Preliminary Resource Management for Dynamic Parallel Applications in the Grid

Hao Liu, Amril Nazir, and Søren-Aksel Sørensen

Department of Computer Science, University College London
United Kingdom
{h.liu,a.nazir,s.sorensen}@cs.ucl.ac.uk

Abstract. Dynamic parallel applications such as CFD-OG impose a new problem for distributed processing because of their dynamic resource requirements at run-time. These applications are difficult to adapt in the current distributed processing model (such as the Grid) due to a lack of interface for them to directly communicate with the runtime system and the delay of resource allocation. In this paper, we propose a novel mechanism, the Application Agent (AA) embedded between an application and the underlying conventional Grid middleware to support the dynamic resource requests on the fly. We introduce AA's dynamic process management functionality and its resource buffer policies which efficiently store resources in advance to maintain the execution performance of the application. To this end, we introduce the implementation of AA.

Keywords: resource management, dynamic parallel application, resource buffer.

1 Introduction

The Grid is commonly used to submit batch and workflow type jobs. However, many scientific applications, such as astrophysics, mineralogy and oceanography, have several distinctive characteristics that differ from batch and workflow type of applications. Some of them are so-called dynamic parallel applications allowing significant changes to be made to the structure of the datasets themselves when necessary. Consequently they may require resources dynamically during the execution to meet their performance benchmarks. CFD-OG (Computational Fluid Dynamics-Object Graph) is one of the examples in that respect. Such applications are hardly applicable with the current distributed processing model, which needs the knowledge of application execution behavior and resource requirements prior to execution.

Our strategy is to introduce an agent that enables a running Grid application to negotiate the computational resources assigned to it at run-time. The agent provides an interface for the application to call for resources on the fly while it communicates with the Grid middleware to allocate resources to satisfy these requests on-demand.

In this paper we introduce the dynamic parallel applications demonstrated by CFD-OG and the necessity of external resource management for them. We propose a novel mechanism, the Application Agent (AA) embedded between the applications and conventional Grid middleware for resource management. We also propose two resource buffer policies to maintain the application performance cost effectively while the resource demands change constantly.

2 Dynamic Parallel Applications

Computational Fluid Dynamics (CFD) is one of the branches of fluid mechanics that uses numerical methods and algorithms to solve and analyze problems that involve fluid flows. One method to compute a continuous fluid is to discretize the spatial domain into small cells to form a volume mesh or grid, and then apply a suitable algorithm such as Eulerian methodology [18] to solve the equations of motion. The approach assumes that the values of physical attributes are the same throughout a volume. If scientists need a higher resolution of the result they have to replace a volume element with a number of smaller ones. Since the physical conditions change very rapidly, high resolution is needed dynamically to represent the flux (amount per time unit per surface unit) of all the physical attributes with sufficient accuracy. Consequently the whole volume grid is changing constantly, which may lead to dynamic resource requirements (Fig. 1).

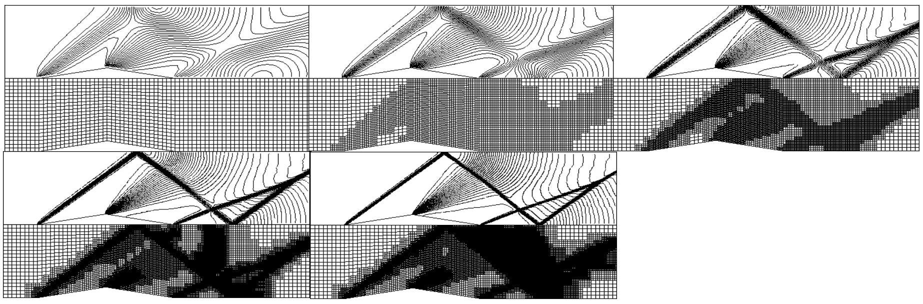

Fig. 1. A series of slides of the flow across a hump. The grid structure is constantly changing at run-time to adjust the reasonable resolution. The resource requirements in the last slide are around 120 times those in the first slide [12].

