Embarrassingly Parallel Jobs Are Not Embarrassingly Easy to Schedule on the Grid

Enis Afgan, Purushotham Bangalore
Department of Computer and Information Sciences
University of Alabama at Birmingham
1300 University Blvd., Birmingham AL 35294-1170
{afgane, puri}@cis.uab.edu

Abstract

Embarrassingly parallel applications represent an important workload in today's grid environments. Scheduling and execution of this class of applications is considered mostly a trivial and well-understood process on homogeneous clusters. However, while grid environments provide the necessary computational resources, associated resource heterogeneity represents a new challenge for efficient task execution for these types of applications across multiple resources. This paper presents a set of examples illustrating how execution characteristics of individual tasks, and consequently a job, are affected by the choice of task execution resources, task invocation parameters, and task input data attributes. It is the aim of this work to highlight this relationship between an application and an execution resource to promote development of better metascheduling techniques for the grid. By exploiting this relationship, application throughput can be maximized, also resulting in higher resource utilization. In order to achieve such benefits, a set of job scheduling and execution concerns is derived leading toward a computational pipeline for scheduling embarrassingly parallel applications in grid environments.

1. Introduction

Embarrassingly parallel (EP) class of applications is likely to represent the most widely deployed class of applications in the world [1]. EP applications, in nature very similar to SPMD (Single Process, Multiple Data) or Parameter Sweep, are characterized by independent, coarsely grained and indivisible tasks. The goal of EP applications is to introduce parallelism into application execution without any application code modification and associated cost. This is realized through multiple invocations of the same application, where each instance is invoked using a different input data set. The number of tasks being instantiated can range greatly from only a few instances to several hundred instances and each instance can execute from several seconds or minutes to many hours. The end result is speedup of application's execution that is only limited by the number of resource available.

Meanwhile, grid computing has emerged as the upcoming distributed computational infrastructure that enables virtualization and aggregation of geographically distributed compute resources into a seamless compute platform [2]. Characterized by heterogeneous resources with a wide range of network speeds between participating sites, high network latency, site and resource instability and a sea of local management policies, it seems the grid presents as many problems as it solves. Nonetheless, from the early days of the grid, EP class of applications has been categorized as the killer application for the grid [3]. This is due to the ability of EP applications to consume large numbers of resources accompanied with minimal requirements and dependencies on network status as well as high tolerance for partial failure. Furthermore, these applications were relatively easy to deploy and initiate execution.

This suitability has materialized through several projects that have explored, exploited, and advanced capabilities of EP application in grid environments, for example AppLeS [4] and Nimrod/G [5]. However, because of the lack of communication between individual tasks, a common assumption is that execution of EP application in grid environments is easy, or at least significantly easier than execution of tightly coupled, MPI applications. On the contrary, execution of EP applications in grid environments is confined by resource availability, optimized through task parameterization, hindered by simultaneous use and management of heterogeneous resources belonging to different administrative domains, and dependent on user requirements. As such, the act of application execution includes application scheduling, or assignment of tasks to appropriate resources. Because of the sheer number of influencing components, scheduling becomes a major component

978-1-4244-2872-4/08/$25.00 ©2008 IEEE

for success of EP applications in the grid and should be handled comprehensively with respect to the application execution environment variables.

As the grid evolves into independently managed clouds relying heavily on commercial aspects, resource owners will try to maximize resource utilization while satisfying users' requirements. Simultaneously, users will impose stringent requirements on their application executions in terms of cost, execution time, reliability or data cost. In order to achieve these objectives an understanding of the components influencing application execution is necessary. In this paper, we offer a discussion of execution variables for EP applications in grid environments and their effects on applications' runtimes. We have used the NCBI BLAST [6] application as an example to highlight many of these issues and illustrate the need for advanced resource and application specific metascheduling techniques.

2. Application Model

As briefly discussed, the Embarrassingly Parallel (EP) class of applications is composed of a class of applications whose computational load consists of executing the same application multiple times, with each instance operating on a varying input data set and perhaps varying parameters [7]. The most significant characteristic of EP applications is that, once started, there is no communication between individual tasks. This model can be seen as a modification of Flynn's original SIMD (Single Instruction, Multiple Data) taxonomy and more general SPMD (Single Process, Multiple Data) model where individual processes can execute independent instructions at the same point in time but are, overall, executing the same code [7]. As such, an application execution with all of its instances is referred to as a job. A job is composed of a set of tasks or instances, where each instance is assigned a different input data set. More formally, a job J composed of a set of tasks t_i can be represented as follows:

$$J = \bigcup (t_1, t_2, \dots, t_n,) \quad (1)$$

In this section we present some of the key factors that affect metascheduling of EP class of applications in a grid along with potential solutions to address some of these issues.

2.1. Task heterogeneity

In more traditional cluster environments, where individual compute nodes are homogeneous, execution times of individual tasks within a job can be expected largely homogeneous. In grid environments, where individual resources are highly heterogeneous, task execution times can vary greatly [8]. Figure 1 shows an example of task runtime variability resulting from resource heterogeneity. Shown tasks were all fed an equally-sized input file and executed across a range of heterogeneous resources resulting in a significant level of load imbalance. Tasks that have experienced approximately equivalent runtimes got assigned to multiple nodes within a single cluster by the high-level scheduler. In the described environment, the runtime of the entire job is determined by the longest running task making a strong requirement for efficient load balancing.

