

Sequential Grid Computing: Models and Computational Experiments

Sam Ransbotham

Carroll School of Management, Boston College, Chestnut Hill, Massachusetts 02467,
sam.ransbotham@bc.edu

Ishwar Murthy

Indian Institute of Management, Bangalore 560076, India, ishwar@iimb.ernet.in

Sabyasachi Mitra, Sridhar Narasimhan

College of Management, Georgia Institute of Technology, Atlanta, Georgia 30332
{saby.mitra@mgt.gatech.edu, sri.narasimhan@mgt.gatech.edu}

Through recent technical advances, multiple resources can be connected to provide a computing grid for processing computationally intensive applications. We build on an approach, termed *sequential grid computing*, that takes advantage of idle processing power by routing jobs that require lengthy processing through a sequence of processors. We present two models that solve the static and dynamic versions of the sequential grid scheduling problem for a single job. In the static and dynamic versions, the model maximizes a reward function tied to the probability of completion within service-level agreement parameters. In the dynamic version, the static model is modified to accommodate real-time deviations from the plan. We then extend the static model to accommodate multiple jobs. Extensive computational experiments highlight situations (a) where the models provide improvements over scheduling the job on a single processor and (b) where certain factors affect the quality of solutions obtained.

Key words: grid computing; stochastic shortest path; dynamic programming

History: Accepted by S. Raghavan, Area Editor for Telecommunications and Electronic Commerce; received October 2007; revised April 2009, February 2010; accepted March 2010. Published online in *Articles in Advance* July 2, 2010.

1. Introduction

Advances in technology have made it possible to connect numerous disparate systems to create a virtual grid of computing resources that can be exploited to solve computationally intensive problems (Rosenberg 2004). Known by various related terms such as grid computing, utility computing, and Web-based computing, the concept has received significant attention recently in the academic and practitioner literature (Bhargava and Sundaresan 2004, Kumar et al. 2009, Meliksetian et al. 2004, Shalf and Bethel 2003, Stockinger 2006). The last few years have also witnessed the growth of computationally demanding applications, particularly in the scientific (Korpela et al. 2001), biological (Deonier et al. 2005, Ellisman et al. 2004), and business (Krass 2003) fields, that are impractical to perform on a single resource. Grid computing has emerged as a cost-effective method for providing an infrastructure for such computationally intensive applications, and several vendors (e.g., IBM, Sun, and Hewlett-Packard) are developing technology to enable a grid computing environment (Chang et al. 2004, Eilam et al. 2004).

Grid computing is largely viewed in the literature as a mechanism for implementing *parallel computing*. In parallel computing, an application is written to execute on multiple machines concurrently by dividing large computations into numerous smaller calculations that are executed in parallel. By enabling multiple machines to work on the application in parallel, the total time taken for completion can be reduced significantly. The topics addressed in the literature on parallel grid computing include grid architectures (Meliksetian et al. 2004), distributed data management (Venugopal et al. 2006), distributed processing for biological and visualization applications (Hansen and Johnson 2003), reliability of grid architectures (Levitin et al. 2006), task scheduling in a grid environment (Kaya and Aykanat 2006, Rosenberg 2004), and market design for grid computing (Bapna et al. 2006, 2008).

Unfortunately, widespread diffusion of parallel computing is not without impediments. In particular, the development of software that can take advantage of grid resources is difficult. As noted by Donald and Martonosi (2006, p. 14), "Writing parallel programs

is much more difficult and costly than sequential programming. . . .” Furthermore, complexities such as synchronization of access to resources and interprocess communications in single-machine environments are exacerbated in the context of grid computing. According to Boeres and Rebello (2004, p. 426), “If writing efficient programs for stable, dedicated parallel machines is difficult, for the grid the problem is even harder. This factor alone is sufficient to inhibit the wide acceptance of grid computing.” In addition, even when parallel programs are feasible, the cost of program conversions can be prohibitive (Donald and Martonosi 2006).

In this research, we explore another dimension of grid computing that avoids some of the grid implementation obstacles described above and increases utilization of a grid infrastructure but has received limited attention in the research literature. In addition to parallel processing, the completion time for computationally intensive applications can be reduced by having machines work on an application *sequentially* over time. It is typical, particularly in a corporate setting, for different computing resources to have utilization rates that vary dramatically over time—machine utilization will experience peak periods as well as lean periods that may not be concurrent for all machines. This is particularly true for grid networks that are geographically dispersed or that are assigned to different functions within an organization. Even when machines are colocated in a centralized data center, such as for a vendor providing on-demand computing resources to a large number of clients, utilization rates of machines allotted to each client will vary based on client usage characteristics. In such a situation, a large background application can be routed sequentially through several machines on the grid, thereby taking advantage of their lean periods to reduce overall job completion time.

Our research builds on this concept of *sequential grid computing* (Berten et al. 2006, Buyya et al. 2002, Sonmez and Gursoy 2007, Yu and Buyya 2006). At first glance, sequential grid computing offers three primary advantages. First, unlike parallel grid computing, it is not necessary to rewrite applications to take advantage of parallel processing—a major implementation bottleneck. In contrast, sequential grid computing requires an interface mechanism that allows for the software to be processed by different machines at different times. Second, relatively simple grid architectures can implement the concept in practice. Once it is estimated where each task will be executed, the relevant software can be sent in advance to those locations. As each task is completed, its intermediate state can be sent to the location where the next task is to be executed. Even in environments where the computing resource availability is stochastic, one can predict the likely

future task assignments and send only the pertinent software modules to those locations. Third, sequential operations are recognized as one of the necessary fundamental building blocks of modern systems (e.g., van der Aalst and Kumar 2003). Thus, for applications specifically designed for parallel computing, sequential grid algorithms can still augment the performance of application segments for which parallel algorithms are not possible. Therefore, our perspective on sequential grid computing is that it is not an alternative to parallel grid computing but rather another mechanism to accrue additional benefits.

In this paper, we develop two models and attendant solution procedures for the sequential grid computing environment that optimally routes a single application or job based on the stochastic availability of idle resources at each time period at each processor. The first model is a static model that determines at the start of processing which machine will process the application in each time period. The second is a dynamic model that provides a policy for allocating the job to a machine in each time period based on the current status of the job. Both models are benchmarked against a single machine assignment in which the entire job is processed by just one machine. The static model is computationally efficient, is simple to implement, and requires little overhead information; the dynamic model is computationally demanding and requires more overhead information but may provide superior solutions. Based on the static model, we also present an heuristic for scheduling multiple jobs on the grid.

This research makes two main contributions to the emerging literature on grid computing in the information systems area. First, although most optimal task scheduling algorithms have focused on parallel grid computing, we address the optimal scheduling problem in the sequential grid computing environment. Second, through extensive computational experiments, we characterize the conditions under which sequential grid computing provides benefits over using a single machine to process the job, and we also identify the conditions under which the dynamic model provides superior schedules compared with the static model. These computational experiments provide a proof of concept for our sequential grid computing scheduling models and yield insights into when the benefits associated with it are the most pronounced.

The rest of this paper is organized as follows. In §2, we describe the sequential grid computing environment and the task assignment problem in detail. In §3, we focus on scheduling a single job and define static and dynamic models along with their respective solution procedures. In §4, we extend the static model to

incorporate the scheduling of multiple jobs simultaneously on the grid. In §§5 and 6, we illustrate the effectiveness of the static and dynamic models using computational experiments with thousands of randomly generated problems that vary in complexity and requirements. Section 7 discusses our computational results, and §8 concludes this paper.

2. Sequential Grid Computing Environment

Following the common architectures of grid computing (e.g., Joseph et al. 2004, Meliksetian et al. 2004), we consider a centralized, software-based grid manager. The grid manager sells idle computing resources to one or more buyers who each have one or more computing jobs. Each such job k has an expected resource requirement of U_k central processing unit (CPU) cycles and a deadline of D_k time units from the time of submission. Similar to recent economic models for grid computing (Bapna et al. 2006, 2008; Sonmez and Gursoy 2007), we map the probability of job completion into economic terms. The price (or reward) for completing a job k of size U_k within deadline D_k is labeled R_k . If job k is not completed within the deadline, the grid manager incurs a penalty cost C_k as part of a service-level agreement (SLA).

For example, banks perform several lengthy jobs at the end of each day as batch processes (e.g., check processing, daily interest calculations, automated clearinghouse transactions, etc.) Consider the end-of-day batch process for automated clearinghouse (ACH) transactions. Because ACH transactions are routinely processed, there is likely to be a reliable estimate of the processing time required based on the number of transactions. Because the account updates are needed before the start of the next business day, there is also an associated deadline. Furthermore, data consistency requirements may dictate that the transactions are processed in sequential order. If the processing is outsourced to a grid provider, the SLA would provide for payments and penalties based on completion within deadlines. Even if the processing is not outsourced, payments and penalties reflect the customer service, inconvenience, and reputation costs associated with noncompletion within deadlines.

