A Distributed System for Genetic Linkage Analysis*

Mark Silberstein, Dan Geiger, and Assaf Schuster

Technion – Israel Institute of Technology
{marks|dang|assaf}@cs.technion.ac.il

Abstract. Linkage analysis is a tool used by geneticists for mapping disease-susceptibility genes in the study of Mendelian and complex diseases. However analyses of large inbred pedigrees with extensive missing data are often beyond the capabilities of a single computer. We present a distributed system called SUPERLINK-ONLINE for computing multipoint LOD scores of large inbred pedigrees. It achieves high performance via efficient parallelization of the algorithms in SUPERLINK, a state-of-the-art serial program for these tasks, and through utilization of thousands of resources residing in multiple opportunistic grid environments. Notably, the system is available online, which allows computationally intensive analyses to be performed with no need for either installation of software, or maintenance of a complicated distributed environment. The main algorithmic challenges have been to efficiently split large tasks for distributed execution in a highly dynamic non-dedicated running environment, as well as to utilize resources in all the available grid environments. Meeting these challenges has provided nearly interactive response time for shorter tasks while simultaneously serving massively parallel ones. The system, which is being used extensively by medical centers worldwide, achieves speedups of up to three orders of magnitude and allows analyses that were previously infeasible.

1 Introduction

Linkage analysis aims at facilitating the understanding of mechanisms of genetic diseases via identification of the areas on the chromosome where disease-susceptibility genes are likely to reside. Computation of a logarithm of odds (LOD) is a valuable tool widely used for the analysis of disease-susceptibility genes in the study of Mendelian and complex diseases. The computation of the LOD score for large inbred pedigrees with extensive missing data is often beyond the computation capabilities of a single computer. Our goal is to facilitate these more demanding linkage computations via parallel execution on multiple computers.

We present a distributed system for exact LOD score computations, called SUPERLINK-ONLINE [1], capable of analyzing inbred families of several hundreds individuals with extended missing data using single and multiple loci disease models. Our approach, made possible by recent advances in distributed computing (e.g., [2]), eliminates the need for expensive hardware by distributing the computations over thousands of non-dedicated PCs and utilizing their idle cycles. Such opportunistic environments,

* This work is supported by the Israeli Ministry of Science.

W. Dubitzky et al. (Eds.): GCCB 2006, LNBI 4360, pp. 110–123, 2007.
© Springer-Verlag Berlin Heidelberg 2007

referred further as *grids*, are characterized by the presence of many computers with different capabilities and operating systems, frequent failures, and extreme fluctuations in the number of computers available. Our algorithm allows a single linkage analysis task to be recursively split into multiple independent subtasks of required size; each subtask can be further split into smaller subtasks for performance reasons. The flexibility to adjust the number of subtasks to the amount of available resources is particularly important in grids. The subtasks are then executed on grid resources in parallel, and their results are combined and outputted as if executed on a single computer.

SUPERLINK-ONLINE delivers the newest information technology to geneticists via a simple Internet interface, which completely hides the complexity of the underlying distributed system. The system allows for concurrent submission and execution of linkage tasks by multiple users, generating a *stream* of parallel and serial tasks with vastly different (though unknown in advance) computational requirements. These requirements range from a few seconds on a single CPU for some tasks to a few days on thousands for others. The stream is dynamically scheduled on a set of independently managed grids with vastly different amounts of resources. In addition, the time spent in a grid on activities other than actual task execution differs for different grids, and the difference can reach several orders of magnitude. We refer to such activities – such as resource allocation or file transfer – as *execution overhead*. The challenge is to provide nearly interactive response time for shorter tasks (up to a few seconds), while simultaneously serving highly parallel heavy tasks in such a multi-grid environment.

Our system implements a novel scheduling algorithm which combines the well-known Multilevel Feedback Queue (MQ) [3] approach with the new concept of *grid execution hierarchy*. All available grids in the hierarchy are sorted according to their size and overhead: upper levels of the hierarchy include smaller grids with faster response time, whereas lower levels consist of one or more large-scale grids with higher execution overhead. The task stream is scheduled on the hierarchy, so that each task is executed at the level that matches the task's computational requirements, or its *complexity*. The intuition behind the scheduling algorithm is as follows. As the task complexity increases, so do the computational requirements and the tolerable overhead. Consequently, more complex tasks should be placed at a lower level of the hierarchy, and parallelized according to the available resources at that level.

The proper execution level for a given task is easy to determine if the task complexity is known or simple to compute. Unfortunately, this is not the case for the genetic linkage analysis tasks submitted to the system. In fact, geneticists usually do not know whether the results can be obtained in a few seconds, or whether the task will require several years of CPU time. Provably, for the linkage analysis tasks estimating the exact complexity of a given task is NP-hard in itself [4].

