Pollarder: An Architecture Concept for Self-adapting Parallel Applications in Computational Science

Andreas Schäfer and Dietmar Fey

Lehrstuhl für Rechnerarchitektur und -kommunikation, Institut für Informatik,
Friedrich-Schiller-Universität, 07737 Jena, Germany
{gentryx,fey}@cs.uni-jena.de

Abstract. Utilizing grid computing resources has become crucial to advances in today's computational science and engineering. To sustain efficiency, applications have to adapt to changing execution environments. Suitable implementations require huge efforts in terms of time and personnel. In this paper we describe the design of the Pollarder framework, a work in progress which offers a new approach to grid application componentization. It is based on a number of specialized design patterns to improve code reusability and flexibility. An adaptation layer handles environment discovery and is able to construct self-adapting applications from a user supplied library of components. We provide first experiences gathered with a prototype implementation.

Keywords: grid computing, scientific computing, self-adaptation.

1 Introduction

The last decades have seen a constantly growing demand for computing resources, scientists and engineers embrace the new opportunities offered by current hardware. Frequently they develop new simulation software on, for instance, their notebook, test it with smaller datasets on workstations and employ large-scale clusters and multi-cluster to analyze real world data.

Such grid applications have to cope with numerous new problems due to the increased complexity and variety of their environment. An exemplary setup is shown in Fig. 1. Suitable designs usually represent the outcome of a multi-disciplinary effort. It is not always possible to invest so much time into the implementation of a new grid application. Therefore it is imperative to simplify the development of adaptable, highly portable scientific applications. This need gave rise to several frameworks, most of them focusing on data decomposition. While this allows comprehensive support for developers, it also limits a framework's applicability to the data structures it supports.

As an alternative we propose a component centric approach. In our framework, software is developed in terms of small self-contained components. Components that offer identical services – but for different environments – share a common interface. An adaptation layer detects the environment (e.g. a cluster

M. Bubak et al. (Eds.): ICCS 2008, Part I, LNCS 5101, pp. 174–183, 2008.

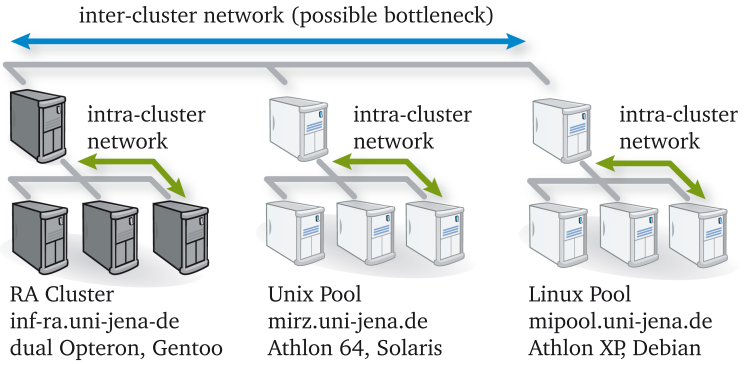

Fig. 1. Small compute grid. This is a multi-cluster setup as we use it at University Jena. An essential difference to a homogeneous setup is the inter-cluster network: all cluster nodes have to share the head nodes' bandwidth for cross-cluster communication.

made of dual-core processors or a multi-cluster) and acts as a factory to build the application from those components that match the environment best.

The rest of the paper is organized as follows. Section 2 gives a brief overview of the current state of the art. Section 3 outlines our design and Sections 4 and 5 describe the two most important employed design patterns in depth. In Section 6 we outline how Pollarder performs its environment discovery and the first results of our prototype are captured in Section 7.

2 Related Work

Parallelization and adaptation have for long been subject of research and consequently there is a huge variety of different solutions. This section outlines two illustrative examples. Cactus is a modular framework for multi-dimensional physical simulations. It consists of a library of components – named *thorns* – and a connecting layer which is called *flesh*, hence the name of the framework. Since all data has to traverse the flesh and the flesh can only pass on multi-dimensional arrays, the interface for the thorns is limited [1]. The user can manually adapt Cactus applications by selecting the thorns to be used. Runtime adaptation is a burden of the *driver* thorns, which are responsible for parallelization (e.g. *Carpet* [1] for adaptive mesh refinement). While Cactus is targeted for a specific application domain and offers a rigid structure therefore, the Common Component Architecture (CCA) is based on a flexible component model [2]. The use of the *provides/uses* design pattern and the Scientific Interface Definition Language allow cross-language component reuse and transparent remote objects via proxy generation. ProActiv [3] is a Java based framework for component construction and deployment. Components can be hierarchically constructed from other components. Similarly to CCA it relies on remote method invocation, as opposed to message passing. It provides solutions for code mobility, security and parallel method invocations.