Before accepting the job, the grid manager must first estimate the probability of completing the job. This requires estimating the resources available (expressed in CPU cycles) on each machine on the grid. However, estimations of available resources vary not only by machine but also by time. To incorporate the variation of resource availability by time, we partition the time horizon D_k into T_k distinct time periods (see Bapna et al. 2008 for a similar discrete treatment of time). The number of distinct time periods should

be selected after considering the trade-off between accuracy and computational efficiency. Smaller time buckets allow the scheduling algorithms to incorporate idle CPU time estimates at a lower level of granularity, but they increase computational effort. Because idle CPU times in each time bucket are estimates, the duration of a time bucket should reflect the periodicity of such estimates. If it is only possible to estimate idle CPU times hourly, there is no benefit obtained from time buckets that are less in duration.

Processors in a grid typically handle a large number of internal (nongrid) tasks for which the CPU requirements are small. Furthermore, the actual CPU cycles required by these tasks, as well the number that would arrive in a given time period, is uncertain. Assuming that these tasks arrive independently, then because of the central limit theorem (Andersen and Dobrić 1987), it follows that the CPU cycles utilized on a machine in a given time period follow a normal distribution. Because the total amount of CPU cycles available in any time period is fixed, the idle CPU cycles available on a processor after processing all the nongrid tasks would also follow a normal distribution. Thus, we assume that in time period t on processor l , the idle CPU cycles available ($c_{l,t}$) follow a normal distribution with a mean of $\mu_{l,t}$ and a variance of $\nu_{l,t}$ and are estimated by the grid manager from historical data.

Consider that the grid manager has at her disposal a set M of m processors ($|M| = m$) for executing a job k requiring U_k CPU cycles with a deadline of D_k . Given the stochastic nature of resource availability, the grid job may not be completed within the deadline. The grid manager assigns a job to processors in each time period to maximize the expected net reward (ENR). If, for some schedule s_k , the probability of meeting the deadline is $\wp(s_k)$, then

$$\text{ENR}_k = [\wp(s_k) \times R_k - (1 - \wp(s_k)) \times C_k]. \quad (1)$$

For a single job, maximizing Equation (1) is the equivalent of maximizing $\wp(s_k)$. When the grid manager must schedule multiple jobs, we assume that the time horizon is divided into similar time buckets, but the job deadlines (D_k) may differ. The ENR for multiple jobs is computed as follows; however, because the schedules are not independent, we cannot maximize Equation (2) by independently maximizing $\wp(s_k)$ for each job:

$$\text{ENR} = \sum_{k=1}^K [\wp(s_k) \times R_k - (1 - \wp(s_k)) \times C_k]. \quad (2)$$

In the context of the banking ACH example, consider that the grid provider has access to a set of mainframes located around the world that are used to process a variety of other online jobs for the bank, such as teller and ATM transactions, account queries,

and internal bank systems processing. For an external grid provider, the mainframes can also be used to process transactions for other clients. An estimate of the processing time required for the ACH transaction batch job can be generated from historical data. Estimates of the CPU cycles available at each mainframe can also be generated from historical availability data. The time buckets can be of fixed duration (e.g., 30-minute intervals), or they can be adjusted to reflect the periodicity of the estimates or fine-tuned over time to optimize performance. Once an optimal schedule is determined, ACH transactions can be distributed in advance to the mainframes based on the optimal schedule (with some overlap because the actual processing may deviate from the optimal schedule as a result of the stochastic nature of available CPU cycles). At the end of each day, the grid manager can receive other batch jobs to process (e.g., for calculating account daily interest, check processing, etc.) and will need to optimize the schedules of multiple batch jobs simultaneously. The reward and penalty functions for each job will reflect the importance of the job and the disutility from missing processing deadlines.

2.1. Related Literature

There is a large body of research in the operations management and operations research areas on stochastic shop-floor scheduling problems that are relevant to the sequential grid models described here (Allahverdi and Mittenthal 1995, Baker and Scudder 1990, Balut 1973, Kise and Ibaraki 1983, Pinedo and Ross 1980). The basic problem studied in this literature is that of scheduling a number of jobs on multiple machines with stochastic processing times and failure probabilities so as to optimize a variety of performance measures such as number of tardy jobs (Balut 1973, Kise and Ibaraki 1983), earliness and tardiness penalties (Baker and Scudder 1990, Cai and

Zhou 1999), total job time and makespan (Allahverdi and Mittenthal 1995), and the number of successful jobs when machines are unreliable (Herbon et al. 2005, Pinedo and Ross 1980). Although a review of this literature is beyond the scope of this paper, there are two unique characteristics of the sequential grid computing environment studied here. First, unlike the stochastic shop-floor scheduling literature, we consider a single large job that is routed from one processor to another utilizing unused CPU cycles until the job is completed. Because processors on a grid may be geographically dispersed or may serve different clients of a grid provider, their peak utilizations do not occur concurrently. Thus, it is advantageous to route a single job through multiple processors to exploit unused CPU cycles, a situation that does not have an equivalent in the stochastic shop-floor scheduling literature. Second, we focus on optimizing completion probabilities within a specified time period. Existing research in the scheduling literature has not modeled both of these dimensions of relevance to the sequential grid computing environment.

2.2. Network Representation

We initially focus on the case where the grid manager is examining the request for a single job that needs to be processed on the grid, and we later extend the analysis to multiple jobs. For simplicity of presentation, we drop the subscript k that denotes a job from our discussion. The problem can be represented on an acyclic directed network $G(N, A)$. As shown in Figure 1, the node set N is arranged into rows and columns. The rows are labeled from 0 to $T + 1$. Rows 0 and $T + 1$ consist of singleton nodes, with the former representing the source node (start of processing) and the latter representing the terminal node n (end of processing). Nodes belonging to rows 1 through T are arranged into m columns, with each column associated with a processor. Thus, a node at the intersection

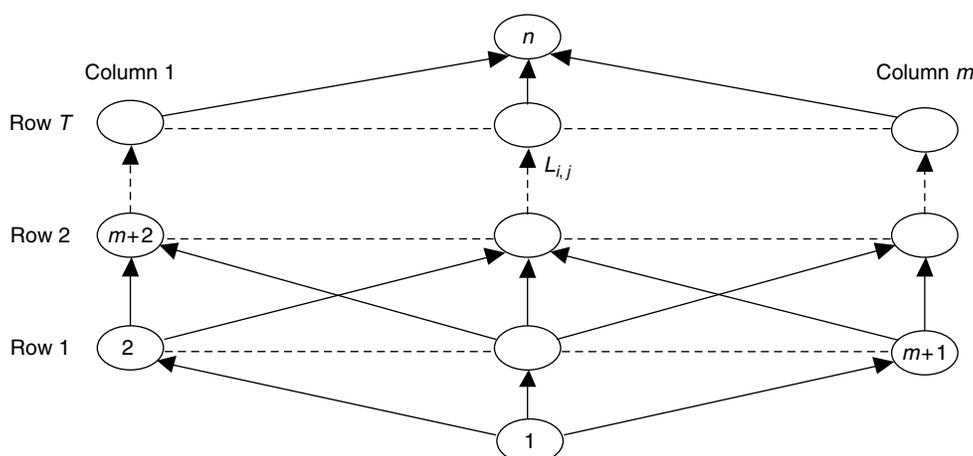


Figure 1 Network Representation of the Sequential Grid

of row t and column i ($1 \leq i \leq m$, $1 \leq t \leq T$) represents processor i in time period t . Hence, the node set N consists of $mT + 2$ nodes, numbered as shown.

The arc set A includes *continuation* arcs and *transfer* arcs. First, there is an arc from source node 1 to every node in row 1. Similarly, there is an arc from each node in row t to every node in row $t + 1$, where $t = 1, \dots, T$. Finally, an arc $(i, j) \in A$ with j lying at the intersection of column l and row t ($t = 1, \dots, T$ and $l = 1, \dots, m$) represents the action of assigning the job to processor l in time period t . The length of each such arc $(L_{i,j})$ represents the stochastic availability of CPU cycles from processor l in time period t . A *complete* path from node 1 to node n represents a *schedule*.

Let p denote a complete path and P denote the collection of all such paths p in $G(N, A)$. We define an arc (i, j) to be a *continuation* arc if the processors corresponding to nodes i and j in $G(N, A)$ are the same and a *transfer* arc if the processors corresponding to i and j in $G(N, A)$ are different. If (i, j) is a *continuation* arc with node j being associated with row (time period) t and column (processor) l , then its length $L_{i,j}$ corresponds to $c_{l,t}$ defined earlier and follows a normal distribution with a mean of $\mu_{l,t}$ and a variance of $\nu_{l,t}$. Also, if (i, j) is a *transfer* arc, then there is a transfer cost f_{l_1,l_2} . We express the transfer cost in terms of processing cycles because processing time is lost as a result of the transfer of the job from l_1 to l_2 . Hence, for such arcs, the length $L_{i,j}$ has a mean of $(\mu_{l_1,t} - f_{l_1,l_2})$ and a variance of $\nu_{l_1,t}$.

