Self-adaptive applications on the Grid

Gosia Wrzesinska Jason Maassen Henri E. Bal

Dept. of Computer Systems, Vrije Universiteit Amsterdam

{gosia, jason, bal}@cs.vu.nl

Abstract

Grids are inherently heterogeneous and dynamic. One important problem in grid computing is resource selection, that is, finding an appropriate resource set for the application. Another problem is adaptation to the changing characteristics of the grid environment. Existing solutions to these two problems require that a performance model for an application is known. However, constructing such models is a complex task. In this paper, we investigate an approach that does not require performance models. We start an application on any set of resources. During the application run, we periodically collect the statistics about the application run and deduce application requirements from these statistics. Then, we adjust the resource set to better fit the application needs. This approach allows us to avoid performance bottlenecks, such as overloaded WAN links or very slow processors, and therefore can yield significant performance improvements. We evaluate our approach in a number of scenarios typical for the Grid.

Keywords self-adaptivity, grid computing

1. Introduction

In recent years, grid computing has become a real alternative to traditional parallel computing. A grid provides much computational power, and thus offers the possibility to solve very large problems, especially if applications can run on multiple sites at the same time [7, 15, 20]. However, the complexity of Grid environments also is many times larger than that of traditional parallel machines like clusters and supercomputers. One important problem is *resource selection* - selecting a set of compute nodes such that the application achieves good performance. Even in traditional, homogeneous parallel environments, finding the optimal number of nodes is a hard problem and is often solved in a trial-and-error fashion. In a grid environment this problem is even more difficult, because of the heterogeneity of resources: the compute nodes have various speeds and the quality of network connections between them varies from low-latency and high-bandwidth local-area networks (LANs) to high-latency and possibly low-bandwidth wide-area networks (WANs). Another important problem is that the performance and availability of grid resources varies over time: the network links or compute nodes may become overloaded, or the compute nodes may become unavailable because of crashes or because they have been claimed by a higher priority application. Also, new, better resources may become available. To maintain a reasonable performance level, the application therefore needs to *adapt* to the changing conditions.

The adaptation problem can be reduced to the resource selection problem: the resource selection phase can be repeated during application execution, either at regular intervals, or when a performance problem is detected, or when new resources become available. This approach has been adopted by a number of systems [5, 14, 18]. For resource selection, the application runtime is estimated for some resource sets and the set that yields the shortest runtime is selected for execution. Predicting the application runtime on a given set of resources, however, requires knowledge about the application. Typically, an analytical *performance model* is used, but constructing such a model is inherently difficult and requires an expertise which application programmers may not have.

In this paper, we introduce and evaluate an alternative approach to application adaptation and resource selection which does not need a performance model. We start an application on *any* set of resources. During the application run, we periodically collect information about the communication times and idle times of the processors. We use these statistics to automatically estimate the resource requirements of the application. Next, we *adjust* the resource set the application is running on by adding or removing compute nodes or even entire clusters. Our adaptation strategy uses the work by Eager et al. [10] to determine the efficiency and tries to keep the efficiency of the application between a lower and upper threshold derived from their theory. Processors are added or deleted to stay between the thresholds, thus adapting automatically to the changing environment.

A major advantage of our approach is that it improves application performance in many different situations that are typical for grid computing. It handles all of the following cases:

- automatically adapting the number of processors to the degree of parallelism in the application, even when this degree changes dynamically during the computation
- migrating (part of) a computation away from overloaded resources
- removing resources with poor communication links that slow down the computation
- adding new resources to replace resources that have crashed

Our work assumes the application is malleable and can run (efficiently) on multiple sites of a grid (i.e., using co-allocation [15]). It should not use static load balancing or be very sensitive to wide-area latencies. We have applied our ideas to divide-and-conquer applications, which satisfy these requirements. Divide-and-conquer has been shown to be an attractive paradigm for programming grid applications [4, 20]. We believe that our approach can be extended to other classes of applications with the given assumptions.

We implemented our strategy in Satin, which is a Java-centric framework for writing grid-enabled divide-and-conquer applications [20]. We evaluate the performance of our approach on the

WXYZ '05 date, City.

DAS-2 wide-area system and we will show that our approach yields major performance improvements (roughly 10-60 %) in the above scenarios.

The rest of this paper is structured as follows. In Section 2, we explain what assumptions we are making about the applications and grid resources. In Section 3, we present our resource selection and adaptation strategy. In Section 4, we describe its implementation in the Satin framework. In Section 5, we evaluate our approach in a number of grid scenarios. In Section 6, we compare our approach with the related work. Finally, in Section 7, we conclude and describe future work.

2. Background and assumptions

In this section, we describe our assumptions about the applications and their resources. We assume the following resource model. The applications are running on *multiple* sites at the same time, where sites are clusters or supercomputers. We also assume that the processors of the sites are accessible using a grid scheduling system, such as Koala [15], Zorilla [9] or GRMS [3]. Processors belonging to one site are connected by a fast LAN with a low latency and high bandwidth. The different sites are connected by a WAN. Communication between sites suffers from high latencies. We assume that the links connecting the sites with the Internet backbone might become bottlenecks causing the inter-site communication to suffer from low bandwidths.

We studied the adaptation problem in the context of divide-and-conquer applications. However, we believe that our methodology can be used for other types of applications as well. In this section we summarize the assumptions about applications that are important to our approach.

The first assumption we make is that the application is *malleable*, i.e., it is able to handle processors joining and leaving the on-going computation. In [23], we showed how divide-and-conquer applications can be made fault tolerant and malleable. Processors can be added or removed at any point in the computation with little overhead. The second assumption is that the application can efficiently run on processors with different speeds. This can be achieved by using a dynamic load balancing strategy, such as work stealing used by divide-and-conquer applications [19]. Also, master-worker applications typically use dynamic load-balancing strategies (e.g., MW – a framework for writing grid-enabled master-worker applications [12]). We find it a reasonable assumption for a grid application, since applications for which the slowest processor becomes a bottleneck will not be able to efficiently utilize grid resources. Finally, the application should be insensitive to wide-area latencies, so it can run efficiently on a wide-area grid [16, 17].