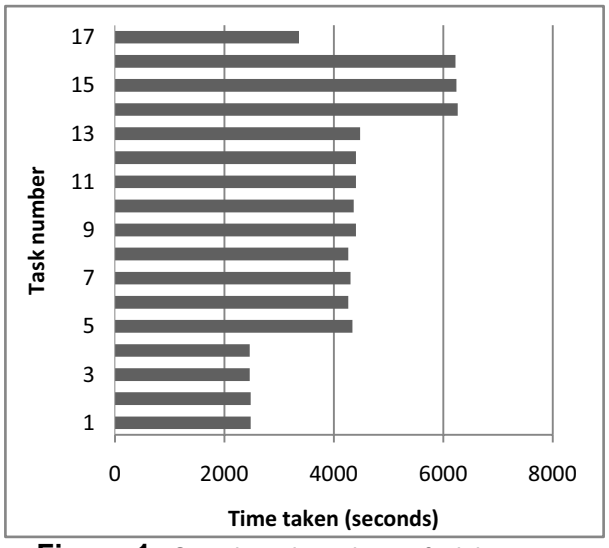

Figure 1. Sample task runtimes of a job across heterogeneous resources.

2.2. Task parameterization

In order to better understand factors influencing runtime of a task, a task t_i can be further decomposed and represented as follows:

$$t_i = f(d_i, r_i, p_i) \quad (2)$$

where d_i represents the task input data, r_i represents the execution resource for task t_i, and p_i represents the parameter set used when invoking the given task. Individually, the input data set is the single most influential factor determining the runtime of a task. However, because the input file or input parameters are the purpose for the computation,the job is constrained by the characteristics of the input. Nonetheless, at the task level, existing file characteristics may be exploited to better meet resource capabilities. For example, Figure 2 is showing runtime of two different NCBI BLAST [6] jobs where a large number of only short search queries

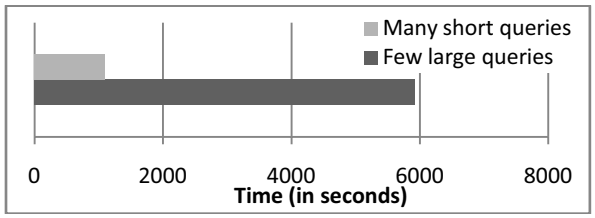

Figure 2. Task runtime affected by input file properties.

where fed to the program as opposed to a small number of long queries. Both files were of approximately same physical size. The figure shows a difference in execution speed of several orders of magnitude. Such observations are application specific, but offer a new level of application-specific scheduling that can be exploited in grid environments. For example, in case of BLAST, variation in task execution time between long and short queries is caused by algorithm's requirement to keep track of various segments of the search query. For the long queries, this process consumes significantly more time than for the short queries. As such, this characteristic can be exploited to divide the input data into sets of only long and only short search queries. Then, long queries can be submitted to fast resources while short queries can be targeted for the slower, cheaper and less loaded resources achieving overall job load balance.

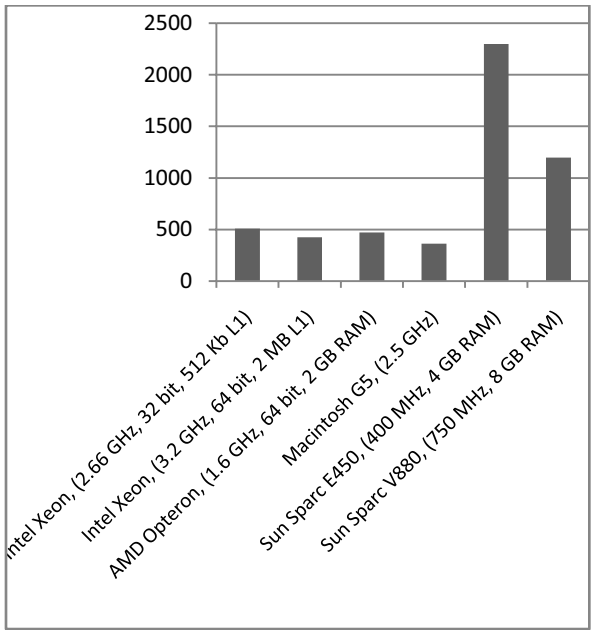

Figure 3. Execution times of a set of jobs across heterogeneous architectures showing runtime influence.

Because of grid resource heterogeneity, the factor offering most flexibility regarding task runtime is the resource selected for task execution. Figure 3 shows a sample runtimes for a set of jobs on heterogeneous resources. From the figure, it is obvious that even small variations in underlying resources can cause noticeable change in task and therefore job runtime.