In the context of our banking example, the columns in Figure 1 represent designated bank mainframes that are available to the grid manager for batch processing. The rows are the time buckets into which the batch processing duration is divided. For example, all U.S. batch processing can be scheduled between midnight and 4:00 A.M. Eastern Standard Time, divided into eight time buckets (rows) of 30 minutes duration each. It is important to note that although a job could potentially be transferred to another processor at any time by communicating its intermediate state (variable values, registers, temporary files, etc.), such transfers are significantly easier at key transition points (e.g., at the end of a module). Clearly, such transition points may not coincide with time buckets because processing times are stochastic. However, if software programs are written in a modular fashion (a common software engineering practice), the grid manager will need to wait a short time at the end of a time bucket for the module to complete before transitioning the job. Thus, deviations from the schedule will be small and our algorithms are likely to provide computationally efficient approximations.

3. Sequential Grid Models for a Single Job

3.1. Problem Formulation

We now construct two models to help the grid manager schedule a single job on the grid. We then describe solution procedures for optimally solving these two models. The first is a *static* model that generates a static schedule (a list of T processors, one for each time period, to which the job is assigned). This schedule is sent with the job so that it can be routed by each processor at the end of each time period. Thus, the overhead information that needs to be transmitted with the job to implement the static model is minimal. The second is a *dynamic* model, the output of which is not a predetermined schedule but rather an optimal policy. Let u_t represent the cumulative CPU cycles obtained by the job thus far in time period t . The optimal policy specifies, for each node in $G(N, A)$ and for each possible value of u_t at that node ($1 \leq u_t \leq U$), the next processor that the job is assigned to in period $t + 1$. Clearly, to implement the dynamic model, significantly more control information needs to be transmitted with the job. In addition, the computational requirements of the dynamic model are several orders of magnitude higher.

3.2. The Static Model for Single-Job Assignment

In the discussion for scheduling a single job, we drop the subscript k (denoting the job) for simplicity of presentation. Let L_p denote the length of a path p in $G(N, A)$; i.e.,

$$L_p = \sum_{(i,j) \in p} L_{i,j}. \quad (3)$$

We maximize ENR for which it suffices to maximize the probability of completion within the deadline. Accordingly, the *static* model, denoted as PS-1 (indicating the scheduling of a single job), can now be stated as selecting a path in $G(N, A)$ that maximizes the probability of completion:

$$(PS-1) \quad \text{Maximize } \{\wp(L_p \geq U) | p \in P\},$$

where \wp denotes probability.

In PS-1, because each path $p \in P$ is composed of arcs whose lengths are independent and normally distributed random variables, the path length L_p is also normally distributed with a mean of μ_p and a variance of ν_p , where $\mu_p = \sum_{(i,j) \in p} \mu_{i,j}$ and $\nu_p = \sum_{(i,j) \in p} \nu_{i,j}$. Because $\wp(L_p \geq U)$ in (PS-1) is monotonic in the reduced Gaussian, $g(\mu_p, \nu_p) = (\mu_p - U) / \sqrt{\nu_p}$, PS-1 is reduced to the deterministic equivalent of maximizing $g(\mu_p, \nu_p)$ over the set P .

To evaluate the computational complexity of PS-1, we consider the decision version of PS-1—does there exist a path $p \in P$ such that $g(\mu_p, \nu_p) \geq L$? For the

special case when $L = 0$, we simply solve the longest-path problem on this acyclic graph with the arc length $\mu_{i,j}$ for each $(i, j) \in A$. If the path length is greater than or equal to U , then the answer is yes; otherwise, it is no. If L is not zero, then path variance makes the problem more complex. When $L > 0$ and the maximum mean path has a length that exceeds U , Nikolova et al. (2006) provide a quasi-polynomial time whose running time is $O(n^{\theta(\log n)})$. Thus, the existence of a true polynomial-time algorithm for such instances is an open question. If $L < 0$ and the maximum mean path length is less than U , the decision version of PS-1 is NP-complete (Karger et al. 1997, Nikolova et al. 2006).

3.2.1. Best Single-Processor Assignment Algorithm. First, we describe an easy algorithm that we refer to in the computational experiments as the best *single-processor* assignment (PA-1). If the path p is restricted to the use of *continuation* arcs alone, then determining the optimal path becomes easy. This is the case when the job is assigned to the *same* machine over all time periods. Let $P' \subset P$ denote the subset of paths that consists of *continuation* arcs alone. Note that because $|P'| = m$, PA-1 can be solved quickly through enumeration:

$$(PA-1) \quad \text{Maximize } \{\phi(L_p \geq U) | p \in P'\}.$$

3.2.2. Characteristics of the Reduced Gaussian. The solution method presented here for PS-1 is a modification of the stochastic shortest-path algorithm (Murthy and Sarkar 1998) to suit the special structure of PS-1. Lemmas 1, 2, and 3 present some results about the nature of the function $g(\mu_p, \nu_p)$ that will be used in our algorithm to solve PS-1. Some of these results are straightforward while others are based on results that have appeared earlier in the literature, most notably in Henig (1990). For that reason, we state Lemmas 1, 2, and 3 here without proof.

LEMMA 1. Consider two paths $p_1, p_2 \in P$ such that (i) $\mu_{p_1} \geq U$ and (ii) $\mu_{p_2} < U$. Then, $g(\mu_{p_1}, \nu_{p_1}) > g(\mu_{p_2}, \nu_{p_2})$ for any $\nu_{p_1}, \nu_{p_2} > 0$.

The significance of this result is that if there exists a path $p \in P$ whose $\mu_p \geq U$, then all paths \hat{p} whose mean length $\mu_{\hat{p}} < U$ can be ignored. The existence question can be answered efficiently by solving a single longest-path problem (not stochastic) on $G(N, A)$ using $\mu_{i,j}$ as arc lengths for each $(i, j) \in A$. If the answer to the existence question is positive (Case 1), then we restrict our attention to only those paths $p \in P$ whose $\mu_p \geq U$. If the answer is negative (Case 2), then we know that $\mu_p < U$ for all $p \in P$. Thus, PS-1 is partitioned into two dichotomous cases.

LEMMA 2. The function g is increasing in μ and decreasing in ν for all $\mu > U$ and $\nu > 0$ (Case 1) while increasing in μ and increasing in ν for all $\mu \leq U$ and $\nu > 0$ (Case 2).

LEMMA 3. The function g is quasi convex in μ and ν for all $\mu > U$ and $\nu > 0$ (Case 1) and quasi concave in μ and ν for all $\mu \leq U$ and $\nu > 0$ (Case 2).

3.2.3. Pruning Rules. Based on the previously stated results, the algorithmic approach we use recognizes and prunes as many subpaths as possible that are not part of the optimal path. The algorithm incorporates two basic approaches to pruning nonoptimal subpaths based on the lemmas: (a) *local preference relations* (based on Lemma 2) and (b) *upper bound comparisons* (based on Lemma 3). The pruning significantly improves the performance of the stochastic shortest-path algorithm.

In pruning based on *local preference relations*, we use two rules based on the two cases identified above. Consider $p_1(j)$ and $p_2(j)$ denoting two subpaths from node 1 (start node) to node j . The subpath $p_1(j)$ dominates $p_2(j)$ if there is at least one feasible extension of $p_1(j)$ to node n that is at least as good as all feasible extensions of $p_2(j)$ to node n . In such a case, the subpath $p_2(j)$ can be discarded (pruned). The rules that determine the conditions when one subpath dominates another are referred to as *local preference relations*. The following two pruning rules are based on Lemma 2 for Case 1 and Case 2, respectively; Rule 1 applies for Case 1 and Rule 2 applies for Case 2.

RULE 1. The subpath $p_1(j)$ dominates $p_2(j)$ if (a) $\mu_{p_1(j)} \geq \mu_{p_2(j)}$ and (b) $\nu_{p_1(j)} \leq \nu_{p_2(j)}$.

RULE 2. The subpath $p_1(j)$ dominates $p_2(j)$ if (a) $\mu_{p_1(j)} \geq \mu_{p_2(j)}$ and (b) $\nu_{p_1(j)} \geq \nu_{p_2(j)}$.

