7-2012

An Energy-Aware Bioinformatics Application for Assembling Short Reads in High Performance Computing Systems

Julia Warnke
University of Nebraska at Omaha

Sachin Pawaskar
University of Nebraska at Omaha, spawaskar@unomaha.edu

Hesham Ali
University of Nebraska at Omaha, hali@unomaha.edu

An Energy-Aware Bioinformatics Application for Assembling Short Reads in High Performance Computing Systems

Julia Warnke, Sachin Pawaskar and Hesham Ali
College of Information Science and Technology
University of Nebraska at Omaha
Omaha, Nebraska 68182
hali@unomaha.edu

ABSTRACT*—Current biomedical technologies are producing massive amounts of data on an unprecedented scale. The increasing complexity and growth rate of biological data has made bioinformatics data processing and analysis a key and computationally intensive task. High performance computing (HPC) has been successfully applied to major bioinformatics applications to reduce computational burden. However, a naïve approach for developing parallel bioinformatics applications may achieve a high degree of parallelism while unnecessarily expending computational resources and consuming high levels of energy. As the wealth of biological data and associated computational burden continues to increase, there has become a need for the development of energy efficient computational approaches in the bioinformatics domain. To address this issue, we have developed an energy-aware scheduling (EAS) model to run computationally intensive applications that takes both deadline requirements and energy factors into consideration. An example of a computationally demanding process that would benefit from our scheduling model is the assembly of short sequencing reads produced by next generation sequencing technologies. Next generation sequencing produces a very large number of short DNA reads from a biological sample. Multiple overlapping fragments must be aligned and merged into long stretches of contiguous sequence before any useful information can be gathered. The assembly problem is extremely difficult due to the complex nature of underlying genome structure and inherent biological error present in current sequencing technologies. We apply our EAS model to a newly proposed assembly algorithm called Merge and Traverse, giving us the ability to generate speed up profiles. Our EAS model was also able to dynamically adjust the number of nodes needed to meet given deadlines for different sets of reads.*

KEYWORDS*-Energy aware scheduling; high performance computing; next generation sequencing; genome assembly*

I. INTRODUCTION

Since its inception in the mid 2000's, next generation sequencing has produced massive amounts of genetic information, making a large impact on numerous research fields. As next generation sequencing systems and centers become more readily available, massively parallel sequencing has become the cornerstone of many diverse research endeavors, including those such as cancer transciptome and gene expression analysis studies [1] and microbiomics [2]. Next generation sequencing technologies are capable of producing millions to even billions of short reads per run. Individually each read represents only a fraction of the original genome and provides no information in itself. However, sequencing reads are produced at a high coverage of the original genome such that many of these reads overlap with one another. Relationships between overlapping sequence reads assist the identification of fragments that are consecutive within the genome, allowing the recursive merging of these overlapping sequences until long stretches of contiguous genetic data, known as contigs, are recovered.

The assembly of next generation sequencing data still remains a challenging task due to the massive size of read datasets, short read lengths, and underlying target sequence composition such as repeat content. The assembly of short reads produced by these devices is a critical and computationally intensive process. Fortunately, many steps of this process are good candidates for parallel computing. The parallel implementation of the read overlap detection phase of assembly is relatively straightforward. High performance computing has been successfully applied to help reduce the computational burden of detecting read overlaps in large datasets [3]. However, straightforward parallel applications developed for overlap detection could achieve an unnecessary high degree of parallelism at the expense of significant energy consumption.

In this paper we introduce an energy-aware scheduling (EAS) model that takes both deadline and energy usage requirements into consideration. We use this EAS model to run the overlap detection algorithm of a newly developed assembly program,

called Merge and Traverse. We conduct multiple experiments to evaluate the computational resources needed to complete the overlapping process while balancing task deadline requirements with energy minimization. These experiments demonstrate the viability of the proposed energy-aware scheduling model and characterize the impact of various parameters on program runtime.