The usual method for introducing high resolution is to replace a volume element with a number of smaller ones. This is a difficult process in traditional CFD because it used a Cartesian of indexed volumes (V_{ijk}). To introduce smaller volumes into such a structure would require further hierarchical indexing leading to complex treatment. An alternative approach which is introduced by Sørensen is the Object Graph (CFD-OG) [12]. A CFD-OG application has two base objects: cells and walls. A cell represents the fluid element and holds the physical values for a volume. It is surrounded by a number of wall objects known to it by their addresses (pointers). A cell simply holds a number of wall pointers. The walls on the other hand only know two objects: the cells on either side of the wall. This is a simple structure that is completely unaware of the physical geometry and topology of the model. The advantage of the object graph is that it uses reference by addressing only. It is therefore possible to change the whole grid topology when smaller volume elements are introduced. This approach also benefits the distributed processing since it is easy to substitute the local addresses (pointers) with global addresses (host, process, pointer) without changing the structure. That being the case, the objects can be distributed on the network computer nodes in any manner.

With object graph, a distributed CFD-OG application can be very dynamic and autonomic. The application is composed of a number of processes executing synchronously and normally one process is running on one nodes. In this context, each process holds a number of computational objects (cells and walls) that can migrate from one process to another. As the application progresses, a built-in physical manager monitors the local conditions. If it detects the local resolution is too low, it will ask the built-in object manager to introduce smaller volumes. If it on the other hand the built-in physical manager detects superfluous resolutions, it asks to replace several smaller cells with fewer larger ones. This may create imbalances in the processing and the object manager may subsequently attempt to balance this by moving objects onto light load nodes.

Such load balancing is limited. The application may have to demand additional resources (processors) immediately to maintain the required performance such as a certain execution time progression. Likewise, the application may want to release redundant allocated resources when the low resolution is acceptable. The dynamic resource requirements on the fly is a huge challenge for current distributed environment. This is because there is no well defined interface for applications to communicate with the Grid to add/release resources at run-time; and further more a significant delay is normally associated with the allocation of a resource in response to a request due both to resource competition between running jobs and the time conflict of concurrent requests, which will influence the smooth execution of the application. For the first problem, we define a mechanism which has several functions that applications can call to add and release resources during the execution in a way that is independent from resource managers. For the second problem, we propose the resource buffer management to hide the delay of resource allocation.

3 Dynamic Resource Support

As introduced, a dynamic parallel application is composed of a number of processes that holds a number of computational objects that can migrate from one process to another by the application itself. Therefore, as the application needs an additional resource, it requests to deploy a new process that it can move other objects into. The resource requirement is therefore interpreted as adding a new process with no objects initially.

We define a mechanism called Application Agent (AA) that stays between applications and Grid middleware to support such dynamic addition/release of processes. Programmers must build applications based on the programming interface provided by AA. Each process's binary file therefore has to be compiled with the AA library. That means each running process is associated with an AA which keeps information synchronized across the whole application. Programmers can perform three basic operations inherited from AA: add a process, stop a process, and exchange messages with a process. As soon as the application is running, AA will start and act as an agent to communicate with the Grid environment on behalf of the application, i.e. to find a new node and deploy a new process, stop a process and return a node, and transport messages during the execution of the application. The process requests are served by a scheduler that is associated with AA. The scheduler can apply different scheduling policies (e.g. FCFS (First Come First Served), priority-based) according to the preference of the application. We do not address the scheduling issue in this paper.

3.1 Adding a Process

AA allows the application programmer to start a named process at anytime during the execution. The process request is performed using AddProcess(Name of executable) which returns an integer ID which subsequently used to address the process. The function AddProcess() is non-blocking. As soon as it is invoked, AA will check if its Resource Buffer (RB) holds an additional prepared process (an idle process that has been deployed on a new node previously). If so, it immediately returns the process ID to the application which can activate this process for migrating objects. This is called a ”successful request”. If not, it will contact the underlying Grid scheduler such as Condor [3], SGE [1] or GRAM [7] to request an allocation, simultaneity returning integer 0 to the application. This is called a ”failed request”. As in a non-dedicated Grid environment the amount of time it takes for a process to be allocated is not bound, it is the programmer’s responsibility to request a process again if previous request has failed. Once the new process is deployed, AA will store the ID of the process into the RB and return it to the application as soon as the next request is performed.

One problem with the dynamic addition of process is that current Grid schedulers do not support dynamic resource co-allocation and deployment. They treat each later added process as an independent single job and do not deploy them communicable with other processes of the parallel application. One approach (approach A) to this problem is that once the process is allocated by a scheduler, the AA that is associated with this new process must broadcast its address to other old processes so that they can communicate. This is archived by registering this process’s address into the RB which is synchronized throughout the application. The new process is assigned an unique global id for further use by the application during the registration. This approach can be implemented by general socket programming or based on distributed computing framework (e.g. CORBA [10], Jini [11]).