Our algorithm allows the proper execution level to be quickly determined via fast estimation of an upper bound on the task complexity. We apply a heuristic algorithm that yields such an upper bound [5] for a fraction of the running time limit assigned to a given hierarchy level. If the complexity is within the complexity range of that level (which is configured before the system is deployed), the task is executed; otherwise it is moved directly to a lower level. The precision of the complexity estimation algorithm improves the longer it executes. Consequently, task complexity is reassessed at each

level prior to the actual task execution. The lower the level, the greater the amount of resources that are allocated for estimating more precisely its complexity, resulting in a better matched hierarchy level.

SUPERLINK-ONLINE is a production system which assists geneticists worldwide, utilizing about 3000 CPUs located in several grids at the Technion in Haifa, and at the University of Wisconsin in Madison. During the last year the system served over 7000 tasks, consuming about 110 CPU years over all available grids.

The paper is organized as follows. In Section 2 we show the algorithm for scheduling a stream of linkage analysis tasks on a set of grids. In Section 3 we describe the parallel algorithm for genetic linkage analysis. The current deployment of the SUPERLINK-ONLINE system is reported in Section 4, followed by performance evaluation, related work and future research directions.

2 The Grid Execution Hierarchy Scheduling Algorithm

The algorithm has two complementary components: organization of multiple grids into a grid execution hierarchy and a set of procedures for scheduling tasks on this hierarchy.

2.1 Grid Execution Hierarchy

The purpose of the execution hierarchy is to classify available grids according to their performance characteristics, such as execution overhead and amount of resources, so that each level of the hierarchy provides the best performance for tasks of a specific complexity range.

Upper levels of the hierarchy include smaller grids with faster response time, whereas lower levels consist of one or more large-scale grids with higher execution overhead. The number of levels in the hierarchy depends on the expected distribution of task complexities in the incoming task stream, as explained in Section 4.

Each level of the execution hierarchy is associated with a set of one or more queues. Each queue is connected to one or more grids at the corresponding level of the hierarchy, allowing submission of jobs into these grids. A task arriving at a given hierarchy level is enqueued into one of the queues. It can be either executed on the grids connected to that queue (after being split into jobs for parallel execution), or migrated to another queue at the same level by employing simple load balancing techniques. If a task does not match the current level of the execution hierarchy, as determined by the scheduling procedure presented in the next subsection, it is migrated to the queue at a lower level of the hierarchy.

2.2 Scheduling Tasks in Grid Hierarchy

The goal of the scheduling algorithm is to quickly find the proper execution hierarchy level for a task of a given complexity . Ideally, if we knew the complexity of each task and the performance of each grid in the system, we could compute the execution time of a task on each grid, placing that task on the one that provides the shortest execution

time. In practice, however, neither the task complexity nor the grid performance can be determined precisely. Thus, the algorithm attempts to schedule a task using approximate estimates of these parameters, dynamically adjusting the scheduling decisions if these estimates turn out to be incorrect.

We describe the algorithm in steps, starting with the simple version, which is then enhanced. Due to space limitations we omit some details available elsewhere [6].

Simple MQ with grid execution hierarchy. Each queue in the system is assigned a maximum time that a task may execute in the queue (execution time)T_e[1]. The queue configured to serve the shortest tasks is connected to the highest level of the execution hierarchy, the queue for somewhat longer tasks is connected to the next level, and so on.

A task is first submitted to the top level queue. If the queue limit T_e is violated, a task is preempted, check-pointed and restarted in the next queue (the one submitting tasks to the next hierarchy level).

Such an algorithm ensures that any submitted task will eventually reach the hierarchy level that provides enough resources for longer tasks and fast response time for shorter tasks. In fact, this is the original MQ algorithm applied to a grid hierarchy.

However, the algorithm fails to provide fast response time to short tasks if a long task is submitted to the system prior to a short one. Recall that the original MQ is used in time-shared systems, and tasks within a queue are scheduled using preemptive round-robin, thus allowing fair sharing of the CPU time [3]. In our case, however, tasks within a queue are served in FCFS manner (though later tasks are allowed to execute if a task at the head of the queue does not occupy all available resources). Consequently, a long task executed in a queue for short tasks may make others wait until its own time limit is exhausted.

Quick evaluation of the expected running times of a task in a given queue can prevent the task from being executed at the wrong hierarchy level. Tasks expected to consume too much time in a given queue are migrated to the lower level queue without being executed. This is described in the next subsection.

