Self-Adaptation in Divide-And-Conquer Systems

Christian Rossow

30th June 2009

Vrije Universiteit, Amsterdam

Master's thesis

Supervisors: Dr. Frank J. Seinstra

Dr. Jason Maassen

Prof. dr. ir. Henri E. Bal

Contents

Chapter 1

Introduction

Yielding a high performance is one of the main motivations behind parallel computing. In the past, different programming models for parallel computing have been developed. Among these, *divide and conquer* is a model, where work is recursively split into subtasks. These subtasks are computed in parallel on a set of nodes and their results are eventually merged again.

In the basic case, the distributed computation is performed on a static set of nodes. These nodes are selected prior to the application launch. However, in general it is unknown in advance which configuration performs best for a given algorithm. *Self-adaptation* is an automated way to optimize the execution of a parallel algorithm to certain criteria, e.g. to maximize parallel speedup.

In this chapter, we first explain the principles of divide and conquer. Next, Satin will be presented, a system implementing a divide and conquer programming model. We will then define the concept of self-adaptation. Finally, we will state the research goal of our work.

1.1 Divide and conquer system characteristics

Divide and conquer (D&C) is a parallel programming model that recursively divides a problem into smaller subproblems. Problem division is continued until the subproblems become trivial or at least simple enough to be solved sequentially. The divided work is distributed over several machines that collaborate in computation.

D&C systems use hierarchically-structured task graphs to spawn subproblems and link them to their parents. Every node participating in the parallel computation is responsible for solving at least one part of this task graph. Figure 1.1 shows an example task graph. In this example, a task is split into two subproblems each. The recursion depth is equal to the depth of the graph. The dotted lines group tasks that are executed locally on one node. Recursion demands that a node waits for results of subtasks that

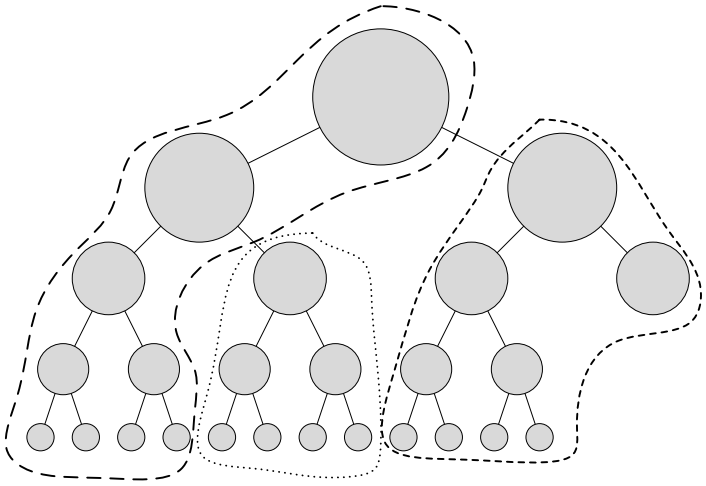


Figure 1.1: An example task graph of a D&C system

are computed by other nodes. A task is finished when the results of all its children are returned and processed.

1.2 The Satin Divide and conquer system

Satin [20] is specifically developed for running D&C applications on distributed memory systems. In Satin, single-threaded Java programs are parallelized by annotating methods that can run in parallel. The Satin compiler enhances the annotated methods with mechanisms to split up work and merge results. It supports two basic primitives of D&C systems, namely the *spawn* and *sync* operations.

In Satin, a *spawn* operation is a special form of a method invocation. Methods that can be spawned are annotated with a marker interface. For each call of an annotated method, new subtasks are spawned and put into a work queue. Items in this queue either can be computed locally or by a remote node. The *sync* operation will wait until all spawned calls in a method invocation are finished. A Satin example application is shown in Appendix A.

Load balancing in Satin is done using a work stealing algorithm called *cluster-aware random stealing* (CRS). A node running out of tasks tries to steal work from other nodes. It sends both an asynchronous request to a random node outside its cluster (to minimize communication latency) and a synchronous request to a random node in its own cluster. Nodes receiving a request will either send a task back or respond with an error if they currently do not possess any tasks. An idle node will continue stealing until it retrieves a task or until the application finishes.

In addition to the basic mechanisms described above, Satin also offers

other features such as shared objects, a global abort mechanism and fault tolerance. For a complete description of the Satin system refer to van Nieuwpoort et al. [20].

1.3 Definition of self-adaptation

Self-adaptation is a broad concept to adapt parts of a parallel system to obtain a better performance. There are two different self-adaptation approaches; *algorithm adaptation* and *resource selection*. The first tries to adapt the algorithm during runtime to improve its performance. Possible actions that can be taken are, for example, increasing the grain size of jobs to minimize communication overhead. This approach requires knowledge of the underlying algorithm. Resource selection aims at finding the set of worker nodes which performs best in a given execution. This can be done by adding nodes to computation, removing nodes or replacing nodes with other nodes. Resource selection can be useful in a wide range of scenarios, for example, if:

- the optimal node configuration is unknown in advance
- algorithm parallelization requirements are dynamic over time
- additional loads decrease the performance of a subset of nodes
- nodes or networks fail and need to be replaced
- new hardware becomes available during runtime

Given this broad applicability, in this thesis we focus on resource selection for self-adaptation.

1.4 Research goal and scope

This work is aimed at developing a general self-adaptation framework for D&C systems. Our goal is to minimize the application runtime, i.e. maximizing the speedup of a parallel algorithm execution. Our approach is developing self-adaptation heuristics that help to converge towards a set of nodes that perform best under the criterion of run-time minimization. These heuristics are to be integrated into a self-adaptation framework that does not require the programmer to change or annotate a parallel algorithm. As stated above, we use resource selection as self-adaptation mechanism that allows for meeting this constraint. The framework is to be made generic for it to be used in any heterogeneous parallel programming environment implementing the D&C paradigm and an integration should require as minimal effort as possible.

Although this work aims at supporting a broad range of parallel algorithms, we define certain requirements to algorithms than can be adapted with our framework. First, we require the parallel algorithm to implement the D&C model. Next, the computational complexity of leaf tasks (i.e. immutable subproblems) must not vary significantly. The performance metrics applied in the developed framework are more accurate if subtasks are homogeneous. Last, although computationally complex leaf tasks are also supported, self-adaptation is more efficient with less complex leaf tasks.

1.5 Main contributions of this work

The following list gives the basic contributions of this work:

- We introduce general applicable performance metrics (*productivity*) that can be used for self-adaptation of a broad range of D&C applications.
- We show that our performance metrics work better than a recent related self-adaptation approach of D&C applications.
- We discuss three self-adaptation strategies (*hill climbing*, *outlier replacement* and *failure watchguard*) that allow for efficient resource selection mechanisms.
- Used in combination, these strategies perform always better than a non-adaptive application run (in our evaluation, up to factor 5). In the worst case, i.e. when the application run does not require any self-adaptation, we show that the self-adaptation overhead is insignificant (in our evaluation, merely 0.66% performance loss).
- We present a generic framework that incorporates our work and can be used by other research groups.

The remainder of this work is structured as follows. In chapter 2, we describe the basic architecture of our self-adaptation framework. From the research goal follows that there are basically two problems that need to be explored. First, we need to gather statistics on each node that allow for an assessment of the current node configuration. We will extensively describe our performance measurement approach in chapter 3. Based on this data, we need to develop a self-adaptation mechanism that meets our research goals. We therefore present the self-adaptation strategies that we implemented in chapter 4. Given a reference integration of the framework into Satin, we evaluate our approach in chapter 5. Next, we show potential future work in chapter 6 and compare our work with related work in chapter 7. Finally, we conclude our work in chapter 8.

Chapter 2

Self-adaptation framework

Self-adaptation raises two substantial challenges that need to be solved. First, we need to develop metrics and mechanisms that measure the performance of a parallel algorithm run. These performance measurements will serve as input for a self-adaptation process. Intermediate measurements, in addition to a performance evaluation after program termination, allow for an integration of feedback loops to the self-adaptation process. Second, intelligent strategies are required to perform self-adaptation based on this performance data. These strategies aim at optimizing the actual performance according to certain criteria, e.g. maximizing the parallel speedup. To separate these two main challenges, we modeled a self-adaptation framework that divides these two task in a client-server architecture. Whereas on server-side performance statistics of the parallel application are gathered, the client-side applies self-adaptation strategies based on this data.

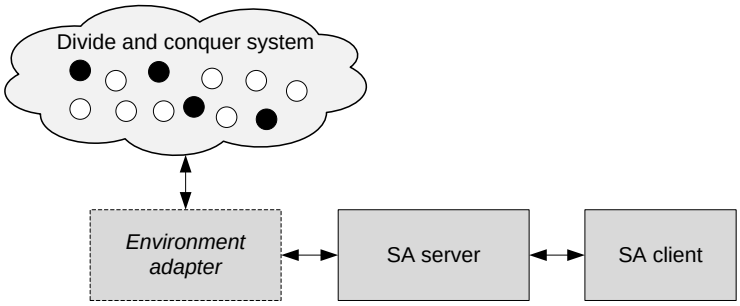


Figure 2.1: The self-adaptation framework structure

Figure 2.1 gives a basic overview of the framework that was integrated into a D&C system. The D&C system comprises of active nodes (i.e., nodes that already participate in the calculation) and inactive nodes (i.e., nodes that are available for computation). The framework is integrated into the D&C system via a pluggable *Environment adapter*. This adapter is D&C system-specific and acts as a communication interface between the D&C sys-

tem and the self-adaptation server (*SA server*). This server is responsible for gathering performance statistics. It regularly requests performance data from worker nodes in the D&C system. Moreover, the SA server provides functionalities to administrate this data and offers a well-defined interface to communicate with a self-adaptation client (*SA client*). This client is responsible for the actual self-adaptation decision making. An SA client retrieves statistical data from the SA server and bases self-adaptation strategies on this data. The strategies lead to adaptation decisions, which are passed to the SA server and in turn forwarded to the D&C system. Finally, the D&C system enforces these adaptation decisions by, for example, adding nodes to computation or removing nodes. Detailed implementation descriptions of the SA server, SA client and environment adapter are attached in Appendix C. In addition, a complete description of the client-server API is attached in Appendix D.

The chosen structure allows for a generic and yet adaptive framework. The environment adapter is a passive element of the framework that translates communication between the SA server and a specific D&C system. In detail, the environment adapter needs to pass statistics requests from the SA server to the D&C system, forward the according responses with statistic data back to the SA server and pass self-adaptation decisions made by the SA client to the D&C system. Thus, a successful integration of the framework into a D&C system requires implementing a specific environment adapter that can communicate with a given D&C system. In addition, the D&C system needs to a) implement an interface to collect statistics and send them on requests and b) implement adaptation functionality such as adding, removing, pausing and resuming nodes. In this work, we will show a reference integration of the framework into Satin.

Another advantage of our design is that it allows for an easy adjustment of the self-adaptation decision making process done by the SA client, without the need to touch the SA server at all. Moreover, the scheme supports multiple SA clients. Although developing a self-adaptation strategy was our primary intend for SA clients, we can also imagine other applications, such as grid monitoring software, that can make use of the statistical data offered by the SA server.

Chapter 3

Self-adaptation statistics

This chapter describes a model of how to gather and edit performance statistics such that they can serve as input for self-adaptation mechanisms. In the first section, we discuss metrics that help to assess the performance of a parallel algorithm execution. Next, we will evaluate our metrics against an approach described in the literature. The last section contains details on how we gather performance statistics from D&C worker nodes.

3.1 Performance metrics

D&C applications split a task into smallest subproblems, the so called *leaf tasks*. Given our assumption of homogeneous leaf tasks, the frequency in which nodes compute leaf tasks is a measure of how fast an algorithm progresses. We will describe metrics based on this measure that help to monitor the performance of a parallel application.

3.1.1 Local performance metrics

This section defines *speed*, *efficiency* and *productivity*, three basic notions to express the performance of a single node. We will start by defining the *speed* of a node k as:

$$speed_k = \#\ of\ leaf\ tasks\ computed\ per\ time\ interval\ without\ parallel\ overhead \tag{3.1}$$

In other words, the speed is the frequency of leaf task computations in a parallel application in an ideal world, excluding all parallel overhead such as load balancing, communication delays or idle times. To also incorporate parallel overhead, we define the *efficiency* of node k as the ratio of the total runtime the node actually spends on computation:

$$efficiency_k = \frac{time\ on\ computation_k}{total\ time_k} \tag{3.2}$$

The product of both measures indicates how much a node contributes to computation. We define this term as *productivity*. It shows, how many tasks a node k actually computes, taking into account parallel overhead:

$$productivity_k = efficiency_k * speed_k \tag{3.3}$$

3.1.2 Global performance metrics

The metrics defined above indicate the performance of a single node only. To measure the performance of a parallel algorithm, it is required to consider statistics of all nodes that actively participate in the computation. We will start by defining the *total speed* of all N nodes:

$$total\ speed = \sum_{k=1}^{N} speed_k \tag{3.4}$$

This measure merely represents the theoretical maximum performance considering linear speedup without any parallel overhead. In particular if many nodes are part of the process, the parallel overhead is not negligible. The *total productivity*, therefore, is a measure of how fast the computation proceeds globally:

$$total\ productivity = \sum_{k=1}^{N} (efficiency_k * speed_k) \tag{3.5}$$

Our work is based on the assumption that this measure is a good estimate of the actual application runtime, which we try to minimize. It is important to realize that a higher total productivity leads to shorter application run times, and vice versa.

As a last measure, we define the *total efficiency* of all computing nodes. The total efficiency indicates which ratio of globally available run time is used for actual algorithm computation. A first approach would be averaging the node efficiencies. This approach, however, does not normalize the efficiency values for the speed of a given node. Instead, in order to estimate which ratio of computational power is effectively used, the total efficiency should take into account different node speeds. We define total efficiency as:

$$total\ efficiency\ = \frac{total\ productivity}{total\ speed}\ = \frac{\sum_{k=1}^{N} (efficiency_k * speed_k)}{\sum_{k=1}^{N} speed_k} \tag{3.6}$$

As a side note, we remark that the total efficiency and total productivity show a correlated behavior. This is of particular importance when the efficiency is to be kept within given bounds. We show the effects of adding and removing nodes on this correlation in Appendix B.

3.2 Related performance metrics

In 2007, Wrzesinska [23] presented a self-adaptation mechanism for algorithms implementing the D&C paradigm. Whereas our work uses the metrics of *productivity*, Wrzesinska defined a measure called *weighted average efficiency* to assess the performance of the system:

$$wa\_efficiency = \frac{\sum_{k=1}^{N} relative\ productivity_k}{N} \tag{3.7}$$

The wa_efficiency is called weighted, since it takes into account the *relative productivity* of a node. The node with the highest speed defines its relative productivity as its own efficiency and determines the maximum productivity. For all other nodes, the relative productivity of node k is computed by $relative\ productivity_k = productivity_k / maximum\_productivity$, i.e. a value $\in (0, 1]$. Wrzesinska states the purpose of this was to model slower nodes as less efficient, under the assumption that adding slower nodes yields less benefit.

The crucial difference between the wa_efficiency and our definition of total efficiency in equation 3.6 is that the wa_efficiency does not normalize the nodes according to their speed. As a consequence, the wa_efficiency can deliver misleading data in certain situations. Table 3.1 shows an example, where the wa_efficiency gives wrong indications about the total efficiency. It shows a parallel computation with four active nodes, where two of them are faster than the others. Both the efficiency (equation 3.6) and the wa_efficiency (equation 3.7) are calculated for each setting.

Table 3.1(a) shows the initial situation of the scenario, with two nodes being slow but efficient and the other two fast but inefficient. In case of table 3.1(b), a slow but efficient node is removed from computation. As a result, the total efficiency decreases (since a highly efficient node left the computation), but the wa_efficiency increases. Similarly, table 3.1(c) simulates removing a fast but inefficient node. In this case, the total efficiency increases (since a very inefficient node left the computation), but the wa_efficiency increases.

This example indicates that wa_efficiency is not a good way to express the total efficiency of the computation. As mentioned earlier, it does not consider the total sum of relative productivities. The formula assumes every node to have a relative speed of 1 when dividing the sum of productivities. Therefore, the wa_efficiency works better for homogeneous systems and clearly disfavors systems where at least one node has a relatively high speed.