Lastly, the parameter set used when invoking a task can have a significant impact on resource utilization and task runtime (see Figure 4). For example, a sample parameter that the user may have control over but that does not influence the results of the computation is number of threads employed within a process. Other such parameters may be application dependent and can include transitional probabilities in Hidden Markov Model (HHM) applications or step size in Monte Carlo simulations. Figure 4 shows the runtime influence of using varying number of threads to complete the same task. For shown application (NCBI BLAST), such functionality is supported directly within the application and requires only an additional parameter when invoking the task. What can also be noticed from the figure is that scalability of presented approach has its limits, and, more so, those limits are resource dependent. As can be seen, speedup of employing two threads as opposed to one thread is nearly linear for all of the resources, but when using four threads as opposed to two, only the dual Opteron resource shows significant speedup. Furthermore, when invoking the task with eight threads, all the machines seem to have reached their scalability maximum with respect to the available parameter. This limit is caused by the total number of processing cores available on underlying resources, *i.e.,* mentioned parameter scales well until the number of threads is equivalent to the number of processing cores available on the compute node. As an additional note on application-resource dependency, if compute nodes with multiple CPUsare considered, one may be tempted to start multiple BLAST processes so that the number of processes corresponds to the number of CPUs on the node. As shown in Figure 5 though, such parameterization results in approximately 7% slowdown when compared to the single process and multiple threads parameterization. These observations apply to NCBI BLAST in particular and have been derived through analysis of application's execution characteristics. Nonetheless, similar observations can be derived for other applications.

With provided examples in mind, the p_i factor may need to be derived individually for each application and can represent a complex structure corresponding closely to application parameters. In general, because of the effects all the components have on the task computation time, a variable e_i can be

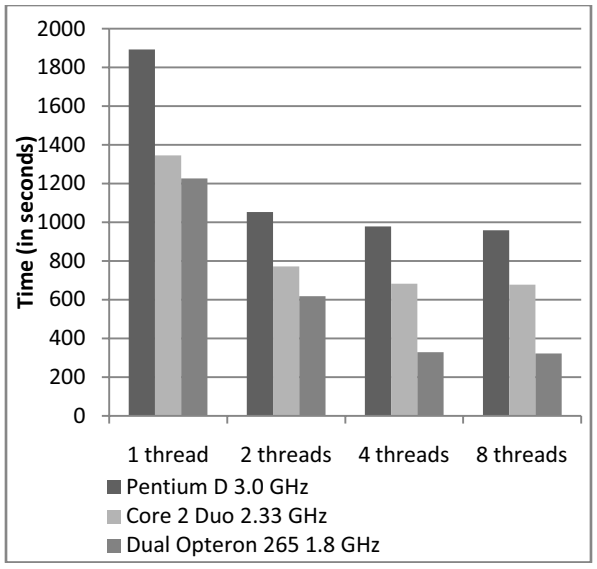

Figure 4. Task runtime dependencies on invocation parameters across architectures.

introduced to represent the expected computation time of a task t_i:

$$e_i \propto (d_i, r_i, p_i) \quad (3)$$

To aim of variable e_i is not necessarily representation of an absolute and definite task runtime that deals with job runtime models and job runtime estimations. Rather, it provides a relative reference among individual task options that exist when submitting a job and provides insight into components influencing task execution time. Finally, overall job execution time can be defined as $max(e_i)$ for all i.

2.3. Job parameterization

Following, the aim of a job submission effort and an accompanying scheduler dealing with EP jobs is to consider available options and factors influencing task execution with the goal of minimizing load imbalance between individual tasks t_i. By minimizing load imbalance among resources selected for execution, the overall job runtime will be minimized. Presented formulation leads to a definition of *job parameterization*, which can be defined as an understanding and selection of job and task parameters that are algorithm, input data, and resource dependent.

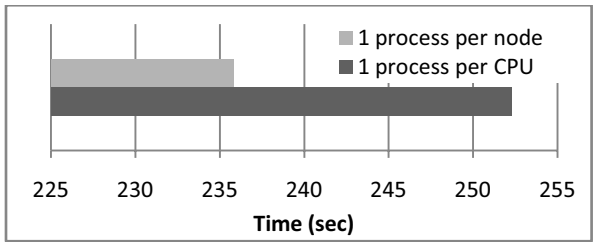

Figure 5. Effects of task parameterization - invoking a task using a single process per node with multiple threads as opposed to single process per CPU.

Because of the broad encompassment of EP class of applications and the versatility of the model describing them, other classes of applications can be encapsulated within the EP class. This derivation can be used to make a case for showing the difficulty involved in scheduling and coordinating execution of EP class of applications. Obviously, a sequential application can be represented by a single task that would also define the entire job. Similarly, a tightly coupled MPI application can be encapsulated within a single task. The value for the parameter factor p_i from Eq. 2 would include needed parallelism information. Because of general inability of MPI applications to cross individual cluster boundaries, currently there has not been such a need for multiple task coordination. This greatly simplifies the scheduling and job submission process because load balance at the job level does not have to be a concern. Master-worker and all-worker classes of applications can be seen as equivalent to EP class where the master process would be handling the resource selection and job parameterization rather than a higher-level scheduler.