Avoiding hierarchy level mismatch. Each queue is associated with a maximum allowed task complexity C_e, derived from the maximum allowed execution time T_e by optimistically assuming linear speedup, availability of all resources at all times, and resource performance equal to the average in the grid. The optimistic approach seems reasonable here, because executing a longer task in an upper level queue is preferred over moving a shorter task to a lower level queue, which could result in unacceptable overhead. The following naive relationship between a complexity limit C_e and a time limit T_e reflects these assumptions:

$$C_e = T_e \cdot (N \cdot P \cdot \beta), \tag{1}$$

where N is the maximum number of resources that can be allocated for a given task, P is the average resource performance, and β is the efficiency coefficient of the

[1] For simplicity we do not add the queue index to the notations of queue parameters, although they can be set differently for each queue.

application on a single CPU, defined as a portion of CPU-bound operations in the overall execution.

For each task arriving to the queue, task's complexity is first estimated[2]. Allocating a small portion $\alpha < 1$ of T_e for complexity estimation often allows quick detection of a task that does not fit in the queue. The task is migrated to the lower level queue if its complexity estimate is higher than C_e.

However, the upper bound on the task's complexity might be much larger than the actual value. Consequently, if the task is migrated directly to the level in the hierarchy that serves the complexity range in which this estimate belongs, it may be moved to too low a level, thus decreasing the application's performance. Therefore, the task is migrated to the next level, where the complexity estimation algorithm is given more resources and time to execute, resulting in a more precise estimate.

The queue complexity mismatch detection improves the performance of both shorter and longer tasks. On the one hand, the computational requirements of longer tasks are quickly discovered, and the tasks are moved to the grid hierarchy level with enough computational resources avoiding the overhead of the running task migration. On the other hand, longer tasks neither delay nor compete with the shorter ones, reducing the response time and providing nearly interactive user experience. In practice, however, a task may stay in the queue longer than initially predicted because of the too optimistic grid performance estimates. Thus, the complexity mismatch detection is combined with the enforcement of the queue time limit described in the previous subsection, by periodically monitoring the queue and migrating tasks which violate the queue time limit.

2.3 Handling Multiple Grids at the Same Level of the Execution Hierarchy

The problem of scheduling in such a configuration is equivalent to the well-studied problem of load sharing in multi-grids. It can be solved in many ways, including using the available meta-schedulers, such as [7], or flocking [2]. If no existing load sharing technologies can be deployed between the grids, we implement load sharing as follows.

Our implementation is based on a push migration mechanism (such as in [8]) between queues, where each queue is connected to a separate grid. Each queue periodically samples the availability of resources in all grids at its level of the execution hierarchy. This information, combined with the data on the total workload complexity in each queue, allows the expected completion time of tasks to be estimated. If the current queue is considered suboptimal, the task is migrated. Conflicts are resolved by reassessing the migration decisions at the moment the task is moved to the target queue. Several common heuristics are implemented to reduce sporadic migrations that may occur as a result of frequent fluctuations in grid resource availability [9]. Such heuristics include, for example, averaging of momentary resource availability data with the historical data, preventing migration of tasks with a small number of pending execution requests, and others.

[2] The complexity estimation can be quite computationally demanding for larger tasks, and in which case it is executed using grid resources.

3 Parallel Algorithm for the Computation of LOD Score

LOD score computation can be represented as the problem of computing an expression of the form

$$\sum_{x_1}\sum_{x_2}\cdots\sum_{x_n}\prod_{i=1}^{m}\Phi_i(\mathbf{X_i}), \tag{2}$$

where $\mathbf{X} = \{x_1, x_2, \ldots, x_n | x_i \in \mathbb{N}\}$ is a set of non-negative discrete variables, $\Phi_i(\mathbf{X_i})$ is a function $\mathbb{N}^k \to \mathbb{R}$ from the subset $\mathbf{X_i} \subset \mathbf{X}$ of these variables of size k to the reals, and m is the total number of functions to be multiplied. Functions are specified by a user as an input. For more details we refer the reader to [10].

3.1 Serial Algorithm

The problem of computing Eq. 2 is known to be NP-complete [11]. One possible algorithm for computing this expression is called variable elimination [12].

The complexity of the algorithm is fully determined by the order in which variables are eliminated. For example, two different orders may result in running time ranging from few seconds to several hours. Finding an optimal elimination order is NP-complete [13]. A close-to-optimal order can be found using the stochastic greedy algorithm proposed in [5]. The algorithm can be stopped at any point, and it produces better results the longer it executes, converging faster for smaller problems. The algorithm yields a possible elimination order and an upper bound on the problem complexity, i.e., the number of operations required to carry out the computations if that order is used. It is this feature of the algorithm that is used during the scheduling phase to quickly estimate the complexity of a given problem prior to execution.

The above algorithm is implemented in SUPERLINK, which is the state-of-the-art program for performing LOD score computations of large pedigrees.