3. Self-adaptation

In this section we will explain how we use application malleability to find a suitable set of resources for a given application and to adapt to changing conditions in the grid environment. In order to monitor the application performance and guide the adaptation, we added an extra process to the computation which we call *adaptation coordinator*. The adaptation coordinator periodically collects performance statistics from the application processors. We introduce a new application performance metric: *weighted average efficiency* which describes the application performance on a heterogeneous set of resources. The coordinator uses statistics from application processors to compute the weighted average efficiency. If the efficiency falls above or below certain thresholds, the coordinator decides on adding or removing processors. A heuristic formula is used to decide which processors have to be removed. During this process the coordinator *learns* the application requirements by remembering the characteristics of the removed processors. These requirements are then used to guide the adding of new processors.

3.1 Weighted average efficiency

In traditional parallel computing, a standard metric describing the performance of a parallel application is *efficiency*. Efficiency is defined as the average utilization of the processors, that is, the fraction of time the processors spend doing useful work rather than being idle or communicating with other processors [10].

$$efficiency = \frac{1}{n} * \sum_{i=0}^{n} (1 - overhead_i)$$

where n is the number of processors and $overhead_i$ is the fraction of time the i^{th} processor spends being idle or communicating. Efficiency indicates the benefit of using multiple processors.

Typically, the efficiency drops as new processors are added to the computation. Therefore, achieving a high speedup (and thus a low execution time) and achieving a high system utilization are conflicting goals [10]. The optimal number of processors is the number for which the ratio of efficiency to execution time is maximized. Adding processors beyond this number yields little benefit. This number is typically hard to find, but in [10] it was theoretically proven that if the optimal number of processors is used, the efficiency is at least 50%. Therefore, adding processors when efficiency is smaller or equal to 50% will only decrease the system utilization without significant performance gains.

For heterogeneous environments with different processor speeds, we extended the notion of efficiency and introduced *weighted average efficiency*.

$$wa\_efficiency = \frac{1}{n} * \sum_{i=0}^{n} speed_i * (1 - overhead_i)$$

The useful work done by a processor $(1 - overhead_i)$ is weighted by multiplying it by the speed of this processor relative to the fastest processor. The fastest processor has $speed = 1$, for others holds: $0 < speed \leq 1$. Therefore, slower processors are modeled as fast ones that spend a large fraction of the time being idle. Weighted average efficiency reflects the fact that adding slow processors yields less benefit than adding fast processors.

In the heterogeneous world, it is hardly beneficial to add processors if the efficiency is lower than 50% unless the added processor is faster than some of the currently used processors. Adding faster processors might be beneficial regardless of the efficiency.

3.2 Application monitoring

Each processor measures the time it spends communicating or being idle. The computation is divided into *monitoring periods*. After each monitoring period, the processors compute their overhead over this period as the percentage of the time they spent being idle or communicating in this period. Apart from total overhead, each processor also computes the overhead of inter-cluster and intra-cluster communication.

To calculate weighted average efficiency, we need to know the relative speeds of the processors, which depend on the application and the problem size used. Since it is impractical to run the whole application on each processor separately, we use application-specific benchmarks. Currently we use the same application with a small problem size as a benchmark and we require the application programmer to specify this problem size. This approach requires extra effort from the programmer to find the right problem size and possibly to produce input files for this problem size, which may be hard. In the future, we are planning to generate benchmarks au-

tomatically by choosing a random subset of the task graph of the original application.

Benchmarks have to be re-run periodically because the speed of a processor might change if it becomes overloaded by another application (for time-shared machines). There is a trade-off between the accuracy of speed measurements and the overhead it incurs. The longer the benchmark, the greater the accuracy of the measurement. The more often it is run, the faster changes in processor speed are detected. In our current implementation, the application programmer specifies the length of the benchmark (by specifying its problem size) and the maximal overhead it is allowed to cause. Processors run the benchmark at such frequency so as not to exceed the specified overhead. In the future, we plan to combine benchmarking with monitoring the load of the processor which would allow us to avoid running the benchmark if no change in processor load is detected. This optimization will further reduce the benchmarking overhead.

Note that the benchmarking overhead could be avoided completely for more regular applications, for example, for master-worker applications with tasks of equal or similar size. The processor speed could then be measured by counting the tasks processed by this processor within one monitoring period. Unfortunately, divide-and-conquer applications typically exhibit a very irregular structure. The sizes of tasks can vary by many orders of magnitude.

At the end of each monitoring period, the processors send the overhead statistics and processor speeds to the coordinator. Periodically, the coordinator computes the weighted average efficiency and other statistics, such as average inter-cluster overhead or overheads in each cluster. The clocks of the processors are not synchronized with each other or with the clock of the coordinator. Each processor decides separately when it is time to send data. Occasionally, the coordinator may miss data at the end of a monitoring period, so it has to use data from the previous monitoring period for these processors. This causes small inaccuracies in the calculations of the coordinator, but does not influence the performance of adaptation.