3 Pollarder Overview

The goal of Pollarder is to automatically assemble a Grid application from user provided components on the basis of an environment discovery. We provide design patterns to aid the componentization of an application.

As hinted in the introduction, the Pollarder framework's most important feature is the adaptation layer. The layer consists of two parts: an environment discovery component and a factory which uses the environment discovery to select appropriate components for a component library. Assume a user wants to perform a parallel computation for a given model, e.g. to numerically integrate a given function. Instead of selecting the appropriate solver by directly (for example one using the Message Passing Interface, MPI) he registers his solvers at Pollarder's component library and requests an instance at Pollarder's factory. The factory uses a scoring function to determine how well a component fits into a given environment. This function can be chosen separately for each component and could even be user-defined to capture performance models, but up to date we did not need such a complex scoring function. For more information on how Pollarder integrates into the source code, see Section 7.

This technique of decoupling is also known as dependency injection (DI), an variation of inversion of control [4]. The original goal of DI was to create easily testable objects by injecting mock objects whilst running in a test environment. We abuse this method to achieve self-adaptation. The advantage of this decoupling is that, if multiple solvers are available, the same program can run without modification in each environment they are targeted at.

4 Model-Parallelization-Balancer

Despite the huge variety of parallel applications, analysis of the parallel algorithm structure design space exhibits a number of recurring patterns. Therefore it is sensible to separate a specific model from its generic parallelization [5]. An efficient parallelization has to maintain a smooth load equality among participating nodes, which suggests to use a dedicated balancer to tune the parallelization's parameters to care for efficient execution. The approach we have distilled from this is that grid applications should be designed in three pieces: *Model*, *Parallelization*, and *Balancer*. According to our experiences, this partition results in a separation of concerns and increased code reuse.

Model. The Model is the core of the application and defines the problem to be solved (e.g. a cellular automaton (CA) to simulate molten metal, see Section 7).

Parallelization. The Parallelization encapsulates a generic parallel algorithm for a certain class of parallel computer (e.g. CA for MPI clusters). Although a generic component is typically harder to write than a specialized one, our experience is that it is a lesser effort in the long run, because it can be repeatedly reused and improved.

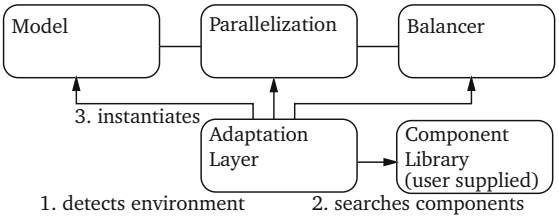

Fig. 2. The Model-Parallelization-Balancer pattern. The adaptation layer detects the environment it is running in. Given a Model, it will select an interface compatible Parallelization and Balancer most suitable for the environment.

Balancer. In general, to optimize a parallelization's performance, several parameters have to be set. This is not limited to the load balancing itself, but also includes – just to mention some examples – load balancing frequency and (for finite difference codes and CA simulations) ghost zone width [6]. Setting these parameters could be left as a user burden, but this would not only be tedious and error-prone, it would even be impossible for parameters that have to change during runtime. Optimizing these parameters is the task of the Balancer.

5 Hierarchical Adaptive Parallelization

A challenge often encountered in compute grids is the heterogeneity of the participating systems. While it is often relatively simple to create a parallelization for a homogeneous system, a grid application has to take into account specific details like different network characteristics or processor performances simultaneously. This can render an application overly complex. The problem has become even more prominent with the advance of multi-core nodes in message passing clusters. It would be much simpler, if the programmer could deal with one system and its properties at a time and ignore the remaining ones meanwhile.

The Hierarchical Adaptive Parallelization pattern depicted in Fig. 3 tries to achieve this by breaking down a parallelization into smaller sub-parallelizations, each of which is responsible for a single subsystem. Using the MPB pattern, the sub-parallelizations use their own balancers to adapt to their sub-system (not shown in the figure). By choosing suitable components for each subsystem, the application can harness each system optimally. The sub-parallelizations are stacked in a hierarchy to reflect the grid's structure. They perform synchronization with their direct neighbors in the tree (parents and children) and possibly with those nodes that share the same parent. If a sub-parallelization is not a leaf node, it will delegate its load the children.