3.2 Parallel Algorithm

The choice of a parallelization strategy is guided by two main requirements. First, subtasks are not allowed to communicate or synchronize their state during the execution. This factor is crucial for performance of parallel applications in a grid environment, where communication between computers is usually slow, or even impossible due to security constraints. Second, parallel applications must tolerate frequent failures of computers during the execution by minimizing the impact of failures on the overall performance.

The algorithm for finding an elimination order consists of a large number of independent iterations. This structure of the algorithm allows to parallelize it by distributing the iterations over different CPUs using the master-worker paradigm, where master process maintains a queue of independent jobs which are subsequently pulled and executed by worker processes running in parallel on multiple CPUs. At the end of the execution the order with the lowest complexity is chosen.

Parallelization of the variable elimination algorithm also fits the master-worker paradigm and is performed as follows. We represent the first summation over x_1 in

Eq.2 as a sum of the results of the remaining computations, performed for every value of x_1. This effectively splits the problem into a set of independent subproblems having exactly the same form as the original one, but with the complexity reduced by a factor approximately equal to the number of values of x_1. We use this principle recursively to create subproblems of a desired complexity, increasing the number of variables used for parallelization. Each subproblem is then executed independently, with the final result computed as the sum of all partial results.

The choice of the number of subtasks and their respective granularity is crucial for efficiency of the parallelization. The inherent overheads of distributed environments, such as scheduling and network delays, often become a dominating factor inhibiting meaningful performance gains, suggesting that long-running subtasks should be preferred. On the other hand, performance degradation as a result of computer failures will be lower for short subtasks, suggesting to reduce the amount of computations per subtask. Furthermore, decreasing the amount of computations per subtask increases the number of subtasks generated for computing a given problem, improving load balancing and utilization of available computers.

Our algorithm controls the subtask granularity by specifying maximum allowable complexity threshold C. Specifying lower values of C increases the number of subtasks, decreasing the subtask complexity and consequently its running time. The value of C for a given problem is determined as follows. We initially set C so that a subtask's running time does not exceed the average time it can execute without interruption on a computer in the Grid environment being used. If such value of C yields that the number of subtasks is below the number of available computers, then C is iteratively reduced to allow division into more subtasks. The lower bound on C is set so that overheads due to scheduling and network delays constitute less than 1% of the subtask's running time.

3.3 Implementation of the Parallel Algorithm for Execution in Grid Hierarchy

Parallel tasks are executed in a distributed environment via Condor [2], which is a general purpose distributed batch system, capable of utilizing idle cycles of thousands of CPUs[3]. Condor hides most of the complexities of job invocation in an opportunistic environment. In particular, it handles job failures that occur because of changes in the system state. Such changes include resource failures, or a situation in which control of a resource needs to revert to its owner. Condor also allows resources to be selected according to the user requirements via a matching mechanism.

There are three stages in running master-worker applications in Condor: the parallelization of a task into a set of independent jobs, their parallel execution via Condor, and the generation of final results upon their completion. In our implementation, this flow is managed by the Condor flow execution engine, called DAGman, which invokes jobs according to the execution dependencies between them, specified as a directed acyclic graph (DAG). The complete genetic linkage analysis task comprises two master-worker applications, namely, parallel ordering estimation and parallel variable elimination. To integrate these two applications into a single execution flow, we use an outer DAG composed of two internal DAGs, one for each parallel application.

[3] Use of Condor is nor assumed, neither required by the algorithm. Moreover, the system is being expanded to use other grid batch systems without making any change to the algorithm.

DAGman is capable of saving a snapshot of the flow state, and then restarting execution from this snapshot at a later time. We use this feature for migration of a task to another queue as follows: the snapshot functionality is triggered, all currently executing subtasks are preempted, the intermediate results are packed, and the task is transferred to another queue where it is restarted.

4 Deployment of SUPERLINK-ONLINE

The current deployment of the SUPERLINK-ONLINE portal is presented in Figure 1.

We configure three levels of the execution hierarchy, for the following reasons. About 60% of the tasks take a few minutes or less, and about 28% take less than three hours, as reflected by the histogram in Figure 4, generated from the traces of about 2000 tasks executed by the system. This suggests that two separate levels, Level 1 and Level 2, should be allocated for these dominating classes, leaving Level 3 for the remaining longer tasks. Yet, the current system is designed to be configured with any number of levels to accommodate more grids as they become available.

Each queue resides on a separate machine, connected to one or several grids. Utilization of multiple grids via the same submission machine is enabled by the Condor flocking mechanism, which automatically forwards job execution requests to the next grid in the list of available (*flocked*) grids, if these jobs remain idle after previous resource allocation attempts in the preceding grids.