An alternative approach (approach B) is to use probes for resource allocation while the actual process startup is accomplished by AA itself. A probe is a small piece of deployment code that does not perform any computation for the application. The probe always runs until AA kills it, in order to hold that node. Once the probe is allocated by the scheduler, it configures the node for AA use and notifies its node address to the master AA which takes charge of process spawning. Then the master AA transfers the process binary onto that node and starts the process. This approach can make full use of existing process management systems (e.g. PVM [6], LAM/MPI [2]) that will return a process identifier for the application to address the process.

Once the new processed is deployed, the object manager of the application can then migrate the target objects into this process for load balancing. The migration procedure is beyond the scope of this paper.

3.2 Stopping a Process

Programmers may want to stop a process and release the node when application's resource requirement is low. The application firstly autonomically vacates the process (computational objects migration), then invokes the function StopProcess(id). This results in the process with the given ID to be removed from the application and the related computer node to be disassociated. However AA may still reserve this prepared process for the possible future requests from the application. The decision is made according to AA's buffer policies (we discuss them in next section). If AA does decide to release the node, it contacts the local scheduler to kill the vacated process and finally return the node to the pool.

3.3 Communication

Technically, AA could either have a new communication mechanism if using approach A for adding processes or have PVM/MPI as the lower communication service if using approach B. In the latter case, programmers can still use PVM/MPI routines for communication, and use AA's routines for process management.

4 Resource Buffer

In non-dedicated Grid environments the amount of time it takes until a process is allocated is not bound. In order to satisfy the process addition request on-demand at run-time, we propose the Resource Buffer (RB) to hide the delay of process allocation. A RB stores a number of prepared processes that can be returned to the application immediately when it requests AddProcess(). AA manages the RB by requesting the allocations of processes from schedulers in advance.

AA manages the RB to satisfy the application demands based on the RB policies. In order to measure the RB performance, we propose two simple metrics: Request Satisfaction S and Resource Waste W. Since the process release requests do not benefit from the policies, the Request Satisfaction is defined as the percentage of successful added processes out of the total number of process addition requests. If S is approaching 1, it indicates that the dynamic application can run smoothly since all of dynamic resource requests are satisfied on-demand. The Resource Waste is defined as the accumulative total time when the prepared processes stay in the RB (W = $\sum_{t=0}^{T_{exe}} R_t$ where T_{exe} = total execution time; R_t = the number of idle processes in the RB at time t). If the RB policy is good enough, W should be approaching 0 while S should be approaching 1.

In order to investigate the RB policies, we made a few assumptions regarding the dynamic parallel application:

– 1. It is a parallel iterative application.
– 2. Its dynamic behavior is not random. Similar to the CFD-OG, the application only requests resources on some stage (e.g. during iteration 100 ~ 110) when physical condition changes.
– 3. It only requests to add processes but not to release processes.

4.1 Policies

We consider two corresponding heuristic policies, the Periodic (P) policy and the Periodic Prediction (PP) policy to manage the RB.

The P policy periodically reviews the RB to ensure that the RB is kept to a predetermined process amount RL at each predefined interval t_p. For every time interval, the number of Nprocess requested is:

$$N = \begin{cases} +(RL - R) : R <= RL \\ -(RL - R) : R > RL \end{cases}$$

where RL is the request (threshold) level and R is the current number of prepared processes in the RB at each periodic level. ”+” means requesting the allocation of a node and deploying a new process and ”-” means requesting the release of a process and returning the node to the pool. The value of RL is crucial for the P policy. RL can be determined by the maximum number of requests that the application could consecutively make. Generally, if the application requests processes frequently, RL will be higher.

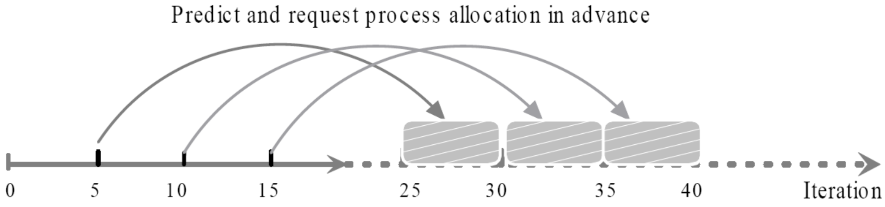

Fig. 2. AA periodically predicts the number of requests the application will make. E.g. when the application is running at iteration 5, AA predicts the period from iteration 25 to 30.