II. Energy Aware Scheduling

Scheduling is a classical field with several interesting problems and results. Due to its wide range of applications, the scheduling problem has been attracting many researchers from a number of different fields. A scheduling problem emerges whenever there is a choice. This choice could be the order in which a number of tasks can be performed and/or in the assignment of those tasks for processing. In general, the scheduling problem assumes a set of resources and a set of consumers serviced by these resources according to a certain policy. Given a set of customers, resources, and constraints, a solution to the scheduling problem attempts to find an efficient policy (schedule) for customer access to resources while optimizing some desired performance measure such as the total service time (schedule length).

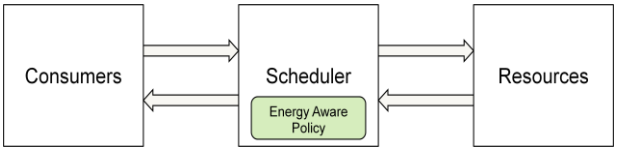


Figure 1. Energy Aware Scheduling System

Over the years several methods have been used to address the sequencing problem including complete enumeration, heuristic rules, integer programming, and sampling methods. It is clear that complete enumeration is impractical because the problem is exponential; hence optimal solutions cannot be obtained in real time [4, 5]. However, many heuristic methods have been successfully applied to most general cases of the scheduling problem. Such methods include traditional priority-based algorithms [6], task merging techniques [7], critical path heuristics [6, 8]. In addition, distributed algorithms have been designed to address different versions of the scheduling problem [9].

Energy aware scheduling is a special case of the general scheduling problem in which our scheduling policy is the optimization of energy in HPC systems or battery power in mobile devices. Minimizing the power utilization which is directly proportional to costs becomes the most important consideration in a system that is energy aware. At the same time this system must still meet other specified parameters such as task deadlines.

Simply put, an energy aware scheduling system is a scheduling problem that assumes a set of resources and a set of consumers serviced by these resources according to an energy aware policy. Given a set of customers, resources, and constraints, a solution to the energy aware scheduling problem attempts to find an efficient policy for customer access to resources while optimizing battery power utilization. Accordingly, an energy aware scheduling system can be considered to consist of a set of consumers, a set of resources, and an energy aware scheduling policy as shown in figure one. Clearly, there is a fundamental similarity to scheduling problems regardless of the difference in the nature of the tasks and the environment.

III. Assembly Algoithm Overview

The Merge and Traverse assembler follows the traditional overlap-layout-consensus paradigm that has been successfully employed by various assemblers [3] [10] [11]. Our algorithm assembles reads into contigs in three stages: 1) overlap detection and alignment, 2) graph construction and manipulation, and 3) consensus sequence generation by multiple alignment [12].

A. Overlap Detection and Alignment

The Merge and Traverse algorithm uses short k-mer words to seed overlaps between reads. These short seed matches are extended into full alignments using dynamic programing. The overlap relationships found during the overlapping phase are placed into two categories by the assembly algorithm. The first type of overlap that the assembly algorithm considers is the dovetail overlap. The dovetail overlap occurs when the reads

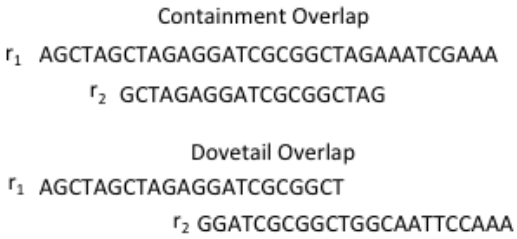


Figure 2. Read Overlaps

align such that they form a suffix-prefix relationship as shown in figure two.

The second type of overlap that the assembly algorithm considers is the containment overlap. The containment overlap occurs when the sequence of one read is fully contained in another read. For the purpose of simplifying the overlap graph in subsequent assembly phases, our algorithm disregards containment overlap relationships. Each read that is contained in one or more other reads is mapped to a suitable representative read using a clustering approach detailed in section four.