3.3 Adaptation strategy

The adaptation coordinator tries to keep the weighted average efficiency between E_{min} and E_{max}. When it exceeds E_{max}, the coordinator requests new processors from the scheduler. The number of requested processors depends on the current efficiency: the higher the efficiency, the more processors are requested. The coordinator starts removing processors when the weighted average efficiency drops below E_{min}. The number of nodes that are removed again depends on the weighted average efficiency. The lower the efficiency, the more nodes are removed. The thresholds we use are $E_{max} = 50\%$, because we know that adding processors when efficiency is lower does not make sense, and $E_{min} = 30\%$. Efficiency of 30% or lower might indicate performance problems such as low bandwidth or overloaded processors. In that case, removing bad processors will be beneficial for the application. Such low efficiency might also indicate that we simply have too many processors. In that case, removing some processors may not be beneficial but it will not harm the application. The coordinator always tries to remove the "worst" processors. The "badness" of a processor is determined by the following formula:

$$proc\_badness_i = \alpha * \frac{1}{speed_i} + \beta * ic\_overhead_i + \gamma * inWorstCluster(i)$$

The processor is considered bad if it has low speed ($\frac{1}{speed}$ is big) and high inter-cluster overhead ($ic\_overhead$). High inter-cluster overhead indicates that the bandwidth to this processor's cluster is insufficient. Removing processors located in a single cluster is desirable since it decreases the amount of wide-area communication. Therefore, processors belonging to the "worst" cluster are preferred. Function $inWorstCluster(i)$ returns 1 for processors belonging to the "worst" cluster and 0 otherwise. The "badness" of clusters is computed similarly to the "badness" of processors:

$$cluster\_badness_i = \alpha * \frac{1}{speed_i} + \beta * ic\_overhead_i$$

The speed of a cluster is the sum of processor speeds normalized to the speed of the fastest cluster. The $ic\_overhead$ of a cluster is an average of processor inter-cluster overheads. The α, β and γ coefficients determine the relative importance of the terms. Those coefficients are established empirically. Currently we are using the following values: $\alpha = 1$, $\beta = 100$ and $\gamma = 10$, based on the observation that $ic\_overhead > 0.2$ indicates bandwidth problems and processors with $speed < 0.05$ do not contribute to the computation.

Additionally, when one of the clusters has an exceptionally high inter-cluster overhead (larger than 0.25), we conclude that the bandwidth on the link between this cluster and the Internet backbone is insufficient for the application. In that case, we simply remove the whole cluster instead of computing node badness and removing the worst nodes. After deciding which nodes are removed, the coordinator sends a message to these nodes and the nodes leave the computation. Figure 1 shows a schematic view of the adaptation strategy. Dashed lines indicate a part that is not supported yet, as will be explained below.

This simple adaptation strategy allows us to improve application performance in several situations typical for the Grid:

- If an application is started on fewer processors than its degree of parallelism allows, it will automatically expand to more processors (as soon as there are extra resources available). Conversely,

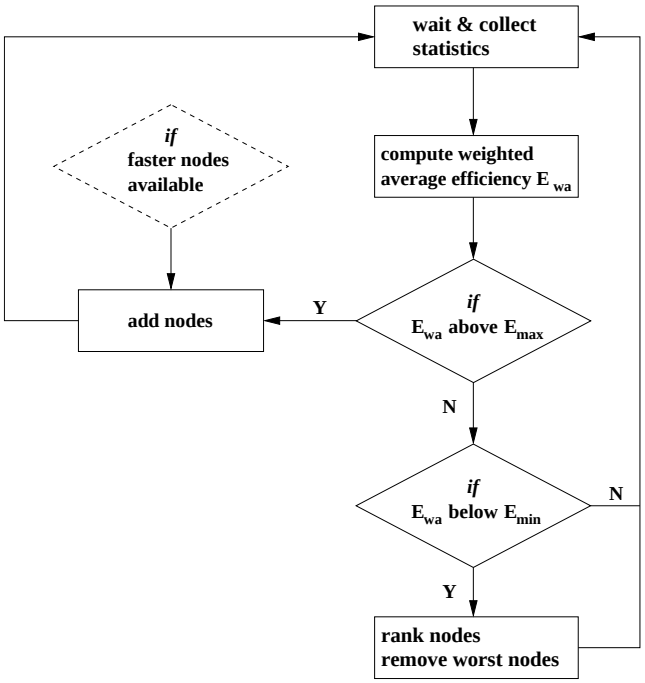

Figure 1. Adaptation strategy

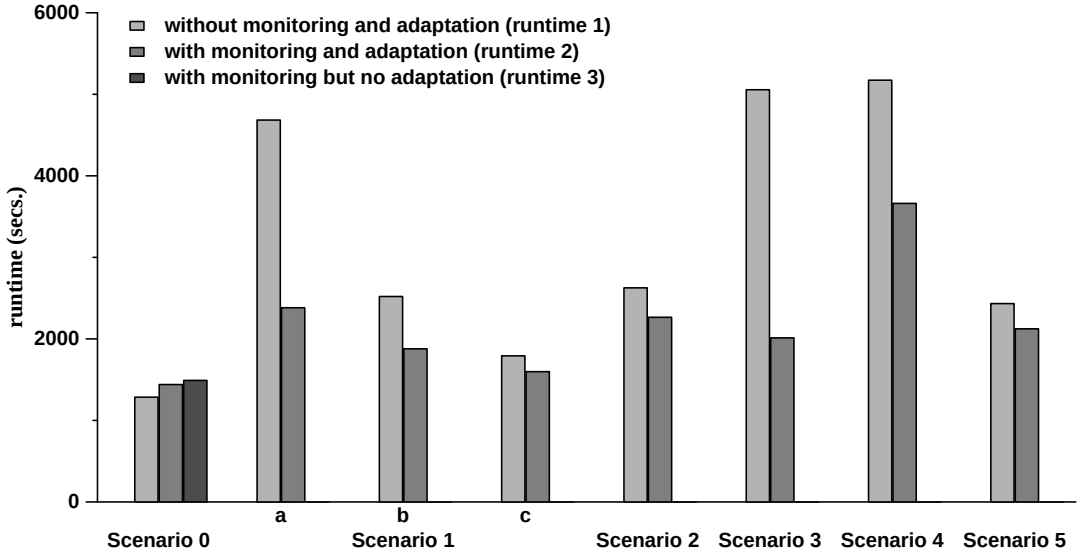

Figure 2. The runtimes of the Barnes-Hut application, scenarios 0-5

if an application is started on more processors than it can efficiently use, a part of the processors will be released.