In terms of pruning based on upper bound comparisons, the basic algorithmic approach is to compare the best extension of a newly created path $p^{\text{new}}(j)$ from node 1 to node j to a current best-known feasible path p^l . If the best extension of $p^{\text{new}}(j)$ results in a path that is no better than p^l , then $p^{\text{new}}(j)$ can be discarded. Let $p(j)$ denote a path from node j to node n (terminal node) whose mean and variance are denoted as μ_j and ν_j , respectively. The best extension of $p^{\text{new}}(j)$ can be obtained by solving the subproblem

$$(SPS-1) \quad \text{Maximize } [g(\bar{\mu}_j, \bar{\nu}_j) | p(j) \in P(j)],$$

where $\bar{\mu}_j = \mu^{\text{new}}(j) + \mu_j$, $\bar{\nu}_j = \nu^{\text{new}}(j) + \nu_j$, and $P(j)$ is the set of all feasible paths from j to node n (terminal node). Of course, SPS-1 is as hard to solve as PS-1 and hence we consider suitable relaxations of SPS-1 that utilize Lemma 3 to obtain an upper bound on the best extension of $p^{\text{new}}(j)$ by taking advantage of the quasi-convex (Case 1) and quasi-concave (Case 2) nature of g . This value is compared to a current best-feasible path p^l and is pruned accordingly.

3.2.4. Algorithmic Approach for the Static Model.

For simplicity of presentation, we omit the details of the algorithm used to solve PS-1. The approach is based on a well-known labeling procedure (see Murthy and Sarkar 1998) that uses the pruning rules described earlier. The procedure starts at node 1 and proceeds towards node n , processing nodes sequentially. At each node, the procedure stores all the *nondominated* paths from node 1 to that node. The two pruning methods described earlier substantially improve the performance of the labeling procedure (Murthy and Sarkar 1998). When node n is reached, the procedure picks the best path from the pruned set of nondominated paths. If \wp^* is the corresponding optimal completion probability for the best path, the optimal ENR from the static model can be obtained from substituting \wp^* in (1).

3.3. The Dynamic Model for Single-Job Assignment

The dynamic model is a stochastic control problem that is solved using *dynamic programming*. To frame this problem as a dynamic program, consider it as consisting of T stages. At each stage t , the job is in state s_t , defined by the tuple $\{i, u_i\}$, where $i \in N$ is a node in $G(N, A)$, and u_i is the cumulative amount of processing units obtained by the job thus far. Furthermore, at stage t , imagine a random process $\omega_t \in W$ that generates the arc lengths $c_{i,j}$ randomly from their respective distributions for each (i, j) emanating out of node i . Traversing arc (i, j) corresponds to assigning the job on machine j from which the actual CPU time obtained is a random variable drawn from a normal distribution with a mean of $\mu_{i,j}$ and a variance of $\nu_{i,j}$, the realization of which is known only *after* the grid manager has taken a decision. The grid manager now has to choose a decision x_t from a *feasible* choice set, $X(s_t)$; i.e., $x_t \in X(s_t)$. Here, $X(s_t)$ constitutes the forward star $f(i)$, the set of all arcs $(i, j) \in A$ that originate from i . Using a *decision rule* $h_t: S \times W \rightarrow X$, the grid manager takes the decision x_t , i.e., $x_t = h_t(s_t, \omega_t)$, which amounts to selecting an arc $(i, j) \in f(i)$. As a result, the job moves to a new state s_{t+1} in stage $t+1$.

The sequence of decision rules $\pi_T = [h_0, h_1, \dots, h_T]$ constitutes a policy. In simple terms, the policy will specify for each node $i \in N$ and for each value of $u_i \leq U$ (i.e., for each state s_t) the optimal decision x_t (which node in $G(N, A)$ to move to in the next stage). As a practical matter (because U is relatively large) an approximation u_i is assumed to take on a discrete set of values (or states), $0, 1, \dots, U$. Let π denote the set of all feasible policies. Because of the finiteness of N , $f(i)$, and U , the state space S and the decision set X are also finite. As a result, π is also finite. Let the value function $\mathfrak{Z}(\pi_T)$ be ENR as defined in Equation (1). The dynamic programming model is

$$(PD-1) \quad \text{Maximize } \{\mathfrak{Z}(\pi_T) \mid \pi_T \in \pi\}.$$

To solve PD-1 using dynamic programming, we frame the recursive Bellman equation (see Equation (4)) in the following way. Suppose that the value function $F_t(i, u_i)$ denotes the optimal ENR from stage t onward given that, as represented by the state s_t , the job is at node i , having obtained u_i cumulative units of CPU thus far. Furthermore, let $p_{i,j}(k)$ denote the probability of obtaining k units of CPU from traversing arc (i, j) , for $k = 0, \dots, U - u_i$. Let the probability of obtaining more than $(U - u_i)$ CPU cycles from traversing arc (i, j) be $p_{i,j}(U+)$. The recursive Bellman equation is

$$F_t(i, u_i) = \text{Max}_{(i,j) \in f(i)} \left(\sum_{k=0}^{U-u_i} p_{i,j}(k) \times F_{t+1}(j, u_i + k) + p_{i,j}(U+) \times F_{t+1}(j, U) \right). \quad (4)$$

The term within the parentheses in (4) is the expected value function in stage $t+1$ if the grid manager chooses to traverse link (i, j) . The optimal value of $F_t(i, u_i)$ is obtained by choosing the link (i, j) that maximizes this expected value. The recursive equation is solved by working backward from the last row T . The boundary conditions that apply for all nodes j in row T is $F_T(j, k) = -C$ (penalty for noncompletion) for $k = 0, \dots, U - 1$ and $F_T(j, k) = R$ (reward for completion) for $k \geq U$. The solution to PD-1 corresponds to the value function F_1 (Allahverdi and Mittenthal 1995). The computational effort required to solve PD-1 using the recursive Equation (4) is $O(n^2U^2)$. In summary, the dynamic model develops a policy that specifies for each node $i \in N$ and for each value of $u_i \leq U$ (where u_i is the CPU cycles obtained thus far by the job) the machine where the job will be processed in the next time period. However, transmitting this policy to the distributed grid manager software at each location requires more control information to be attached and significantly greater computation time.

3.3.1. Comparing the Dynamic and Static Models. The optimal policy obtained from solving PD-1 is superior to the optimal solution obtained from solving the static model (PS-1) because the dynamic model implicitly includes the static solution and therefore evolves a policy that is at least as good as the static solution. To illustrate, consider the simple graph shown in Figure 2 that illustrates the mean and variance $(\mu_{i,j}, \nu_{i,j})$ of the CPU obtained by traversing each arc (i, j) . Suppose that the CPU required $U = 45$ units. From the static model, the optimal path is $1 - 2 - 4 - 5$ and not $1 - 2 - 3 - 5$ because the standard normal associated with path $1 - 2 - 4 - 5$ is $z_1 = (60 - 45)/\sqrt{73} = 1.76$, whereas that associated with $1 - 2 - 3 - 5$ is $z_2 = (50 - 45)/\sqrt{25} = 1.00$. This

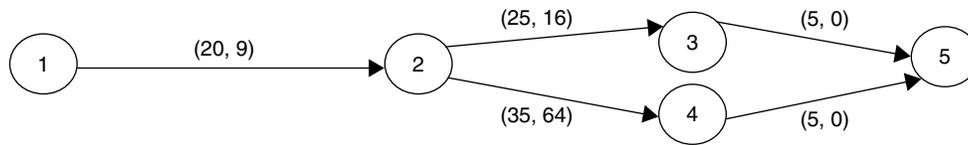


Figure 2 Static and Dynamic Paths for a Simple Graph

implies that path 1 – 2 – 4 – 5 must be traversed irrespective of the actual CPU obtained upon arriving at node 2. Instead, after reaching node 2, if it is discovered that 30 units have been obtained so far, traversing path 2 – 3 – 5 would yield a better chance of meeting the requirement of 45 units than the path 2 – 4 – 5. The z value associated with the former is $z = (30 + 25 + 5 - 45)/\sqrt{16} = 3.75$ and that associated with the latter is $z = (30 + 35 + 5 - 45)/\sqrt{64} = 3.13$. Therefore, the chance of meeting the deadline requirement is better by following a policy that allows for varying the route based on information available at node 2.

4. Sequential Grid Models for Multiple Jobs

We now examine the case where buyers approach the grid manager with requests for processing K jobs on the grid with $K > 1$. Each job k requires U_k units and carries a reward R_k if it is completed on time and a penalty C_k otherwise. We assume that there are a sufficient number of processors on the grid; i.e., $m \gg K$. Furthermore, each processor can process only one grid-supplied job at a time. We consider two heuristic approaches for scheduling the K jobs on the grid. Both maximize the ENR (Equation (2)). The first approach is the single-period assignment problem, which is a direct extension of PA-1. Each job k is assigned to a different processor l , and this assignment remains unchanged over the entire duration of T time periods. The second approach is the multiperiod static assignment problem and is a direct extension of the static model (PS-1). Each job k is assigned to a different processor l , but each job is allowed to be processed by different processors in each time period. However, like PS-1, the schedule that is developed is considered static because it does not change based on the state achieved at a node.