Queue Q1 is connected to the dedicated dual CPU server and invokes tasks directly without parallelization. Queue Q2 resides on a submission machine connected to the flock of two Condor pools at the Technion. However, due to the small number of resources at Q2, we increased the throughput at Level 2 of the execution hierarchy by activating load sharing between Q2 and Q3, which is connected to the flock of three Condor pools at the University of Wisconsin in Madison. The tasks arrive to Q3 only from Q2. Queue Q4 is also connected to the same Condor pools in Madison as Q3, and may receive tasks from both Q2 and Q3.

In fact, Q3 exhibits rather high invocation latencies and does not fit Level 2 of the execution hierarchy well. Alternatively, Q3 could have been set as an additional level between Q2 and Q4, and the queue time limit of Q2 could have been adjusted to handle smaller tasks. However, because both queues can execute larger tasks efficiently, such partitioning would have resulted in unnecessary fragmentation of resources. Migration allows for a more flexible setup, which takes into account load, resource availability and overheads in both queues, and moves a whole task from Q2 to Q3 only if this yields better task performance. Typically, small tasks are not migrated, while larger tasks usually benefit from execution in a larger grid as they are allocated more execution resources. This configuration results in better performance for smaller tasks than does flocking between these two grids, as it ensures their execution on the low-overhead grid.

To ensure that larger tasks of Q4 do not delay smaller tasks of Q3, jobs of tasks in Q3 are assigned higher priority and may preempt jobs from Q4. Starvation is avoided via internal Condor dynamic priority mechanisms [2].

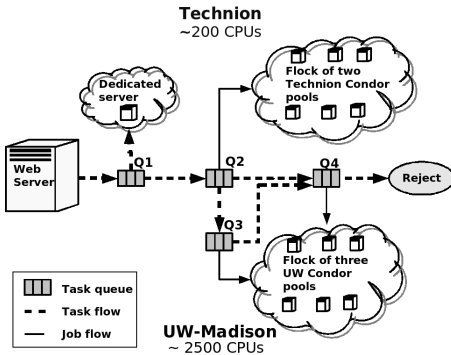

Fig. 1. SUPERLINK-ONLINE deployment

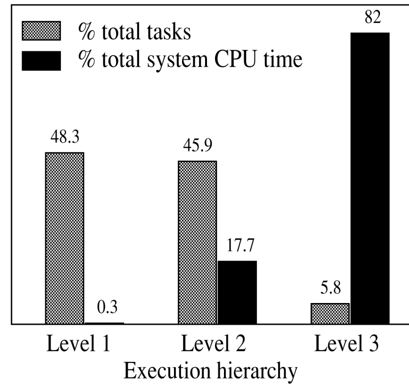

Fig. 2. Tasks handled versus the overall system CPU time (per level)

5 Results

We evaluate two different aspects of the performance of the SUPERLINK-ONLINE system. First, we show the speedups in the running time achieved by the parallel algorithm implementation over the serial state-of-the-art program for linkage analysis on a set of challenging inputs. Next, we demonstrate the performance of the scheduling algorithm by analyzing the traces of about 2,000 tasks served by SUPERLINK-ONLINE during second half of 2005.

5.1 Parallel Algorithm Performance

We measured the total run time of computing the LOD score for a few real-life challenging inputs. The results reflect the time a sole user would wait for a task to complete, from submission to the SUPERLINK-ONLINE system via a web interface until an e-mail notification about task completion is sent back. Results are summarized in Table 1.

We also compared the running time with that of the newest serial version of SUPERLINK, invoked on Intel Pentium Xeon 64bit 3.0 Ghz, 2GB RAM. The entries of the running time exceeding two days were obtained by measuring the portion of the problem completed within two days, as made available by SUPERLINK, and extrapolating the running time assuming similar progress. The time saving of the online system versus a single computer ranged from a factor of 10 to 700. The results (in particular, rows 5,7,9,10) demonstrate the system capabilities to perform the analyses which are infeasible by SUPERLINK, considered a state-of-the-art program for linkage analysis on such inputs.

In a large multiuser Grid environment used by SUPERLINK-ONLINE, the number of computers employed in computations of a given task may fluctuate during the execution between only few to several hundreds. Table 1 presents the average and the maximum number of computers used during execution. We believe that the performance can be improved significantly if the system is deployed in a dedicated environment. Running times of SUPERLINK-ONLINE include network delays and resource failures (handled

Table 1. SUPERLINK-ONLINE vs. serial SUPERLINK V1.5

	Running time		#CPU used	
Input	SUPERLINK V1.5	SUPERLINK-ONLINE	Average	Maximum
1	5000sec	1050sec	10	10
2	5600sec	520sec	11	11
3	20hours	2hours	23	30
4	450min	47min	82	83
5	˜300hours	7.1hours	38	91
6	297min	27min	82	100
7	˜138days	6.2hours	349	450
8	˜2092sec	1100sec	7	8
9	˜231hours	3hours	139	500
10	˜160days	8hours	310	360

automatically by the system). The column average is computed from the number of computers sampled every 5 minutes during the run.