B. Graph Construction and Manipulation

The second phase of the assembly process builds an overlap graph using high quality dovetail overlaps between the

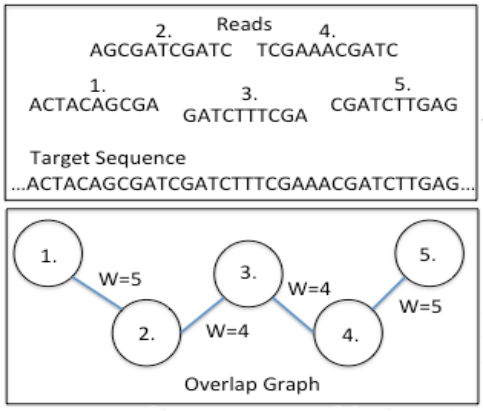


Figure 3. The overlap graph. Reads map to nodes. Overlaps map to edges. Each edge is assigned a weight representing the length of the overlap shared between the reads.

remaining representative reads. In this graph theoretic model, each node represents a sequencing read. An edge joins two nodes if their corresponding reads overlap.

After graph construction is complete, the algorithm preforms transitive reduction of the graph [13] revealing non-branching paths that likely correspond to unique regions of the target sequence being assembled. The algorithm identifies and merges these non-branching paths into super-nodes in the overlap graph. Remaining graph structural features such as dead-end paths and bubbles, where two paths start and end at a common node, are in many cases caused by sequencing error present in the read data set. The algorithm identifies this noise using a Dijkstra shortest path method. Each dead-end path that is shorter than a user-provided threshold is removed from the overlap graph. For each bubble whose component paths are shorter than the user-provided threshold, the least covered path in the bubble is removed. After graph trimming is complete, the algorithm extracts all maximal non-branching paths from the graph for use in the consensus phase of the assembly process to construct contigs.

C. Consensus Sequence Generation

In the final consensus phase, progressive multiple alignment guided by the read path layout is used to determine contig consensus sequence.

IV. Read Overlap Detection

In this section, we provide a description of our three-step approach for read overlap detection. The first step orders a read dataset *S* in descending read length and partitions it into subsets. The second step maps each read that forms a containment overlap with one or more other reads to a suitable representative read following a hierarchical clustering scheme introduced by CD-Hit [13]. After clustering is complete, the final step identifies dovetail overlap relationships among the remaining representative reads.

A. Read Preprocessing

The containment clustering step of the overlap detection phase requires that the reads are sorted by descending length. First the reverse complements of an input read dataset R are generated to form the read set $S = (R, \overline{R})$. It then sorts S into descending order of length by a merge sort algorithm, and partitions S into n subsets = $\{S_0, S_1, \ldots S_{n-1}\}$ of size m, where n is specified by the user. Each read subset S_k is sorted in descending read length and the subsets are ordered such that $readLengths(S_0) \geq readLengths(S_1) \geq \ldots \geq readLengths(S_{n-1})$.

B. Containment Clustering

The initial read clustering step follows the greedy hierarchical clustering scheme introduced by the CD-hit algorithm [14]. The longest read becomes the first representative. It is used to search for containment overlaps among the remaining reads using the exact matching and alignment methods described in the section three. If a read forms a containment overlap with the current representative and its alignment meets minimum length and alignment identity requirements, it is mapped to that representative read. The algorithm considers each read in the order of descending length. If a read is not already mapped to an existing representative, it becomes a new representative read and is used to query the remaining reads in the dataset for containment overlaps. A read that has been mapped to a previous representative read but forms a containment overlap with the current representative is remapped to the current representative if its alignment identity with the current representative is greater than its alignment identity with the previous representative. After this process has completed, all

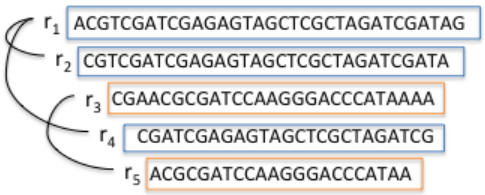


Figure 4. Containment clustering. Reads two and four cluster to read one, and read five clusters to read three.

read to representative mappings are recorded for use in the consensus phase of the assembly process.

