Dynamic Resizing of Parallel Scientific Simulations: A Case Study Using LAMMPS

Rajesh Sudarsan[1], Calvin J. Ribbens[1], and Diana Farkas[2]

[1] Department of Computer Science, Virginia Tech, Blacksburg, VA 24060
sudarsar@vt.edu, ribbens@vt.edu
[2] Department of Materials Science and Engineering, Virginia Tech, Blacksburg, VA 24061

Abstract. Large-scale computational science simulations are a dominant component of the workload on modern supercomputers. Efficient use of high-end resources for these large computations is of considerable scientific and economic importance. However, conventional job schedulers limit flexibility in that they are 'static', i.e., the number of processors allocated to an application can not be changed at runtime. In earlier work, we described ReSHAPE a system that eliminates this drawback by supporting dynamic resizability in distributed-memory parallel applications. The goal of this paper is to present a case study highlighting the steps involved in adapting a production scientific simulation code to take advantage of ReSHAPE. LAMMPS, a widely used molecular dynamics code, is the test case. Minor extensions to LAMMPS allow it to be resized using ReSHAPE, and experimental results show that resizing significantly improves overall system utilization as well as performance of an individual LAMMPS job.

Keywords: LAMMPS, parallel clusters, dynamic scheduling, data redistribution.

1 Introduction

Today's terascale and petascale computers exist primarily to enable large-scale computational science and engineering simulations. While high-throughput parallel applications are important and often yield valuable scientific insight, the primary motivation for high-end supercomputers is applications requiring hundreds of compute cores, with data sets distributed across a large aggregate memory, and with relatively high interprocess communication requirements. Such calculations are also characterized by long running times, often measured in weeks or months. In this high-capability computing context, efficient utilization of resources is of paramount importance. Consequently, considerable attention has been given to issues such as parallel cluster scheduling and load balancing, data redistribution, performance monitoring and debugging, and fault-tolerance. The goal is to get the maximum amount of science and engineering insight out of these powerful (and expensive) computing resources.

A constraint imposed by existing cluster schedulers is that they are 'static,' i.e., once a job is allocated a set of processors, it continues to use those processors until it finishes execution. Even if there are idle processors available, parallel applications cannot use them because the scheduler cannot allocate more processors to an application at runtime. A more flexible approach would allow the set of processors assigned to a job to be expanded or contracted at runtime. This is the focus of our research—dynamically reconfiguring, or *resizing*, parallel applications.

We are developing *ReSHAPE*, a software framework designed to facilitate and exploit dynamic resizing. In [1] we describe the design and initial implementations of ReSHAPE and illustrate its potential for simple applications and synthetic workloads. An obvious potential benefit of resizing is reduced turn-around time for a single application; but we are also investigating benefits such as improved cluster utilization, opportunities for new priority-based scheduling policies, and better mechanisms to meet quality-of-service or advance reservation requirements. Potential benefits for cluster utilization under various scheduling scenarios and policies are considered in [2]. Efficient data redistribution schemes are described in [3].

In this paper we investigate the potential of ReSHAPE for resizing production computational science codes. As a test case we consider LAMMPS [4,5], a widely used molecular dynamics (MD) simulation code. One of us uses this code on a regular basis to study the mechanical behavior of nanocrystalline structures [6]. In a typical case we run LAMMPS on 100-200 processors for hundreds of hours. LAMMPS has three characteristics typical of most production computational science codes: a large and complex code base, large distributed data structures, and support for file-based checkpoint and recovery. The first two characteristics are a challenge for resizing LAMMPS jobs. However, file-based checkpointing offers a simple but effective way to resize LAMMPS using ReSHAPE. We describe the changes required in the LAMMPS source to use it with ReSHAPE. Experimental results show that resizing significantly improves overall system utilization as well as the performance of an individual LAMMPS job.