Example of a previously infeasible analysis. We replicated the analysis performed for studying the Cold Induced Sweating Syndrome in a Norwegian family [14]. The pedigree consists of 93 individuals, two of which affected, and only four were typed. The original analysis was done using FASTLINK [15]. The maximum LOD score of 1.75 was reported using markers D19S895, D19S566 and D19S603, with the analysis limited to only three markers due to computational constraints[4]. According to the authors, using more markers for the analysis was particularly important in this study as "*in the absence of ancestral genotypes, the probability that a shared segment is inherited IBD from a common ancestor increases with the number of informative markers contained in the segment; that is, a shared segment containing only three markers has a significant probability of being simply identical by state, whereas a segment containing a large number of shared markers is much more likely to be IBD from a common ancestor. Thus, the four-point LOD score on the basis of only three markers from within an interval containing 13 shared markers, is an underestimate of the true LOD score*". Study of another pedigree as well as application of additional statistical methods were employed to confirm the significance of the findings.

Using SUPERLINK-ONLINE we computed six point LOD score with the markers D19S895, M4A, D19S566, D19S443 and D19S603, yielding the LOD=3.10 at marker D19S895, which would facilitate the linkage conclusion based on the study of this pedigree alone.

5.2 Performance of the Grid Execution Hierarchy Scheduling Algorithm

We analyzed the traces of 2300 tasks, submitted to the SUPERLINK-ONLINE system via Internet by users worldwide for the period between the 1st of June and the 31st of December 2005. During this time, the system utilized about 460,000 CPU hours (52.5

[4] LOD score above 3.3 is considered an indication of linkage.

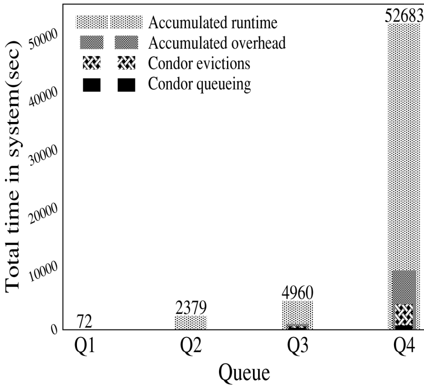

Fig. 3. Average accumulated time (from arrival to termination) of tasks in each queue

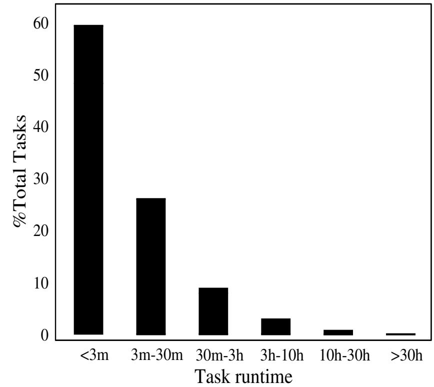

Fig. 4. Distribution of real task runtimes as registered by the system

CPU years) over all Condor pools connected to it (according to the Condor accounting statistics). This time reflects the time that would have been spent if all tasks were executed on single CPU. About 70% of the time was utilized by 1971 successfully completed tasks. Another 3% was wasted because of system failures and user-initiated task removals. The remaining 27% of the time was spent executing tasks which failed to complete within the queue time limit of Q4, and were forcefully terminated. However, this time should not be considered as lost since users were able to use partial results. Still, for clarity, we do not include these tasks in the analysis.

We compared the total CPU time required to compute tasks by each level of the execution hierarchy relative to the total system CPU consumption by all levels together. As expected, the system spent most of its time handling the tasks at Level 3, comprising 82% of the accumulated running time of the system (see Figure 2). The tasks at Level 2 consumed only 17.7% of the total system bandwidth, and only 0.3% of the time was consumed by the tasks at Level 1. If we consider the total number of tasks served by each level, the picture is reversed: the first two levels served significantly more tasks than the lower level. This result proves that the system was able to provide short response time to the tasks which were served at the upper levels of the hierarchy.

This conclusion is further supported by the graph in Figure 3, which depicts the average accumulated time of tasks in each queue, computed from the time a task is submitted to the system until it terminates. This time includes accumulated overheads, which are computed by excluding the time of actual computations from the total time. As previously, the graph shows only the tasks which completed successfully. Observe that very short tasks which require less than three minutes of CPU time and are served by Q1 stay in the system only 72 seconds on average regardless of the load in the other queues.This is an important property of the scheduling algorithm.

The graph also shows average accumulated overhead for tasks in each queue, which is the time a task spent in the system performing any activity other than the actual computations.