C. Dovetail Overlaps

After containment clustering is complete, the remaining representative reads are used to query the read dataset for dovetail overlaps with other representative reads. The exact matching and alignment methods of section three are used to locate dovetail overlap relationships. If a dovetail overlap meets minimum alignment length and alignment identity requirements, it is recorded for use in the graph construction phase of the assembly algorithm.

D. Implementation Details

The containment clustering and dovetail overlapping steps accept two read subsets S_i and S_j as input. The subset S_i is the query dataset and the subset S_j is the reference dataset, where $i \leq j$.

To facilitate the identification of exact matches between reads, a suffix array constructed by Larsson and Sadakane's algorithm [15] is used to index the reference dataset. In succession, each read in the query dataset is broken into all of its possible subwords of size k (denoted as k-mers). These k-mers are used to query the suffix array for exact matches. If one or more exact matches are found between the query read and a reference read indexed by the suffix array, then both reads are passed to an alignment algorithm for evaluation. The k-mers shared by the reads are chained [16] and the Needle-Wunsh algorithm [17] is used to align the regions between k-mers and to align the beginning and end regions of the reads.

After the alignment of the two reads is complete, the computed overlap is evaluated by its alignment length and alignment percent identity. If the overlap does not meet the user-provided minimums for these measurements, it is not included in subsequent steps of the assembly process.

Since the containment clustering step is dependent on the read ordering, each subset S_j must be ran against each S_i as a reference dataset, where $i < j$, before it can be used as a query dataset against any other read subset. The dovetail-overlapping step is not dependent on read ordering and can accept read subsets in any order.

V. Parallel Implementation and EAS Model

The input read dataset S is partitioned into *n* subsets = $\{S_0, S_1, \ldots S_{n-1}\}$ of size *m* during the initial read sorting and preprocessing step. A master thread sends each unique subset combination of size two as input to worker processors running serial versions of the containment clustering and dovetail overlapping algorithms. The master thread manages the execution order constraints of the containment clustering step.

A. Solution Overview

The EAS engine runs the pre-processor on the input fasta file, the output of which is the n-split read subsets. Let us assume that the large file has m sequences, and then each of the smaller files will contain (m/n) sequences in sorted order.

The files created in the pre-processing step become inputs to the EAS engine. The EAS engine runs the alignment program in a 2-step process. The first step finds the containment overlaps and the second step determines the dovetails overlaps among the remaining representative reads. The containment part of the execution is not naively parallel; the execution of certain pairs of subsets (tasks) has to be done in order, only then can dependent subsets be processed. The main process flow is shown in figure six below.

B. Containment Execution – Step 1

The execution dependencies are shown in figure seven for the following set of containment tasks T = {(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 1), (1, 2), (1, 3), (1, 4), (2, 2), (2, 3), (2, 4), (3, 3), (3, 4), (4, 4)}, where each integer represents a read subset. The tasks along the diagonal (0, 0), (1, 1) (2, 2), (3, 3) and (4, 4) are considered to be higher priority tasks because they have a greater number of child/dependent tasks. All other tasks have a normal priority in terms of execution. After a task gets released, meaning that all of its predecessors have been executed, it is sent to the EAS execution queue. When the task has completed executing, the EAS engine checks to see if any dependent tasks can be released for execution.