3. Scheduling Considerations

Execution of EP jobs in grid environments enables large reduction in overall job execution time, increase in resource utilization and reduction in resource operating cost. This job execution though is perplexed with application and resource dependencies that, if not understood, can easily invalidate many of the available benefits. Scheduling of grid jobs, EP jobs in particular, is thus the most relevant activity that can realize these benefits. The first step toward efficient job execution is understanding of the existing application-resource relationship. Once understood, these relationships can be leveraged to deliver sought goals. Such understanding of an application's relationship to the numerous grid resources may seem as a far fetched goal requiring un-proportional effort for the benefit received. If decomposed, this process can be classified into several main categories lending themselves as a set of guidelines for deepening the understanding of the application-resource relationship. Furthermore, depending on the desired results, the level of understanding of the existing relationship can range from rudimentary, where merely a small number of most obvious application characteristics are recognized, all the way to a well-understood and well-developed mathematical model for application's

behavior that can later be mapped to individual resources.

In the context of the EP applications, efficient job execution can be warranted by minimization of the load imbalance across instantiated tasks. This load imbalance is minimized through coordination of employed parameters in the context of Eq. 2 above across job spectrum. Eq. 2 alone provides a baseline for understanding individual components that affect task's runtime. As stated earlier in Section 2, advancing understanding of effects each individual component has on application's execution is the first step in achieving efficient task execution. The next step is the derivation and coordination of multiple tasks that are making up the current job. This is also the most demanding step. With the ultimate goal of achieving adequate load balance across tasks, derivation of the tasks must be done simultaneously. Thus, this problem can easily be described as a multi-constraint multi-objective optimization problem. Fortunately, because the grid is such a heterogeneous and dynamic system, perfect solution to the problem is rarely necessary and, even if attained, questionably realizable. An approximate solution that recognizes stated attributes and employs heuristics and best-effort practices will often suffice. As such, the following is a set of concerns and suggestion that should be kept in mind during scheduling and execution of EP class of applications:

- *Job-level coordination*– it is beneficial to consider a job in its entirety prior to its execution rather than simply submitting jobs on first come, first serve basis for example. Global overview and comprehension of the job with respect to input file size, input file format and characteristics, data locality, available resources and user requirements is desirable. Through acknowledgement of these characteristics, the scheduling objectives, and thus the scheduling algorithm, can be adjusted more adequately. By considering a job as a unit, a global plan can be developed a priori eliminating many uncertainties as job execution gets under way (*e.g.,* job budget, input file distribution, data accuracy, expected task failure rate). At the same time, this job plan cannot be considered definitive due to the core characteristics of the grid (*e.g.,* unreliability, dynamic state) requiring provisions to be made that enable in-execution plan adjustments. An effective way to cope with the uncertainty is to adopt dynamic task distribution as opposed to the static distribution. The dynamic task distribution offers higher tolerance for individual task failure as well as dynamic resource availability. The major concern regarding dynamic task distribution is job granularity. Due to the overhead associated with task instantiation, a balance between number of tasks created and individual task's computation time needs to be adequately tuned.
- *Task parameterization* – once created, each task can be seen as an individual job. This is especially true from the resource perspective. Depending on the characteristics of the execution resource, parameterization of a task should be independently handled and optimized for the resource at hand. Based on the amount of data available for current application and resource as well as the level of understanding of an application performance model, this process can require a greatly variable amount of effort. Effects of task parameterization can turn a poorly behaving resource into a competitive one. An example is shown in Figure 6 where a properly parameterized task (Parameter set 2) executing on an old Sun Sparc machine is comparable in performance to an improperly parameterized task (Parameter set 1) on a much newer Intel Xeon based resource.
- *Load balancing* – although each task of an EP application executes independently, maximization of job performance is achieved through minimization of load imbalance across tasks. At the job level, aiming at achieving load balance results in task dependencies making efficient scheduling of this class of applications significantly more difficult. This is further complicated by the suggested task-level parameterization where significant variability among task execution rates can be observed. An approach to managing this issue can be seen in the job plan management that is aware of individual task-level optimizations and can thus guide the computation toward set goals. Other possible directions for achieving load balance in grid applications include adaptation of techniques used in parallel and distributed environments (*e.g.,* [9, 10, 11]).

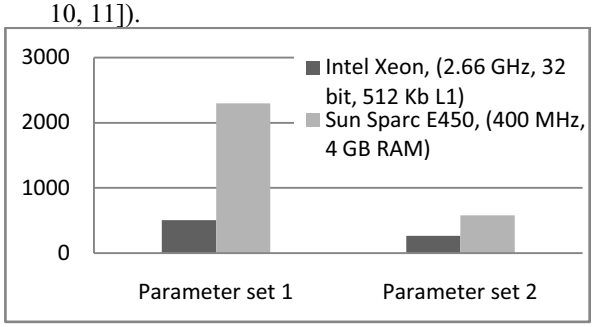

Figure 6. Significance of task-level parameterization - improperly parameterized task can result in a resource behaving very poorly.