- If an application is running on an appropriate set of resources but after a while some of the resources (processors and/or network links) become overloaded and slow down the computation, the overloaded resources will be removed. After removing the overloaded resources, the weighted average efficiency will increase to above the E_{max} threshold and the adaptation coordinator will try to add new resources. Therefore, the application will be *migrated* from overloaded resources.
- If some of the original resources chosen by the user are inappropriate for the application, for example the bandwidth to one of the clusters is too small, the inappropriate resources will be removed. If necessary, the adaptation component will try to add other resources.
- If during the computation a substantial part of the processors will crash, the adaptation component will try to add new resources to replace the crashed processors.
- If the application degree of parallelism is changing during the computation, the number of nodes the application is running on will be automatically adjusted.

Further improvements are possible, but require extra functionality from the grid scheduler and/or integration with monitoring services such as NWS [22]. For example, adding nodes to a computation can be improved. Currently, we add any nodes the scheduler gives us. However, it would be more efficient to ask for the *fastest* processors among the available ones. This could be done, for example, by passing a benchmark to the grid scheduler, so that it can measure processor speeds in an application specific way. Typically, it would be enough to measure the speed of one processor per site, since clusters and supercomputers are usually homogeneous. An alternative approach would be ranking the processors based on parameters such as clock speed and cache size. This approach is sometimes used for resource selection for sequential applications [14]. However, it is less accurate than using an application specific benchmark.

Also, during application execution, we can learn some application requirements and pass them to the scheduler. One example is minimal bandwidth required by the application. The lower bound on minimal required bandwidth is tightened each time a cluster with high inter-cluster overhead is removed. The bandwidth between each pair of clusters is estimated during the computation by measuring data transfer times, and the bandwidth to the removed cluster is set as a minimum. Alternatively, information from a grid monitoring system can be used. Such bounds can be passed to the scheduler to avoid adding inappropriate resources. It is especially important when migrating from resources that cause performance problems: we have to be careful not to add the resources we have just removed. Currently we use *blacklisting* - we simply do not allow adding resources we removed before. This means, however, that we cannot use these resources even if the cause of the performance problem disappears, e.g. the bandwidth of a link might improve if the background traffic diminishes.

We are currently not able to perform opportunistic migration - migrating to better resources when they are discovered. If an application runs with efficiency between E_{min} and E_{max}, the adaptation component will not undertake any action, even if better resources become available. Enabling opportunistic migration requires, again, the ability to specify to the scheduler what "better" resources are (faster, with a certain minimal bandwidth) and receiving notifications when such resources become available.

Existing grid schedulers such as GRAM from the Globus Toolkit [11] do not support such functionality. The developers of the KOALA metascheduler [15] have recently started a project whose goal is to provide support for adaptive applications. We are currently discussing with them the possibility of providing the functionalities required by us, aiming to extend our adaptivity strategy to support opportunistic migration and to improve the initial resource selection.

4. Implementation

We incorporated our adaptation mechanism into Satin – a Java framework for creating grid-enabled divide-and-conquer applications. With Satin, the programmer annotates the sequential code with divide-and-conquer primitives and compiles the annotated code with a special Satin compiler that generates the necessary communication and load balancing code. Satin uses a very efficient, grid-aware load balancing algorithm – Cluster-aware Ran-

dom Work Stealing (CRS) [19], which hides wide-area latencies by overlapping local and remote stealing. Satin also provides transparent fault tolerance and malleability [23]. With Satin, removing and adding processors from/to an ongoing computation incurs little overhead.

We instrumented the Satin runtime system to collect runtime statistics and send them to the adaptation coordinator. The coordinator is implemented as a separate process. Both coordinator and Satin are implemented entirely in Java on top of the Ibis communication library [21]. The core of Ibis is also implemented in Java. The resulting system therefore is highly portable (due to Java's "write once, run anywhere" property) allowing the software to run unmodified on a heterogeneous grid.

Ibis also provides the Ibis Registry. The Registry provides, among others, a membership service to the processors taking part in the computation. The adaptation coordinator uses the Registry to discover the application processes, and the application processes use this service to discover each other. The Registry also offers fault detection (additional to the fault detection provided by the communication channels). Finally, the Registry provides the possibility to send signals to application processes. The coordinator uses this functionality to notify the processors that they need to leave the computation. Currently the Registry is implemented as a centralized server.

For requesting new nodes, the Zorilla [9] system is used – a peer-to-peer supercomputing middleware which allows straightforward allocation of processors in multiple clusters and/or supercomputers. Zorilla provides *locality-aware scheduling*, which tries to allocate processors that are located close to each other in terms of communication latency. In the future, Zorilla will also support bandwidth-aware scheduling, which tries to maximize the total bandwidth in the system. Zorilla can be easily replaced with another grid scheduler. In the future, we are planning to integrate our adaptation component with GAT [3] which is becoming a standard in the grid community and KOALA [15] a scheduler that provides co-allocation on top of standard grid middleware, such as the Globus Toolkit [11].

5. Performance evaluation

In this section, we will evaluate our approach. We will demonstrate the performance of our mechanism in a few scenarios. The first scenario is an "ideal" situation: the application runs on a reasonable set of nodes (i.e., such that the efficiency is around 50%) and no problems such as overloaded networks and processors, crashing processors etc. occur. This scenario allows us to measure the overhead of the adaptation support. The remaining scenarios are typical for grid environments and demonstrate that with our adaptation support the application can avoid serious performance bottlenecks such as overloaded processors or network links. For each scenario, we compare the performance of an application with adaptation support to a non-adaptive version. In the non-adaptive version, the coordinator does not collect statistics and no benchmarking (for measuring processor speeds) is performed. In the "ideal" scenario, we additionally measure the performance of an application with collecting statistics and benchmarking turned on but without doing adaptation, that is, without allowing it to change the number of nodes. This allows us to measure the overhead of benchmarking and collecting statistics. In all experiments we used a monitoring period of 3 minutes for the adaptive versions of the applications. All the experiments were carried out on the DAS-2 wide-area system [8], which consists of five clusters located at five Dutch universities. One of the clusters consists of 72 nodes, the others of 32 nodes. Each node contains two 1 GHz Pentium processors. Within a cluster, the nodes are connected by Fast Ethernet. The clusters are connected by the Dutch university Internet backbone. In our