As a consequence, the wa_efficiency is an unreliable source for self-adaptation. Despite this, Wrzesinska [23] performed self-adaptation by keeping the wa_efficiency within the bounds of 30%–50%. Whenever the

(a) Initial configuration

node ID	speed	efficiency	productivity	relative pr.
1	50 T/s	40%	20 T/s	0.4
2	50 T/s	40%	20 T/s	0.4
3	10 T/s	80%	8 T/s	0.16
4	10 T/s	80%	8 T/s	0.16
			productivity	56 T/s
			efficiency	46.6%
			wa_efficiency	28.0%

(b) Removed slow, but efficient node

node ID	speed	efficiency	productivity	relative pr.
1	50 T/s	40%	20 T/s	0.4
2	50 T/s	40%	20 T/s	0.4
3		node removed		
4	10 T/s	80%	8 T/s	0.16
			productivity	48 T/s
			efficiency	43.6%
			wa_efficiency	32.0%

(c) Removed fast, but inefficient node

node ID	speed	efficiency	productivity	relative pr.
1	50 T/s	40%	20 T/s	0.4
2		node removed		
3	10 T/s	80%	8 T/s	0.16
4	10 T/s	80%	8 T/s	0.16
			productivity	36 T/s
			efficiency	51.4%
			wa_efficiency	24.0%

Table 3.1: Example where wa_efficiency misbehaves. After (a) the initial situation, both (b) a slow node and (c) a fast node are removed.

wa_efficiency drops below 30%, Wrzesinska removed nodes from computation to increase the wa_efficiency. As table 3.2 on page 11 indicates, this strategy can fail in heterogeneous environments. Although all nodes have an excellent efficiency of 100% and adding a node would in general raise the speedup, the strategy suggests removing a node. In this work, we therefore base our work on the definition of productivity.

node ID	speed	efficiency	productivity	relative pr.
1	60 T/s	100%	60 T/s	1.00
2	10 T/s	100%	10 T/s	0.17
3	10 T/s	100%	10 T/s	0.17
4	10 T/s	100%	10 T/s	0.17
5	10 T/s	100%	10 T/s	0.17
6	10 T/s	100%	10 T/s	0.17
7	10 T/s	100%	10 T/s	0.17
			productivity	120 T/s
			efficiency	100%
			wa_efficiency	28.6%

Table 3.2: Scenario where the wa_efficiency drops below the 30% bound

3.3 Statistics gathering

The self-adaptation server of our framework regularly queries any available node for a statistical report. This report contains performance statistics of a particular node in a given measurement interval, according to the metrics described above. To overcome temporal deviations in the performance statistics, we aggregate λ subsequent node statistics to a single statistical block. Deviations in performance, e.g. caused by short-term work loads at nodes, can lead to wrong assumptions when relying on the most recent measurement only. A higher weight is given to more recent measurements, since this data is more up-to-date than older statistics. As an example, Equation 3.8 computes the productivity of the most recent λ measurements. ρ determines the age of a measurements, i.e. $productivity_1$ is the most recent.

$$block\_productivity = \frac{\sum_{\rho=1}^{\lambda}(\alpha^{\rho} * productivity_{\rho})}{\sum_{\rho=1}^{\lambda}\alpha^{\rho}} \tag{3.8}$$

In equation 3.8, the parameter $\alpha \in (0, 1]$ determines the weight for more recent statistics. The higher α, the less weight is given to more recent data. $\alpha = 1$ equals a normal (unweighted) average of all λ values. In our implementation, we chose to aggregate $\lambda = 5$ measurements to a single block. The aggregation is done with $\alpha = 0.8$, i.e. the most recent measurement makes up $\sim 30\%$ of the total average.

In addition, we included a mechanism to normalize statistics for measurement inaccuracies. In practice, both broadcasting the request to all nodes, as well as locally processing the statistics requests takes some time, say T_{proc}. If this time T_{proc} is irregular, the locally collected statistics at a node are not relative to the measurement interval T_{int} anymore and slightly differ per interval. Therefore, every node additionally includes the time T_{passed} in the statistical data. T_{passed} is the time interval in which the provided statistics

were actually measured. Upon receipt of the statistics along with T_{passed}, the collector process can normalize the statistics back to T_{int} by calculating $normalized\ stats = actual\ stats\ /\ T_{passed} * T_{int}$. As a result, the normalized statistics of each node share a common base.

Moreover, there is a need to automatically adapt the length of the measurement interval when gathering statistics. In cases where the measurement interval is too short, the collected statistics are inaccurate. For example, let the interval $T_{int} = 1\ second$, and computing a single task also takes 1 second. This would practically allow the statistics to vary between 0–2 performed tasks per interval. A possible counter measure is to increase T_{int} for as long as the average productivity of all nodes is below a *minimum productivity threshold*. In general, however, the measurement interval should be kept small. More frequent measurements allow for quicker reactions on performance changes. This is the main reason why computationally cheap leaf tasks allow for a faster adaptation (as mentioned in section 1.4)

Chapter 4

Self-adaptation strategies

In this chapter, we show the need to develop intelligent self-adaptation strategies. After that, we present three strategies that are integrated in our self-adaptation framework. First, we examine the hill climbing algorithm that strives to find a good set of nodes during algorithm runtime. Next, we will discuss the outlier replacement strategy that allows for migrating work from poorly performing nodes to better locations. Lastly, we focus on the failure watchguard that checks if nodes failed and replaces them accordingly.

4.1 Motivation for self-adaptation strategies

A first approach to find the ideal set of nodes is exploring all combinatorial possible node configurations. Given a static system, this strategy is guaranteed to find the optimal solution. However, exploring the entire search space will add a tremendously high overhead to the self-adaptation process. In this section, we will discuss how to deal with the complexity of finding the best solution.

In theory, having N different nodes, exactly $2^N - 1$ possible node configuration exist. Considering the overhead of exploring all these configurations, it is impossible to evaluate each configuration separately. In addition, the underlying algorithm or node/network performance may change over time. Thus, comparing two configurations explored at two different points in time is prone to errors.

Instead of performing a brute-force approach, we require intelligent self-adaptation strategies that strive towards finding a well-performing configuration. These strategies avoid exploring the entire search space and take into account that previous evaluations may become outdated.

4.2 Hill climbing

The hill climbing strategy is a powerful tool for finding a node configuration that performs well without exploring the entire search space. It basically adds and removes nodes from the set of computing sites and remembers the set that performed best.

4.2.1 Basic algorithm description

The pseudocode of Algorithm 1 summarizes the behavior of the basic hill climbing algorithm. The hill climbing algorithm starts by adding random nodes as long as the overall productivity increases. After each adaptation, the algorithm compares the previous and current total productivity values. As soon as the productivity decreased with an adaptation, it switches the climbing direction and starts removing the least productive node. Again, it continues removing nodes as long as the productivity is rising. At the point in time when removals also decrease the productivity, the hill climbing algorithm keeps the last configuration and terminates.

```
direction ← ADD
maxProductivity ← currentProductivity

while direction ≠ FINISHED do
  currentProductivity ← recently measured total productivity
  if currentProductivity < maxProductivity then
    /* last change was not succesful */
    if direction = ADD then
      maxProductivity ← currentProductivity
      direction ← REMOVE /* switch to remove now */
    else
      direction ← FINISHED /* terminate */
    end if
  else
    /* last change was succesful, redo */
    maxProductivity ← currentProductivity
    if direction = ADD then
      add random node to computation, if any available
    else
      remove least productive node from computation
    end if
  end if
end while
```

Algorithm 1: The basic hill climbing algorithm

The major advantage of the hill climbing algorithm is that it limits the adaptation overhead to a minimum. The basic algorithm involves a total of only two adaptation steps that decrease the productivity; all other steps increase the productivity.

4.2.2 Algorithm enhancements

In practice, it is not guaranteed that the basic hill climbing algorithm finds the optimal configuration. Moreover, it will take very long until the algorithm terminates in certain scenarios. In this section, we expand the basic algorithm with several improvements.

So far we assumed that changes due to adaptation by the algorithm will have an effect on the performance metrics immediately. Adding and removing nodes however requires some time. In addition, an added node needs to get work from other nodes before it actually helps to raise the productivity. The same holds for a node removal, where the leaving nodes need to spread their work to the remaining nodes. As a result, it may take some time before the effects of the adaptation can be observed in the gathered statistics. Therefore we include a so called *grace period* to the algorithm. This period defines an interval after adaptation, in which statistics are ignored for further processing. In other words, the algorithm sleeps for the grace period after an adaptation step and gives the new configuration a chance to converge towards stable productivity data. Figure 4.1 illustrates the integration of a grace period to the hill climbing algorithm.

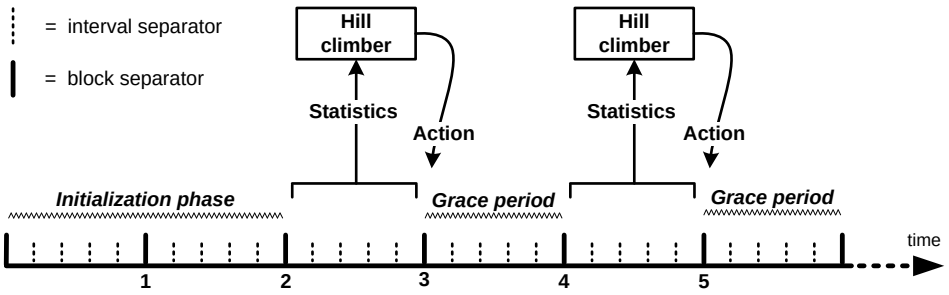


Figure 4.1: Interaction between hill climber and grace periods

In an ideal world, the number of tasks performed by a node does not vary over time. In practice, however, small irregularities in productivity measurements are observable even in static systems. In addition, the non-deterministic measurement process slightly influences the number of tasks completed per interval. Therefore, it is sensible to introduce a tolerance against these deviations to the algorithm, referred to as *toleranceFactor*. The hill climbing process switches its direction if and only if $currentProductivity * toleranceFactor < maxProductivity$. This allows the algorithm to continue

processing even if it is moving away from a local maximum already. This mechanism is crucial to overcome small deviations in performance data, although it may introduce some adaptation overhead.

Once the algorithm terminates, the hill climbing algorithm has found a good configuration of nodes that performs well. In case of dynamic algorithm behavior, this configuration may perform well at one moment, but may result in significant under- or over-provisioning of resources at a later moment. Therefore it is sensible to include a restart mechanism into the hill climbing algorithm. The algorithm restarts, if more recent statistics significantly deviate from earlier measurements or if new hardware becomes available. In addition, the client enforces a restart after $maximumInactivityLenght$ intervals of inactivity.

4.2.3 Accelerating the adaptation

Given the fact that a grace period follows each adaptation, converging towards a good set of nodes may be a time consuming task. We added a feature called *accelerator* to mitigate this issue. Depending on the impact on productivity of the previous adaptation, the accelerator enforces multiple adaptations in the same direction at once. This allows for a faster convergence to a good set of nodes. Therefore, the accelerator feature first computes the *productivity gain* of an adaptation as:

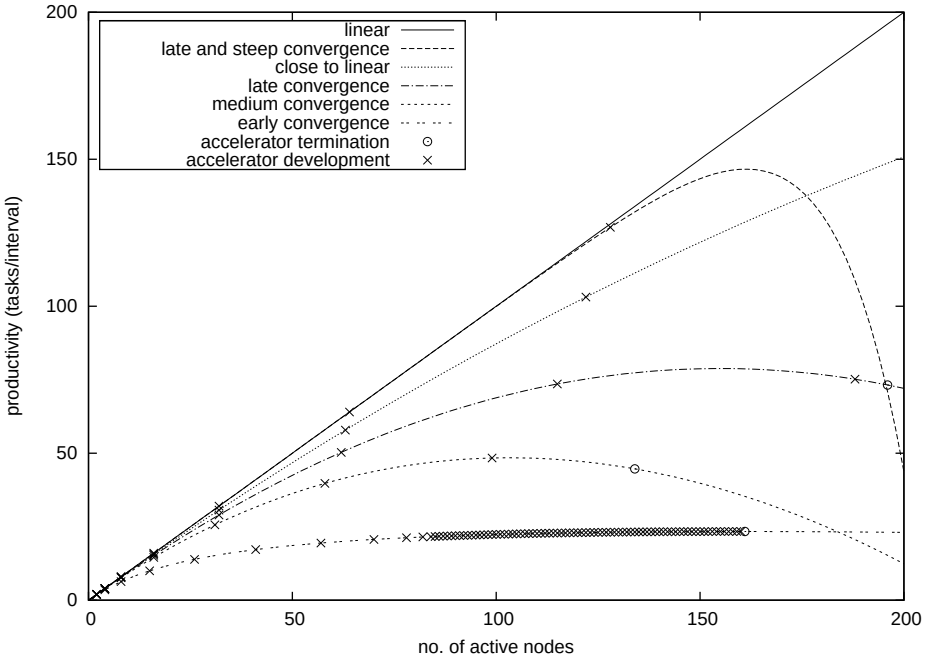
$$productivity\ gain = total\ productivity_{new} - total\ productivity_{old} \quad (4.1)$$

Depending on the number of active nodes N and the accelerator factor β, the *adaptation step size* for the next adaptation is computed as:

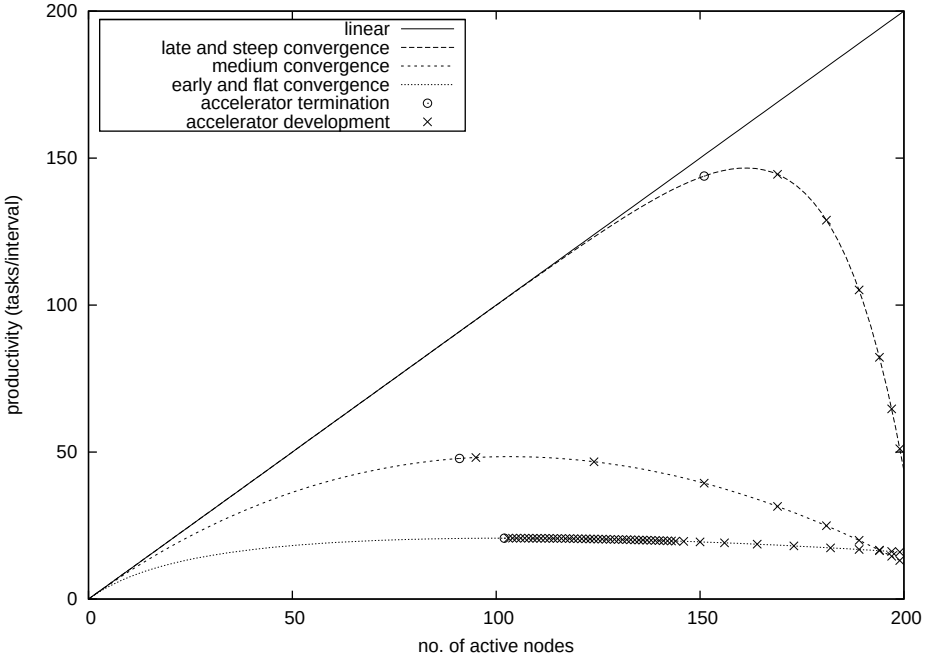
$$adaptation\ step\ size = \frac{productivity\ gain}{total\ productivity} * \beta * N \quad (4.2)$$

According to equation 4.2, the adaptation step size is high, if the previous adaptation yielded a close-to-linear improvement in productivity. A decrease in performance forces the accelerator to stop, small improvements lead to smaller future step sizes. The factor β indicates how aggressive the accelerator should adapt the number of nodes.