Recent research has focused on dynamic reconfiguration of applications in a grid environment [7,8]. These frameworks aim at improving the resources assigned to an application by replacement rather than increasing or decreasing the number of resources. Vadhiyar and Dongarra [9] apply a user-level checkpointing technique to reconfigure applications for the Grid. During reconfiguration, the application writes a checkpoint file. After the application has been migrated to the new set of resources, the checkpointed information is read and redistributed across the new processor set. The DRMS framework proposed by Moreira and Naik [10] also uses file-based data redistribution to redistribute data across a reconfigured processor set. Cirne and Berman [11] describe an application-aware job scheduler for reconfiguring moldable applications. The scheduler requires a user to specify legal processor partition sizes ahead of time.

The remainder of the paper is organized as follows. Section 2 introduces ReSHAPE and describes the modifications required to use LAMMPS with ReSHAPE. We summarize experimental results in Section 3. Section 4 summarizes and concludes the paper.

2 ReSHAPE Applied to LAMMPS

2.1 ReSHAPE Framework

The architecture of the ReSHAPE framework consists of two main components. The first component is an application scheduling and monitoring module which schedules and monitors jobs and gathers performance data in order to make resizing decisions based on scheduling policies, application performance, available system resources, and the state of other running and enqueued jobs. The second component of the framework consists of a programming model for resizing applications. This includes a resizing library and an API for applications to communicate with the scheduler to send performance data and actuate resizing decisions. The resizing library includes algorithms for mapping processor topologies and redistributing data from one processor topology to another. ReSHAPE targets applications that are *homogeneous* in two important ways. First, the approach is best suited to applications where data and computations are relatively uniformly distributed across processors. Second, at a high-level the application should be iterative, with the amount of computation done in each iteration being roughly the same. While these assumptions do not hold for all cluster jobs, they do hold for a significant number of large-scale scientific simulations. A more detailed discussion on ReSHAPE is available in [1,3].

2.2 Extending LAMMPS for Resizing

LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator) is a classical molecular dynamics code which models particles interacting in a liquid, solid, or gaseous state. Like many computational science codes, LAMMPS uses domain-decomposition to partition the simulation domain into 3D sub-domains, one of which is assigned to each processor. LAMMPS is a good candidate for ReSHAPE-enabled resizing since it scales well and includes an outer loop executed many thousands of times, with a relatively constant amount of work done during each iteration of that outer loop. The point at which an iteration of this outer loop finishes is a natural candidate for a *resize* point, i.e., a point at which the code should contact the ReSHAPE scheduler to potentially be resized. To explore the potential of using LAMMPS with ReSHAPE, we modified the LAMMPS source to insert ReSHAPE API calls. By leveraging the existing checkpoint-recovery capabilities of LAMMPS, we can extend the code to support resizing with only a few small changes, described next.

LAMMPS reads an input script to set problem-specific parameter values and runs the simulation. A similar script is used to restart a computation using a restart file. Figure 1 shows a sample ReSHAPE-instrumented LAMMPS input script and restart script, with our changes marked in bold. These input files would be very familiar to a LAMMPS user. Only two additional commands—*reshapeinit* and *reshape*—are needed to support resizing. The LAMMPS command parser must be extended to recognize these two commands. The only other modifications to the existing LAMMPS code base are replacing all occurrences of MPI_COMM_WORLD with RESHAPE_COMM_-WORLD, and including an additional call to the ReSHAPE initialization method in the LAMMPS main program, executed only by newly spawned processes. The *reshapeinit* command in the input script causes ReSHAPE's initialization method to be called. The *reshape* command takes two arguments: the total number of iterations to be run and a restart file name. The existing LAMMPS *restart* command is used to indicate the number of iterations after which a checkpoint file will be generated. We use these restart points as potential resize points for ReSHAPE. In particular, we use the *run* command to specify the number of iterations to execute before stopping. This value must be the same as the value given in the *restart* command, e.g., 1000 in Figure 1. In this way, LAMMPS executes a predetermined number of iterations, generates a restart file, contacts the

```
# bulk Cu lattice
units           metal
atom_style      atomic
lattice         fcc 3.615
region          box block 0 20 0 20 0 20
create_box      1 box
create_atoms    1 box
pair_style      eam
pair_coeff      1 1 Cu_u3.eam
velocity        all create 1600.0 376847 loop geom
neighbor        1.0 bin
neigh_modify    every 1 delay 5 check yes
fix             1 all nve
timestep        0.005
thermo          50
restart         1000 restart.Metal
reshapeinit
run             1000
reshape         13000 restart.Metal
```

```
pair_coeff      1 1 Cu_u3.eam
fix             1 all nve
thermo          50
restart         1000 restart.Metal
run             1000
reshape         13000 restart.Metal
```

