in this section serve to demonstrate that SAND can function correctly in such a hostile environment.

Pipelining. Above, we presented the candidate selection and alignment steps independently. In practice, the two steps can be pipelined, because the aligner can begin constructing

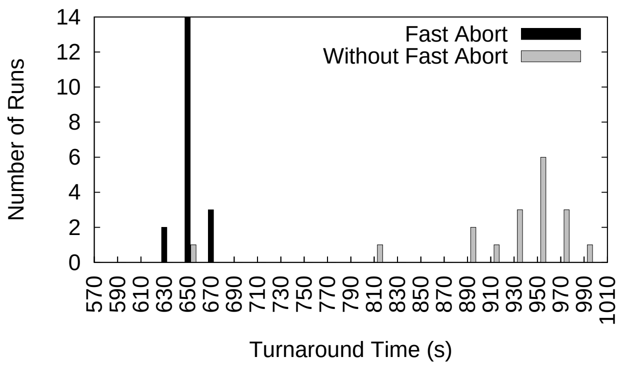


Fig. 11. Preventing Long Tails with Fast Abort

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. This is accomplished by having each worker job submitted for the candidate selection run two copies of the worker, one after the other. The first points to the candidate selection master and the second points to the alignment master When candidate selection finishes, it tells all its workers to shut down and the second command begins, allowing those additional workers to work for the alignment master. Using this method can improve overall runtime. 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 each stage finished in 654 seconds.

Long Tails. All candidate pairs are independently computable, and thus during a workload even very slow machines do useful work. At the end of a workload, however, slow machines may take work that would be completed faster on other available resources. In the worst case, this can hold up completion of the workload significantly and cause a long-tail effect at the end of the workload.

Because it is designed for heterogeneous environments, the Work Queue infrastructure has a mechanism to preemptively abandon jobs that are taking too long – *fast abort*. This is critical in workloads where later computation is dependent on earlier computation [44]. We aim to avoid long-tails while still allowing slow machines to contribute by activating fast abort only in the finishing stages of a workload.

In order to evaluate the fast abort mechanism, we allocated 64 nodes from our dedicated cluster, in which one of those nodes was handicapped to take 5-10x longer to complete tasks than the other nodes. This environment is much more prone to delays in the workload due to a single very slow node.

Figure 11 shows a histogram of completion times for 38 workloads with the small dataset. The light boxes show counts of workloads in which fast abort is not activated and the dark boxes are counts of those in which fast abort was activated after all tasks had been submitted. Though variations in workload timings didn't result in long tails every time without fast abort, it is clear that a significant amount of the trials took much longer to complete. Upon inspection, this delay resulted from having one remaining task being computed on the slow worker while all other workers were idle. The version with fast abort enabled to cut off a worker after it has exceeded the average completion time by 50% does not suffer from these extreme tails.

Another approach would be to replicate tasks at the end of a workload, as in [13] and [14]. To a first approximation, replication and fast abort achieve the same result, althought future work may determine which is more efficent.

Waiting for Out-of-Core Task Data. In our first implementation, complete alignment saw a marked decrease in performance at 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.

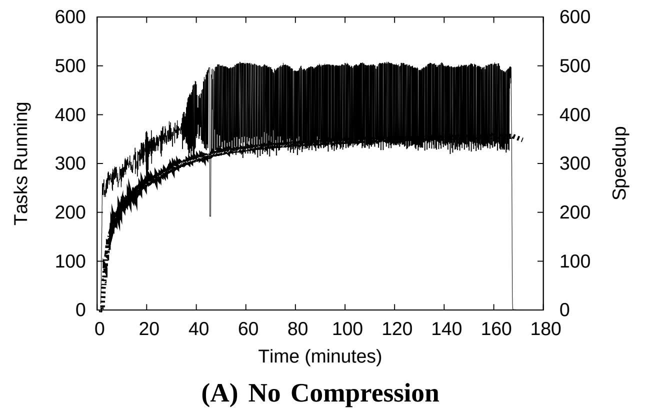
With 512 workers, the additional buffered tasks caused the master to exceed physical memory. When the master began to need paging for its task management, performance began to degrade. The effect of this can be seen in Figure 12(A). Because it takes significantly longer to create the number of tasks 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 compressed the data. Because DNA consists of only 4 letters, we may represent a single base with two bits. The master does not need to interpret the sequence data itself, so the data stays compressed on disk, in the master's memory, and while in transit to the worker nodes. It is uncompressed as a side effect of loading into the memory of the alignment program.

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.

This continues to be necessary even as resources scale up with the data set sizes. For example, the master for the human dataset was run on a machine with 32GB of RAM, which was enough for its 20GB requirement.