A problem caused by the HAP pattern is that such a tree of object is difficult to set up as each node has to decide, which components it needs. As Fig. 3 shows, this can result in multiple components residing on one node, e.g. in a multi-cluster like the one in Fig. 1, a cluster's head node will end up hosting

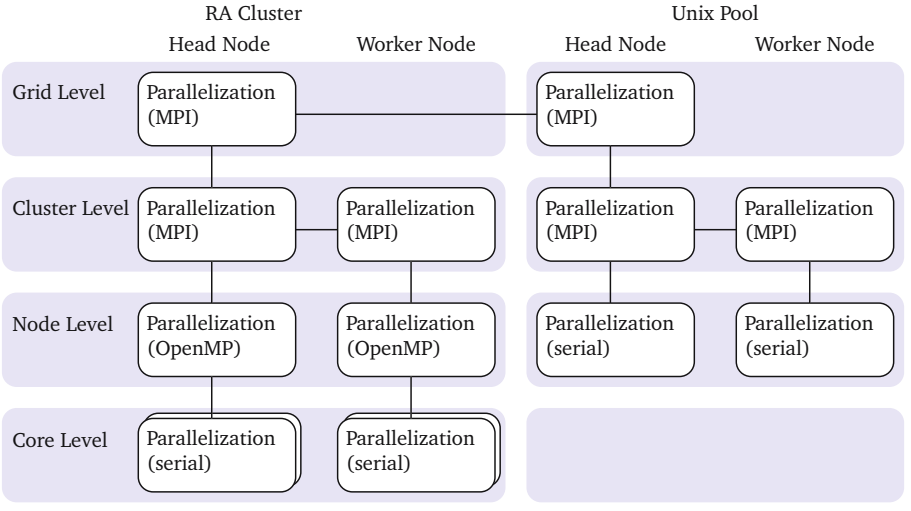

Fig. 3. Exemplary stacking of sub-parallelizations according to the HAP pattern as it would be created for our numerical integration demo (see Section 7)

at least two sub-parallelizations: one to synchronize with other clusters and another one to synchronize with its peers in the same cluster. It is their task to aggregate external communication, thus shielding the worker nodes from external synchronization. This leads to fewer high-latency cross-cluster connections and also reduces the number of nodes participating in collective MPI operations. As the employed RA cluster's nodes are dual CPU machines, they have an additional layer below to care for shared memory parallelization via OpenMP, thereby cutting the necessary number of MPI processes on the RA Cluster in half. With Pollarder this is not a problem because the factory inside the adaptation layer takes over object instantiation and hide the complexity from the user.

6 Environment Discovery

By the term environment discovery we refer to automatically detecting the system's static properties that are relevant for application adaptation. During our experiments we found three properties to be most important: the middleware used for networking, the number of cores on each node and the network structure. While the former two are easily detected (e.g. by checking `MPI::COMM_WORLD.size()` and `/proc/cpuinfo`), the latter is a lot harder to find out. For the HAP pattern it is only important that nodes with similar properties are grouped together. Therefore Pollarder does not need to perform a complex discovery of the exact network topology, but can resort to a hierarchical clustering algorithm. It yields a tree of nodes clustered according to a heuristic we have

developed as a distance measure. A full description of our clustering algorithm would be beyond the scope of this paper, but the following paragraphs should yield a brief overview.

Our method consists of two main components: the distance measure to get an estimate of how close two compute nodes are together and the clustering algorithm itself, which yields a hierarchy based upon the measure. For the distance measure we sought a heuristic that would be suited for use with the HAP pattern. Close nodes should have a good connection and within each cluster the interconnect should be relatively homogeneous to ease the efficient programming of each subsystem. Initially we though measuring the ping pong latency between each node (time to send one byte from one node and back) would be a sufficient measure, but this would ignore different bandwidths in the corresponding networks. Similarly, the throughput alone wouldn't be sufficient. Even together, these two measures would often failed to detect the (for multi-clusters important) case when two nodes are directly connected to the same switch, but this connection has to be shared by multiple nodes (as it is the case for cluster head nodes which often route the inner nodes' traffic to the outside. To detect this case, we found a third heuristic to be useful: the similarity of the hostname. Let s and t be two hostname strings. Our measure takes the domain-wise reversed hostnames and then extracts their common prefix p. The distance d is then defined as $d = (max\{|s|, |t|\} - |p|)/max\{|s|, |t|\})$[1]. To cope with different scales, the three resulting distance matrices (one each for latency, throughput and hostname similarity) were multiplied by the reciprocal of their maximum element. The three dimensional distance vectors were reduced to a scalar using the Euclidean norm.