If the time T_{alloc} that the Grid scheduler takes to allocate a process in the cluster is known and relatively stable, we can use the PP policy. The PP policy periodically (at interval tp) predicts the number of requests that the application will make after θ time (Figure 2). If AA predicts that after θ the application will make n requests, AA will place n requests to the scheduler immediately. We let $\theta = \max T_{alloc} \approx \forall T_{alloc}$. Then the application would be able to use the advanced placed process right after it has been allocated.

PP uses the application’s historical execution behaviors for the prediction. Since the application execution behavior may vary due to its execution environment, PP predicts the probability of a request in an iteration range (e.g. iteration 25 ~ iteration 30, Figure 2) rather than a single iteration. The probability $p_{i \sim i + range}$ in the future iteration i ~ i + range is calculated as $P_{i \sim i+range} = \sum_{i}^{i+range} r_i / times$, where r_i is the total number of requests at iteration i during the recorded executions, times is number of recorded historical executions, and range is the number of iteration the prediction covers. The number of request $n_{i\sim i+range}$ the application is predicted to make in iteration i ~ i + range can be calculated by $\lfloor P_{i\sim i+range} \rfloor$.

In order to avoid the redundancy of process in the RB, PP also has a request (threshold) level RL. As the current number of process R reaches beyond RL, AA initiates to release processes. The full Pseudo-code for PP policy in AA is shown in Figure 3.

```
 1. Get $\max T_{alloc}$
 2. Let $\Theta = \max T_{alloc}$
 3. Get the average execution speed $sp_{exe}$ of the
    application. (the time it takes to run 1 iteration)
 4. Let $\Delta = \lceil \Theta / sp_{exe} \rceil$

 5. Get the current iteration $ci$ of the application.
 6. Let $i = ci + \Delta$
 7. Let $P_{i\sim i+range} = \sum_{i}^{i+range} r_i / times$
 8. If ($\lfloor P_{i\sim i+range} \rfloor > 0$)
    Request to allocate $\lfloor P_{i\sim i+range} \rfloor$ processes
 9. If ($R > RL$)
    Request to release $R - RL$ processes
10. Sleep for range iterations. $// t_p = range$ iterations
11. Go to step 5
```

Fig. 3. Pseudo-code for PP policy

4.2 Simulation and Results

Based on the assumptions we made, we use Clown [17], a discrete-event simulator to simulate a simple iterative application that has a behavior similar to the real dynamic parallel application. Initially, this application has three processes, each of which has 10 computational objects. To simulate the dynamic behavior, one of the processes generates 10 objects every 100 iterations. Due to the increase of objects, the application will detect when the execution speed is getting slower and subsequently responds by requesting additional resources/processes by AddProcess() to balance the computation in ensuring smooth execution. If AA cannot satisfy the requests on some stage, the application runs slower with insufficient resources and it will request to add processes again in next iteration. In order to simulate the execution of the application, we simulate 500 homogeneous computer nodes where the average speed sp_{node} is 11.2 (it takes 11.2 simulation time to complete 1 objects, and takes 112 to complete 10 objects) . The execution speed fluctuates in each run according to a Gaussian distribution with variance equals to 1.0. In the cluster, the time T_{alloc} it takes to allocate a node is Gaussian distributed with average 1000 and variance 100. The simulated execution speed sp_{exe} (the time it takes to run 1 iteration) with S = 1 can be monitored as ≈ 120. The main purpose of the simulation is to evaluate the effectiveness of the proposed buffer policies under the simulated environment.

For each policy, we run the application for 100 times. For each execution, the application is run for 3000 iterations. Figure 4 shows the value S of both policies during the 100 executions. We can see the Request Satisfaction of the P policy is slightly higher than what the PP policy can provide. For the PP policy, S is very low in the beginning and increases to 90% around the 5th execution. This is because PP is based on historical information and there is not enough information for predicting the execution behavior in the beginning. During the 100 executions, both policies can provide reasonable S (S > 80%). The small fluctuations are caused by the unstable T_{alloc} and sp_{node}.