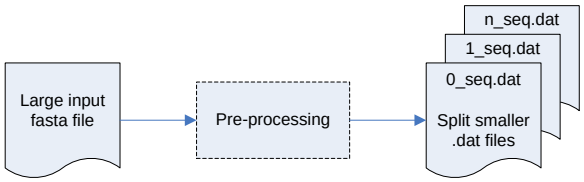


Figure 5. Pre-processing step

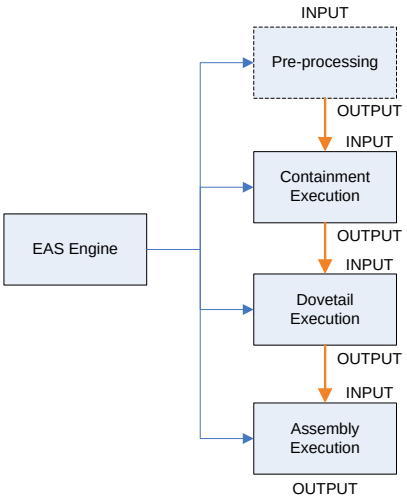


Figure 6. Process flow diagram

Now let us take a look at the example where we have five read subsets. When the task (0, 0) is complete, it releases all the tasks in that row which are tasks (0, 1), (0, 2), (0, 3) and (0, 4). It cannot release (1, 1) because task (1, 1) still has another dependency on (0, 1). When (0, 1) is completed, it will release task (1, 1). Completion of task (1, 1) will flag (1, 2), (1, 3), and (1, 4) but they will only be released when both (1, 1) and the tasks above them namely (0, 2), (0, 3), and (0, 4) have completed execution. This will continue until all tasks are executed. The last task to be executed will be task (4, 4) in our example. Note that the total number of tasks executed would be fifteen. This can be calculated easily using equation one. We would like to point out that the containment phase is

bounded by the number of files (in this case five). We cannot use more than five nodes at any given time due to task

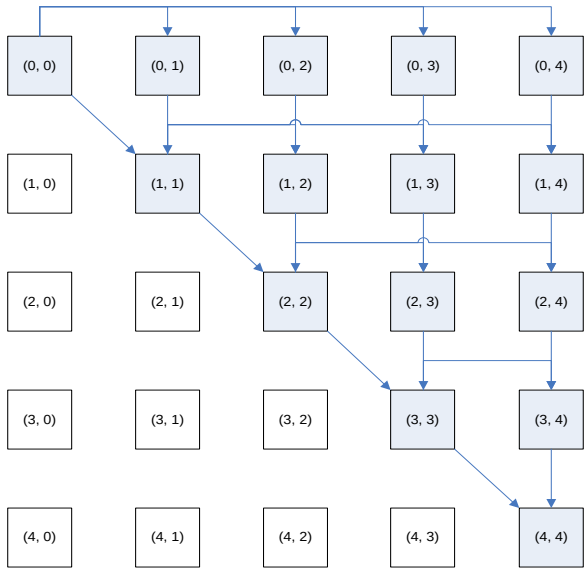


Figure 7. Execution dependencies of containment tasks

dependencies even though we have a total of fifteen containment tasks.

C. Dovetail Execution – Step 2

The execution dependencies of the dovetail tasks are much more straightforward than those for the containment tasks. The dovetail tasks do not have any dependencies on each other and hence can be run in a naively parallel way, allowing us to use as many processors as possible. Continuing with our previous example with fifteen tasks, we could execute (0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 1), (1, 2), (1, 3), (1, 4), (2, 2), (2, 3), (2, 4), (3, 3), (3, 4), (4, 4) all at the same time during the dovetail phase.

The total number of tasks that need to be executed in each of the above steps (containment and dovetail steps) is given by the equation below, where n is the number of read subsets and T is the total number of tasks.

$$T = \frac{n(n+1)}{2} \qquad (1)$$

VI. Results

We downloaded *Escherichia coli W* reads produced by the 454 Titanium technology from the NCBI [18] sequence read archive (accession no. SRR060736 and SRR060737, made public by JCVI). The sequences were trimmed to remove adaptors. The final result was 337,294 trimmed reads. For our experiment in the pre-processing step we decided to split these into 16,866 sequence reads per file, i.e. read subset (except for the last file which contained 16,814 reads). This resulted in 40 files and a total of 674,588 reads. (The preprocessing step generates the reverse complement of each read.) We then used the EAS engine to run the assembly algorithm using 1 to 31 nodes. For our experiments we used the HPC environments available at UNO (University of Nebraska at Omaha). We initially start out with the Blackforest cluster (16 nodes) [19] and then move to a true commercial strength HPC named Firefly cluster (1100 nodes) at the Holland Computing Center [20].