Fig. 1. LAMMPS input script (left) and restart script (right) extended for ReSHAPE

ReSHAPE scheduler, and then depending on the response from the scheduler, either does a restart on the current processor set or on some larger or smaller processor set.

The *reshape* command executes a new method which uses the ReSHAPE API to communicate with the ReSHAPE scheduler. Depending on the scheduler's decision, the resizing library expands, contracts or maintains the processor size for an application. The application clears its old simulation box and re-initializes the system with the new processor size. A new method is implemented to rebuild the process universe each time the processor set size changes after resizing. The application re-initializes the system with new parametric values from the restart file and resumes execution. All the newly spawned processes initialize ReSHAPE before receiving the restart information from processor rank 0.

3 Experimental Results and Discussions

This section presents experimental results to demonstrate the potential of dynamic resizing for MD simulation. The experiments were conducted on 50 nodes of Virginia Tech's System X. Each node has two 2.3 GHz PowerPC 970 processors and 4GB of main memory. Message passing was done using OpenMPI [12] over an Infiniband interconnection network.

We present results from two sets of experiments. The first set focuses on benefits of dynamic resizing for individual MD applications to improve their execution turn-around time; the second set looks at the improvements in overall cluster utilization and throughput which result when multiple static and resizable applications are executing concurrently. In our experiments, we use a total of five different applications—LAMMPS plus four applications from the NAS parallel benchmark suite [13]: CG, FT, IS, LU. We use class A and class B problem sizes for each NAS benchmark, for a total of nine different jobs. For LAMMPS we use an EAM metallic solid benchmark problem. The problem computes the potentials among the metallic copper atoms using an embedded atom potential method (EAM) [14]. It uses NVE time integration with a force cutoff of 4.95 Angstroms and has 45 neighbors per atom. The different problem sizes used for the MD benchmark are listed in Figure 2. We generate a workload of 17 jobs from these

Table 1. Job workloads and descriptions

(a) Experiment workloads

Workload	nApps	Applications
W1	17	LMP2048 (1), IS-B (1), FT-A (4), CG-A (3), CG-B (3), IS-A (2), LU-A (1), LU-B (2)
W2	17	LMP256 (2), LMP2048 (1), LMP864 (3), LU-B (1), FT-A (2), FT-B (1), IS-A (1), CG-B (2), CG-A (2), LU-A (1), IS-B (1)

(b) Application description

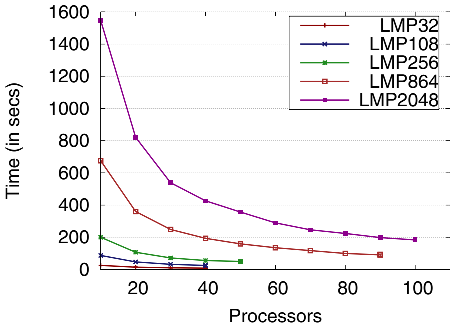
App Name	nProcs	nIters
LMP2048, LMP864	30	16000
LMP256	20	16000
CG-A	16	1000
CG-B	32	40
IS-A, FT-A	8	200
FT-B	32	100
IS-B	16	200
LU-A	64	200
LU-B	32	10