Figure 4.2 indicates how the accelerator behaves when adding and removing nodes. We chose $\beta = 2$ when adding nodes, and a more conservative setting of $\beta = 1.5$ when removing nodes. Each curve represents the possible productivity development (y-axis) in relation with the number of nodes (x-axis). We show in Figure 4.2(a) how the accelerator behaves when adding nodes, with an initial configuration of 1 active node. In most of the scenarios, an aggressive setting of $\beta = 2$ overshoots the productivity maximum. As a consequence, the optimal configuration is reached only if the hill climbing starts removing nodes again. In the case of an early and slow convergence



(a) Accelerator adds nodes with $\beta = 2$ having initially 1 active node



(b) Accelerator removes nodes with $\beta = 1.5$ having initially 200 active node

Figure 4.2: Accelerator development when adding and removing nodes

(lowest line), the accelerator becomes inactive having 82 active nodes and only single nodes are added to computation from this point on. This behavior is not ideal, but the deviation from the ideal set of nodes has a negligible effect on the total performance.

In figure 4.2(b) we show the node removal process, with an initial configuration of 200 active nodes. A more conservative setting of $\beta = 1.5$ limits the risk of skipping the maximum. Choosing the right accelerator factor is a trade-off between the convergence speed and the adaptation correctness. We see that the flat convergence (bottommost line) forces the accelerator to stop early. Again, in this case, the negative effect of this deviation from the ideal number of nodes is negligible. In the other two cases, the removal proceeds quickly and the accelerator terminates after 7 and 9 adaptation phases, respectively.

Although the ideal configurations were missed in most of the scenarios, the accelerator is still a powerful tool. Its main goal is to quickly find a good solution, even if it is not ideal. If the hill climbing algorithm is restarted after a while, the initial configuration is different from the first run. The accelerator will most likely not be active in a second run, allowing the hill climbing algorithm to climb towards the ideal configuration. In section 5.2 we will return to this issue and show the behavior and impact of the accelerator for a real execution scenario.

4.3 Outlier replacement

Apart from the adding and removing of nodes, a second important self-adaptation strategy addresses work migration away from slow nodes. The outlier replacement strategy replaces poorly performing nodes with better ones.

4.3.1 Basic algorithm description

The outlier replacement strategy focuses on nodes that perform worse than average. The strategy calculates how the productivity of a node performs compared to the median productivity. If the relative difference exceeds the so called *maximum deviation threshold*, it is replaced by another node chosen randomly from the set of available nodes. Algorithm 2 describes the procedures of the basic outlier replacement algorithm.

4.3.2 Algorithm enhancements

In certain situations, the outlier replacement algorithm will iterate replacing identical nodes. Consider a grid of 10 available nodes, where two nodes A and B are slower candidates. Let the faster 8 nodes and A be active in an initial situation. As A performs very badly compared to the other nodes,

```
sort active nodes Θ by ascending productivity
currentMedian ← median(productivities in Θ)

for n ∈ Θ do
  /* iterate over all nodes, from slowest to fastest */
  slowerThanMedian ← 1 − n.productivity/currentMedian
  if slowerThanMedian > max_deviation_threshold then
    /* deviates too much from median productivity */
    replace n with random inactive node
  else
    /* productivity in boundaries, next nodes perform better */
    return
  end if
end for
```

Algorithm 2: Basic outlier replacement algorithm

it is replaced by a random node. In our case, only B is available, and replaces A. B in turn does not perform better, and is going to be replaced by A again. In our work, we mitigated this issue by adapting the maximum deviation threshold according to the feedback of a node replacement. If the productivity gain of the previous adaptation done by the outlier replacement strategy is negative, the maximum deviation threshold from the mean is increased by a factor ξ. In the scenario of iterative replacements mentioned above, the threshold will eventually be set high enough such that the replacement condition is not valid anymore.

4.4 Failure watchguard

A third and last implemented strategy is the failure watchguard. This strategy is responsible for tracking which nodes are currently running and providing statistics. If a node failed sending statistics the watchguard starts observing this node. The failure watchguard tolerates a node temporarily not sending any data, e.g. due to high temporal loads. However, if a node fails communicating during the entire *toleration period*, it will be considered as failed and be replaced. There is a trade-off in choosing the length of the toleration period. If the period is too long, a node failure is detected quite late and performance is lost by replacing the failed node relatively late. Too short tolerance periods, on the other hand, may dismiss nodes that temporarily failed to send statistics but that still actively participated in the computation.

Chapter 5

Evaluation

This chapter gives an evaluation of our work. In the first section, we describe the setup that was used for the evaluation. The next three sections give an isolated evaluation of the three strategies that we developed in Chapter 4. Finally, the last section evaluates an extensive scenario in which all three strategies are used in collaboration.

5.1 Evaluation setup

The first part of our evaluation setup description explains the reference integration of our framework into a D&C system. Next, we will discuss a D&C algorithm that was used during the evaluation. The last subsection will describe the self-adaptation strategy configuration used for the evaluation.

Reference integration

For the evaluation, we integrated the self-adaptation framework into the Satin D&C system as discussed in section 1.2. The integration makes use of two mechanisms that are present in the communication subsystem underlying the Satin implementation [21], namely *message upcalls* and *registry events*. With message upcalls, a subscriber is informed as soon as a Satin instance received a message by some other node. This mechanism is used to send statistical data from every node to the statistic collector process. In addition, registry events are received by the environment adapter. Events help to inform the adapter whenever a node joins or leaves the computation. The adapter forwards these notifications to the SA server, that in turn keeps track of active worker nodes.

In addition, the integration required to change some parts of the Satin framework. Most importantly, mechanisms to remove nodes from computation and add nodes to computation were added. Basically, we have two options here. First, whenever a new worker should be added, a new Satin instance can be spawned. Accordingly, a Satin instance can entirely shutdown

when it is removed from computation. Results showed that the adaptation speed of this solution is slow however, in particular if new nodes are added to the system (e.g. due to the need to go through a cluster reservation system). Moreover, spawning new Satin processes requires additional process information, such as e.g. program parameters.

Therefore, we favored the second option: having a static set of worker nodes. Whenever a node is removed from computation, it does not terminate but merely stops computing jobs and sends off the work of his local queue to other nodes. If a node receives such data, it does not consider the sending node as active anymore. Also, nodes trying to steal work from inactive nodes are notified that the node is currently paused. Reducing the set of active nodes is of particular importance, as parallel overhead is reduced only if work steal requests are more likely to be answered positively. As soon as a node resumes computation, it starts stealing work from other nodes. If a node receives a steal request, it considers the stealing node as active again. This scheme assures that eventually all nodes know if other nodes are active.

Evaluation algorithm

For our evaluations, we implemented a D&C algorithm that recursively divides a task into four subtasks up to a recursion depth of 15. The leaf tasks of this algorithm have a negligible small computational complexity. To be able to control the productivity of each node, we introduced a limitation of the productivity in Satin that takes into account, how many nodes actively participate in computation. In general, due to parallel overhead, the higher the number of active nodes, the lower is the productivity of a single node. To evaluate dynamic algorithms, we also vary the productivity depending on the time that passed since the program launch.

This setting allows for an evaluation of our work in a real D&C system. Having the discussed productivity limitation in place, we can evaluate multiple productivity search spaces with our self-adaptation framework. We parameterized our algorithm setup such that we are able to determine the ideal configuration for each run according to the needs of the given self-adaptation scenario. Figure 5.1 illustrates an example productivity search space in that the total productivity increases up to a maximum of 10000 with 17 active nodes. In this example, if more than 17 nodes are active, the total productivity decreases again due to parallel overhead. We varied the specific productivity search space according to the needs of any of the following evaluations. A self-determined search space enables us to compare the actual self-adaptation process with theoretically ideal node configurations. In addition, if required by the scenario, we added events such as having additional load to every node and simulated node failures. To summarize, our evaluation algorithm allows for diverse evaluations of our work under a controlled but realistic D&C application.

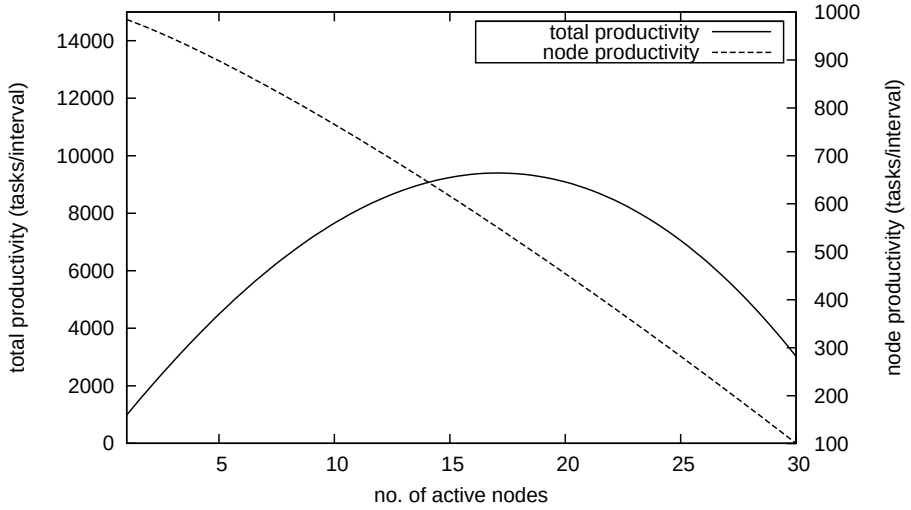


Figure 5.1: Example productivity search space that can be controlled using our evaluation algorithm

Evaluation strategy settings

As the self-adaptation strategies are parameterized, we will specify here which settings we used for our evaluation. All self-adaptation strategies take effect after the algorithm was running for more than 15 measurement intervals. For all measurements, we had a *grace period = 5* measurement intervals (i.e., 5 seconds) in place. In addition, we instrumented the hill climbing strategy to use the *tolerance factor = 1.03*. Furthermore, we enforced the inactive hill climbing strategy to restart after *maximumInactivityLenght = 20* intervals. Moreover, we let the failure watchguard ignore a node to failure up to *toleration period = 10* intervals. Finally, the outlier replacement strategy replaced a node only if it was *max_deviation_threshold = 0.2* slower than the median node.

5.2 Finding an optimal configuration

We start by evaluating the hill-climbing strategy that aims to find a well-performing configuration by adapting the number of active nodes. Parallel algorithms can have a static ideal configuration, but can also change their parallelization requirements during runtime. We split this section therefore in two parts. First, we evaluate whether our strategies find well-performing configurations with algorithms that do not show dynamic behavior and discuss the introduced overhead. Second, we show how the hill-climbing strategy behaves in dynamic settings.

5.2.1 Static algorithms

Converging towards a well-performing configuration

Figure 5.2 on page 24 shows two self-adaptation scenarios in a static settings. In Figure 5.2(a), we show an evaluation of our strategies in a scenario where at program launch too few nodes were active. Starting from a single active node, the hill-climbing strategy converges towards the ideal total productivity. Due to an aggressive accelerator setting, the number of nodes rockets within 6 subsequent adaptation steps from 1 to 32. Although the strategy already exceeded the ideal number of nodes after the fifth adaptation, it continues due to positive feedback of the fifth adaptation. The reason for this is that having 12 nodes (5 too few) yielded a lower total productivity than with 20 nodes (3 too many). As soon as the hill-climbing algorithm switches its direction to remove nodes, the accelerator uses a more conservative setting to strive for the ideal configuration. This less aggressive setting helps to reach the optimum on the way back. After a number of node removals, the actual number of active nodes converges to the ideal setting. Although the configuration is constantly adapted and on average 2–3 nodes away from the ideal setting, it is important to note that the performance as expressed by the total productivity is close to ideal.

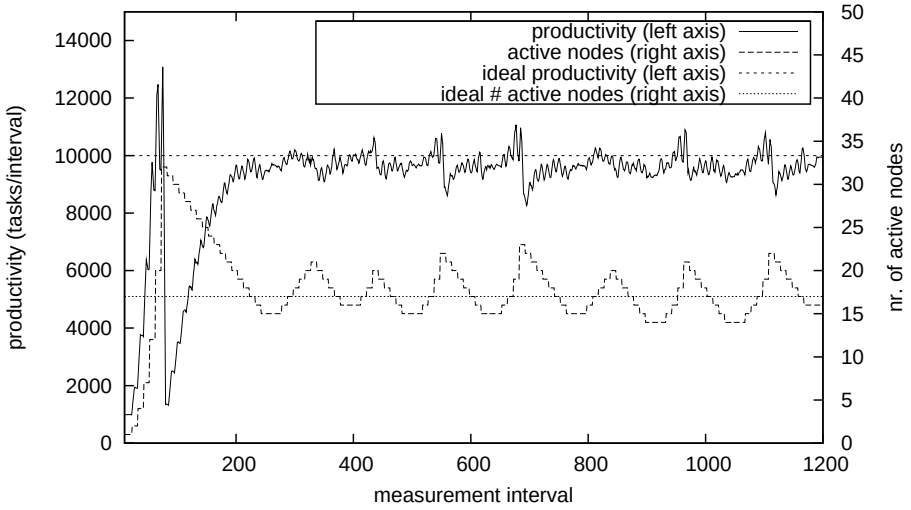
Figure 5.2(b) shows the self-adaptation behavior if initially too many nodes are active. The hill-climbing algorithm starts to constantly remove nodes until it exceeds the ideal configuration after 180 measurement intervals. Similar to the first scenario, it continues changing the ideal configuration, but stays close to the optimal performance.

The two scenarios show that the hill-climbing algorithm is capable of finding a well-performing configuration with a static algorithm parallelization. We saw that the accelerator is very aggressive when adding nodes and noted that the node removal process is slow. In addition, the algorithm does not terminate if it finds a well-performing configuration. This is desired for adaptation of algorithms with static parallelization requirements, whereas it would be suboptimal in dynamic settings. We leave it as future work to further optimize the hill-climbing procedure for static settings.

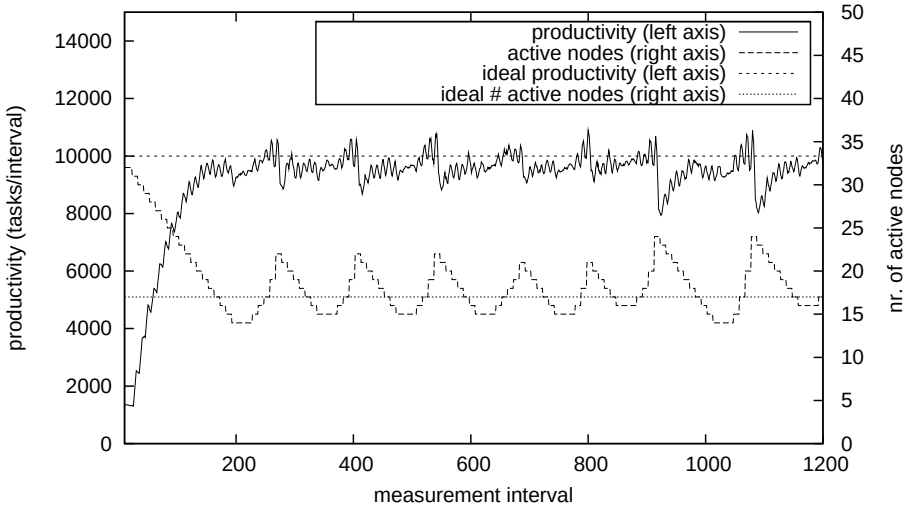
Overhead of the self-adaptation process

An important quality to measure is the added overhead of the self-adaptive system with respect to a constant algorithmic behavior. We evaluate the overhead that was introduced by our self-adaptation framework with two different static productivity search spaces.