Firefly Cluster: The firefly cluster is a large commercial strength cluster at the Holland Computing Center which comprises of 1,151-node supercomputer cluster of Dell SC1435 servers. Each node contains two sockets, and each socket holds a quad-core (four 64-bit AMD Opteron 2.2 GHz processors). The computational network utilizes an 800 MB/sec Infiniband interconnect. Each node has its own 8 GB of memory, and 73 GB of disk space.

Chart (a) in figure 8 shows the execution time of the algorithm in seconds versus the number of nodes used for each run. It shows that after 11 to 12 nodes we do not see any significant performance gain. Along with the total execution time, we captured the average execution time per worker node and the overhead. We find that as we increase the number of nodes the overhead curve follows the execution time curve. It is important to note that in a HPC a significant portion of the master process's work is distributing the tasks and managing the task dependency among the worker processes along with handling of the communication between master and worker processes. This is clearly depicted by chart (b) in figure 8.

It is important to note that given the nature of the task dependencies in the containment phase not all nodes are working all the time, and hence we see a smaller overall curve for the average worker time per node. This leads us to ask the question, "How parallelizable is the program?" For the purpose of answering this question we plotted the program speedup against the number of nodes and integrated this curve with a plot of Amdahl's law in chart (c) in figure 8. Amdahl's law is defined by the formula:

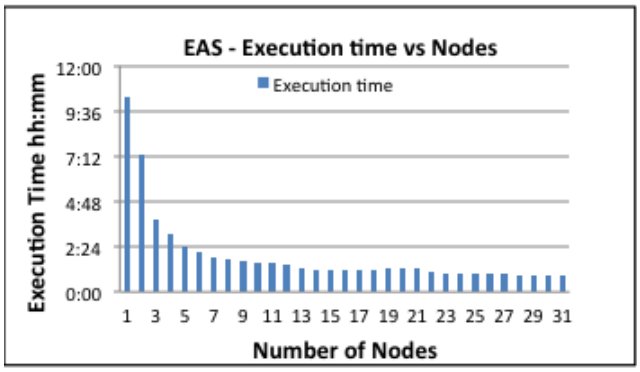
$$\frac{1}{(1-P) + \frac{P}{N}}$$

As $N \to \infty$, the maximum speedup tends to $1/(1-P)$. In practice, performance/price falls rapidly as N is increased once there is even a small component of $(1-P)$. A great part of the craft of parallel programming consists of attempting to reduce $(1-P)$ to the smallest possible value.

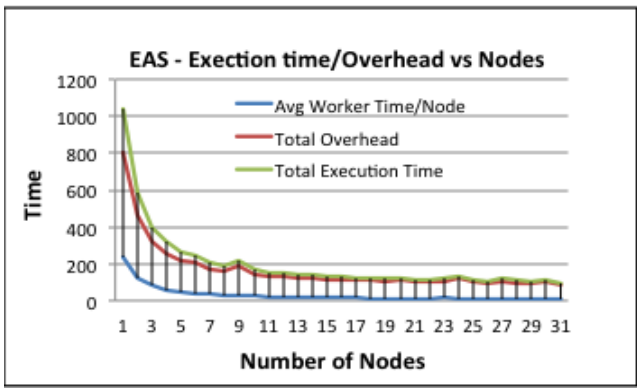
We can conclude that the overlap detection algorithm of the Merge and Traverse assembler has a speedup between 20 - 25 times (which is between 90% - 95% parallelizable).

Next we set up experiments to see if the EAS engine would be able to dynamically adjust the number of nodes to meet a given deadline. We used four groups of read datasets generated from SRR060736 and SRR060737. Each group was partitioned into a different number of files as shown in table one.