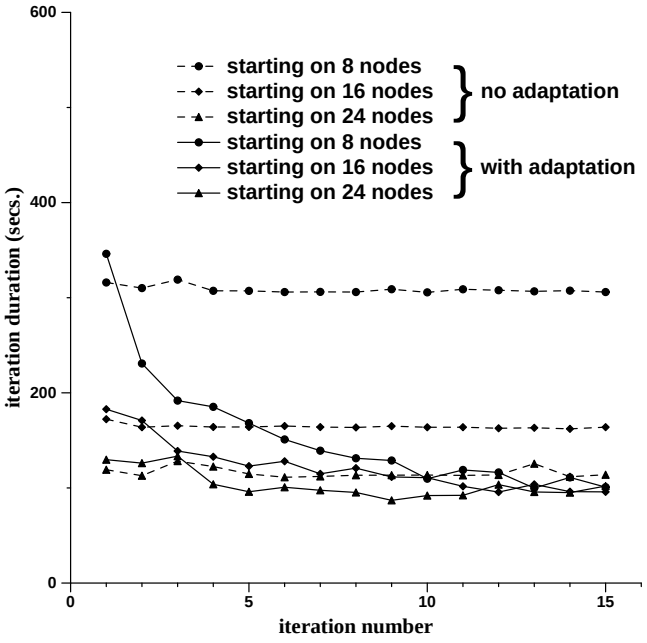

Figure 3. Barnes-Hut iteration durations with/without adaptation, too few processors

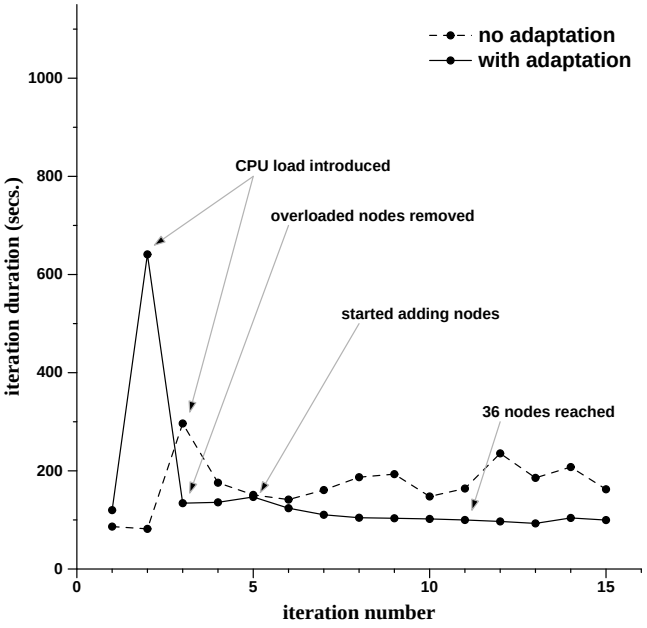

Figure 4. Barnes-Hut iteration durations with/without adaptation, overloaded CPUs

experiments, we used the Barnes-Hut N-body simulation. Barnes-Hut simulates the evolution of a large set of bodies under influence of (gravitational or electrostatic) forces. The evolution of N bodies is simulated in iterations of discrete time steps.

5.1 Scenario 0: adaptivity overhead

In this scenario, the application is started on 36 nodes. The nodes are equally divided over 3 clusters (12 nodes in each cluster). On

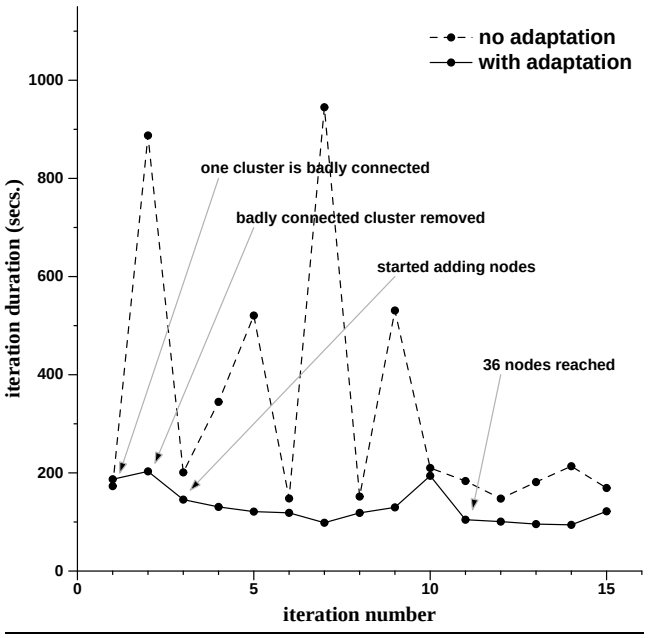

Figure 5. Barnes-Hut iteration durations with/without adaptation, overloaded network link

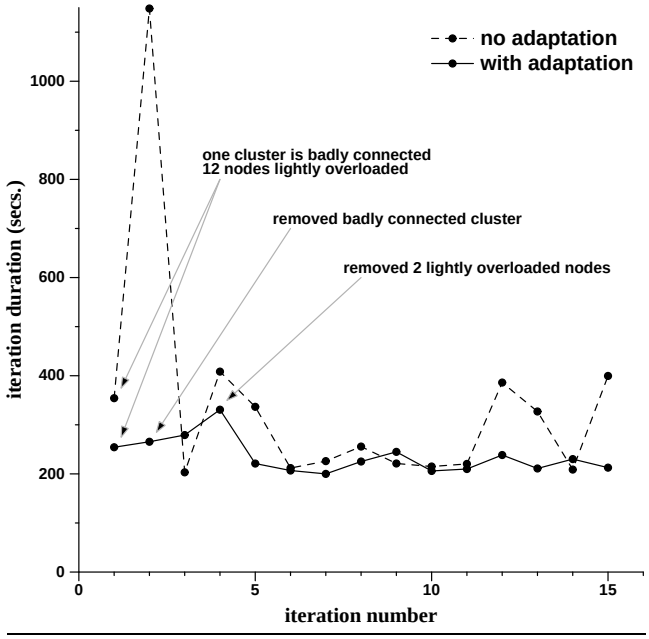

Figure 6. Barnes-Hut iteration durations with/without adaptation, overloaded CPUs and an overloaded network link