In the first scenario, our evaluation algorithm performs best having a static set of $N = 50$ nodes. Figure 5.4(a) on page 26 draws the productivity search space dependent on the number of nodes and measurement interval. The self-adaptation was performed as shown in figure 5.4(b). The adaptation



(a) Too few nodes at algorithm launch

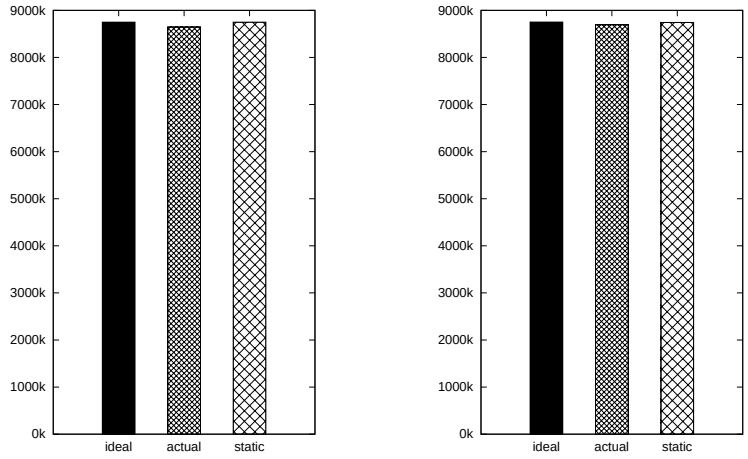


(b) Too many nodes at algorithm launch

Figure 5.2: Performance of self-adaptation in finding the optimal configuration

strategies constantly adapt the set of active nodes. During the runtime, there is rarely a point in time, where the ideal configuration is hit. On the other hand, the productivity does not vary more than the tolerance factor allows for. In this case it becomes obvious that the tolerance factor needs to considered very carefully.

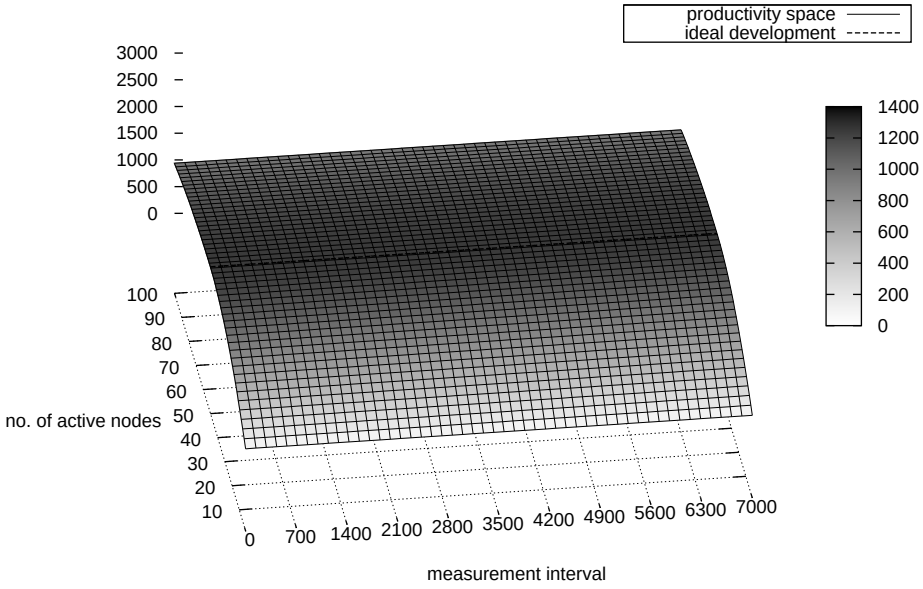
As additional evaluation of this scenario, we cumulated the productivity in each time interval to a productivity sum that is achieved during the entire runtime of 7000 intervals. Figure 5.3(a) shows the overhead in this scenario that was introduced by our self-adaptation framework. The first value in each graph is the productivity that was achieved by having an ideal set of nodes in place. The second bar indicates the performance that was achieved using the adaptation framework. The third bar gives a comparison to the best performing static set of nodes. For this comparison we assume that the ideal configuration of nodes is known in beforehand. In this specific case, the adaptation yields a performance loss of 1.18% compared to the static and ideal set of nodes (both are equal).



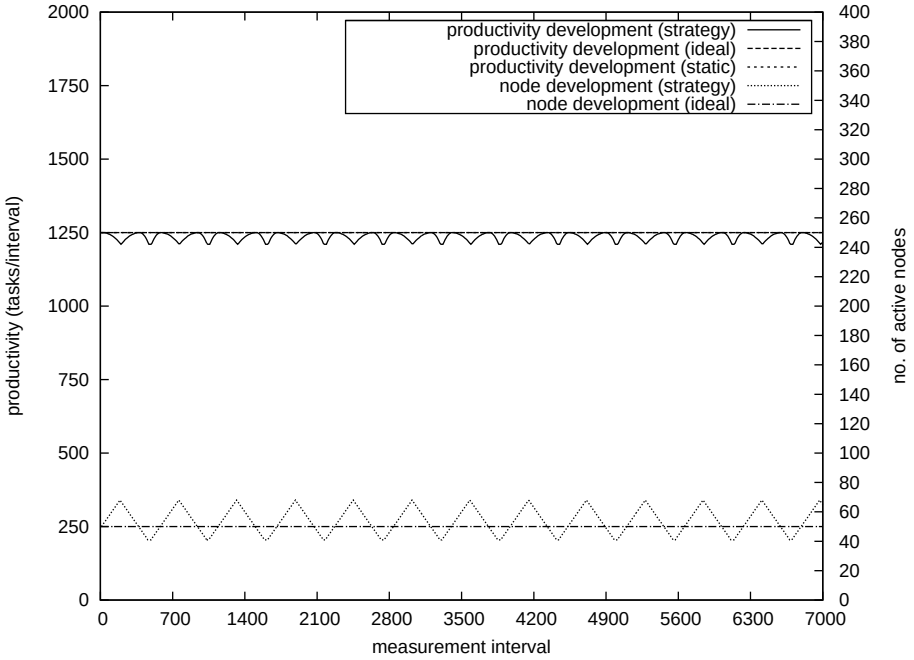
(a) Static productivity (b) Random variations

Figure 5.3: Productivity sum during runs with static algorithms, measured in the total number of computed leaf tasks during a runtime of 7000 seconds.

In the second scenario, instead of having a constant productivity, the productivity search space has some small deviations over time. This scenario tries to address the issue that, in practice, there are fluctuations in productivity measures over time, though the average productivity is constant over time. Figure 5.5(a) on page 27 draws a productivity graph of a possible scenario, in which the ideal number of nodes is always close to $N = 50$. According to figure 5.5(b), the self-adaptation strategy is stable against this scenario. It can be seen that the strategy rarely adapts towards an outstanding slow configuration. Figure 5.3(b) shows that the added over-

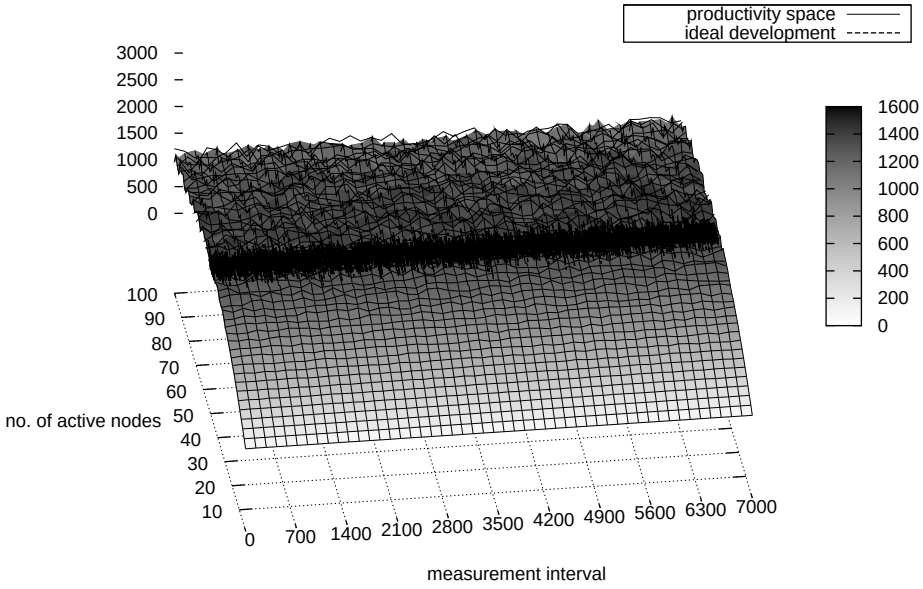


(a) Productivity search space dependent on time and number of active nodes

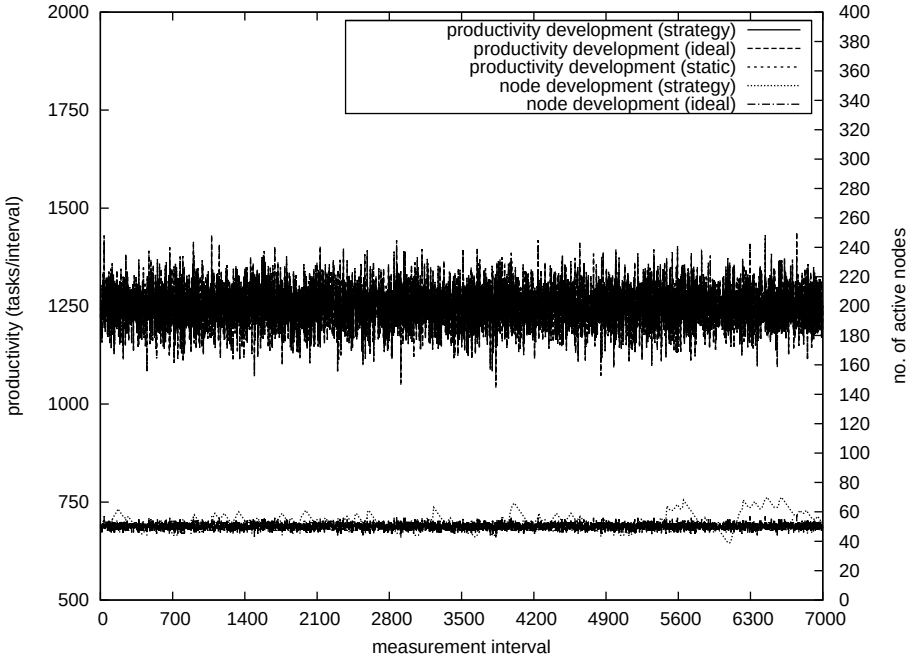


(b) Comparison between productivities of ideal, adapted and static configuration

Figure 5.4: The ideal number of nodes is constant



(a) Productivity search space dependent on time and number of active nodes



(b) Comparison between productivities of ideal, adapted and static configuration

Figure 5.5: The ideal number of nodes varies only slightly from 50

head is less if the productivity has slightly fluctuations. Again we assume that the ideal configuration is known prior to the algorithm launch. This time, due to the small variations in the productivity search space, the hill climbing algorithm terminates faster and yields a smaller error. The added overhead in this scenario is a loss of 0.56% compared to the static run, and merely 0.66% compared to the ideal performance.

Possible improvements to scenarios with static algorithms could be to stop adaptation, if the total productivity does not change over time. In addition, in order to continue running with the ideal set of nodes, one would then need to restore to the best configuration found so far. Although this scheme works very well for static algorithms, it virtually always fails in dynamic environments. In such environments, a configuration that achieved the highest productivity in the past is most likely not the current best configuration any longer. Clearly there is a trade-off to make between having a general adaptation scheme with limited overhead, or a scheme with virtually no overhead but not supporting dynamic algorithms.

It is very likely, however, that the self-adaptation scenarios overwhelm or at least compensate the small overhead. In addition, for the scenarios discussed above, we assumed that the ideal configuration was known. As shown in this section, our self-adaptation framework is able to converge to this ideal configuration even if it is unknown. In cases where the estimation of the ideal configuration as done by a human is wrong, our self-adaptation framework will virtually always outperform even static algorithms with a static configuration.

5.2.2 Dynamic algorithms

In this section, we discuss four scenarios with dynamic algorithm behavior, a subset of all possible situations that show distinctive patterns in dynamics. For the evaluation, we created productivity search spaces that depend on the time that was passed since the application launch. At any point in time, a specific configuration is ideal. For simplicity, we focus our evaluation in this section on homogeneous environments and express a configuration by N, the number of active nodes. For the evaluation of dynamic algorithms, we further limited the maximum number of active nodes to $N = 100$. The following subsections shows how effectively the hill-climbing strategy handles dynamic algorithm behavior. In each of the four scenario, we will compare a self-adapted algorithm execution with both an execution with an ideal static set of nodes and a dynamic ideal configuration dependent on the time.

Steady increase of active nodes

If the parallel overhead decreases over time, more nodes can be added to the computation to achieve a higher productivity. Figure 5.7(a) shows a possible

productivity search space for such a scenario. Whereas in the beginning a few nodes achieve the highest speedup, at a later stage the ideal number of active nodes increases.

The self-adaptation mechanism adapts to these changing circumstances, as plotted in figure 5.8(b). During the first half of the run, the strategy is only adding nodes. Although it adds more nodes than ideal, the total productivity increases due to the development of the productivity search space over time. In this specific scenario, all three possible decisions by the hill-climbing strategy (i.e., adding a node, removing a node, doing nothing) would lead to a gain in productivity. Therefore, first when reaching the maximum of 100 active nodes after 3,600 intervals, the algorithm switches the adaptation direction to node removal. Not having the limit of 100 nodes in place, the strategy would continue adding nodes until it reached a decrease in productivity when adding nodes. Adding speculative direction changes, i.e. switching the direction although the productivity is increasing, may be a reasonable option to further improve the hill-climber algorithm. The number of active nodes however does not necessarily need to be always close to ideal, as long as the actual productivity is high. Regardless of the fact that the number of active nodes is far from ideal until measurement interval 4,000, the actual productivity in this scenario is mostly better than with a static set of nodes.

Once the hill-climbing direction was switched for the first time at interval 3,600, the hill-climbing algorithm starts removing nodes and terminates at interval 4,200. As described in the evaluation setup in section 5.1, the hill-climbing algorithm restarts after an inactivity period of 20 intervals. At this point in time, adding nodes lets the node configuration converge towards the ideal configuration, yielding a higher productivity gain by the adaptation than in the initial run of the hill-climbing algorithm. Therefore, the accelerator adds nodes more quickly now. From this point on, the strategy repeatedly adds nodes to computation, overshooting the ideal configuration, and removes other nodes later again. The height of this zigzag behavior is determined by the tolerance factor, as discussed in section 4.2.2. The higher the tolerance factor, the bigger are the fluctuations in the number of nodes. On the other hand, to allow small productivity inaccuracies over time, the tolerance factor should not be too small. Regardless of the zigzag behavior, however, the number of active nodes increases on long-term and always stays close to the ideal node configuration.

Steady decrease of active nodes

In contrast to the previous scenario, the number of active nodes may also shrink constantly over time. Figure 5.8(a) shows the corresponding productivity graph. The adaptation to this scenario performs close to optimum, as drawn in figure 5.8(b). Whereas the static configuration of $N = 100$

nodes looses its performance during the runtime, our framework adapts the number of active nodes and keeps the performance close to ideal.

Similar to the scenario discussed before, nodes are not constantly removed from the system. Again, the zigzag of removing and adding nodes can be mitigated by shrinking the tolerance factor, with similar counter-effects as discussed before. In contrast to the scenario before, the node removal process terminates much faster than adding nodes in the previous scenario. This is because the total productivity is decreasing due to the given productivity search space, forcing the hill-climbing algorithm to terminate.

Exponential decrease of active nodes

In contrast to a linear decrease that was discussed before, the number of active nodes more likely decreases exponentially. Some real applications, such as the parallel ion recombinition in nonpolar liquids as discussed in [18], adhere to this behavior. Figure 5.9(a) shows how a possible productivity search space may look like.

The self-adaptation strategies covered this scenario very well. As plotted in figure 5.9(b), the strategy adapts close to the ideal number of nodes. In contrast, according to our special scenario, a static set of nodes behaves bad. If the number of active nodes leaves the ideal number of nodes, a high penalty highly influences the productivity. We observe that the bigger the difference between productivities of an ideal and of a non-ideal set of nodes, the worse will a static set of nodes perform.