A)



B)



C)

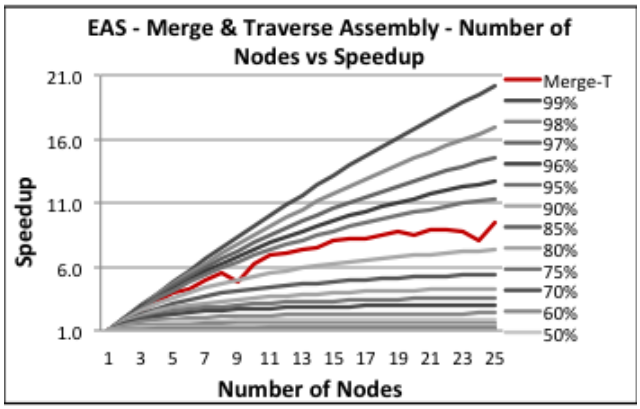


Figure 8. Chart (a): EAS - Execution time v/s Nodes. Chart (b): Execution time/Overhead v/s nodes. Chart (c): Speedup curve for the assembly program

Each group of files was ran against five different deadlines (30, 60, 90, 120, and 150 minutes). Each of these jobs was assigned a starting number of nodes by the EAS engine based on the run profile/speedup curve. As the tasks were completed, variances between EET (Expected Execution Time) and AET (Actual Execution Time) resulted in the EAS engine adjusting the number of nodes up (+N) or down (–N), if there were equal number of (+N) and (–N) adjustments it resulted in a net (0) adjustment and finally the scenario of no adjustments being made (–). The experimental results showed that the EAS engine was able to dynamically adjust nodes to minimize energy utilized while meeting the deadlines.

Table 1. Read subset groups used for analysis

Group	Number of Files	Number of Sequences
G1	5	84330
G2	10	168660
G3	15	337320
G4	20	674588

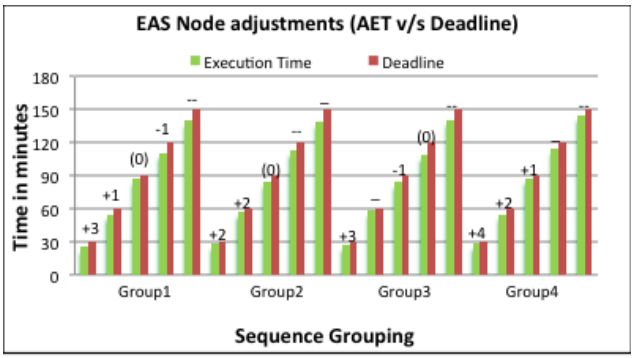


Figure 9. EAS engine - dynamic node adjustments

VII. CONCLUSIONS

Based on the results we can clearly observe that given a deadline we can choose the appropriate number of nodes to run the overlap detection phase of the assembler on based on our new understanding of the run-profile we just produced. This will allow us to apportion just enough nodes to meet the deadline thus maximizing the objective of performance with minimum energy utilization. We also observed that with a smaller number of nodes we have larger gains in performance and above a certain number of nodes the performance gain is only modest at best. In fact as we add additional nodes our communication costs and related overhead is higher.

Clearly different bioinformatics applications and algorithms will have different run profiles and understanding each one of them will allow us to best assign the appropriate number of nodes to meet a given deadline. It was also important to see how the number of read subsets impacted the performance/energy criterion. Our experiments suggest a bowl shaped curve when we varied the number of files for the same number of nodes. Clearly there must be some optimum value for the number of files for each set.

This paper highlights the importance of understanding the degree of parallelism for the program, which is done by establishing the run profile/speedup curve. The EAS engine uses the knowledge from the run profile to make intelligent and dynamic decisions about number of nodes to use to minimize energy utilization and still provide necessary

performance. Clearly it is no longer sufficient to simply run a program in a HPC environment. It is important and essential to understand the data, its characteristics, and the application domain to build a parallel program that is energy aware.

In designing these experiments, we have several parameters we could study and the relationship between them. These parameters are (1) Number of files; (2) Number of sequences per file; (3) Number of nodes used and (4) Average sequence length. In this paper we have only looked at number of nodes used as a parameter for our experimental design. In the future we plan to investigate how adjusting the different tuning parameters such as number of files, number of sequences per file, number of nodes impacts the performance and energy efficiency. We also plan on including the pre-processing step and final assembly as part of the EAS processing. Our main motivation is to move this from a simple speedup to the realm of energy awareness. Our EAS model for the purposes of the experiments conducted calculated energy as a function of resources used in this case number of nodes. The energy function could be made more complex; we leave that for a future study.