- *Task failure* –superficially, execution of EP class of applications with no task communication seems like a perfect platform for easy handling of partial task failures. However, under the goals discussed throughout this paper, task failure introduces a significant issue. The aim of developing a job execution plan is to provide additional insight into job execution and try to satisfy user objectives. Once a task fails, the job plan is invalidated and, depending on the failure situation, could completely disrupt outlook of successful job completion. As a general observation, tasks executing on grid resources have the highest failure rate during their initialization and startup process. This can be due to resource miss-configurations, task miss-configuration, policies or events local to a resource as well as any of other host of variables. As such, a much higher probability of success can be assigned to a task that has reported as being under execution. Alternatively, multiple submission of a task can be started, albeit resulting in wastage of resources. If balance between computation time and number of tasks is well tuned, using dynamic task distribution can offer highest tolerance for task failure [12]. However, other issues, such as consistent resource availability, may pose alternate problems then. While there is no safe-guard against task failure, observations and models such as these can be incorporated into the job planning process leading to a more robust execution.
- *Cost* – considering the general direction of grid computing toward enterprise and cloud computing where economical aspects are being increasingly prevalent [13], cost associated with job execution is becoming a major concern. Because job cost is obtained from cumulative task cost, utilization realized by each task is directly proportional to the end cost. From the perspective of job scheduling, it is thus important to generate efficient job execution plan that satisfies user's requirements while yielding profit to the resource owner. Thus, cost is another major variable when scheduling EP jobs. Adopting the job plan model discussed leads toward desired estimations.
- *User requirements* – in the context of enterprise and cloud computing, user requirements will become the major driving force behind job scheduling policies. Unlike the more traditional cluster computing environments where resource utilization was the main objective, service oriented architecture promoted by the grid infrastructure is advocating a user centric orientation. Here, the quality of a system is realized by user satisfaction and measured through user utility. These notions are realized through the Quality of Service (QoS) requirements imposed by the users and agreed upon by the resource provider through the Service Level Agreements (SLAs). With future advancements of pervasive computing, user will require detailed insight into their consumption of computational power further complicated with imposed requirements on alternative job execution plans. While current efforts in scheduling EP applications primarily focus on runtime minimization, largely because runtime is equated with cost, in future systems cost will take on different forms (*e.g.,* result accuracy, system responsiveness, power consumption, availability) and will become a primary scheduling consideration.

4. EP Application Scheduling Framework

Considering described components that affect execution of EP applications in grid environments, scheduling of this class of applications needs to move beyond using a generic resource comparison as a guideline for task distribution and submission. Scheduling needs to involve coordination of resource capabilities, matching those to application's observed potential, coupled with adequate data distribution and finalized through individual task parameterization. A single scheduling action must coordinate execution parameters of such heterogeneous tasks with the goal of meeting user requirements. As such, an EP application with its derived tasks becomes a heterogeneous unit whose interactions and execution characteristics need to be simultaneously balanced and coordinated.

A diagram of the framework outlining the general process of effective EP application scheduling is provided Figure 7. As can be seen in the figure, effective EP application scheduling is a two-step process. Initially, the application needs to be analyzed, yielding needed application-specific information. Following, as user jobs arrive, the derived information is exploited to account for the application-specifics and in turn provide a job execution plan that aims at optimizing job's performance. Once a job plan has been derived, the control is transferred to a job submission engine for execution on grid resources.

Analyzer and planner are the two main components of described framework. Analyzer operates at the application level by performing a semi-permanent analysis of an application. Such analysis attempts at deriving application-specific relationships between input data, input parameters, runtime modes,

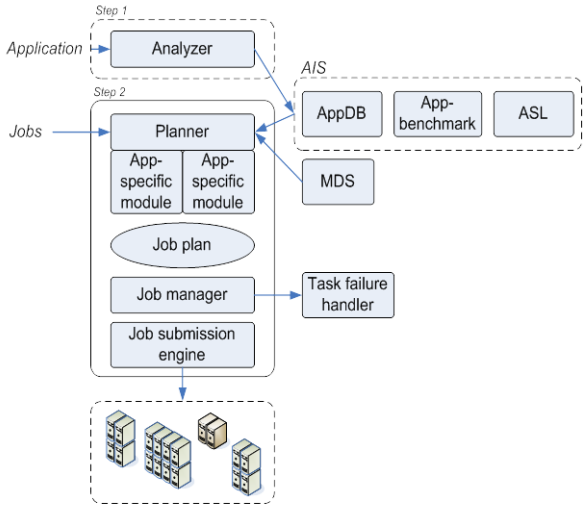

Figure 7. A sample EP application scheduling framework that accounts for application-specific behavior across grid resources.

and execution resources. Unfortunately, application analysis can often be a non-automated process that requires acquiring individual's familiarity with the application execution patterns. Tools and services such as AIS [14] can help in guiding and performing the analysis as well as storing derived conclusions for later retrieval and use. In the simplest format, application analysis involves benchmarking of various resources enabling relative comparison of those resources for later job execution (sample benchmarking tools: [15, 16]). In the most complicated format, application analysis involves development of a mathematical model that describes application execution patterns and dependencies. Once an application has been analyzed, user jobs can exploit derived information to improve their performance. Job planner is the component that operates at the job level and it integrates application analysis information along with resource availability information to produce a job plan. Job planner may require an application-specific module that can perform application data distribution or input data re-formatting in order for the job to maximize its performance. Once a job plan has been created, it is handed to the job manager and a job submission engine. The job manager component is an optional component than can implement application-specific logic for partial task failure and changes in resource availability.

5. Scheduling Examples

This section serves as a brief outlook into application execution characteristics and benefits as a result of applying considerations presented earlier. Two EP applications were used to perform needed analysis followed by job execution. Runtime results show an order of magnitude improvement in job execution characteristic.