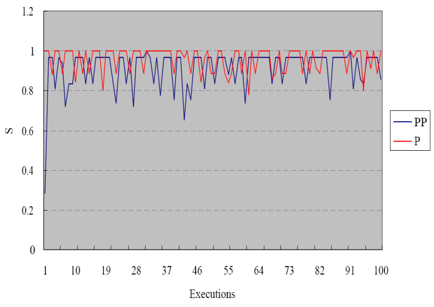

Fig. 4. The Request Satisfaction S of both policies during 100 executions. PP: range = 5, $RL = 1$, t_p = 5iterations. P: $RL = 1$, $t_p = 600$.

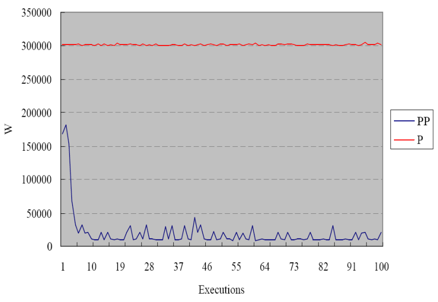

Fig. 5. The Resource Waste W of both policies during 100 executions. PP: range = 5, $RL = 1$, t_p = 5iterations. P: $RL = 1$, $t_p = 600$.

Figure 5 shows the value W of both policies during the 100 executions. We can see that the PP policy is far more efficient than the P policy. The W of PP in the first few executions are relatively high since historical information is insufficient. W then drops rapidly through the learning process and maintains a low value with small fluctuations in the rest of the executions.

The results show that both proposed policies have their pros and cons. The P policy can provide slightly higher Request Satisfaction while leads to very high Resource Waste too. The PP policy is considered more advanced. It can perform simple prediction and make requests at the right time. However the current PP policy is not suitable for the cluster where T_{alloc} is random and unpredictable. While the P policy is suitable for any environment.

5 Implementation

The current implementation of AA is developed based on PVM. The process management follows approach B. Requests are served according to FCFS policy. The test environment is a Condor pool which has 50 Linux machines. The cluster has NFS (Network File System) installed and each machine has PVM installed.

All the AA-enabled application's binaries and related files must be put on a NFS mounted directory. The application is firstly started on the Condor submitting machine. The first process started is called master process and manages the whole application. The correlative AA is called master AA. When AA decides to add a process (receiving a request or according to the RB policies) for the application, it firstly asks the master AA to submit a probe program to Condor specifying the resource requirements (e.g. Arch == "INTEL") of the application in the submit file. Once the probe is allocated, it notifies the master AA by writing the node information into a XML file on the NFS. AA keeps reading this file. Once it finds that a new node is added, it immediately adds that node into its virtual machine by pvm addhosts(). Then it starts a process on that node by pvm spawn(), and stores the process id into its RB. Since all the binaries are on the NFS, AA does not need to transfer any files. AA in its current implementation does not support heterogeneous deployment. It does allow heterogeneous processing as long as a suitable executable is present on the target node.

When AA releases a process, it first stops the process by pvm kill(), then excludes the node from its virtual machine by pvm delhosts(). It finally returns this node to the pool by killing the probe via condor rm().

AA's message passing module is a C++ wrapper of PVM's interface.

6 Related Work

Condor-PVM [3] provides a dynamic resource management for PVM applications executing in a Condor cluster. Whenever a PVM program asks for nodes, the request is re-mapped to Condor, which then finds a machine in the Condor pool via the usual mechanisms, and adds it to the PVM virtual machine. This system is intended to integrate the resource management (RM) systems (Condor) with the parallel programming environment (PVM) [14]. Although it supports runtime resource requests similar to what AA supports, it does not put any effort into the performance of the application, e.g buffer management. Moreover, the request scheduling for the application is totally managed by Condor, which has no scalability to add other application-level scheduling policies.

Gropp et al. [9] introduce an extension of MPI for MPI applications to communicate the job scheduler to add and subtract processes during the computation. It proposed a non-blocking call MPI IALLOCATE to reserve processors from the scheduler with returning a set of MPI requests. Then the actual process startup can be accomplished with the conventional MPI START or MPI STARTALL calls. This paper however does not provide detailed implementation information.