Additionally to the distance measure, we had to find a clustering algorithm. One could group a system's compute nodes in two extreme forms of hierarchies. The first one is a flat tree of height one where each compute node forms a leaf and all leafs are connected via the root. The second one is a binary tree in which subsequently the two closest clusters are grouped via a new union node. Both examples are undesirable for use with the HAP pattern. In a heterogeneous multi-cluster the flat tree would conceal which groups of nodes should be handled separately while the binary tree would yield no information on which nodes could be handled as a group. Thus we were looking for an algorithm that would yield a compromise, a tree not too high, but with nodes that are not too fat. In order to facilitate automation, the clustering algorithm should require as little user input as possible.

The QT (quality threshold) algorithm [7], originally developed for gene clustering, seemed to be a promising candidate. It does not require the user to chose a number of clusters up front, but rather a maximum diameter for the clusters (a cluster's diameter is defined as the maximum distance between two elements, i.e. complete linkage). Given a distance measure, QT will employ a greedy

[1] For instance `racl00.inf-ra.uni-jena.de` and `ppc660.mirz.uni.jena.de` would be reversed to `de.uni-jena.inf-ra.racl00` and `de.uni-jena.mirz.ppc660`. p would be `de.uni-jena.`, leading to a distance of $s = (25 - 12)/25 = 0.52$.

algorithm to subsequently partition the initial set of elements into clusters, starting with the element wise largest cluster of valid diameter. Nodes already assigned to a cluster are taken out of consideration. The disadvantage of this algorithm is that it will not necessarily yield a connected graph. Some nodes may not even belong to any cluster.

As a solution, we use a multiphase variation of QT. It takes as input parameters, similarly to QT, a distance measure and a maximum cluster diameter. Additionally it requires a diameter multiplier. In the first phase nodes are clustered using QT. Each cluster is represented by a new *group node*. For the next round the maximum diameter is enlarged by the diameter multiplier and QT is run again on the remaining set of nodes and clusters. The distance between two clusters is computed via single linkage (minimum distance of two elements). For a maximum diameter greater than 0 and a diameter multiplier greater than 1 this will yield a complete hierarchy in every case. Figure 5 shows the result of our algorithm based on data gathered on the multi-cluster shown in Fig. 1.

Finally, according to the HAP pattern, the artificially introduced group nodes have to be assigned to physical machines. Our algorithm uses a bottom up strategy: starting with the lowest level of clusters, for each the single linkage between its elements and all nodes outside of this cluster is computed. The cluster side element belonging to the single linkage pair is then associated with the group node (not shown in Fig. 5).

7 Evaluation

We are currently evaluating a C++ prototype of Pollarder. In collaboration with the Chair of Metallic Materials at FSU Jena, our working group has developed the computational science application MuCluDent (Multi-Cluster DendriTe), a simulation code for dendritic growth in freezing metal alloys [8]. It is based on a combination of cellular automaton and finite difference method and is parallelized via geometric decomposition. MuCluDent has to run efficiently on a variety of hardware: small scaled model tests are typically done on workstations and notebooks, the parallelization is mostly tested on one of our clusters and long simulations with relevant domain sizes are run on our multi-cluster setup. MuCluDent should be able to adapt itself to the current system without source code changes and user interaction.