this number of nodes, the application runs with 50% efficiency, so we consider it a reasonable number of nodes. As mentioned above, in this scenario we measured three runtimes: the runtime of the application without adaptation support (runtime 1), the runtime with adaptation support (runtime 2) and the runtime with monitoring (i.e., collection of statistics and benchmarking) turned on but without allowing it to change the number of nodes (runtime 3). Those runtimes are shown in Figure 2, first group of bars. The comparison between runtime 3 and 1 shows the overhead of adaptation support. In this experiment it is around 15%. Almost all overhead comes from benchmarking. The benchmark is run 1-2 times per monitoring period. This overhead can be made smaller by increasing the length of the monitoring period and decreasing the benchmarking frequency. The monitoring period we used (3 minutes) is relatively short, because the runtime of the application was also relatively short (30–60 minutes). Using longer running applications would not allow us to finish the experimentation in a reasonable time. However, real-world grid applications typically need hours, days or even weeks to complete. For such applications, a much longer monitoring period can be used and the adaptation overhead can be kept much lower. For example, with the Barnes-Hut application, if the monitoring period is extended to 10 minutes, the overhead drops to 6%. Note that combining benchmarking with monitoring processor load (as described in Section 3.2) would reduce the benchmarking overhead to almost zero: since the processor load is not changing, the benchmarks would only need to be run at the beginning of the computation.

5.2 Scenario 1: expanding to more nodes

In this scenario, the application is started on fewer nodes than the application can efficiently use. This may happen because the user does not know the right number of nodes or because insufficient nodes were available at the moment the application was started. We tried 3 initial numbers of nodes: 8 (Scenario 1a), 16 (Scenario 1b) and 24 (Scenario 1c). The nodes were located in 1 or 2 clusters. In each of the three sub-scenarios, the application gradually expanded to 36-40 nodes located in 4 clusters. This allowed to reduce the application runtimes by 50% (Scenario 1a), 35% (Scenario 1b) and 12% (Scenario 1c) with respect to the non-adaptive version. Those runtimes are shown in Figure 2. Since Barnes-Hut is an iterative application, we also measured the time of each iteration, as shown in Figure 3. Adaptation reduces the iteration time by a factor of 3 (Scenario 1a), 1.7 (Scenario 1b) and 1.2 (Scenario 1c) which allows us to conclude that the gains in the total runtime would be even bigger if the application were run longer than for 15 iterations.

5.3 Scenario 2: overloaded processors

In this scenario, we started the application on 36 nodes in 3 clusters. After 200 seconds, we introduced a heavy, artificial load on the processors in one of the clusters. Such a situation may happen when an application with a higher priority is started on some of the resources. Figure 4 shows the iteration durations of both the adaptive and non-adaptive versions. After introducing the load, the iteration duration increased by a factor of 2 to 3. Also, the iteration times became very variable. The adaptive version reacted by removing the overloaded nodes. After removing these nodes, the weighted average efficiency rose to around 35% which triggered adding new nodes and the application expanded back to 38 nodes. So, the overloaded nodes were replaced by better nodes, which brought the iteration duration back to the initial values. This reduced the total runtime by 14%. The runtimes are shown in Figure 2.

5.4 Scenario 3: overloaded network link

In this scenario, we ran the application on 36 nodes in 3 clusters. We simulated that the uplink to one of the clusters was overloaded and the bandwidth on this uplink was reduced to approximately 100 KB/s. To simulate low bandwidth we use the traffic-shaping techniques described in [6]. The iteration durations in this experiment are shown in Figure 5. The iteration durations of the non-adaptive version exhibit enormous variation: from 170 to 890 seconds. The adaptive version removed the badly connected cluster after the first monitoring period. As a result, the weighted average efficiency rose to around 35% and new nodes were gradually added until their number reached 38. This brought the iteration

times down to around 100 seconds. The total runtime was reduced by 60% (Figure 2).

5.5 Scenario 4: overloaded processors and an overloaded network link

In this scenario, we ran the application on 36 nodes in 3 clusters. Again, we simulated an overloaded uplink to one of the clusters. Additionally, we simulated processors with heterogeneous speeds by inserting a relatively light artificial load on the processors in one of the remaining clusters. The iteration durations are shown in Figure 6. Again, the non-adaptive version exhibits a great variation in iteration durations: from 200 to 1150 seconds. The adaptive version removes the badly connected cluster after the first monitoring period which brings the iteration duration down to 210 seconds on average. After removing one of the clusters, since some of the processors are slower (approximately 5 times), the weighted average efficiency raises only to around 40%. Since this value lies between E_{min} and E_{max}, no nodes are added or removed. This example illustrates what the advantages of *opportunistic migration* would be. There were faster nodes available in the system. If these nodes were added to the application (which could trigger removing the slower nodes) the iteration duration could be reduced even further. Still, the adaptation reduced the total runtime by 30% (Figure 2).

5.6 Scenario 5: crashing nodes

In the last scenario, we also run the application on 36 nodes in 3 clusters. After 500 seconds, 2 out of 3 clusters crash. The iteration durations are shown in Figure 7. After the crash, the iteration duration raised from 100 to 200 seconds. The weighted efficiency rose to around 30% which triggered adding new nodes in the adaptive version. The number of nodes gradually went back to 35 which brought the iteration duration back to around 100 seconds. The total runtime was reduced by 13% (Figure 2).