Waiting for Task Assignment. When a master has many workers connected to it, it takes the master longer to assign tasks to all the workers in round-robin fashion. If task assignment is slow, it takes the master longer to assign tasks to all workers in the pool 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. In this case, the main problem is waiting for the master to transfer task data to every worker. To induce this problem, we began running workers on Condor pools at Purdue University and



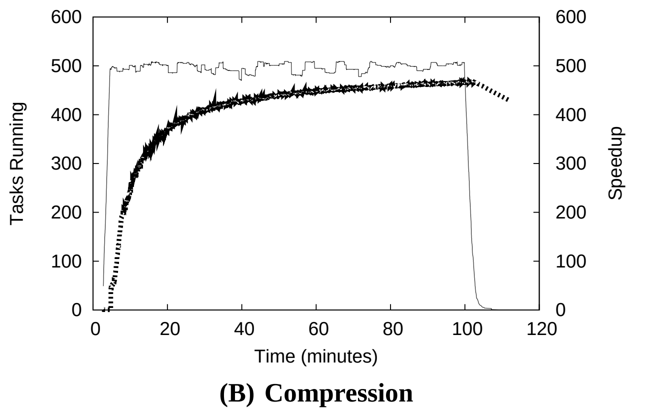
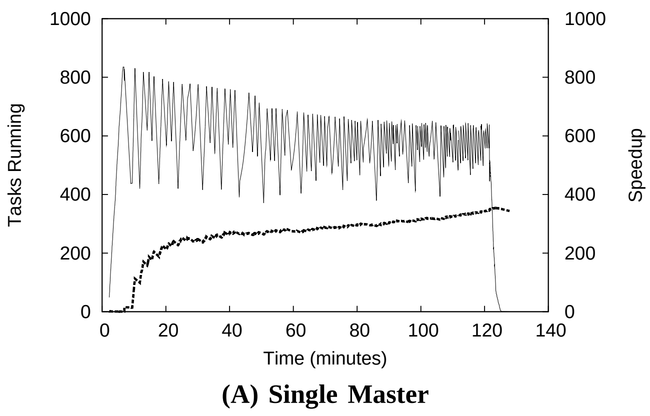


Fig. 12. The Effect of Data Compression.
These graphs show the effect of data compression on the master's ability to dispatch tasks using the large dataset. Each shows a timeline of a single run, with the number of tasks running, the cumulative speedup, and the percent complete over time. Figure 12(A) 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.



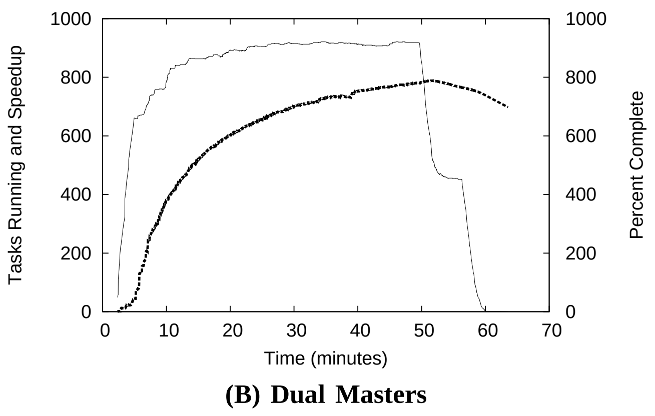


Fig. 13. The Effect of Splitting Masters.
When using a sufficiently large number of workers on the large dataset, the master does not have enough network bandwidth to keep all of them busy. These figures show a timeline of a single run with approximately 950 workers using one master (A) and two masters (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 50 min.

the University of Wisconsin.

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

Ideally more nodes with fast connections could be added in place of machines at other institutions, however this is not always feasible. In order to take advantage of remote computing resources that have slower network transfer times without compromising the efficiency of our workload, we divide the workers (including the slow connections) between two controlling masters. To do this, we split the list of candidate pairs in half and run the master program on two separate machines with the same list of sequences but different halves of the candidate pairs list. 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.

The multiple-master technique is not limited in application to workloads with large number of workers with slow connections. Various other system resources limitations can cause workers to experience starvation even if network speeds are fast enough to support all workers. For instance, many Linux systems have hard limits on file descriptors open by a process (usually 1024), and users might not have permission to increase this limit. Using multiple masters multiplies the number of connections, and thus supportable workers.

We have shown that there is benefit significant benefit to manually splitting into two masters, but a more automatic division of work would be better. Future work may use, for example, the scheduling algorithm developed by Beaumont et al [8] for hierarchical masters.