We use Pollarder to simplify the wire-up code which instantiates a parallelization, load balancer and IO objects. Figure 4 shows a simplified version of this part. The macro `POLLARDER_REGISTER_PARALLELIZATION` will expand to a specialization of Pollarder's class template based component library. As parameters it takes the parallelization's base class, the class itself, a slot number (for alternatives) and a scoring function. Similarly `POLLARDER_SUPPORTS_HAP` is used to hand on type information to Pollarder. Even though MuCluDent currently only uses Pollarder's MPB pattern and support for IO objects (not described in the patterns sections), we could reduce the length of the wire-up code from about 250 lines to circa 50.

```
template<class MODEL> class SerialSimulator {...};
template<class MODEL> class StripingSimulator {...};
template<class MODEL> class PartitioningSimulator {...};
POLLARDER_REGISTER_PARALLELIZATION(Simulator, SerialSimulator, 0,
    Pollarder::isSerial);
POLLARDER_REGISTER_PARALLELIZATION(Simulator, StripingSimulator, 1,
    Pollarder::isThreaded);
POLLARDER_REGISTER_PARALLELIZATION(Simulator, PartitioningSimulator, 2,
    Pollarder::isMPI);

void run(Initializer init) {
    Simulator<Cell> *sim =
        Pollarder::Factory<Simulator>().get<Cell>();
    sim->run(init);
}
```

Listing 1.1. Shortened Wire-up Code from MuCluDent

```
template<class MODEL> class ThreadedIntegrator {...};
template<class MODEL> class MPIIntegrator {...};
POLLARDER_REGISTER_PARALLELIZATION(Integrator, ThreadedIntegrator, 0,
    Pollarder::isThreaded);
POLLARDER_REGISTER_PARALLELIZATION(Integrator, MPIIntegrator, 1,
    Pollarder::isMPI);
POLLARDER_SUPPORTS_HAP(ThreadedIntegrator);
POLLARDER_SUPPORTS_HAP(MPIIntegrator);

double run() {
    Integrator<Parbola> i =
        Pollarder::Factory<Integrator>().get<Parabola>();
    return i->integrate(0, 1);
}
```

Listing 1.2. Wire-up Code for our Numerical Integration Demo

Fig. 4. Exemplary use of Pollarder. Notice how similar the componentization and wire-up are, even though the applications have to perform very different computations.

The heterogeneous networks we use in our multi-cluster setup have proven to be problematic for MuCluDent, as the slowest node will determine the whole system's performance. With overlapping computation and communication, a load balancer can reduce the time needed for computation on slower nodes, but to compensate high latency networks, the ghost zone would have to be enlarged. While enlarged ghost zones on all nodes would be undesirable (as they come at the expense of increase overhead and would be unnecessary between nodes sharing a low latency network), handling a locally increased ghost zone width only between selected nodes turned out to be overly complex. We plan to smooth this out with a HAP capable parallelization for MuCluDent, but, unlike our other parallelizations, this implementation is not yet able to perform load balancing. As MuCluDent's computational load is distributed very unevenly across the simulation grid, we could not gather meaningful benchmark results for this parallelization so far.

We did however test the HAP pattern with a demo application which implements a simple numerical integration for one dimensional functions. Figure 4 shows a small code excerpt. For the test run we used three dual Opteron nodes from our RA cluster. For comparison we did integrate $f(x) = x^2$ on the interval

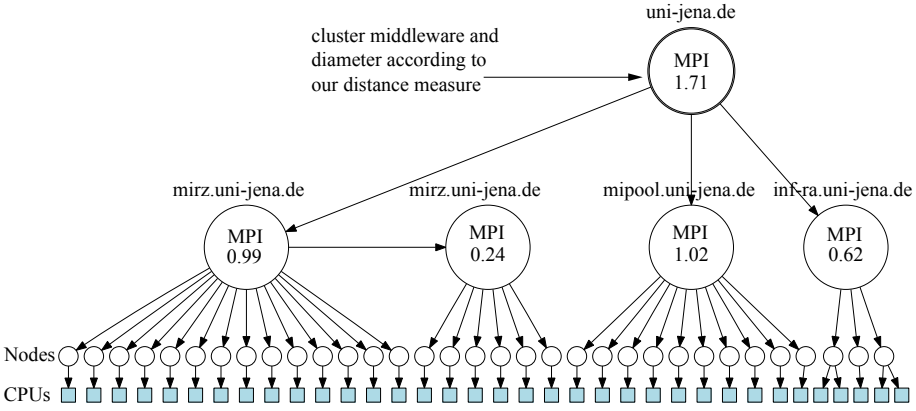

Fig. 5. Cluster analysis from a test run on our multi-cluster system. The initial maximum diameter was 0.7, the diameter multiplier was 1.6.