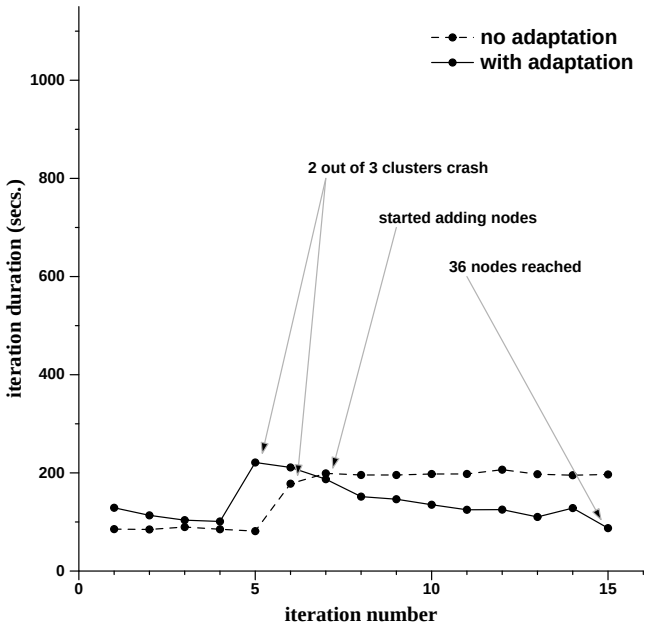

Figure 7. Barnes-Hut iteration durations with/without adaptation, crashing CPUs

6. Related work

A number of Grid projects address the question of resource selection and adaptation. In GrADS [18] and ASSIST [1], resource selection and adaptation requires a performance model that allows predicting application runtimes. In the resource selection phase, a number of possible resource sets is examined and the set of resources with the shortest predicted runtime is selected. If performance degradation is detected during the computation, the resource selection phase is repeated. GrADS uses the ratio of the predicted execution times (of certain application phases) to the real execution times as an indicator of application performance. ASSIST uses the number of iterations per time unit (for iterative applications) or the number of tasks per time unit (for regular master-worker applications) as a performance indicator. The main difference between these approaches and our approach is the use of performance models. The main advantage is that once the performance model is known, the system is able to take more accurate migration decisions than with our approach. However, even if the performance model is known, the problem of finding an *optimal* resource set (i.e. the resource set with the minimal execution time) is NP-complete. Currently, both GrADS and ASSIST examine only a subset of all possible resource sets and therefore there is no guarantee that the resulting resource set will be optimal. As the number of available grid resources increases, the accuracy of this approach diminishes, as the subset of possible resource sets that can be examined in a reasonable time becomes smaller. Another disadvantage of these systems is that the performance degradation detection is suitable only for iterative or regular applications.

Cactus [2] and GridWay [14] do not use performance models. However, these frameworks are only suitable for sequential (GridWay) or single-site applications (Cactus). In that case, the resource selection problem boils down to selecting the fastest machine or cluster. Processor clock speed, average load and a number of processors in a cluster (Cactus) are used to rank resources and the resource with the highest rank is selected. The application is migrated if performance degradation is detected or better resources are discovered. Both Cactus and GridWay use the number of iterations per time unit as the performance indicator. The main limitation of this methodology is that it is suitable only for sequential or single-site applications. Moreover, resource selection based on clock speed is not always accurate. Finally, performance degradation detection is suitable only for iterative applications and cannot be used for irregular computations such as search and optimization problems.

The resource selection problem was also studied by the AppLeS project [5]. In the context of this project, a number of applications were studied and performance models for these applications were created. Based on such a model a scheduling agent is built that uses the performance model to select the best resource set and the best application schedule on this set. AppLeS scheduling agents are written on a case-by-case basis and cannot be reused for another application. Two reusable *templates* were also developed for specific classes of applications, namely master-worker (AMWAT template) and parameter sweep (APST template) applications. Migration is not supported by the AppLeS software.

In [13], the problem of scheduling master-worker applications is studied. The authors assume homogeneous processors (i.e., with the same speed) and do not take communication costs into account. Therefore, the problem is reduced to finding the right number of workers. The approach here is similar to ours in that no performance model is used. Instead, the system tries to deduce the application requirements at runtime and adjusts the number of workers to approach the ideal number.

7. Conclusions and future work

In this paper, we investigated the problem of resource selection and adaptation in grid environments. Existing approaches to these problems typically assume the existence of a performance model that

allows predicting application runtimes on various sets of resources. However, creating performance models is inherently difficult and requires knowledge about the application. We propose an approach that does not require in-depth knowledge about the application. We start the application on an arbitrary set of resources and monitor its performance. The performance monitoring allows us to learn certain application requirements such as the number of processors needed by the application or the application's bandwidth requirements. We use this knowledge to gradually refine the resource set by removing inadequate nodes or adding new nodes if necessary. This approach does not result in the optimal resource set, but in a *reasonable* resource set, i.e. a set free from various performance bottlenecks such as slow network connections or overloaded processors. Our approach also allows the application to adapt to the changing grid conditions.

The adaptation decisions are based on the weighted average efficiency – an extension of the concept of parallel efficiency defined for traditional, homogeneous parallel machines. If the weighted average efficiency drops below a certain level, the adaptation coordinator starts removing "worst" nodes. The "badness" of the nodes is defined by a heuristic formula. If the weighted average efficiency raises above a certain level, new nodes are added. Our simple adaptation strategy allows us to handle multiple scenarios typical for grid environments: expand to more nodes or shrink to fewer nodes if the application was started on an inappropriate number of processors, remove inadequate nodes and replace them with better ones, replace crashed processors, etc. The application adapts *fully automatically* to changing conditions. We implemented our approach in the Satin divide-and-conquer framework and evaluated it on the DAS-2 distributed supercomputer and demonstrate that our approach can yield significant performance improvements (up to 60% in our experiments).