As throughout the rest of this paper, NCBI BLAST application was chosen as one of the applications for execution. Figure 9 shows the runtimes of job segments as they were distributed across individual resources. A segment is defined as a set of tasks that executes on any one resource. Analyzing job turnaround time at the segment level is semantically equivalent to analyzing job turnaround time at individual task level with the benefit of reducing number of entities to display and keep track of, and has thus been used as the method of choice for result display.

Application analysis included benchmarking of each machine using BLAST and systematically changing input parameters. Following, based on benchmark data, application-specific modules for data distribution were developed. Given an input file, resource-application benchmarks were used to distribute and allocate appropriate chunks of data to selected resources with the goal of reducing job load imbalance. Initial and adjusted job data distributions can be seen in Figure 8. Furthermore, in order to exploit some of the observed dependencies [8] on job input file format, additional application-specific module has been developed and used to rearrange data in the input files enabling a finer level of control over reducing load imbalance. This module allows a key step in optimizing job execution parameters. Lastly, each task was parameterized to use appropriate number of threads to meet resource capabilities (*i.e.,* meet number of cores on a node) and thus minimize tasks' runtime.

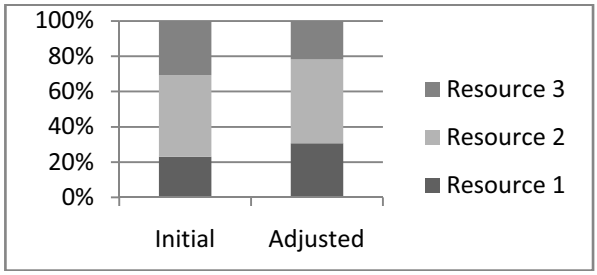

Figure 8. BLAST job data distribution across resources between initial and optimized job executions.

In short, the following components were addressed to obtain optimized job execution: resources were benchmarked, results of the application benchmarks were used to perform data distribution of for optimized tasks, input file for each task was re-formatted providing finer level of load balance

control, and lastly, each task was parameterized to use number of threads suitable to execution host.

From the Figure 9, it can be seen that the reduction in job turnaround time of the optimized case is on the order of 50%. In all of the test cases, the same number of tasks and the same number of compute nodes was used as in the initial case; thus, the increase in resource utilization is obtained purely through more appropriate job parameterization. Furthermore, it is obvious that the load imbalance across the machines has been drastically reduced and almost eliminated in the optimized case. The only difference in execution parameters of the adjusted and optimized cases is use of the input file re-formatting module. From the results, it can be concluded that for true optimization of application's execution, a fine-level understanding and analysis of an application is needed. Obviously, such understanding may be quite complicated and time consuming. Therefore, depending on expected application execution benefits and needs, highly detailed optimization of application's execution patterns may or may not be needed.

Presented methods have also been applied and used during job submission of statistical R code. Similar to execution of BLAST jobs, executing R code targeted at exploiting knowledge about individual resources and application's suitability to those. Unlike BLAST, R code does not support multiple threads and thus majority of speedup was achieved through data distribution that would match resource capabilities more closely. As was case with BLAST, the same number of compute nodes was used for all jobs; the only difference was in how individual tasks were parameterized. In case of selected R code, this required heterogeneous parameterization of tasks that corresponded to benchmarked relative resource capabilities. Furthermore, due to the lack of multi-threaded support in the application but due to the

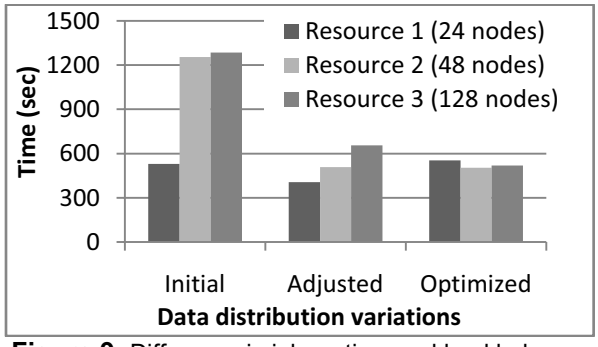

Figure 9. Difference in job runtime and load balance between a naïve job submission and an optimized job submission of a BLAST job.

publicized scheduling policies of available resources, resource specific knowledge was exploited to create multiple processes within a single assigned compute node resulting in higher overall throughput. After initial resource benchmarks, a targeted job plan was created by developing an R-specific data distribution module. The first iteration of job's execution resulted in significant reduction of job turnaround time ('adjusted' column in Figure 10). However, considerable load imbalance was still observed. In order to remove such load imbalance, similar to heterogeneous parameterization of tasks assigned to individual resources, tasks within a resource were then parameterized in heterogeneous fashion (based on runtime data observed during the 'adjusted' run). Final runtime results are shown under 'optimized' column in Figure 10 where it can be seen that load imbalance was further reduced leading to 75% reduction in runtime when compared to the initial case.