$[0, 1]$ once using a "flat" MPI parallelization with six processes (two on each machine) and once with a stacked HAP parallelization that used three MPI processes which forwarded their sub-intervals to a threaded parallelization. Although the problem scales well, the HAP parallelization turned out to be 31% faster than the flat parallelization. This is because of the low number of samples (2000), the initial scatter of interval borders and the final gather of results did dominate the running time and a reduced number of MPI processes sped them up. But still it substantiates our claim that HAP may benefit a system's efficient usage. As the flat parallelizations in the MuCluDent project suffer from the same problem that communication may be the dominating factor in a multi-cluster, we expect a comparable gain for our geometric decomposition codes.

Figure 5 shows the result of Pollarder's environment detection using our cluster analysis algorithm. For the test 20 nodes from the Unix pool were used along with 10 from the Linux pool and three from the RA cluster (two of which were dual Opterons). Despite its early stage, our prototype was able to reliably detect the system's structure, including the two dual Opterons on the right. An interesting observation is the sub-cluster of diameter 0.24 in the Unix pool (`mirz.uni-jena.de`). Initially this seemed to be a bug in our algorithm, but it turned out that the nodes in this sub-cluster have gigabit Ethernet, which contrasts the other Unix pool nodes that only use Fast Ethernet.

8 Summary and Outlook

Complexity and variety of contemporary grid systems have become major challenges for scientific computing. We have presented a new approach to grid application componentization, specially targeted at adaptive parallelizations. The presented design patterns can break down an application's functionality into small, reusable components. Our prototype suggests that these pattern are

generic enough to be employed in a variety of applications, ranging from loosely coupled problems like simple function integration to tightly coupled geometric decomposition codes. The Model-Parallelization-Balancer pattern takes care for coarse grained adaptation, while Hierarchical Adaptive Parallelization can decompose complex parallelizations into smaller sub-parallelizations. This is especially important in the face of increasingly popular combined multi-core and MPI cluster setups. An factory takes over environment discovery and assembles the application's components, thereby enabling self-adaption to multiple environments and relieving the user from manual interaction. While the adaptation provided by the factory is of static nature, the balancer in the MPB pattern can provide dynamic adaptation during at runtime. Despite being only a prototype, our current implementation has already proven itself in a real application and is able to reliably detect even complex multi-cluster setups.

References

1. Goodale, T., Allen, G., Lanfermann, G., Masso, J., Radke, T., Seidel, E., Shalf, J.: The Cactus Framework and Toolkit: Design and Applications. In: Palma, J.M.L.M., Sousa, A.A., Dongarra, J., Hernández, V. (eds.) VECPAR 2002. LNCS, vol. 2565. Springer, Heidelberg (2003)
2. Bernholdt, D.E., Allan, B.A., Armstrong, R.C., Bertrand, F., Chiu, K., Dahlgren, T.L., Damevski, K., Elwasif, W.R., Epperly, T.G., Govindaraju, M., Katz, D.S., Kohl, J.A., Krishnan, M.K., Kumfert, G.K., Larson, J.W., Lefantzi, S., Lewis, M.J., Malony, A.D., McInnes, L.C., Nieplocha, J., Norris, B., Parker, S.G., Ray, J., Shende, S., Windus, T.L., Zhou, S.: A Component Architecture for High-Performance Scientific Computing. International Journal of High Performance Computing Applications 20, 163–202 (2006)
3. Baduel, L., Baude, F., Caromel, D., Contes, A., Huet, F., Morel, M., Quilici, R.: Programming, Deploying, Composing, for the Grid. In: Grid Computing: Software Environments and Tools. Springer, Heidelberg (2006)
4. Fowler, M.: Inversion of Control Containers and the Dependency Injection Pattern (2004)
5. Mattson, T.G., Sanders, B.A., Massingil, B.L.: Patterns for Parallel Programming. Addison Wesley Professional, Reading (2004)
6. Quinn, M.J. (ed.): Parallel Programming in C with MPI and OpenMP, vol. 1. Mc Graw Hill, New York (2003)
7. Heyer, L., Kruglyak, S., Yooseph, S.: Exploring expression data: identication and analysis of coexpressed genes. Genome Research 9, 1106–1115 (1999)
8. Schäfer, A., Erdmann, J., Fey, D.: Simulation of Dendritic Growth for Materials Science in Multi-Cluster Environments. In: Workshop Grid4TS, vol. 3 (2007)