Periodical changes of ideal configurations

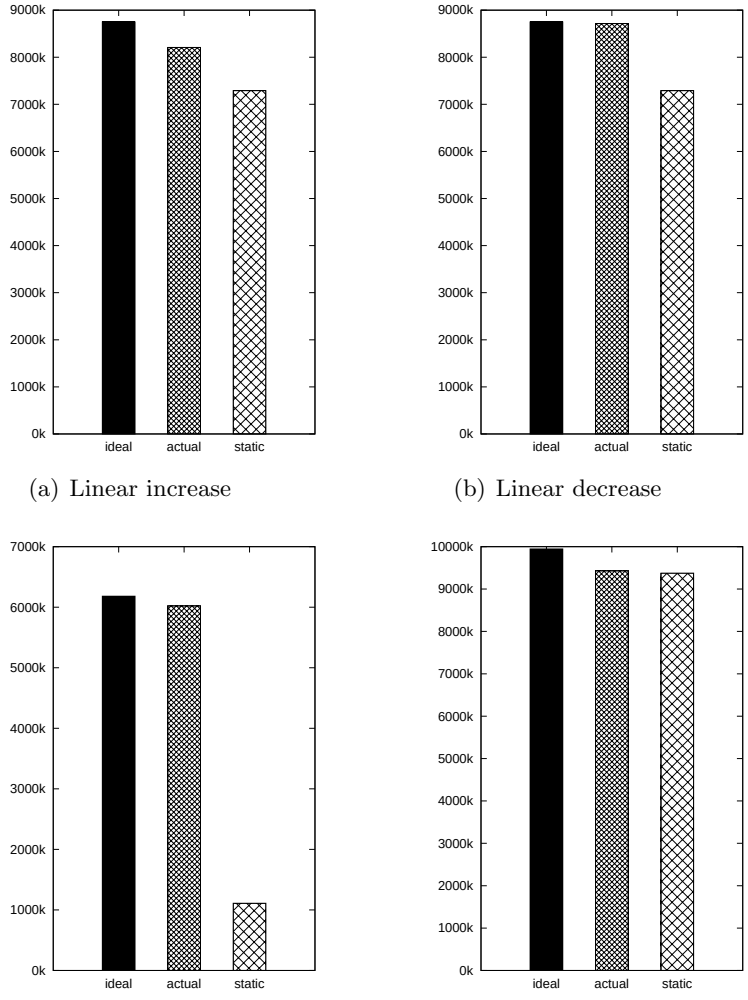
In some D&C applications, bursts of work are sent and need to be synchronized, before new work is spread. In such systems, the ideal set of active nodes might periodically change over time. Figure 5.10(a) shows a possible productivity search space of such a scenario. An adaptation mechanism needs to quickly react to changes, just before they are reversed again.

Figure 5.10(b) shows how the self-adaptation framework works with such graphs. It can be observed that the adaptation strategy is not always ideal in this scenario. In particular, in periods where the ideal productivity rapidly increases, the strategy misinterprets this and adds too many nodes to computation. This behavior is similar to the one we observed in section 5.2.2 and might be mitigated by speculative direction changes of the strategy. The overall performance of the strategy is however very good and has its strengths when the productivity space asks for many active nodes.

Overall evaluation of adhering to dynamics

As a final evaluation, we will compare the application run times of the four dynamic algorithm scenarios. Similarly as done before with the static algo-

rithms, figure 5.6 summarizes the productivity yielded in 7000 measurement intervals. In these scenarios, the self-adaptation works always better than a static set of nodes. The higher the static set of nodes deviates from an ideal set of nodes, the more our self-adaptation strategy outperforms the static set of nodes. In particular when the maximum productivity decreases over time (figures 5.6(b) and 5.6(c)), the adaptation strategies perform close to ideal. In the linear decrease scenario, a performance gain of 19.5% was achieved by the self-adaptive system. In the exponential decrease scenario, a gain of 442.3% (!) was measured. If the maximum productivity increases (figure 5.6(a)), a gain of 12.6% was possible. In situations, where the productivity search space changes fast and periodically (figure 5.6(d)), the adaptation yielded just a small benefit of 0.6% compared to a static set.



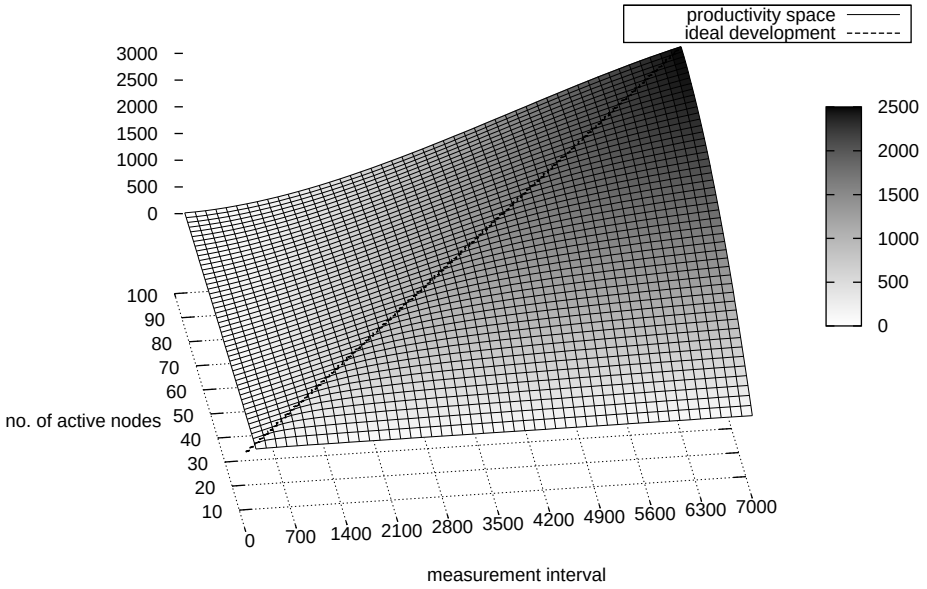
(a) Linear increase

(b) Linear decrease

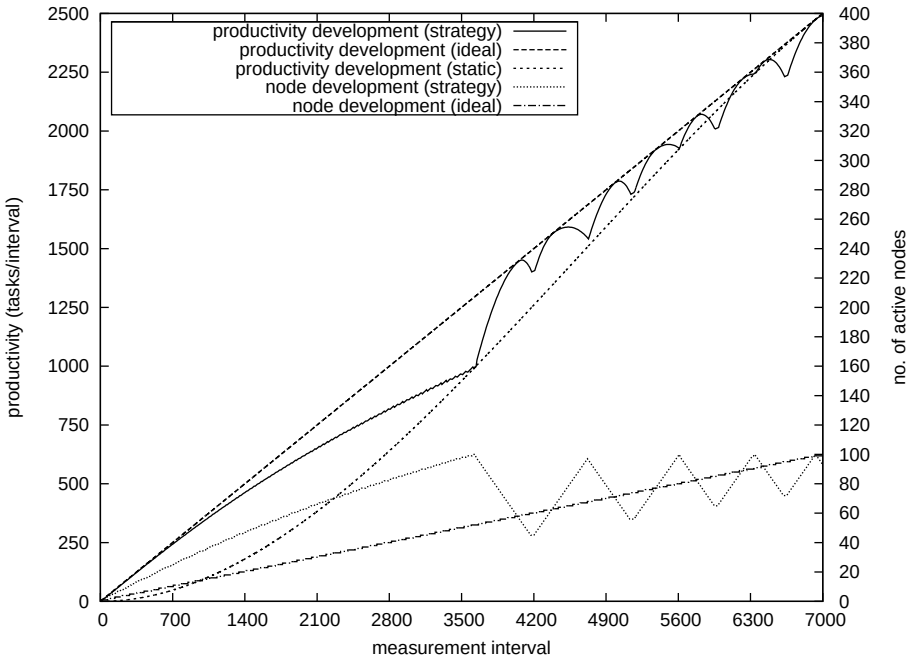
(c) Exponential decrease

(d) Sinus-shaped variations

Figure 5.6: Productivity sums during runs with dynamic algorithms.