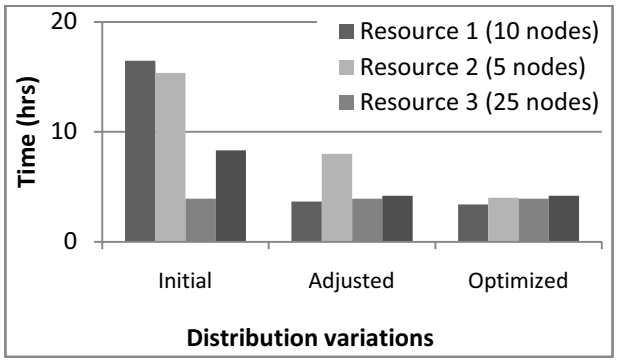

Figure 10. Job runtime reduction as a result of exploiting application-resource relationship and knowledge when executing R code.

Obviously, customizations and optimizations taking place during shown job submissions can be very application specific. It can thus be concluded that there is no single silver bullet that can be applied to any and all applications resulting in shown benefits. Rather, and as indicated in the introduction, the purpose of this paper is to provide a set of insights and tentative guidelines that can be considered when developing metaschedulers or executing jobs in grid environments. Currently, provided framework architecture can be used as performance optimization model rather than a concrete implementation. Obtaining desired results automatically requires an application-specific module or scheduler that understands and exploits existing relationships. The major drawback of this model is the need to, on application-per-application basis, understand and learn existing application-resource relationships followed by scheduler development. However, presented

guidelines present a broad framework offering one a directed path in achieving sought result.

6. Future Trends

Existing EP scheduling approaches and algorithms focus on minimizing job turnaround time, cost or both. Furthermore, current models require the user to specify job requirements and select most of the job execution parameters. Examples of options requested from the user can include specification of a preferred execution resource, number of processors to use, required amount of memory, and even creation of a virtual machine with all the necessary components and invocation parameters. Consequently, the user is still deeply involved in the job submission and scheduling process where the scheduler primarily performs actions requested. Even though today's schedulers advertise cost and runtime optimality, because they operate under constraints specified by the user, the schedulers are likely not exploring the entire application execution space. An analogy can be drawn here between searching for a local optimum versus a global optimum that actually exists in the application-resource space but is not even being explored due to the constraints imposed by uninformed user.

Future systems should advance the scheduling environment and alleviate the user of otherwise required effort by providing options and solutions to the user. The level of understanding of the relationship between an application and a resource should be understood by the scheduler to the point where advice can be provided to the user automatically and prior to job's execution. Such advice can come in form of concrete values such as compute time, cost, or data accuracy (for sample methods of estimating needed values, interested reader can refer to [17]) or more abstractly where only a window of various execution alternatives are presented. The latter alternatives may not be mapped to concrete values but could be related only to each other, thus providing insight to the user about the existing job execution alternatives. Without such support, the user may not even be aware of existing options and is thus less likely to realize attainable goals. By enabling generation and presentation of described alternatives, job submission can be tailored to each individual user and each individual job to better match user's requirements. This is the goal of service oriented technology.

An example of what the user may be presented with in future is provided in Figure 11, where a sample of application's execution space is made available to the user. Such an execution space should be derived automatically, following guidelines presented above and merged with user's job requirements (*e.g.,* job file size). Upon collection of needed information, job execution options are generated providing desired insight to the user. The user can then simply select the

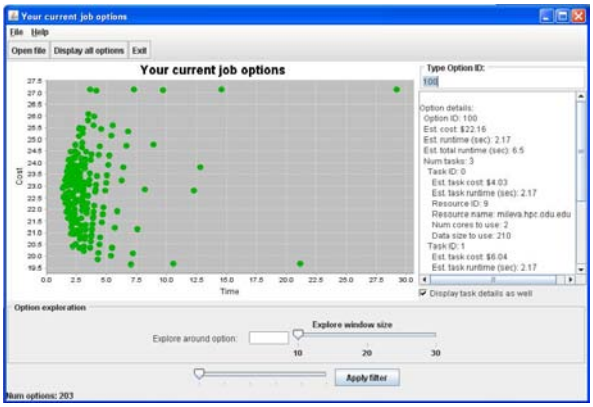

Figure 11. A sample job execution option space presented to the user showing job execution tradeoffs.

execution option based on their current situation, thus satisfying their needs more adequately.

In order to achieve described functionality, a deep understanding of an application needs to be attained. Initial works on understanding and exploiting this functionality have already begun (*e.g.,* [18, 19, 20]) with more coming. Furthermore, each application needs by be analyzed and described independently. Examples of application-specific characteristics can include scalability levels, input data characteristic, as well as resource compatibility. Added difficulty stems from the dynamics of the grid where resources fail, come online during job's execution or vary their performance rate based on current load. Some of these issues may be alleviated through existing and developing systems (*e.g.,* advance reservation), while others cannot be remedied for but only dealt with as they arise. Additional technologies, such as machine learning, may need to be employed to enable and automate aspirations such as these. Altogether, this is what makes scheduling a complex issue, but also one that is a cornerstone of a successful paradigm.