4.1. Single-Period Assignment for Multiple Jobs

Because this problem is a direct extension of PA-1, we will refer to it as PA- K , where K jobs have to be assigned to m different machines. As described for PA-1, let $P' \subset P$ denote the set of paths in $G(N, A)$ consisting of only continuation arcs. Hence, $|P'| = m$ and traversing each such path amounts to the job being processed by a single specific machine. All paths $p_l \in P'$ are node disjoint except for the starting and ending nodes. Associated with each path $p_l \in P'$,

the mean-variance pair (μ_l, ν_l) can be obtained as $\mu_l = \sum_{(i,j) \in p_l} \mu_{i,j}$ and $\nu_l = \sum_{(i,j) \in p_l} \nu_{i,j}$. If a job k is assigned to machine l , then the probability of its completion, $\wp_{k,l} = \Pr[z \leq (\mu_l - U_k)/\sqrt{\nu_l}]$, can be determined using the normal distribution. Accordingly, the ENR obtained from assigning job k to machine l can be determined as $ENR_{k,l} = (R_k + C_k)\wp_{k,l} - C_k$ for each $k = 1, \dots, K$ and $l = 1, \dots, m$. Because $m \gg K$, additional $(m - K)$ dummy jobs are created whose ENR is zero when assigned to any machine l . PA- K can be solved as the classical single assignment problem, where m jobs are assigned to m processors so that the total ENR is maximized.

4.2. Multiperiod Static Assignment Problem with Multiple Jobs

Although PA- K can be solved efficiently, the quality of the solution obtained may not be good because it does not take advantage of sequential grid computing. We now consider the assignment of K jobs to K of the m available machines while allowing the assignment to vary over the T periods. However, the assignments over the T periods are determined a priori and are hence static. Relating this problem to the graph in Figure 1, each job k traverses the acyclic network from node 1 to node n . Such a traversal amounts to assigning job k to different processors over T periods. Because each processor can process at most one grid-supplied job in each time period, the K paths are node disjoint except for node 1 and node n . The problem then is to determine K node disjoint paths, one for each job, so that the total ENR is maximized. This problem is a direct extension of PS-1 to K jobs and is therefore referred to as PS- K .

4.2.1. Computational Complexity of PS- K . It can be shown that problem PS- K is NP-hard. The decision version of PS- K is as follows: Does there exist K node disjoint paths in the acyclic graph $G(N, A)$ such that the total ENR is at least W ? We show in the Online Supplement (available at <http://joc.pubs.informs.org/ecompanion.html>) that the decision version of PS- K is NP-complete. Given an acyclic graph $G(N, A)$, where each $(i, j) \in A$ has an integer-valued arc length $c_{i,j}$, problem MaxMinD- K is defined as that of finding K node disjoint paths so that the path length of the longest path amongst these K paths is minimized, which is known to be NP-hard. We show that the decision version of MaxMinD- K

reduces to an instance of the decision version of PS-K. The theorem is stated here without proof.

THEOREM 1. *The decision version of PS-K for $K \geq 2$ is NP-complete (proof is in the Online Supplement).*

4.2.2. An Efficient Heuristic for PS-K. Because PS-K is shown to be NP-hard, it is reasonable to explore fast heuristics that derive good workable schedules. In the next section, we empirically explore the following simple heuristic. The K jobs are sorted in decreasing order of $(R_k + C_k)$, the sum of the reward and penalty. It is assumed that this ordering is consistent with the ordering by U_k ; that is, jobs with greater computational requirements carry a greater price and penalty. The heuristic involves solving K number of PS-1 in sequence. The first PS-1 problem solved uses the original parameters $(\mu_{i,j}, \nu_{i,j})$ for each $(i, j) \in A$. As a result, a static path is obtained where each intermediate node corresponds to a machine assignment. After determining the ENR associated with the first job, the machines used are removed from consideration for subsequent jobs. This process is repeated K times, after which we have schedules for all K jobs.

5. Computational Results for Single-Job Models

To evaluate the performance of the static and dynamic models for a single-job assignment, we coded the two models PS-1 and PD-1 using C++ and ran several thousand instances using randomly generated input data. The purpose of our computational experiments was twofold: (a) to understand the factors that affect the benefits from sequential grid computing by comparing the completion probabilities provided by the static and dynamic models (PS-1 and PD-1, respectively) with that obtained by performing the job on the same machine (PA-1), and (b) to understand the factors that affect the difference in completion probabilities obtained by the static versus dynamic models. The first analysis determines the conditions when sequential grid computing provides the greatest benefits, and the second analysis explores whether the benefits from the dynamic model outweigh its additional complexity.

We focus on the impact of three characteristics of the sequential grid-computing environment on completion probabilities—(1) the job size (using the CPU cycles required as a representative metric), (2) the grid resources available (using the number of processors available as a representative metric), and (3) the heterogeneity of available grid resources (using the variance of CPU cycles available at each processor as a representative metric). These three factors capture key differences in grid environments likely to affect the benefits from sequential grid computing.

5.1. Parameters for Problem Instances

The results excerpted for presentation are based on a subset of 3,100 instances with varying job sizes and estimates of the mean and variance of CPU cycles available at each processor in each time period. As a benchmark for the static and dynamic models we also estimated the probability of job completion for each of the 3,100 instances, assuming that the job was assigned to the single best processor for all time periods (PA-1). PA-1 estimates the best-case completion probability without sequential grid computing. The mean CPU cycles available at the processor in each time period were randomly selected from a uniform distribution [95, 105] units. The corresponding variance was also selected from a uniform distribution [5, 10] units. Transfer cost was fixed at one CPU cycle to evaluate situations where transfer costs are low because high transfer costs will simply impede sequential grid computing. To simulate peak loads of machines, during one-third of randomly chosen time periods the available CPU cycles were reduced to 20% of the maximum capacity. The metric of CPU cycles is intended to be an abstract relative measure of resources required to resources available rather than a specific absolute measure; the models can also be applied to specific resources (e.g., CPU, memory, or storage). Interestingly, a grid of 15 personal computers was used to run the computational experiments.

5.2. Model Performance and Job Size

First, we investigated the effects of job size on the relative performances of PS-1 and PD-1. We used the CPU requirements of the submitted job as the focal metric. The grid was composed of 100 machines operating over five time periods. Figure 3 shows that the improvement over PA-1 is most pronounced within a range in the middle section of the figure, with a 100% maximum improvement in probability of completion. The intuition behind these results is straightforward. For a small job where the probability of completion is nearly one, there is little benefit in routing the job through multiple processors because even a single processor provides good solutions. Conversely, when

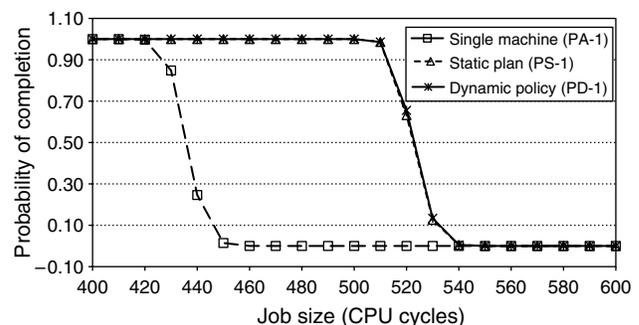


Figure 3 Performance and Job Size

the job size is so high that even the static and dynamic models yield low probabilities of completion, there is once again little benefit from multiple processors. Within these extremes, sequential grid computing provides significant improvement over the single machine best case. Although these general results hold for any variation in parameters of the peak period, the benefits of the sequential grid-computing models are more pronounced as either (a) the length of the peak period increases or (b) the resource availability during the peak period is reduced. Little performance difference is seen between the static plan and dynamic policy.

5.3. Model Performance and Resources Available

Next, we investigated the impact of the resources available to the grid manager on the performance of the two models (PS-1 and PD-1) and the benefits from sequential grid computing. We used the number of processors on the grid as the focal metric. The job required 430 CPU cycles and the grid operated over five time periods. Based on these parameters, the results depicted in Figure 4 show the improvement in probability of completion relative to PA-1, the best single-machine case.