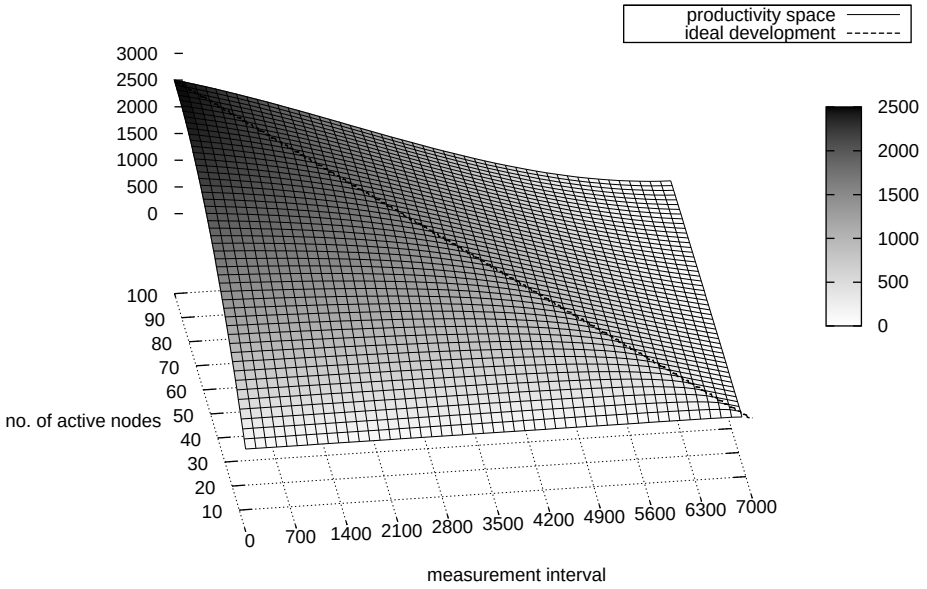


(a) Productivity search space dependent on time and number of active nodes

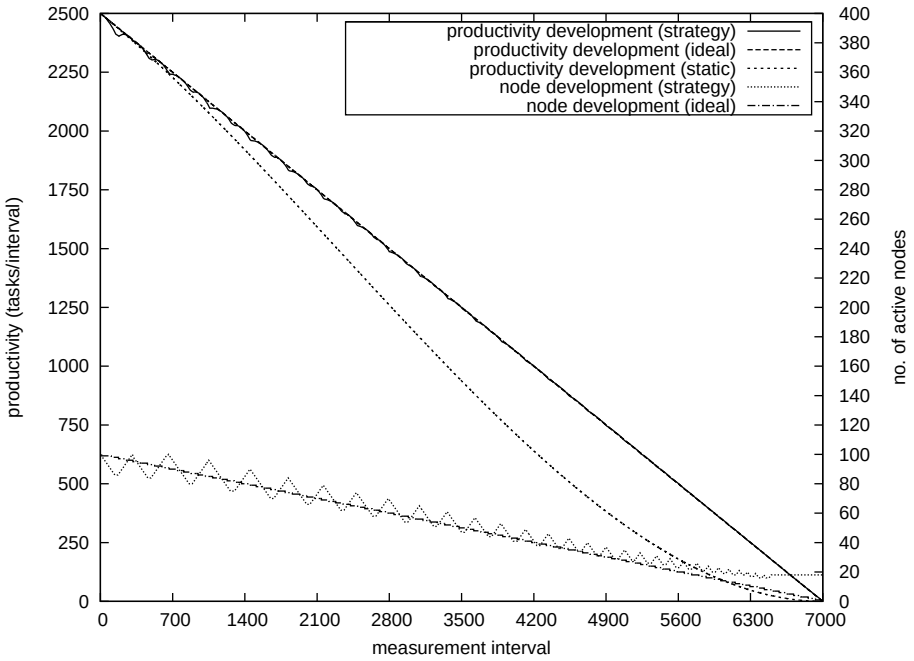


(b) Comparison between productivities of ideal, adapted and static configuration

Figure 5.7: The ideal number of nodes steadily increases

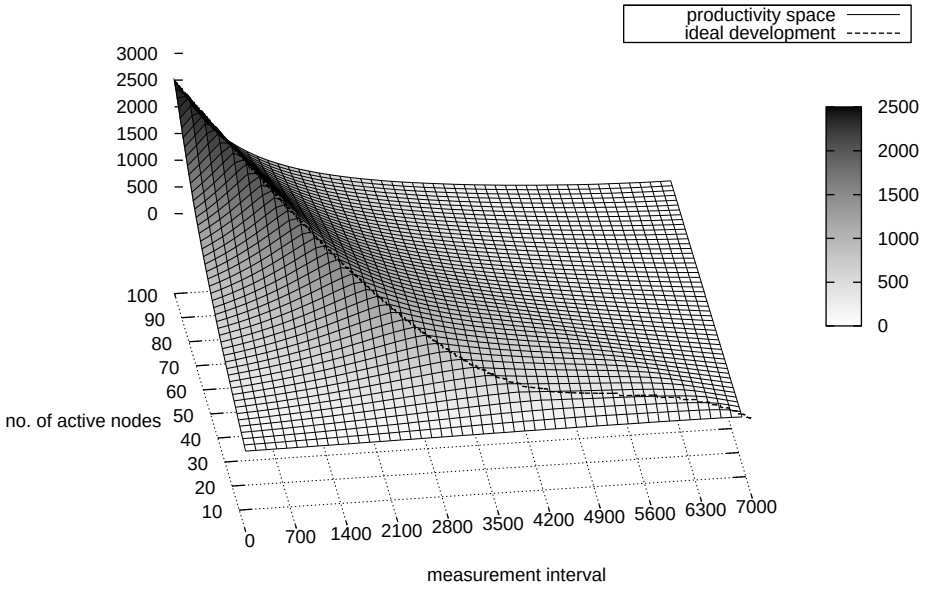


(a) Productivity search space dependent on time and number of active nodes

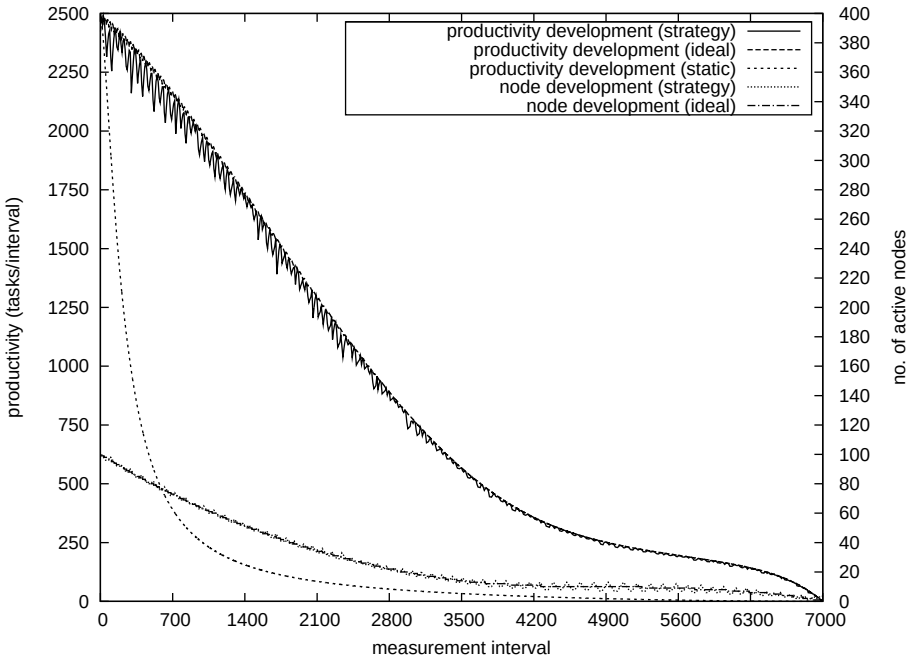


(b) Comparison between productivities of ideal, adapted and static configuration

Figure 5.8: The ideal number of nodes steadily decreases

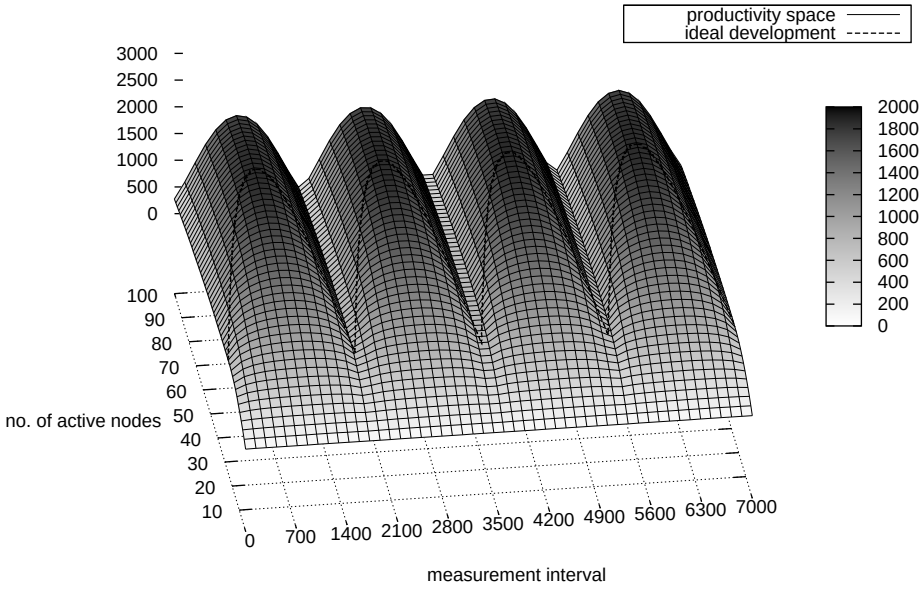


(a) Productivity search space dependent on time and number of active nodes

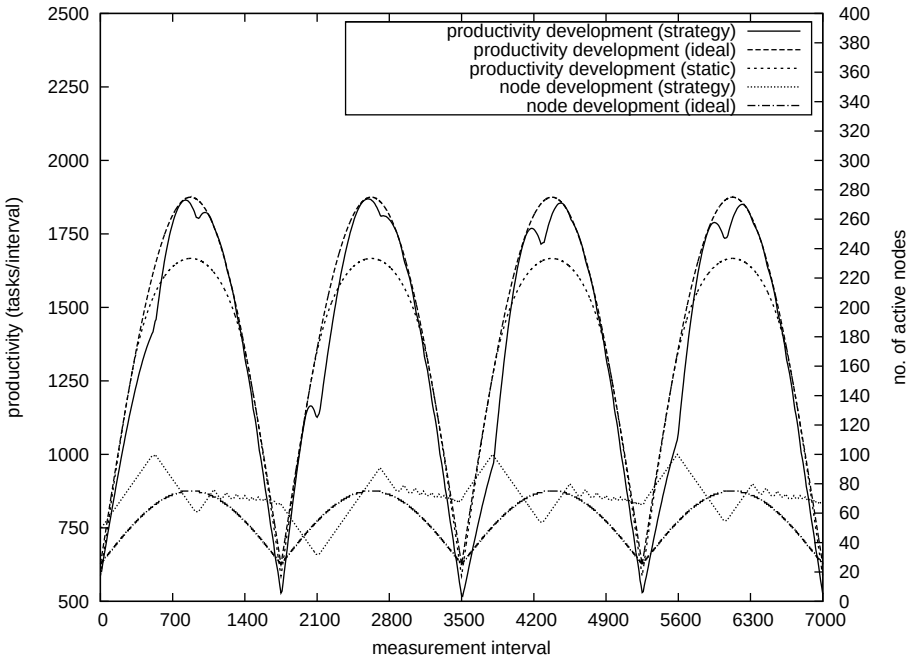


(b) Comparison between productivities of ideal, adapted and static configuration

Figure 5.9: The ideal number of nodes decreases exponentially



(a) Productivity search space dependent on time and number of active nodes

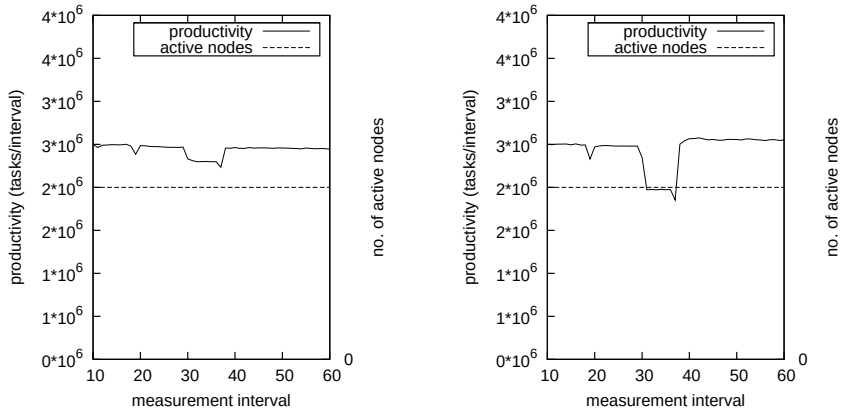


(b) Comparison between productivities of ideal, adapted and static configuration

Figure 5.10: The ideal number of nodes periodically changes

5.3 Reacting to dynamic network- or node loads

This section will evaluate the outlier replacement strategy against two scenarios. In both scenarios, 10 out of 16 available nodes are active in computation. After 28 measurement intervals, we increase the load of some of these active nodes. Figure 5.11 shows the strategy behavior to the scenarios.



(a) Single outlier (b) Three outliers

Figure 5.11: Evaluation of the outlier replacement strategy

In case of figure 5.11(a), we added a significant load to a single node, which has an immediate effect on the total productivity. Although the node has a high load already at the first test of the outlier replacement strategy at interval 30, the strategy does not replace the node due to the aggregated block statistics. First in interval 35 the node is replaced by a better node. Shortly after node replacement, the total productivity slightly drops, because the leaving node needs to send its work away to the remaining nodes. After three interval, the total productivity increases again.

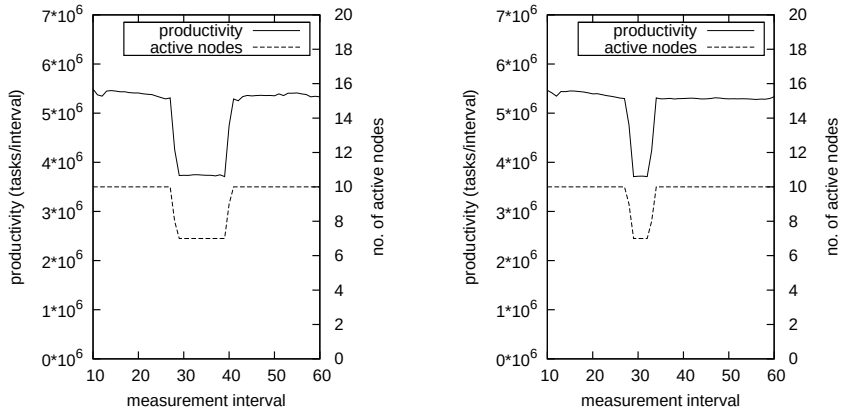
Figure 5.11(b) shows a similar scenario. However, this time significant load is added to three nodes at once. Again, the replacement is deferred for an entire statistics block. Similar to the first scenario, the three poorly performing nodes are replaced. This time, however, the total productivity after adaptation even exceeds the productivity before the load was added. Apparently, three slightly slower nodes were replaced by faster nodes.

To summarize, the two scenarios showed that the outlier replacement strategy is able to deal with nodes that have a poor performance.

5.4 Repairing network- or node failures

In this section, we will evaluate the failure watchguard strategy, as discussed in section 4.4. We developed two test scenarios for the evaluation. In both

scenarios, 10 out of 16 available nodes actively participate in a distributed computation. Starting after 28 measurement intervals however, three of the active nodes fail sending statistics. In the first scenario, the node failures are permanent. In the second scenario, the nodes are only temporarily inactive for 5 measurement intervals.



(a) Permanent failure (b) Temporal failure

Figure 5.12: Behavior of the failure watchguard when nodes fail.

Figure 5.12 shows how the self-adaptation framework reacts to node failures. Figure 5.12(a) shows that the failure watchguard replaces a single failed nodes after a tolerance period of 10 intervals. The total productivity drops after the nodes fail and stabilizes again after the replacement of the failed nodes. Figure 5.12(b), on the other hand, shows that the failure watchguard takes into account its tolerance period. Although the nodes seem to be failed during 5 consecutive intervals, the watchguard does not replace them. After 5 intervals, the nodes become active again without interference of the self-adaptation strategy.

5.5 Overall evaluation

As a final evaluation, we evaluate a scenario that requires a collaboration of all developed strategies to find the optimal configuration. We integrated a dynamic productivity search space as well as events such as node failures to our evaluation algorithm, as discussed in section 5.1. Figure 5.13 shows how our framework adapted the node configuration in a 35 minutes lasting program execution of this algorithm. The dashed lines show the theoretical ideal behavior, whereas the solid lines depict the actual performance we achieved. The x-axis shows the measurement intervals of 1 second length.

The scenario starts with 5 active nodes, whereas 18 nodes is the ideal configuration. The hill-climbing algorithm manages to converge close to the

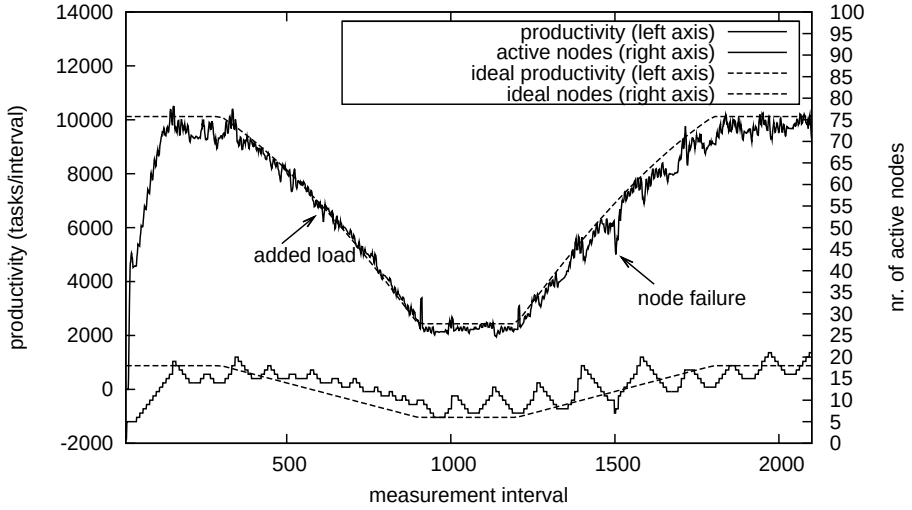


Figure 5.13: Overall evaluation of combined self-adaptation scenarios

optimal configuration after 103 seconds. Starting after 300 seconds, we artificially change the ideal number of nodes and let it slowly drop from 18 to 6 nodes. During this drop in performance, the hill-climbing algorithm adapted the number of active nodes towards a smaller set. Although constantly a few more nodes than ideal were active during this time, the productivity was close to ideal. In our evaluation setup, too few nodes have a higher negative effect on the productivity than having too many active nodes.

After 600 seconds, we added some additional load to three active nodes. As the small dip in productivity indicates, the poorly performing nodes were replaced by the framework immediately after 2 measurement blocks. Then, the number of ideal nodes was increased from 6 to 18 nodes again between the time intervals 1200–1800. The hill-climbing algorithm adhered to these algorithm dynamics. In between, at interval 1500, we purposely let three nodes fail. Tolerating temporary failure, the failure watchguard replaced these nodes in time interval 1511. After 30 minutes were passed, the strategies adapted the system close to the ideal configuration of 18 nodes.

The overall evaluation shows that our framework is able to find a well-performing configuration of nodes in many different scenarios. First, the strategies were able to adapt the initial number of active nodes. Moreover, the strategies managed to replace failed nodes or nodes with poor performance. In addition, the strategies handled situations with a dynamic ideal configuration. Regardless of the fact that the number of active nodes varies from the ideal configuration, the achieved productivity is close to ideal. Finally, this last evaluation has shown that all strategies work well together, although they interfere with each other. We will discuss possible improvements to the strategies in the next chapter.

Chapter 6

Future work

This chapter discusses possible enhancements of our self-adaptation framework that we left for future work.

6.1 Assessment of offline nodes

The self-adaptation strategies that we discussed in this work all use a probabilistic approach to pick a new node that is added to computation. Beforehand, nothing can be said about this node, and it can only be assessed after adding it. If node qualities vary, on average only a node with medium qualities is added to computation. Thus the more heterogeneous the nodes are, the more important it is to find the optimum node to pick. In this section, we will discuss two approaches that might help choosing the ideal nodes to expand the set of calculating nodes.

One approach is based on measurements done by the operating systems of each node. Possible indications of performance are for example CPU power, number of CPU cores, CPU cache size, available memory, available network bandwidth, network latency, etc. A mechanism that can measure and weigh these metrics is presumable able to tell how well a node fits into a set of computing nodes. In addition, OS based measurements are lightweight and do not cause much overhead. Unfortunately, there are some disadvantages that prevent an easy use of this technique. It depends on the operating system and on the grid environment if and how this data can be obtained. Moreover, it deviates from algorithm to algorithm which of the performance data is crucial for it. A computationally expensive task might favor nodes with high CPU power, whereas other algorithms require a generous memory size. Thus, this approach also requires the algorithm designer to specify the characteristics of a task.

Another approach lets each node measure its performance while computing a subtask of the actual problem. Each node can test how good it fits to a parallel computation by running the same parallel algorithm locally. The

yielded performance gives an indication how suitable a node is for addition. As long as defining a reasonable subtask size is automated, this approach is entirely independent from the algorithm and the underlying platform. On the other hand, it does not provide very reliable performance data. A local computation does not incorporate parallel overhead, such as connection bandwidth and latency. In addition, this technique requires each available node to run a small benchmark first. During its runtime, a node is reserved by the benchmark, causing additional overhead or costs to the system.

Finally, we observe that there is a trade-off to make. On the one hand, adding the optimum node will increase the performance faster and eventually higher. On the other hand, the presented mechanisms either require additional overhead, are inaccurate or are platform dependent.

6.2 Hill-climbing strategy improvements

As the evaluation has shown, there are potential optimizations to the hill-climbing strategy described in section 4.2.

First, the accelerator feature behaved very aggressively when adding nodes and too conservative when removing nodes. A possible improvement is finding an β that performs better in the given scenarios. Moreover, the calculation of the accelerator adaptation magnitude, as given in equation 4.2, assumes a homogeneous environment. A faster node however will have a bigger impact on productivity than adding a relatively slow node. To incorporate this heterogeneity it may be reasonable to take into account the speed of an added node when assessing the improvement the adaptation brought.

In general, the hill-climbing strategy relies on the productivity difference before and after an adaptation took place. This may raise problems in certain scenarios, as the evaluation has shown. Having algorithms with dynamic parallelization requirements, for example, force the strategy to compare current statistics with outdated measures. This situation becomes problematic when the strategy assesses a node removal as negative due to a decreasing overall performance, whereas decreasing the number of nodes would be the right option. A similar situation occurs with an increasing overall performance, if the strategy keeps on adding nodes although it exceeds the ideal number of active nodes already. These problems can possibly be handled by introducing direction switches to the hill-climbing algorithm, although the productivity is still increasing.

In addition, depending on the way parallel overhead emerges during a computation, the measured productivity may not be stable. Figure 6.1 gives an example of such a development that compares a normalized productivity development with a productivity search space that undergoes oscillations. This scenario appears in settings, where the parallel overhead is mainly influ-

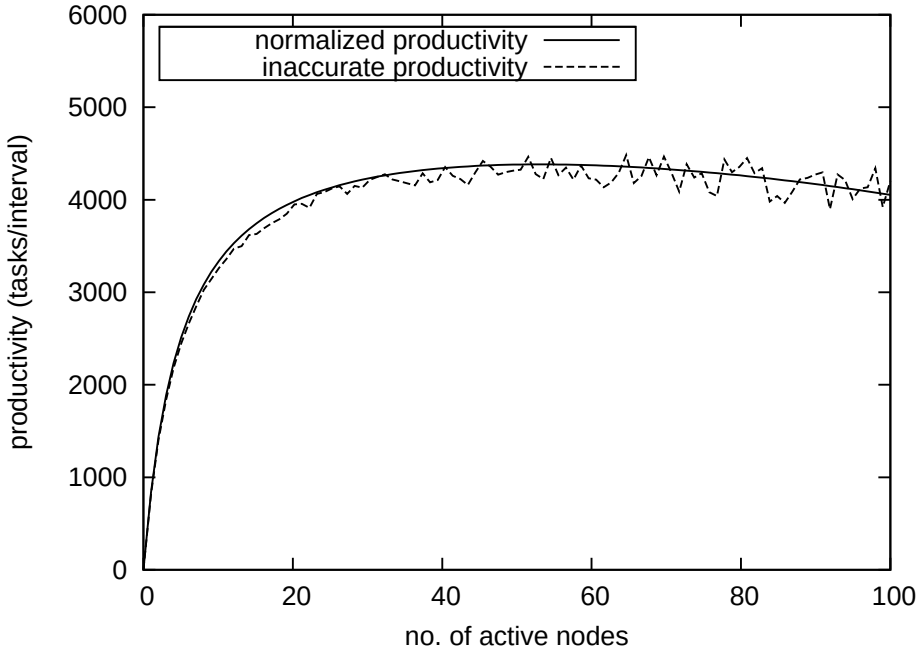


Figure 6.1: Inaccurate productivity development

enced by non-deterministic events, such as work steal requests, that cause a high overhead. In this example, the more nodes participate in computation, the higher are the fluctuations in the steal request behavior of the nodes, causing non-stable productivity measures. A possible approach to overcome this issue is taking into account other measures than the productivity. For example, the total efficiency or the number of steal requests can also be integrated into the decision making. The approaches mentioned above and other strategy improvements will be the subject of future work.