7. Conclusions

EP class of applications represents an ever-growing workload on today's compute resources. This workload is generated by larger problems as well as a desire for more accurate results. Grid computing presents itself as a viable solution to the resource shortage problem because independent institution can now share their resources, thus increasing resources' overall utilization. Integration of the independent resources was the first step towards achieving goals set by grid computing ideals. The second step is

efficient usage of those resources. This means that resource should not only be used continuously (*i.e.,* increasing their utilization) but also effectively (*i.e.,* increasing their throughput). In order to achieve such execution, relationship between an application and a resource needs to be understood and exploited by the metascheduler. Then, the third step of customizing the execution of an application to user's current needs becomes a realizable goal. At that point, the user abstracts themselves from underlying application execution details and is able to focus on the original problem. Work presented in this paper aims at promoting existence and importance of the application-resource relationship followed by a set of general concerns that can be used in order to develop more effective metascheduling techniques and solutions.

8. References

[1] A. Iosup, C. Dumitrescu, D. H. Epema, H. Li, and L. Wolters, "How are real grids used? The analysis of four grid traces and its implications," in *International Conference on Grid Computing 2006*, Barcelona, Spain, 2006, pp. 262-269.

[2] I. Foster and C. Kesselman, Eds. *The Grid 2*, Second ed., New York: Morgan Kaufmann, 2004.

[3] D. Abramson, J. Giddy, and L. Kotler, "High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid," in *International Parallel and Distributed Processing Symposium (IPDPS)*, Cancun, Mexico, 2000, pp. 520-528.

[4] H. Casanova, G. Obertelli, F. Berman, and R. Wolski, "The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid," in *Supercomputing 2000*, Dallas, TX, 2000.

[5] R. Buyya, D. Abramson, and J. Giddy, "Nimrod-G Resource Broker for Service-Oriented Grid Computing," *IEEE Distributed Systems Online,* 2(7), November 2001,

[6] T. Madden, "The BLAST Sequence Analysis Tool," NCBI August 13 2003.

[7] V. Kumar, *Introduction to Parallel Computing*: Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2002.

[8] E. Afgan and P. Bangalore, "Performance Characterization of BLAST for the Grid," in *IEEE 7th International Symposium on Bioinformatics & Bioengineering (IEEE BIBE 2007)* Boston, MA, 2007, pp. 1394-1398.

[9] I. Banicescu and Z. Liu., "Adaptive Factoring: A Dynamic Scheduling Method Tuned to the Rate of Weight Changes," in *High Performance Computing Symposium (HPC 2000)*, Washington, D.C., 2000, pp. 122-129.

[10] J. Cao, D. P. Spooner, S. A. Jarvis, S. Saini, and G. R. Nudd, "Agent-Based Grid Load Balancing Using Performance-Driven Task Scheduling," in *17th International Symposium on Parallel and Distributed Processing (IPDPS) 2003*, Nice, France, 2003, p. 49.2.

[11] R. U. Payli, E. Yilmaz, A. Ecer, H. U. Akay, and S. Chien, "DLB – A Dynamic Load Balancing Tool for Grid Computing," in *Parallel CFD Conference*, Grand Canaria, Canary Islands, Spain, 2004, p. 8.

[12] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI'04: Sixth Symposium on Operating System Design and Implementation* San Francisco, CA, 2004, p. 13.

[13] J. Broberg, S. Venugopal, and R. Buyya, "Market-oriented Grids and Utility Computing: The State-of-the-art and Future Directions," *Journal of Grid Computing,* 5(4), December 28 2007,

[14] E. Afgan and P. Bangalore, "Application Specification Language (ASL) – A Language for Describing Applications in Grid Computing," in *The 4th International Conference on Grid Services Engineering and Management - GSEM 2007* Leipzig, Germany, 2007, pp. 24-38.

[15] G. Tsouloupas and M. D. Dikaiakos, "Grid Resource Ranking Using Low-Level Performance Measurements," in *13th International Euro-Par Conference 2007 on Parallel Processing*, Rennes, France, 2007, pp. 467-476.

[16] G. Tsouloupas and M. Dikaiakos, "GridBench: A Tool for Benchmarking Grids," in *4th International Workshop on Grid Computing (Grid2003)*, Phoenix, AZ, 2003, pp. 60-67.

[17] E. Elmroth, J. Tordsson, T. Fahringer, F. Nadeem, R. Gruber, and V. Keller, "Three Complementary Performance Prediction Methods For Grid Applications," in *CoreGRID Integration Workshop 2008*, Heraklion, Greece, 2008.

[18] G. Tan, L. Xu, Z. Dai, S. Feng, and N. Sun, "A Study of Architectural Optimization Methods in Bioinformatics Applications," *International Journal of High Performance Computing Applications,* 21(3), August 2007, pp. 371-384.

[19] H. Stockinger, M. Pagni, L. Cerutti, and L. Falquet, "Grid Approach to Embarrassingly Parallel CPU-Intensive Bioinformatics Problems," in *IEEE International Conference on e-Science and Grid Computing* Amsterdam, Netherlands: IEEE 2006, pp. 58-68.

[20] A. Tirado-Ramos, G. Tsouloupas, M. Dikaiakos, and P. Sloot, "Grid Resource Selection by Application Benchmarking for Computational Haemodynamics Applications," in *International Conference on Computational Science (ICCS) 2005*, Kassel, Germany, 2005, pp. 534-543.