ACKNOWLEDGMENT

This project was supported by the NIH grant number P20 RR016469 from the INBRE Program of the National Center for Research Resources (NCRR).

REFERENCES

[1] M. Meyerson et al, "Advances in Understanding Cancer Genomes through Second-generation Sequencing," *Nat. Rev. Genet.*, vol. 11, no. 10, pp. 685-696, Oct. 2010.

[2] J Qin et al, "A Human Gut Microbial Gene Catalogue Established by Metagenomic Sequencing," *Nature*, vol. 464 no. 7285 pp. 59-65, Mar. 2010.

[3] X. Huang et al, "PCAP: A Whole-Genome Assembly Program," *Genome Res.*, vol. 13, no. 9, pp. 2164-2170, Sept. 2003.

[4] J. Ullman, "NP-complete Scheduling Problems," *J. of Comput. and Syst. Sci.* vol. 10 no. 3 pp. 384-393. June 1975.

[5] E. G. Coffman et al, *Computer and Job-Shop Scheduling Theory*, NY: John Wiley & Sons Inc., 1976.

[6] H. El-Rewini et al, *Task Scheduling in Parallel and Distributed Systems*, Upper Saddle River, NJ: Prentice Hall, 1994.

[7] P. Aronsson and P. Fritzson, "Task Merging and Replication using Graph Rewriting," in the *Tenth International Workshop on Compilers for Parallel Computers*, Amsterdam, The Netherlands, 2003, doi: 10.1.1.7.9285.

[8] A. A. Khan et al, "A Comparison of Multiprocessor Scheduling Heuristics," in the *International Conference on Parallel Processing*, North Carolina State Univeristy, NC. 1994, pp. 243-250.

[9] R. Xie et al, "Scheduling Multi-Task Agents," in the *Fifth IEEE International Conference on Mobile Agents*, Atlanta, GA., 2001, pp 260-276.

[10] D.D. Sommer et al, "A Fast, Lightweight Genome Assembler," *BMC Bioinformatics* vol. 8, no. 1, Feb. 2007.

[11] E.W. Myers et al, "A whole-genome assembly of Drosophila," *Science*.2000, vol. 287 , no. 5461, pp. 2196–204, Mar. 2000.

[12] J.R Miller et al, "Assembly Algorithms for next-generation sequencing data," *Genomics*, vol. 95, no. 6, pp. 315-327, June 2010.

[13] E.W. Myers, "The Fragment Assembly String Graph," *Bioinformatics*, vol. 21, no. 2, pp. 79-85, Sept. 2005.

[14] W. Li, and A. Godzik. "Cd-hit: A Fast Program for Clustering and Comparing Large Sets of Protein or Nucleotide Sequences," *Bioinformatics*, vol. 22, no.13, pp. 1658-659, July 2006.

[15] N. J. Larsson and K. Sadakane. "Faster suffix sorting," Lund University, Lund, Sweden, Tech. Rep. LU-CS-TR:99-214, 1999.

[16] E. Ohlebusch and M. I. Abouelhoda. "Chaining Algorithms and Applications in Comparative Genomics," in the *Handbook of Computational Molecular Biology*, Boca Raton, FL: Chapman and Hall/CRC Computer and Information Science Series, 2006, ch. 15

[17] S.B. Needleman, and C.D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequences of two proteins," *J. Mol. Biol.*, vol. 48, no. 3, pp. 443–453, 1970

[18] NCBI Database. Retrieved Nov 2010 from http://www.ncbi.nlm.nih.gov/sra

[19] Blackforest Computing Cluster. Retrieved Dec 2010 from http://blackforest.gds.unomaha.edu/about.php

[20] Holland Computing Center. Retrieved Dec 2010 from http://www.hollandhpc.com/index.shtml.