DUROC [4] implements an interactive transaction strategy and mechanism for resource co-allocation in a multi-cluster environment. It accepts multiple requests, each written in a RSL expression and each specifying a subjob of an application. In order to tolerate resource failures and timeouts, some resources may be specified as "interactive" and "optional". Successful completion of a DUROC co-allocation request results in the creation of a set of application processes that are able to communicate with one another. An important issue in the resource co-allocation is that the required resources have to be available at the same time otherwise the computation cannot proceed. While in our model, a dynamic parallel application can continue computation with insufficient resources and request additional resources via AA during the computation to maintain its ideal performance.

7 Conclusion and Future Direction

The contribution of this paper is the proposal of an application agent AA that supports the dynamic resource requirements of dynamic parallel applications such as CFD-OG. An AA-enabled application is able to add new resource (deploy a new process) and release surplus (release a process) at run-time. To maintain the smooth execution of the application, the Resource Buffer service is proposed that is embedded in AA to relieve the cost for waiting resources. Two heuristic policies are introduced to examine how the RB concept can be managed more effectively and efficiently.

The current version of AA is implemented with approach B (detailed in section 3) and tested in a Condor cluster. As we mentioned, this approach is restricted by existing systems. For example, PVM is bounded to join resources that are located in the same network domain and so AA cannot perform wide-area computing based on PVM. Some extensions (e.g. PMVM [13]) enable PVM to create multi-domain virtual machines. The future work will involve implementing and testing AA in a multi-domain environment. We aim to investigate whether the virtual machine architecture would apply in this setting or it is more appropriate to apply approach A that distributed loosely links processes across the network. The security problem arising from multi-domain environment will be also addressed.

The RB policies also need more precise investigation. Two polices will be further tested by two real world dynamic parallel applications CFD-OG and RUNOUT [16]. The policies will be further extended to intelligently react to the change of resource environment to ensure that the smooth execution of application is not affected.

References

1. N1 grid engine6 administration guide. Technical report, Sun Microsystems, Inc.
2. Burns, G., Daoud, R., Vaigl, J.: LAM: An Open Cluster Environment for MPI. In: Proceedings of Supercomputing Symposium, pp. 379–386 (1994)
3. Condor. Condor online manual version 6.5, http://www.cs.wisc.edu/condor/manual/v6.5
4. Czajkowski, K., Foster, I.T., Kesselman, C.: Resource co-allocation in computational grids. In: HPDC (1999)
5. Foster, I.: Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering. Addison-Wesley Longman Publishing Co., Inc., Boston (1995)
6. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V.S.: PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing. MIT Press, Cambridge (1994)
7. Globus Alliance. Globus Toolkit, http://www.globus.org/toolkit/
8. Goux, J.-P., Kulkarni, S., Linderoth, J., Yoder, M.: An enabling framework for master-worker applications on the computational grid. In: HPDC, pp. 43–50 (2000)
9. Gropp, W., Lusk, E.: Dynamic process management in an mpi setting. spdp, 530 (1995)

10. Idl, N.S.: The common object request broker: Architecture and specification
11. S. Microsystems. Jini network technology. Technical report, `http://www.sun.com/software/jini/`
12. Nazir, A., Liu, H., Sørensen, S.-A.: Powerpoint presentation: Steering dynamic behaviour. In: Open Grid Forum 20, Manchester, UK (2007)
13. Petrone, M., Zarrelli, R.: Enabling pvm to build parallel multidomain virtual machines. In: PDP 2006: Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, Washington, DC, USA, pp. 187–194. IEEE Computer Society Press, Los Alamitos (2006)
14. Pruyne, J., Livny, M.: Providing resource management services to parallel applications (1995)
15. Shao, G.: Adaptive scheduling of master/worker applications on distributed computational resources (2001)
16. Sørensen, S.-A., Bauer, B.: On the dynamics of the kofels sturzstrom. In: Geomorphology (2003)
17. Sorensen, S.-A., Jones, M.G.W.: The clown network simulator. In: 7th UK Computer and Telecommunications Performance Engineering Workshop, London, UK, pp. 123–130. Springer, Heidelberg (1992)
18. Trac, H., Pen, U.-L.: A primer on eulerian computational fluid dynamics for astrophysics. Publications of the Astronomical Society of the Pacific 115, 303 (2003)
19. Wolski, R., Spring, N.T., Hayes, J.: The network weather service: a distributed resource performance forecasting service for metacomputing. Future Generation Computer Systems 15(5–6), 757–768 (1999)