Future work will involve extending our adaptation strategy to support opportunistic migration. This, however, requires grid schedulers with more sophisticated functionality than currently exists. Further research is also needed to decrease the benchmarking overhead. For example, the information about CPU load could be used to decrease the benchmarking frequency. Another line of research that we wish to investigate is using *feedback control* to refine the adaptation strategy during the application run. For example, the node "badness" formula could be refined at runtime based on the effectiveness of the previous adaptation decisions. Finally, the centralized implementation of the adaptation coordinator might become a bottleneck for applications which are running on very large numbers of nodes (hundreds or thousands). This problem can be solved by implementing a *hierarchy* of coordinators: one sub-coordinator per cluster which collects and processes statistics from its cluster and one main coordinator which collects the information from the sub-coordinators.

Acknowledgments

This work was carried out in the context of Virtual Laboratory for e-Science project (ww.vl-e.nl). This project is supported by a BSIK grant from the Dutch Ministry of Education, Culture and Science (OC&W) and is part of the ICT innovation program of the Ministry of Economic Affairs (EZ).

References

[1] M. Aldinucci, F. Andre, J. Buisson, S. Campa, M. Coppola, M. Danelutto, and C. Zoccolo. Parallel program/component adaptivity management. In *ParCo 2005*, Sept. 2005.

[2] G. Allen, D. Angulo, I. Foster, G. Lanfermann, C. Liu, T. Radke, E. Seidel, and J. Shalf. The cactus worm: Experiments with resource discovery and allocation in a grid environment. *Int'l Journal of High Performance Computing Applications*, 15(4):345–358, 2001.

[3] G. Allen, K. Davis, K. N. Dolkas, N. D. Doulamis, T. Goodale, T. Kielmann, A. Merzky, J. Nabrzyski, J. Pukacki, T. Radke, M. Russell, E. Seidel, J. Shalf, and I. Taylor. Enabling applications on the grid – a gridlab overview. *Int'l Journal of High-Performance Computing Applications*, 17(4):449–466, Aug. 2003.

[4] J. E. Baldeschwieler, R. D. Blumofe, and E. A. Brewer. ATLAS: An Infrastructure for Global Computing. In *7th ACM SIGOPS European Workshop on System Support for Worldwide Applications*, pages 165–172, Sept. 1996.

[5] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov. Adaptive Computing on the Grid Using AppLeS. *IEEE Trans. on Parallel and Distributed Systems*, 14(4):369–382, Apr. 2003.

[6] D.-M. Chiu, M. Kadansky, J. Provino, and J. Wesley. Experiences in programming a traffic shaper. In *5th IEEE Symp. on Computers and Communications*, pages 470–476, 2000.

[7] W. Chrabakh and R. Wolski. GridSAT: A Chaff-based Distributed SAT Solver for the Grid. In *2003 ACM/IEEE conference on Supercomputing*, page 37, 2003.

[8] The Distributed ASCI Supercomputer (DAS). http://www.cs.vu.nl/das2/.

[9] N. Drost, R. V. van Nieuwport, and H. E. Bal. Simple locality-aware co-allocation in peer-to-peer supercomputing. In *6th Int'l Workshop on Global Peer-2-Peer Computing*, May 2005.

[10] D. L. Eager, J. Zahorjan, and E. D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, Mar. 1989.

[11] I. Foster. Globus toolkit version 4: Software for service-oriented systems. In *IFIP International Conference on Network and Parallel Computing*, pages 2–13. Springer-Verlag LNCS 3779, 2005.

[12] J.-P. Goux, S. Kulkarni, M. Yoder, and J. Linderoth. An Enabling Framework for Master-Worker Applications on the Computational Grid. In *9th IEEE Int'l Symp. on High Performance Distributed Computing*, pages 43–50, Aug. 2000.

[13] E. Heymann, M. A. Senar, E. Luque, and M. Livny. Adaptive scheduling for master-worker applications on the computational grid. In *1st IEEE/ACM International Workshop on Grid Computing*, pages 214–227. Springer Verlag LNCS 1971, 2000.

[14] E. Huedo, R. S. Montero, and I. M. Llorente. A framework for adaptive execution in grids. *Software – Practice & Experience*, 34(7):631–651, 2004.

[15] H. H. Mohamed and D. H. Epema. Experiences with the KOALA Co-Allocating Scheduler in Multiclusters. In *5th IEEE/ACM Int'l Symp. on Cluster Computing and the GRID*, pages 640–650, May 2005.

[16] A. Plaat, H. E. Bal, and R. F. H. Hofman. Sensitivity of parallel applications to large differences in bandwidth and latency in two-layer interconnects. In *5th Int'l Symp. On High Performance Computer Architecture*, pages 244–253, Jan. 1999.

[17] J. W. Romein, H. E. Bal, J. Schaeffer, and A. Plaat. A performance analysis of transposition-table-driven work scheduling

in distributed search. *IEEE Trans. on Parallel and Distributed Systems*, 13(5):447–459, May 2002.

[18] S. S. Vadhiyar and J. J. Dongarra. Self adaptivity in Grid computing. *Concurrency and Computation: Practice and Experience*, 17(2–4):235–257, 2005.

[19] R. V. van Nieuwpoort, T. Kielmann, and H. E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. In *8th ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming*, pages 34–43, 2001.

[20] R. V. van Nieuwpoort, J. Maassen, T. Kielmann, and H. E. Bal. Satin: Simple and Efficient Java-based Grid Programming. *Scalable Computing: Practice and Experience*, 6(3):19–32, Sept. 2004.

[21] R. V. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. E. Bal. Ibis: a Flexible and Efficient Java-based Grid Programming Environment. *Concurrency & Computation: Practice & Experience*, 17(7–8):1079–1107, 2005.

[22] R. Wolski, N. Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems*, 15(5–6):757–768, Oct. 1999.

[23] G. Wrzesinska, R. V. van Nieuwport, J. Maassen, and H. E. Bal. Fault-tolerance, Malleability and Migration for Divide-and-Conquer Applications on the Grid. In *Int'l Parallel and Distributed Processing Symposium*, Apr. 2005.