The form of the graph requires explanation. Assuming a uniform distribution of task runtimes and availability of an appropriate grid for each task, the task accumulated time is expected to increase linearly towards lower levels of the hierarchy. However in practice these assumptions do not hold. There are exponentially more shorter tasks requiring up to 3 minutes on single CPU (see Figure 4). This induces a high load on Q1, forcing short tasks to migrate to Q2 and thus reducing the average accumulated time of tasks in Q2. This time is further reduced by the load sharing between Q2 and Q3, which causes larger tasks to migrate from Q2 to Q3. Thus, shorter tasks are served by Q2, while longer ones are executed in Q3, resulting in the observed difference between the accumulated times in these queues. To explain the observed steep increase in the accumulated time in Q4, we examined the distribution of running times in this queue. We found that shorter tasks (while exceeding Q3's allowed task complexity limit) were delayed by longer tasks that preceded them. Indeed, over 70% of the overhead in that queue is due to the time the tasks were delayed because of other tasks executing in that queue. Availability of additional grids for the execution of higher complexity tasks would allow for the queueing and turnaround time to be reduced.

6 Related Work and Conclusions

In this work we presented the SUPERLINK-ONLINE system, which delivers the power of grid computing to geneticists worldwide, and allows them to perform previously infeasible analyses in their quest for disease-provoking genes. We described our parallel algorithm for genetic linkage analysis, suitable for execution in grids, and the method for organizing multiple grids and scheduling user-submitted linkage analysis tasks in such a multi-grid environment.

Future work will include on-the-fly adaptation of the hierarchy to the changing properties of the grids, designing a cost model to take into account locality of applications and execution platforms, improving the efficiency of the parallel algorithm, as well as connection to other grids, such as EGEE-2.

Related work. Parallelization of linkage analysis has been reported in [15–22]. However, the use of these programs by geneticists has been quite limited due to their dependency on the availability of high performance execution environments, such as a cluster of high performance dedicated machines or a supercomputer, and due to their operational complexity.

Execution of parallel tasks in grid environments has been thoroughly studied by grid researchers (e.g., [23–28]). In particular, [28] addressed the problem of resource management for short-lived tasks on desktop grids, demonstrating the suitability of grid platforms for execution of short-lived applications. Various scheduling approaches have been reported in many works(e.g. meta-schedulers [7, 29][30, 31], streams of tasks scheduling [32], steady-state scheduling [33] and others), however some of the are either not applicable in a real environment due to unrealistic assumptions. Others, such as [9] and [8], inspired the implementation of some system components.

References

1. Superlink-online: Superlink-online genetic linkage analysis system. http://bioinfo.cs.technion.ac.il/superlink-online (2006)
2. Thain, D., Livny, M.: Building reliable clients and servers. In Foster, I., Kesselman, C., eds.: The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, San-Francisco (2003)
3. Kleinrock, L., Muntz, R.: Processor sharing queueing models of mixed scheduling disciplines for time shared systems. Journal of ACM **19** (1972) 464–482
4. Fishelson, M., Geiger, D.: Exact genetic linkage computations for general pedigrees. Bioinformatics **18**(Suppl. 1) (2002) S189–S198
5. Fishelson, M., Dovgolevsky, N., Geiger, D.: Maximum likelihood haplotyping for general pedigrees. Human Heredity **59** (2005) 41–60
6. Silberstein, M., Geiger, D., Schuster, A., Livny, M.: Scheduling of mixed workloads in multi-grids: The grid execution hierarchy. In: 15th IEEE International Symposium on High Performance Distributed Computing (HPDC-15 2006). (2006)
7. CSF: Community scheduler framework. http://www.globus.org/toolkit/docs/4.0/contributions/csf (2006)
8. England, D., Weissman, J.: Costs and benefits of load sharing in the computational grid. In Feitelson, D.G., Rudolph, L., eds.: 10th Workshop on Job Scheduling Strategies for Parallel Processing. (2004)
9. Vadhiyar, S., Dongarra, J.: Self adaptivity in grid computing. Concurrency and Computation: Practice and Experience **17**(2-4) (2005) 235–257
10. Friedman, N., Geiger, D., Lotner, N.: Likelihood computation with value abstraction. In: Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence (UAI), Morgan Kaufmann (2000) 192–200
11. Cooper, G.: The computational complexity of probabilistic inference using bayesian belief networks. Artificial Intelligence **42** (1990) 393–405
12. Dechter, R.: Bucket elimination: A unifying framework for probabilistic inference. In Jordan, M., ed.: Learning in Graphical Models. Kluwer Academic Press. (1998) 75–104
13. Arnborg, S., Corneil, D.G., Proskurowski, A.: Complexity of finding embeddings in a k-tree. SIAM Journal of Algorithms and Discrete Methods **8** (1987) 277–284
14. Knappskog, P., Majewski, J., Livneh, A., Nilsen, P., Bringsli, J., Ott, J., Boman, H.: Cold-Induced Sweating Syndrome is caused by mutations in the CRLF1 Gene. American Journal of Human Genetics **72(2)** (2003) 375–383
15. Miller, P., Nadkarni, P., Gelernter, G., Carriero, N., Pakstis, A., Kidd, K.: Parallelizing genetic linkage analysis: a case study for applying parallel computation in molecular biology. Computing and Biomedical Research **24**(3) (1991) 234–248
16. Dwarkadas, S., Schäffer, A., Cottingham, R., Cox, A., Keleher, P., Zwaenepoel, W.: Parallelization of general linkage analysis problems. Human Heredity **44** (1994) 127–141
17. Matise, T., Schroeder, M., Chiarulli, D., Weeks, D.: Parallel computation of genetic likelihoods using CRI-MAP, PVM, and a network of distributed workstations. Human Heredity **45** (1995) 103–116
18. Gupta, S., Schäffer, A., Cox, A., Dwarkadas, S., Zwaenepoel, W.: Integrating parallelization strategies for linkage analysis. Computing and Biomedical Research **28** (1995) 116–139
19. Rai, A., Lopez-Benitez, N., Hargis, J., Poduslo, S.: On the parallelization of Linkmap from the LINKAGE/FASTLINK package. Computing and Biomedical Research **33**(5) (2000) 350–364
20. Kothari, K., Lopez-Benitez, N., Poduslo, S.: High-performance implementation and analysis of the linkmap program. Computing and Biomedical Research **34**(6) (2001) 406–414