As the number of processors available increases, under the sequential grid models (both the static plan and dynamic policy), the completion probability increases dramatically at the initial stages (Figure 4) while the improvement flattens out as the completion probability reaches close to one. On the other hand, the completion probability in the single-processor case exhibits slower stepwise improvement as the number of available processors increases. The identity of the best processor changes infrequently as processors are added in the single-processor case. For example, in our reported result, the 14th machine added has a large capacity and dramatically increases the completion probability. This machine is selected in future samples because no subsequent machines match its capacity. The single-processor case is not

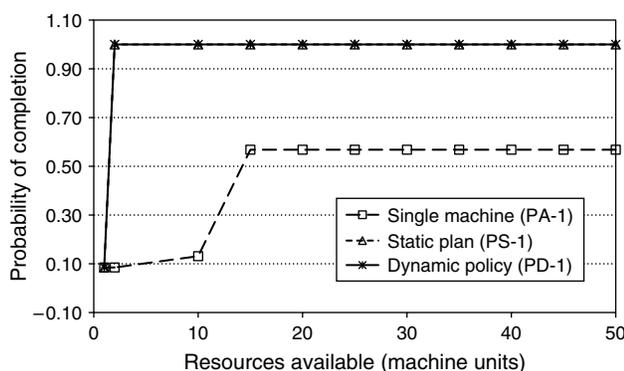


Figure 4 Performance and Resources

able to achieve better than a 50% probability of completion. In contrast, with the static and dynamic models, each new processor added to the grid provides incrementally more flexibility to the grid manager and quickly increases the probability of job completion to one. The improvement is highest at smaller grid sizes and then diminishes but still remains substantial throughout the experiments. Again, little performance difference is seen between the static plan and dynamic policy.

5.4. Model Performance and Resource Heterogeneity

We investigated the effects of the resource heterogeneity on completion probability. In this experiment, we used the variance of CPU cycles available at each processor in each time period as the focal metric. The grid was composed of 100 machines operating over five time periods. The variance of CPU cycles available in each period for each processor was randomly generated from a uniform distribution $([1, V])$ units, where V is the value shown on the x axis of Figures 5, 6, and 7. This experiment explored three demand scenarios—low (425 CPU cycles required; see Figure 5), medium (475 CPU cycles required; see Figure 6), and high (525 CPU cycles required; see Figure 7).

For the results shown in Figure 5, a small enough job demand was selected such that it was likely that a single machine had mean CPU cycles available to complete the job. Thus, with low variance in CPU cycles available, the probability of job completion in the single-machine best case is high. The probability of completion diminishes in the single-machine case as the variance in CPU cycles increases. However, the sequential grid computing models are robust to the increase in variance because the grid manager is able to work around potential problems.

For the results shown in Figure 6, a medium-sized job demand was selected such that it was unlikely that a single machine had mean CPU cycles available to complete the job, but there was still a relatively high availability of processing power on the grid compared to the job size. The medium job size is independent of resource heterogeneity. The single-processor case is unlikely to complete the job; the sequential grid models are almost guaranteed to finish. Again, there is minimal difference between the static plan and dynamic policy.

For the results shown in Figure 7, a high-demand job size was selected such that it was unlikely that a single machine had mean CPU cycles available to complete the job, and there was a relatively low availability of processing power on the grid compared to the job size. Single-machine assignment is unlikely to complete the job with any variation in

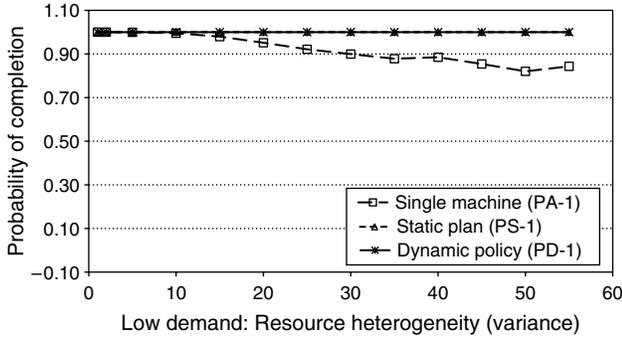


Figure 5 Performance and Heterogeneity with Low Demand

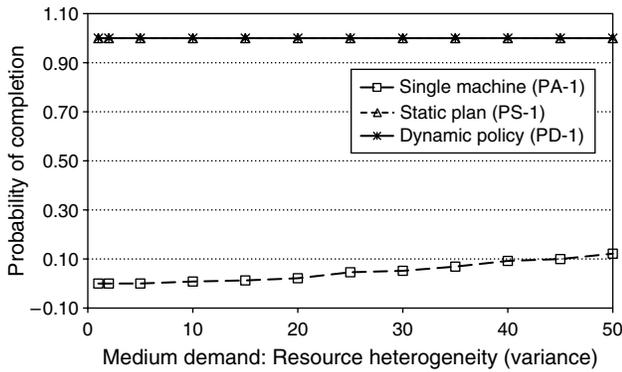


Figure 6 Performance and Heterogeneity with Medium Demand

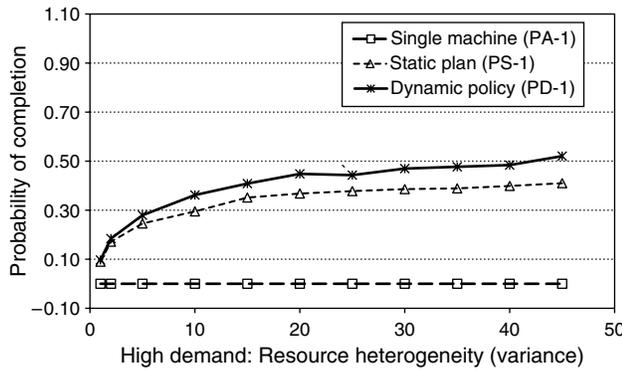


Figure 7 Performance and Heterogeneity with High Demand

resource heterogeneity; however, the sequential grid models begin with a low probability of completion but rapidly increase as the heterogeneity increases. The sequential models achieve 50% or higher probability of completion at higher levels of variance in machine availability. In this scenario, again there is clear value to sequential grid computing as the sequential models route the job intelligently through the grid. Interestingly, the dynamic policy shows a slight increase in performance; we explore this difference further in the next section.

5.5. Comparison of a Dynamic Policy vs. a Static Plan

The computational experiments show definite evidence of performance benefits from using the sequential grid models. However, in the majority of cases, there were few differences between the static plan and the dynamic policy. The dynamic policy subsumes the static plan; therefore, it is possible to use the dynamic policy alone. Unfortunately, the dynamic policy requires significantly more computational time and routing overhead.

To provide evidence of the contrast in computational requirements, we investigated the effects of the instance size on the calculation runtimes of the static (PS-1) and dynamic (PD-1) models. With a similar setup as in previous experiments, this trial used five processing periods to complete a job of 500 CPU cycles. The results are shown in Figure 8. As expected, PS-1 requires little processing time compared to PD-1. Furthermore, as the problem size increases, the processing time for the dynamic model increases significantly while the corresponding processing times for the static model remains at a relatively constant low level. For reference, the runtimes reported were found using a 2.13 GHz Pentium processor with 2.0 GB of memory.

Therefore, if sequential grid models are used, when should a grid manager select a dynamic policy over a static plan? We investigated thousands of problem instances with varying parameters and discovered two situations when the dynamic policy can be advantageous over the static plan—when the job is behind schedule and when this deviation occurs during the early stages of the job. The experiments described below have a similar setup as prior experiments with a grid of 50 machines available over 10 time periods for a job requiring 1,040 CPU cycles. We compare the probability of completion from the dynamic policy over the static plan.

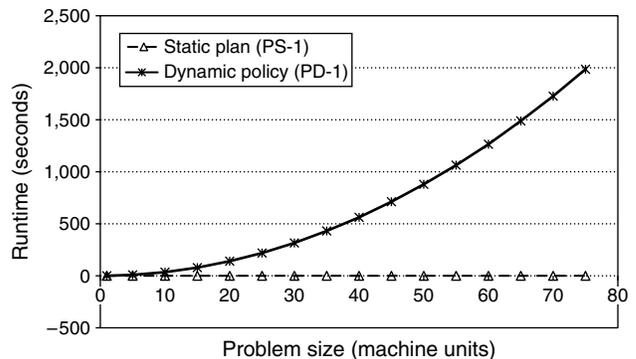


Figure 8 Runtime for the Static and Dynamic Models

5.6. Comparative Performance by Time Until Deadline

First, we examine the relative performance of the dynamic policy versus the static plan as the deadline for completion approaches and the job is behind or ahead of schedule. At any time period t , let u_t be the cumulative amount of CPU time obtained by the job thus far. We quantify the deviation of the job from plan through the variable Z_t , defined as the difference between the expected value of the remaining available CPU times on the static path (reduced by any applicable transfer costs), $\sum_{k=t+1}^{T-1} (c_{l,k} - f_{k,k+1})$, and the amount of processing required to complete the job, $U - u_t$, divided by the square root of the variance in the remaining available processing, $\sum_{k=t+1}^{T-1} v_{l,k}$. Thus, positive values of Z_t represent a job ahead of schedule, and negative values of Z_t represent a job behind schedule. For the static plan, we first determine the overall static schedule and calculate the probability of completion assuming that the job remains on the original static schedule irrespective of the value of u_t (and hence Z_t). For the dynamic plan, we use the value of u_t to determine the new optimal path from the stored dynamic policy for that node.