6.3 Analyzing intra-cluster links

Parallel overhead is considerably higher in heterogeneous environments. Multiple clusters are, compared to a local cluster, most likely connected via network links with higher latencies and a lower bandwidth. Analyzing the intra-cluster links allows for better adaptation strategies that take into account communication characteristics.

A possible way of analyzing communication characteristics is measuring the intra-cluster latencies and bandwidth capacities. It is possible, for example, to measure how long it takes to contact other nodes, or to transfer data to them. Based on this measurements, aggregated statistics can tell performance data of each node and cluster. A possible drawback of this

approach is the dynamics of the measurements. As an extreme example, consider a) two clusters with 5 nodes each and b) two clusters having 1 and 9 nodes, respectively. The usual behavior of D&C systems requires nodes to exchange tasks and results. Obviously, in a) both clusters have a high chance of local communication, whereas in b) the single node cluster requires only intra-cluster communication. In addition, the available bandwidth of each cluster is most likely shared by all nodes. On the one hand, measuring the network performance introduces some overhead. Yet, the analysis can be integrated to self-adaptation strategies that consider this data for their decision finding.

6.4 Machine learning techniques

All self-adaptation strategies that we discussed applied some heuristics to find a well-performing set of computing nodes. Machine learning techniques could be used to support the decision making. We will shortly summarize possible approaches of machine learning techniques that may be suitable for this task.

As we discussed in section 4.1, the possible combinations of node configurations form a huge search space. A good approach of exploring such spaces is Evolutionary Computing (EC). Similar to biology, in EC some individuals evolve to a good and mature population. In the case of self-adaptation, this could mean that a number of simultaneous runs with different configurations are changed over time by adding and removing nodes or mixing (crossing) them. EC is a good way of achieving a good, but not necessarily optimal, solution. There exist even EC variants that can deal with dynamic environments, such as node loads or algorithm dynamics. Unfortunately, it requires many evolution steps. This requirement brings lots of adaptation overhead and multiplies the number of nodes used for a strategy. This makes EC impracticable, when resources are limited.

Another machine learning technique is the k-nearest neighbors algorithm (KNN). KNN classifies objects in a multidimensional future space. This future space is based on history data, which in our case are represented by previous configurations and their performances. Given the history data, KNN classifies a future configuration by deriving information from the most similar (historical) neighbors. Due to possible dynamics such as temporary node loads, history data might be outdated over time. This should be taken into consideration in a KNN network. A disadvantage of KNN is that a considerable big set of history data (also called training data) is necessary for sensible results. It is therefore not able to deal with initial algorithm runs, and has its strengths after a longer overall parallel run time. Instead of replacing the basic strategies, KNN can possibly be used in combination with these.

Chapter 7

Related work

Adaptation of parallel systems was explored since the 1990s and lots of profound research has been done in this area. Our work distinguishes from the work done before in that it focuses on a specific programming model of parallel computing: divide and conquer applications. This chapter will summarize important related work that has previously be done.

In 1989, Eager et al. [7] investigated how far Amdahl's law should be respected when searching for the fastest algorithm execution time. The article describes that, given both a static algorithm behavior and comparable input, performance bounds can be given. The authors included an interesting discussion about trade-offs between *speedup* and *efficiency* that is inherent to software systems. Our discussion about proper performance metrics, as done in section 3.1, was motivated by this.

During the 1990s, research started to focus on measuring the performance of parallel systems [14, 15, 24]. The work done by Miller et al. [14] was a first approach to get insight to the execution behavior of distributed applications. Miller et al. used operating system hooks to count system procedures calls and recorded the performance data. Similarly, Reed et al. [15] and Yan [24] developed modified C and Fortran parsers that instrument source code to interactively measure execution times of code parts. The outcome of all approaches, dynamic performance metrics, can partly be used to dynamically adapt a parallel system. Self-adaptation however was just mentioned as future work and was not further discussed in these works. In addition to these approaches, the Globus Toolkit [8] was designed with the long-term goal to enable applications to adapt to heterogeneous and dynamically changing metacomputing environments. Although Foster and Kesselman integrated resource discovery and resource selection mechanisms, the actual self-adaptation was left for future work.

First approaches of real-time self-adaptive systems were discussed in 1997 by Ribler et al. [17]. Next to traditional approaches of measuring execution times of code parts, the *Autopilot* framework uses sensor threads to period-

ically monitor the application progress. It is up to the algorithm at hand which data is measured to calculate performance metrics. An example was given for I/O software, where offsets in files may be useful to indicate the execution progress. Based on the sensor data, so called *actuators* can adapt local variable values or call local functions of the parallel algorithm. Although the described solutions are highly dependent on the given algorithm, they describe the first real-time mechanism to adapt a parallel algorithm. Similarly, Liu [12] developed a self-adaptive scheme for data allocation in grid environments. Liu expands the hash algorithm commonly used for data distribution such that the distribution takes heterogeneity into account.

Rencuzogullari et al. [16] later developed an approach focusing on networks of workstations (NOWs). Due to the highly heterogeneous environment they focus on mitigating load-imbalance. At compile time, Rencuzogullari et al. analyze the program to capture the access patterns of the code and instrument the code with access information on critical program parts, such as program loops. The statistical feedback is used to partition the data according to the recorded patterns. Although this approach automatically adapts to any algorithm with distinctive patterns, it addresses load imbalances only. The work does not cover any of the self-adaptation scenarios that we described in section 1.3.

Instead of adapting the algorithm, Allen et al. [2] described resource selection methods and developed the *Cactus* framework. One of the main contribution of Allen et al. was the discussion on code migration, that is inherent for any resource selection mechanism. In addition, resource discovery mechanisms were developed that are a way to detect newly available resources. The underlying performance metrics rely on measures of flops, transfer rates or application specific data. So called *contracts* enforce the program to stay in certain boundaries. A comprehensive discussion of these policies and correlation between the performance metrics is not part of this work.

Huedo et al. [9] present the *GridWay* framework based on Globus [8]. GridWay adapts grid schedules based on performance and ranking data provided by an application. Huedo et al. require the programmer to enhance applications to measure the performance of a given algorithm and to rank machines that execute this algorithm. Another self-adaptive scheduler was developed by Lawson and Smirni [11]. This scheduler uses history workload data to predict future workload, but does not take into account algorithm dynamics. Recently, Yu and Shi [25] also presented an adaptive rescheduling strategy that incorporates history data to find the best execution schedule on a grid. All three approaches mentioned in this paragraph make use of self-adaptation techniques to find a good execution schedule. The approaches rely on performance feedback either from the grid or from the executed algorithms. Our work distinguishes from these approaches as it aims at minimizing the runtime of a single job, instead of finding the best schedule

for multiple jobs.

Vadhiyar and Dongarra [19] implemented a resource selection mechanism that uses the *Network Weather Service* (NWS, [22]) to predict performance data of nodes that participate in a parallel execution. Rescheduling algorithms are able to migrate work away from nodes that might perform bad in the future. Using mathematical models, the algorithms investigate whether or not a change is worthwhile on long term. This approach works well in environments where periodically changing performance data allows for reliable predictions. It is uncertain, however, if the NWS models accurate information for any grid.

The NWS was used by other researchers as well. Dail et al. [4] developed a platform- and application-dependent environment. To initiate an application-run, the user submits a machine list to the adaptation environment. A scheduler collects performance information (e.g. via the NWS) of these machines. A so called *mapper* maps application data or tasks to physical resources. Thus, the mapper provides an application specific performance model that tries to find the best configuration by having inside knowledge of the algorithm. For any algorithm, this performance model needs to be developed separately. Dail et al. implemented specific algorithm mappers to evaluate their approaches. Similarly, Kennedy et al. [10] added a possibility to describe performance models of a specific parallel application to the GrADS framework. Next to the manual information specified by programmers, however, Kennedy et al. make also use of trial executions to determine run times of important components, such as e.g. communication costs.

In the recent years, the urgent need for self-adaptation in grids was discovered. In 2004, Dini et al. [5] stated that self-adaptation mechanisms are necessary because of two reasons; similar to our observations, Dini et al. state that a) applications are multi-phased and thus dynamic and moreover b) the grid infrastructure is heterogeneous and also dynamic. One year later, an abstract scheme of self-adaptation was modeled by Aldinucci et al. [1], generalizing previous efforts. Similarly, Buisson et al. [3] present a generic framework that aims at optimizing an underlying application whenever its execution environment changes. Both abstract discussions, however, do not present actual implementations and evaluations. In 2009, self-adaptation was once more highlighted as means to automatically manage to the growing number of available resources by Dongarra [6]. Dongarra underlines that manual adaptation is too inefficient and automated mechanisms are required.

Recently, a self-adaptation mechanism for divide and conquer applications was presented by Wrzesinska [23]. Our work was inspired by the described mechanisms. As in our work, Wrzesinska describes a framework that is independent from the parallel algorithm and instruments resource selection as self-adaptation system. The approach of Wrzesinska however

significantly differs from our framework. First, as discussed in section 3.2, Wrzesinska bases self-adaptation strategies on performance metrics that can fail in certain scenarios. A second significant difference with the work done by Wrzesinska is the way of statistics gathering. The previous work made use of additional performance measures by running smaller problem sizes, thus introducing significant overhead to the parallel application. Our work limits the overhead by using statistics that are present anyway, i.e. the frequency at which tasks are computed. We further distinguish from the work by using more advanced adaptation strategies to optimize system productivity, rather than merely adding nodes as long as the efficiency is in given boundaries. Our work is moreover entirely independent from any specification by the algorithm designer and can run in any divide and conquer system.

As discussed in this chapter, measuring performance of parallel systems and adding self-adaptation to them was extensively studied in the past. This work contributes to this research by presenting a framework that a) is independent from the algorithm used, b) does not require any specifications by the application developer and c) is able to deal with common scenarios that ask for self-adaptive systems. To the best of our knowledge, so far no work has been done to extensively study self-adaptation strategies to find well-performing resource configurations in divide and conquer systems.

Chapter 8

Conclusion

Divide and conquer systems allow for measuring the performance of an algorithm execution during runtime. In this work, we introduced the term of productivity to measure this performance. We described a generic framework that is capable of gathering productivity statistics from nodes that participate in a distributed computation. The framework bases self-adaptation strategies on this data to find a (close-to) ideal configuration of nodes, using resource selection. We learned a few lessons during the development of the framework:

- The definition of productivity allows for intermediate performance measurements during an application run. This enables strategies to use performance measurements as feedback for the adaptation process.
- Although efficiency is a good indicator of the node utilization [23], we have shown that strategies aiming at maximizing the speedup may not solely rely on this measure for decision making.
- Self-adaptation strategies need to take algorithm dynamics into account and should reassess node configurations, since previous measurement results may be outdated.
- In certain scenarios with dynamic algorithms, both adding and removing a node can increase (or decrease, respectively) the overall performance. In future work, we will examine if *speculative direction changes* help the hill climbing algorithm to deal with these scenarios.

We integrated the framework into Satin and evaluated it against a range of common self-adaptation scenarios. This evaluation has shown that our approach is a well-working self-adaptation scheme. The developed framework is generic enough to be integrated into any parallel computing environment implementing the divide and conquer paradigm.

Summarizing, the basic achievements of our work are:

- The self-adaptation framework is able to quickly converge towards a well-performing configuration. An evaluation has shown that the framework can both deal with situations where too many or too few nodes are active.
- Algorithms with dynamic parallelization requirements are particularly supported by the developed self-adaptation strategies. In our evaluation, given such dynamic algorithms, the strategies always outperform an ideal static set of nodes.
- Nodes that have additional load, or entirely failed, are detected by our framework and replaced by well-performing nodes.
- The overhead that was introduced by the framework is marginal. We achieved this by using light-weight measurements and by our approach to minimize the number of wrong adaptation decisions.

Any divide and conquer system can be enhanced by integrating our framework. As long as the algorithm requirements of homogeneous and small leaf tasks are met, similar performances can be expected.

Appendix A

Satin example application

```
interface FibInterface extends ibis.satin.Spawnable {
    public long fib(int n);
}

final class Fib extends SatinObject implements FibInterface {
    public long fib(int n) {
        if (n < 2) { return n; }

        long x = fib(n - 1);
        long y = fib(n - 2);

        sync();
        return x + y;
    }

    public static void main(String[] args) {
        Fib f = new Fib();

        long result = f.fib(10);
        f.sync();

        System.out.println("result fib (10) = " + result);
    }
}
```

The code above shows an example Satin application, the calculation of Fibonacci numbers. Note that this program is not an efficient way to calculate Fibonacci numbers in parallel and is merely an educative example. The interface *FibInter* represents the marker interface, and the method $fib()$ in class *Fib* implements it. After spawning two children per each task, the invocations are synchronized to retrieve a final result.

Appendix B

Correlation of efficiency/productivity

If the total efficiency shall be kept within certain bounds, it is necessary to know ways how to increase and decrease the efficiency. Usually, adding nodes to computation increases the parallel overhead, and thus decreases the overall efficiency. Similarly, reducing the set of worker nodes reduces the parallel overhead and raises the efficiency. In addition, the productivity is usually higher if more nodes contribute to the distributed computation.

There are however certain scenarios, where these rules of thumb do not apply. Table B.1 on page 51 indicates four scenarios that show all possible correlations between total efficiency and total productivity when adding a node to computation. It shows if simultaneous increases and decreases in productivity and efficiency are possible in both homogeneous and heterogeneous environments. As described in table B.1, the first three correlations are possible. The fourth scenario, that after adding a node the efficiency increases and the productivity decreases, is impossible.

Proof. Let a node with positive speed be added to the system. Thus the total speed increases, i.e.:

$$total\ speed_{old} < total\ speed_{new} \tag{B.1}$$

Following the scenario, the total efficiency of the system increases:

$$total\ efficiency_{old} < total\ efficiency_{new} \tag{B.2}$$

From observations B.1 and B.2, as well as equation 3.5 follows that:

$$total\ speed_{old} * total\ efficiency_{old} < total\ speed_{new} * total\ efficiency_{new}$$
$$\Leftrightarrow$$
$$total\ productivity_{old} < total\ productivity_{new}$$

□

eff.	prod.	real?	reason
-	+	yes	Adding a node to computation usually raises the productivity, but an increased load balance overhead lowers the efficiency.
-	-	yes	In certain situations, the parallelization achieved by a set of nodes is supersaturated. Again, adding a node decreases the efficiency. If the introduced overhead is too high compared to the benefit of an extra node, the productivity decreases.
+	+	yes	In some situations, e.g. where positive cache effects or pruning benefits can be observed, adding nodes to the computation increases the efficiency. Parallel applications showing super linear speedup are an example of this scenario.
+	-	no	Adding a node always increases the global speed of the currently active nodes. If efficiency increases at the same time, then the productivity must necessarily also grow.

Table B.1: Possible productivity and efficiency correlations when adding nodes to the computation

Appendix C

Framework implementation description

This appendix describes details of the framework implementation. In particular, implementations of the self-adaptation server, client and environment adapter are described.