21. Conant, G., Plimpton, S., Old, W., Wagner, A., Fain, P., Pacheco, T., Heffelfinger, G.: Parallel Genehunter: implementation of a linkage analysis package for distributed-memory architectures. Journal of Parallel and Distributed Computing **63**(7-8) (2003) 674–682
22. Dietter, J., Spiegel, A., an Mey, D., Pflug, H.J., al Kateb, H., Hoffmann, K., Wienker, T., Strauch, K.: Efficient two-trait-locus linkage analysis through program optimization and parallelization: application to hypercholesterolemia. European Journal of Human Genetics **12** (2005) 542–50
23. Berman, F., Wolski, R.: Scheduling from the perspective of the application. In: 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03), Washington, DC, USA, IEEE Computer Society (1996) 100–111
24. Yang, Y., Casanova, H.: Rumr: Robust scheduling for divisible workloads. In: 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03), Washington, DC, USA, IEEE Computer Society (2003) 114
25. Berman, F., Wolski, R., Casanova, H., Cirne, W., Dail, H., Faerman, M., Figueira, S., Hayes, J., Obertelli, G., Schopf, J., Shao, G., Smallen, S., Spring, S., Su, A., Zagorodnov, D.: Adaptive computing on the grid using AppLeS. IEEE Transactions on Parallel and Distributed Systems **14(4)** (2003) 369–382
26. Heymann, E., Senar, M.A., Luque, E., Livny, M.: Adaptive scheduling for master-worker applications on the computational grid. In: GRID 2000. (2000) 214–227
27. Beaumont, O., Legrand, A., Robert, Y.: Scheduling divisible workloads on heterogeneous platforms. Parallel Computing **29**(9) (2003) 1121–1152
28. Kondo, D., Chien, A.A., Casanova, H.: Resource management for rapid application turnaround on enterprise desktop grids. In: ACM/IEEE Conference on Supercomputing (SC'04), Washington, DC, USA, IEEE Computer Society (2004) 17
29. MOAB Grid Suite: Moab grid suite. http://www.clusterresources.com/pages/ products/moab-grid-suite.php (2006)
30. Dail, H., Sievert, O., Berman, F., Casanova, H., YarKhan, A., Vadhiyar, S., Dongarra, J., Liu, C., Yang, L., Angulo, D., Foster, I.: Scheduling in the grid application development software project. Grid Resource Management: State-of-the-art and Future Trends (2004) 73–98
31. Vadhiyar, S., Dongarra, J.: A metascheduler for the grid. In: 11th IEEE International Symposium on High Performance Distributed Computing (HPDC'02), Washington, DC, USA, IEEE Computer Society (2002)
32. Sabin, G., Kettimuthu, R., Rajan, A., Sadayappan, P.: Scheduling of parallel jobs in a heterogeneous multi-site environment. In Feitelson, D.G., Rudolph, L., Schwiegelshohn, U., eds.: Job Scheduling Strategies for Parallel Processing. Springer Verlag (2003) 87–104
33. Marchal, L., Yang, Y., Casanova, H., Robert, Y.: Steady-state scheduling of multiple divisible load applications on wide-area distributed computing platforms. International Journal of High Performance Computing Applications (2006, to appear)