Figure 9 depicts the increase in probability of completion from using the dynamic policy over the static plan for deviations that occur at different time periods. In Figure 9, we use the specific Z_t -value shown to calculate u_t and the corresponding completion probabilities from the static and dynamic models. For jobs that are significantly behind schedule ($Z_t \ll 0$), there is some increase in probability of completion by using the dynamic policy when the deviations occur during early periods. However, as the deadline approaches there is little chance of recovery for either the dynamic policy or static plan. Alternatively, for jobs that are significantly ahead of schedule, the dynamic policy provides little increase in probability of completion

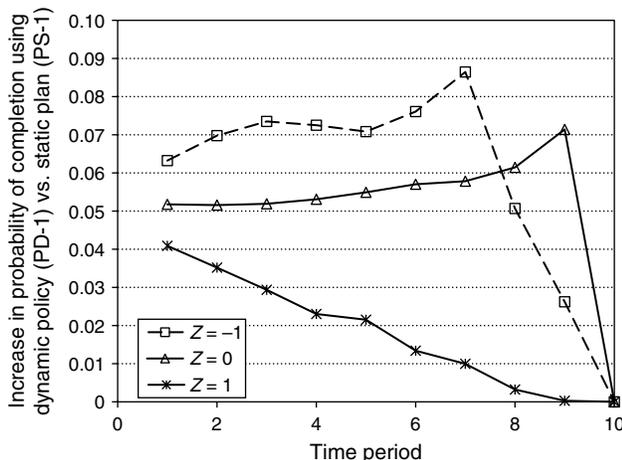


Figure 9 Comparison of Dynamic vs. Static by the Time Until Deadline

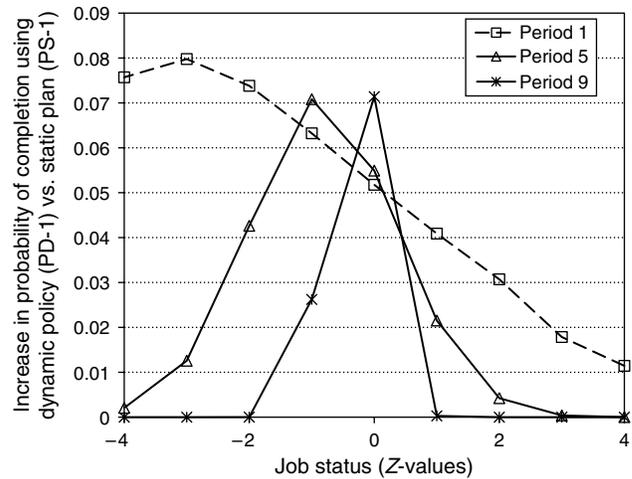


Figure 10 Comparison of Dynamic vs. Static by Job Status

because both models are likely to complete successfully. Thus, for jobs that are behind schedule, the dynamic policy preserves more options for completing jobs until much later in the processing schedule.

5.7. Comparative Performance by Job Status

Alternatively, we can view the results from the perspective of the relative performance of the dynamic policy versus the static plan by the job status. Figure 10 depicts the increase in probability of completion from using the dynamic policy over the static plan for a range of job states (Z_t -values). The data are generated in exactly the same way as in Figure 9. However, in Figure 10, each line in represents the time period in the processing schedule where the deviation occurs. When the deviation occurs early (period 1), the dynamic model provides improvements even when the job is significantly behind schedule ($Z_t \ll 0$). When the deviation occurs during later periods, the dynamic model shows improvements only when the deviation is small (Z_t is close to 0). Overall, the value of the dynamic program is highest when the deviations occur early in the processing schedule.

6. Model Performance with Multiple Jobs

We now consider the efficacy of the static model when multiple jobs are scheduled. We use the number of jobs available as the focal metric, keeping the grid size constant. Because we do not evaluate the dynamic alternative, we can consider larger grid sizes. For the experiment reported, the grid contains 100 machines evaluated across 10 time periods. Jobs generated required an average of 1,040 CPU cycles. Expected revenue for each job was set at US\$2 per CPU cycle requested, and the penalty was allowed to vary uniformly from \$100 to \$500. Transfer costs

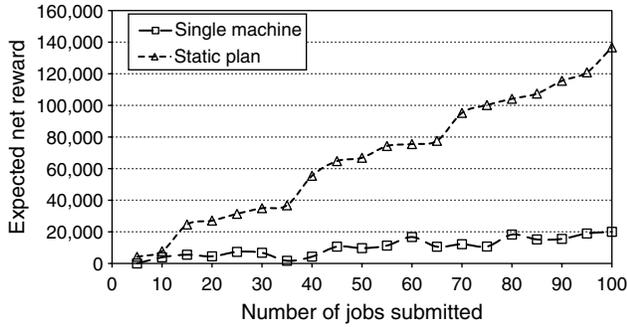


Figure 11 Expected Net Reward for Multiple Jobs

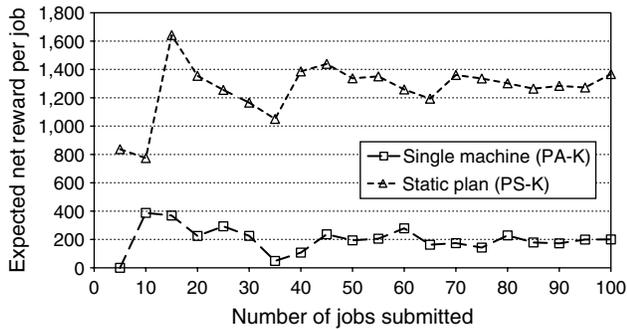


Figure 12 Expected Net Reward per Job for Multiple Jobs

were kept constant at one CPU cycle for any change of machine. As mentioned previously, the optimal scheduling of multiple jobs (PS-K) is itself a difficult problem. For this experiment, a greedy heuristic was used where jobs were scheduled on the grid sequentially based on a descending order of potential revenue. This was compared to PA-K, i.e., where each of the K jobs was assigned to one of K ($K \leq m$) machines. The result is depicted in Figure 11.

At very low numbers of requested jobs, the differences in expected net reward of the static plan relative to the same machine assignment are fairly small. Relatively quickly, however, the static plan is able to take advantage of grid resources to provide increased positive ENR. Because negative ENR jobs would not be accepted, ENR increases with the number of jobs submitted. However, Figure 12 depicts the ENR per job and illustrates the consistent superiority of the static plan over single-machine assignment. The static plan allows the grid manager to accept many jobs that could not be accepted because of negative ENR in the single-machine assignment case.

7. Discussion of Computational Results

The computational results highlight several interesting observations about the sequential grid-computing environment.

Improvements over the single-processor best case. For small jobs with relatively low CPU requirements, the single-machine case provides a high completion probability, and the improvements that result from the sequential grid models (both static and dynamic) are low. Likewise, for large jobs with low completion probability, the sequential grid models provide little improvement because they too cannot complete the job. Between these two extremes, the static and dynamic models provide significant benefits. Experiments indicate that when the single-processor best case provides low probabilities of completion (around 20%), sequential grid models provide the greatest benefits, increasing the completion probability to around 70%.

Increasing resources. As resources are added to a sequential grid, completion probability increases dramatically but reaches saturation quickly. Unlike the parallel grid environment, sequential grid models use only one processor in each period. Thus, additional resources initially increase the options available to the grid manager, but the benefits are muted as resources increase further. In the single-processor best case, the improvement in probability of completion is stepwise and idiosyncratic. An interesting implication is that in sequential grid computing, the grid size can be kept fairly small to obtain most of the benefits without significantly increasing the complexity for the grid manager.

Impact of heterogeneity in grid resources. As uncertainty regarding the idle capacity at each processor in each time period increases, the benefits of the sequential grid over the single-processor case increases. At low levels of demand relative to the capacity of a single machine, the increase is marginal. At medium demand levels, the sequential models allow for job completion irrespective of the resource heterogeneity. At high levels of demand, both the single-processor and sequential grid models can benefit from increased variability; however, the sequential models consistently outperform the single-processor case.

Difference between dynamic and static models. Although the dynamic model always provides superior solutions to the static model by design, extensive experiments indicate that the difference is small. At the same time, the dynamic model is computationally intensive, requires large overhead information and goes through dramatically longer runtimes. However, when a job is behind schedule especially during the early stages, the dynamic policy has greater ability to recover. The static model is computationally efficient and easy to implement, and it also provides good solutions.

Handling of multiple jobs. The sequential grid models prove robust in multiple-job scenarios. The limits of single-machine assignment are quickly reached

while sequential grid models continue to extract additional value with each additional job considered. Furthermore, the average per-job expected net reward is stable and consistently higher than single-machine assignment.