Figure C.1 gives a UML representation of the most important components of the self-adaptation server implementation. The main work is done by a singleton object of the class *GeneralSACollector*. This instance is responsible for storing collected statistics and information about active worker nodes. The statistics collector was modeled such that it can run in any parallel environment. To communicate with the parallel environment, the collector keeps a reference to an *SAEnvironmentAdapter* instance, that is responsible for communication with a specific parallel environment via SmartSockets [13].

All remaining three classes are responsible to keep track of self-adaptation clients. An object of *SAServer* spawns the *SAServerAcceptThread*, which in turn accepts client connections and hands them off to *SAServerThread* instances. Such an instance is in charge of reacting to commands issued by a client, such as statistic requests or node configuration modifications. Once the server was started, the user can also use the standard input interface to issue regular client commands.

The self-adaptation client implementation is structured as illustrated by the UML diagram in figure C.2. A singleton instance of *SAClient* is responsible for querying statistics from the server via the API. Any collected statistics is handed over to an instance of *StatsCollecter*, which administrates a complete history of *Statistic* instances. Each such instance represents the statistical data of a single measurement interval and has a list of nodes with associated performance metrics. After querying new statistics, the *SAClient* instance asks a specific implementation of the abstract class *SAStrategy* for self-adaptation steps. The class *MaximizeSpeedup* is a concrete implemen-

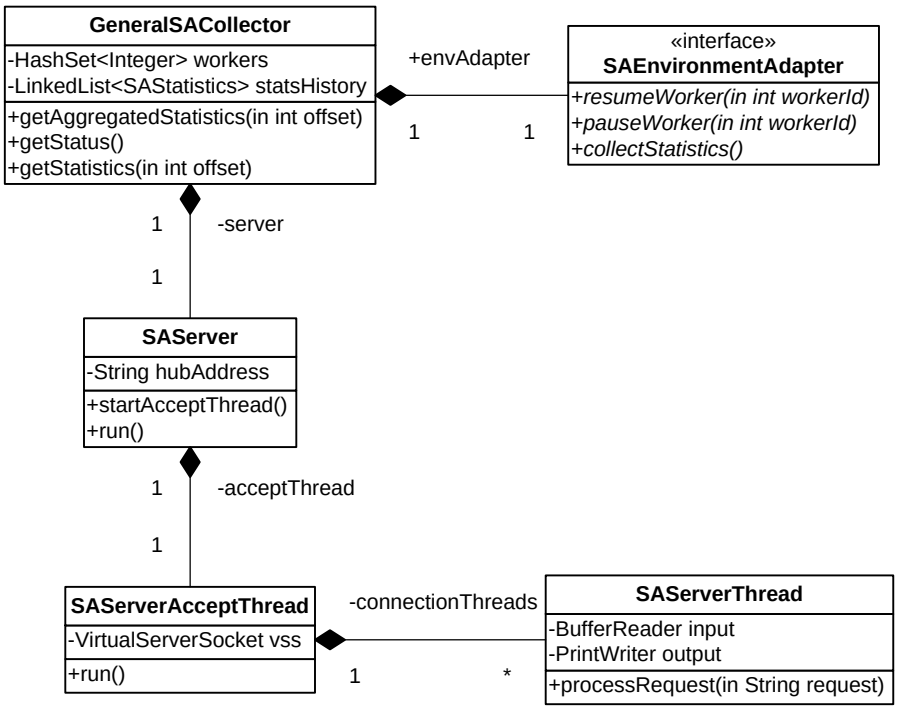


Figure C.1: UML graph of the statistics collector process

tation of the strategy and comprises all strategies that we discussed before.

A current state of the network is always kept in a *Network* singleton. The strategies implemented by the client will change this singleton if adaptation is required. The network singleton object follows a publish-subscribe mechanism. With every adaptation, all registered instances of the type *NetworkListener* are notified about the change. In our implementation, an instance of *SAClientConnection* subscribed for changes and forwards every modification as adaptation command to the self-adaptation server. The server, in turn, will process the requests and change the current node configuration in the parallel environment accordingly.

Figure C.3 illustrates the structure of a reference implementation of an environment adapter that we integrated into *Satin*. By definition, an environment adapter needs to implement the *SAEnvironmentAdapter* interface. This interface allows the collector to communicate with the parallel environment and stay general, such that no source modification is required when integrating the framework to a specific parallel environment. The adapter, in turn, needs a reference to the *GeneralSACollector* in order to forward statistic replies and update the collector with current node configuration information.

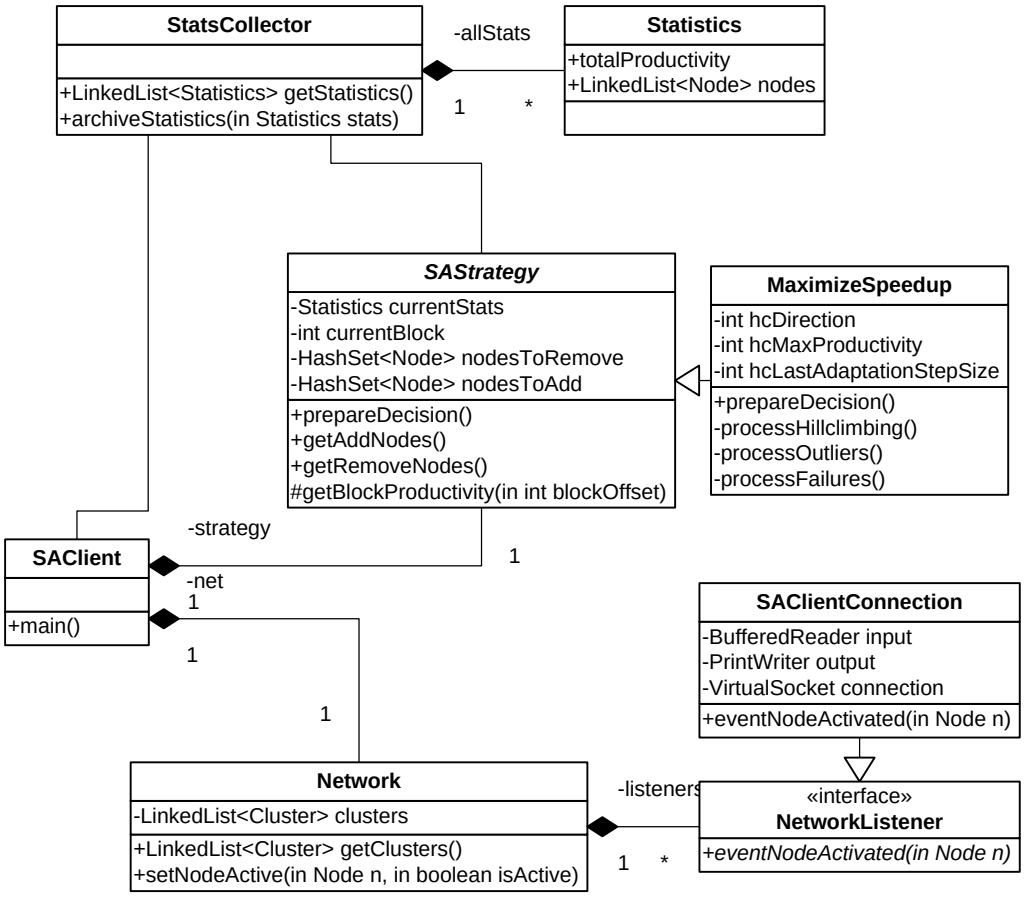


Figure C.2: UML graph of the self-adaptation client

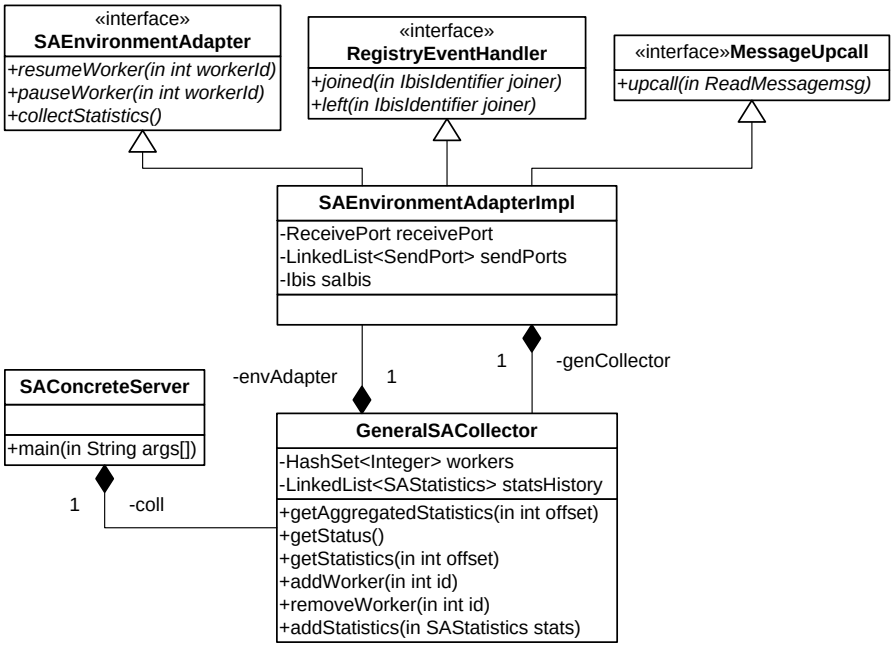


Figure C.3: UML graph of the environment adapter reference implementation

Appendix D

Client-server communication API

The server offers self-adaptation clients an interface to retrieve the collected statistics as well as issue adaptation commands via a well-defined API. Table D.1 gives an overview of the API that is used in our self-adaptation framework. The first three entries enable the client to retrieve collected statistics, or get a list of (in)active worker nodes. The remaining four commands can be used to control the set of running nodes and perform the actual adaptation.

command	description
STATS_ALL *offset*	return a list of statistics per interval and per node
STATUS	return a list of connected worker nodes
ACTIVE	return a list of connected and active worker nodes
PAUSE_NODE *nodeId*	pause computation of worker node
RESUME_NODE *nodeId*	resume computation of worker node
ADD_NODE	add new worker node to computation
REMOVE_NODE *nodeId*	remove worker node entirely from computation

Table D.1: API definition for communication between SA server and client

Bibliography

[1] M. Aldinucci, F. Andre, J. Buisson, S. Campa, M. Coppola, M. Danelutto, and C. Zoccolo. Parallel Program/Component Adaptivity Management. In *Proceedings of Parallel Computing 2005*, 2005.

[2] Gabrielle Allen, David Angulo, Ian Foster, Gerd Lanfermann, Chuang Liu, Thomas Radke, Ed Seidel, and John Shalf. The Cactus Worm: Experiments with Dynamic Resource Discovery and Allocation in a Grid Environment. *International Journal of High Performance Computing Applications*, 15:2001, 2001.

[3] Jeremy Buisson, Francoise Andre, and Jean-Louis Pazat. Dynamic adaptation for Grid computing. *Lecture Notes in Computer Science*, 3470:538–547, June 2005.

[4] H. Dail, H. Casanova, and F. Berman. A Decoupled Scheduling Approach for the GrADS Program Development Environment. In *Proceedings of Supercomputing, ACM/IEEE 2002 Conference*, 2002.

[5] Petre Dini, Wolfgang Gentzsch, Mark Potts, Alexander Clemm, Mazin Yousif, and Andreas Polze. Internet, GRID, Self-Adaptability and Beyond: Are We Ready? In *15th International Workshop on Database and Expert Systems Applications*, volume 0, pages 782–788, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[6] Jack Dongarra. An Overview of High Performance Computing and Challenges for the Future. In *Keynote of International Symposium on High Performance Distributed Computing (HPDC 2009)*, 2009.

[7] D.L. Eager, J. Zahorjan, and E.D. Lazowska. Speedup Versus Efficiency in Parallel Systems. *IEEE Transactions on Computers*, 38(3):408–423, 1989. ISSN 0018-9340.

[8] Ian Foster and Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 11: 115–128, 1996.

[9] E. Huedo, R.S. Montero, and I.M. Llorente. The GridWay Framework for Adaptive Scheduling and Execution on Grids. *Scalable Computing - Practice and Experience*, 6:Nova Science, 2005.

[10] Ken Kennedy, Mark Mazina, John M. Mellor-Crummey, Keith D. Cooper, Linda Torczon, Francine Berman, Andrew A. Chien, Holly Dail, Otto Sievert, Dave Angulo, Ian T. Foster, Ruth A. Aydt, Daniel A. Reed, Dennis Gannon, S. Lennart Johnsson, Carl Kesselman, Jack Dongarra, Sathish S. Vadhiyar, and Richard Wolski. Toward a Framework for Preparing and Executing Adaptive Grid Programs. In *IPDPS*, 2002.

[11] Barry Lawson and Evgenia Smirni. Self-Adaptive Scheduler Parameterization via Online Simulation. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*, page 29.1, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2312-9.

[12] Zhaobin Liu. Saha: A Self-Adaptive Hierarchical Allocation Strategy for Heterogeneous Grid Environments. *Semantics, Knowledge and Grid, International Conference on*, 0:14, 2006.

[13] Jason Maassen and Henri E. Bal. Smartsockets: Solving the Connectivity Problems in Grid Computing. In *Proceedings of The 16th IEEE International Symposium on High-Performance Distributed Computing (HPDC)*, Monterey, CA, USA, June 2007.

[14] Barton P. Miller, Morgan Clark, Jeff Hollingsworth, Steven Kierstead, Sek-See Lim, and Timothy Torzewski. IPS-2: The Second Generation of a Parallel Program Measurement System. *IEEE Transactions on Parallel and Distributed Systems*, 1:206–217, 1990.

[15] Daniel A. Reed, Ruth A. Aydt, Roger J. Noe, Phillip C. Roth, Keith A. Shields, Bradley W. Schwartz, and Luis F. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proceedings of the Scalable parallel libraries conference*, pages 104–113. IEEE Computer Society, 1993.

[16] Umit Rencuzogullari, Sandhya Dwarkadas, and Hya Dwarkadas. Dynamic Adaptation to Available Resources for Parallel Computing in an Autonomous Network of Workstations. In *Proc. of the 8th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '01)*, pages 72–81, 2001.

[17] Randy L. Ribler, Huseyin Simitci, and Daniel A. Reed. The Autopilot Performance-Directed Adaptive Control System. In *Future Generation Computer Systems*, pages 175–187, 1997.

[18] Frank J. Seinstra, Henri E. Bal, and Hans J.W. Spoelder. Parallel Simulation of Ion Recombination in Nonpolar Liquids. *Future Generation Computer Systems*, 13(4-5)(4-5):261–268, 1998. ISSN 0167-739X.

[19] Sathish S. Vadhiyar and Jack J. Dongarra. Self Adaptivity in Grid Computing. *Concurrency & Computation: Practice & Experience*, 17: 235–257, 2005.

[20] Rob V. van Nieuwpoort, Jason Maassen, Rutger Hofman, Thilo Kielmann, and Henri E. Bal. Satin: Simple and Efficient Java-based Grid Programming. In *AGridM Workshop on Adaptive Grid Middleware*, New Orleans, Louisiana, USA, September 2003.

[21] Rob V. van Nieuwpoort, Jason Maassen, Gosia Wrzesinska, Rutger Hofman, Ceriel Jacobs, Thilo Kielmann, and Henri E. Bal. Ibis: a Flexible and Efficient Java based Grid Programming Environment. *Concurrency and Computation: Practice and Experience*, 17(7-8):1079–1107, June 2005.

[22] Rich Wolski, Neil T. Spring, and Jim Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768, 1999.

[23] Gosia Wrzesinska. *Handling complexity and change in grid computing*. PhD thesis, Vrije Universiteit, Amsterdam, The Netherlands, May 2007.

[24] Jerry C. Yan. Performance Tuning with AIMS - An Automated Instrumentation and Monitoring System for Multicomputers. In *HICSS*, volume System Sciences, 1994. Vol.II: Software Technology, pages 625–633, 1994.

[25] Zhifeng Yu and Weisong Shi. An Adaptive Rescheduling Strategy for Grid Workflow Applications. *Parallel and Distributed Processing Symposium, International*, 0:115, 2007.