8. Conclusions

In this paper, we defined a grid-computing model (termed sequential grid computing) that has significant advantages in processing large jobs. In sequential grid computing, a computationally intensive job is routed through several processors toward completion but is assigned to one processor during each time period. We also defined two models (static and dynamic) that solve the routing problem associated with sequential grid computing—that is, determining the processor to which the job is assigned for each time period. The static model is computationally efficient and easy to implement, and it also provides good solutions under a variety of conditions, whereas the dynamic model is computationally intensive and requires more overhead information to be transmitted with the job. Our computational experiments provide evidence that the sequential grid computing models have significant benefit when compared to the single-processor best case.

The research can be extended in several ways. First, although we have shown the benefits of sequential grid computing and provided a proof of concept, the software architecture and protocols required to implement the environment are a significant future research issue. Second, we have assumed fixed time buckets in both the static and dynamic models. Determining the optimal size of the scheduling time interval is a difficult research problem and will depend on the modularity of the application, the transfer cost, and the size of the state information that must be transmitted with the job. Third, the procedures and protocols required to implement the models in a peer-to-peer environment (without a centralized grid manager) are also future research issues and are of particular relevance in the Internet environment. Fourth, we have assumed independence of available processing times at each processor, an assumption that may not be realistic if processor failures or nongrid jobs are related. Fifth, the amount of CPU time required by a job may be difficult to determine and can be treated as a random variable in the models. Finally, models that combine parallel and sequential grid computing will enable the benefits of both grid computing paradigms.

References

Allahverdi, A., J. Mittenthal. 1995. Scheduling on a two-machine flowshop subject to random breakdowns with a makespan objective function. *Eur. J. Oper. Res.* **81**(2) 376–387.

Andersen, N. T., V. Dobrić. 1987. The central limit theorem for stochastic processes. *Ann. Probab.* **15**(1) 164–177.

Baker, K. R., G. D. Scudder. 1990. Sequencing with earliness and tardiness penalties: A review. *Oper. Res.* **38**(1) 22–36.

Balut, S. J. 1973. Scheduling to minimize the number of late jobs when set-up and processing times are uncertain. *Management Sci.* **19**(11) 1283–1288.

Bapna, R., S. Das, R. Garfinkel, J. Stallaert. 2006. A continuous auction model for stochastic grid resource pricing and allocation. *Workshop Inform. Tech. Systems (WITS), Milwaukee*, Association of Information Systems, Atlanta, 1–6.

Bapna, R., S. Das, R. Garfinkel, J. Stallaert. 2008. A market design for grid computing. *INFORMS J. Comput.* **20**(1) 100–111.

Berten, V., J. Goossens, E. Jeannot. 2006. On the distribution of sequential jobs in random brokering for heterogeneous computational grids. *IEEE Trans. Parallel Distrib. Systems* **17**(2) 113–124.

Bhargava, H. K., S. Sundaresan. 2004. Computing as utility: Managing availability, commitment, and pricing through contingent bid auctions. *J. Management Inform. Systems* **21**(2) 201–227.

Boeres, C., V. E. F. Rebello. 2004. EasyGrid: Towards a framework for the automatic Grid enabling of legacy MPI applications. *Concurrency Comput.: Practice Experience* **16**(5) 425–432.

Buyya, R., D. Abramson, J. Giddy, H. Stockinger. 2002. Economic models for resource management and scheduling in grid computing. *Concurrency Comput.: Practice Experience* **14**(13–15) 1507–1542.

Cai, X., S. Zhou. 1999. Stochastic scheduling on parallel machines subject to random breakdowns to minimize expected costs for earliness and tardy jobs. *Oper. Res.* **47**(3) 422–437.

Chang, K., A. Dasari, H. Madduri, A. Mendoza, J. Mims. 2004. Design of an enablement process for on demand applications. *IBM Systems J.* **43**(1) 190–203.

Deonier, R. C., S. Tavaré, M. S. Waterman. 2005. *Computational Genome Analysis: An Introduction*. Springer, New York.

Donald, J., M. Martonosi. 2006. An efficient, practical parallelization methodology for multicore architecture simulation. *IEEE Comput. Architecture Lett.* **5**(2) 14–17.

Eilam, T., K. Appleby, J. Breh, G. Breiter, H. Daur, S. A. Fakhouri, G. D. H. Hunt et al. 2004. Using a utility computing framework to develop utility systems. *IBM Systems J.* **43**(1) 97–120.

Ellisman, M., M. Brady, D. Hart, F.-P. Lin, M. Müller, L. Smarr. 2004. The emerging role of biogrids. *Comm. ACM* **47**(11) 52–57.

Hansen, C., C. Johnson. 2003. Graphics applications for grid computing. *IEEE Comput. Graph. Appl.* **23**(2) 20–21.

Henig, M. I. 1990. Risk criteria in a stochastic knapsack problem. *Oper. Res.* **38**(5) 820–825.

Herbon, A., E. Khmelnitsky, I. Ben-Gal. 2005. Using a pseudo-stochastic approach for multiple-parts scheduling on an unreliable machine. *IIE Trans.* **37**(3) 189–199.

Joseph, J., M. Ernest, C. Fellenstein. 2004. Evolution of grid computing architecture and grid adoption models. *IBM Systems J.* **43**(4) 624–645.

Karger, D., R. Motwani, G. D. S. Ramkumar. 1997. On approximating the longest path in a graph. *Algorithmica* **18**(1) 82–98.

Kaya, K., C. Aykanat. 2006. Iterative-improvement-based heuristics for adaptive scheduling of tasks sharing files on heterogeneous master-slave environments. *IEEE Trans. Parallel Distrib. Systems* **17**(8) 883–896.

Kise, H., T. Ibaraki. 1983. On Balut's algorithm and NP-completeness for a chance-constrained scheduling problem. *Management Sci.* **29**(3) 384–388.

Korpela, E., D. Werthimer, D. Anderson, J. Cobb, M. Leboisky. 2001. SETI@home—Massively distributed computing for SETI. *Comput. Sci. Engrg.* **3** 78–83.

Krass, P. 2003. Grid computing. CFO.com (November 17), <http://www.cfo.com/article.cfm/3010943>.

Kumar, S., K. Dutta, V. Mookerjee. 2009. Maximizing business value by optimal assignment of jobs to resources in grid computing. *Eur. J. Oper. Res.* **194**(3) 856–872.

Levitin, G., Y.-S. Dai, H. Ben-Haim. 2006. Reliability and performance of star topology grid service with precedence constraints on subtask execution. *IEEE Trans. Reliab.* **55**(3) 507–515.

- Meliksetian, D. S., J.-P. Prost, A. S. Bahl, I. Boutboul, D. P. Currier, S. Fibra, J.-Y. Girard et al. 2004. Design and implementation of an enterprise grid. *IBM Systems J.* **43**(4) 646–664.
- Murthy, I., S. Sarkar. 1998. Stochastic shortest path problems with piecewise-linear concave utility functions. *Management Sci.* **44**(11, Part 2) S125–S136.
- Nikolova, E., J. A. Kelner, M. Brand, M. Mitzenmacher. 2006. Stochastic shortest paths via quasi-convex maximization. *Proc. 2006 Eur. Sympos. Algorithms (ESA '06), Zurich*. Lecture Notes in Computer Science, Vol. 4168. Springer, Berlin, 552–563.
- Pinedo, M. L., S. M. Ross. 1980. Scheduling jobs subject to nonhomogeneous Poisson shocks. *Management Sci.* **26**(12) 1250–1257.
- Rosenberg, A. L. 2004. On scheduling mesh-structured computations for Internet-based computing. *IEEE Trans. Comput.* **53**(9) 1176–1186.
- Shalf, J., E. W. Bethel. 2003. The grid and future visualization system architectures. *IEEE Comput. Graph. Appl.* **23**(2) 6–9.
- Sonmez, O. O., A. Gursoy. 2007. A novel economic-based scheduling heuristic for computational grids. *Internat. J. High Performance Comput. Appl.* **21**(1) 21–29.
- Stockinger, H. 2006. Grid computing: A critical discussion on business applicability. *IEEE Distrib. Systems Online* **7**(6) 1–8.
- van der Aalst, W. M. P., A. Kumar. 2003. XML-based schema definition for support of interorganizational workflow. *Inform. Systems Res.* **14**(1) 23–46.
- Venugopal, S., R. Buyya, K. Ramamohanarao. 2006. A taxonomy of data grids for distributed data sharing, management, and processing. *ACM Comput. Surv.* **38**(1) Article 3.
- Yu, J., R. Buyya. 2006. A taxonomy of workflow management systems for grid computing. *J. Grid Comput.* **3**(3–4) 171–200.