nine job instances, with each newly arriving job randomly selected with equal probability. Table 1(a) lists the different workloads used in our experiments. The number listed in parenthesis for each job is the number of jobs for each application in the job trace for that particular workload. All the LAMMPS jobs execute for 16000 iterations and the timing results are recorded after every 1000 iterations. The jobs reach their resize points after every 1000 iterations. All NAS benchmark applications are configured to expand only in powers-of-2 processor sizes, i.e., 2, 4, 8, 16, 32 and 64. The starting processor size and the number of iterations for each job are listed in Table 1(b). The arrival time for each job in the workload is randomly determined using a uniform distribution between 50 and 650 seconds.

3.1 Performance Benefit for Individual MD Applications

Although in practice it is not always possible to run a single application on an entire cluster, it is not uncommon to have a large number of processors available at some point during a long running application. These processors can be alloted to a running application, so as to probe for a processor configuration beyond which adding more processors will not benefit the application's performance, i.e., to look for a 'sweet-spot' processor allocation for that job. ReSHAPE uses this technique to probe for sweet spots for resizable applications. It uses an application's past performance results, and a simple performance model, to predict whether the application will benefit from additional processors. Based on the prediction, the scheduler decides whether an application should expand, contract or maintain its current processor size.

To get an idea of the potential of sweet spot detection, consider the data in Figure 2, which shows the performance of the LAMMPS benchmark with various problem sizes at different processor configurations. All jobs start with 10 processors and gradually add more processors at every resize point as long as the expand potential [2] of an application remains greater than the fixed threshold performance potential. The threshold value of the performance potential can vary based on the scheduling policy implemented in the system. We observe that the performance benefits due to resizing for small jobs (LMP32 and LMP108) are small; these jobs reach their sweet spots at only 40 processors. As expected, jobs with larger problem sizes show greater performance benefit



Job	Number of atoms
LMP32	32000
LMP108	108000
LMP256	256000
LMP864	864000
LMP2048	2048000

Fig. 2. Identifying the execution sweet spot for LAMMPS. The table shows the number of atoms for different LAMMPS problem sizes.

Table 2. Performance improvement in LAMMPS due to resizing. Time in secs.

Number of Atoms	nResize	Iteration time		Improvement		Overhead	
		Static	ReSHAPE	Time	%	Time	%
32000	3	330.98	168	162.98	49.24	29.31	17.98
108000	3	1132.56	443	689.56	60.72	29.42	4.27
256000	4	2601.30	909	1692.30	65.05	48.25	2.85
864000	8	8778.02	2529	6249.02	71.18	145.68	2.33
2048000	9	20100.86	5498	14602.00	72.64	198.51	1.36

with more processors. For example, LMP864 reaches its sweet spot at 90 processors. LMP2048 shows a performance improvement of 11.6% when expanding from 90 to 100 processors and has the potential to expand beyond 100 processors. The jobs continue to execute at their sweet spot configuration till they finish execution. Table 2 compares the improvement in performance due to resizing for LAMMPS jobs for different problem sizes with the associated overhead. For jobs with smaller problem sizes, the cost of spawning new processors and redistributing data contributes a significant percentage of the total overhead. By reducing the frequency of resize points for smaller jobs, the overhead can be outweighed by performance improvements over multiple iterations at the new processor size. For a production science code such as LAMMPS, ReSHAPE leverages the existing checkpoint-restart mechanism and uses the restart files to redistribute data after resizing. The table shows the number of resize operations performed for each problem size. We observe that as the problem size increases, LAMMPS benefits more from resizing, with relatively low overhead cost. For example, as the problem size increased from 32000 to 2048000 atoms, the performance improvement increased from 49.24% to 72.64% whereas the redistribution overhead decreased from 17.98% to 1.36%. The performance improvements listed in Table 2 include the resizing overhead.

3.2 Performance Benefits for MD Applications in Typical Workloads

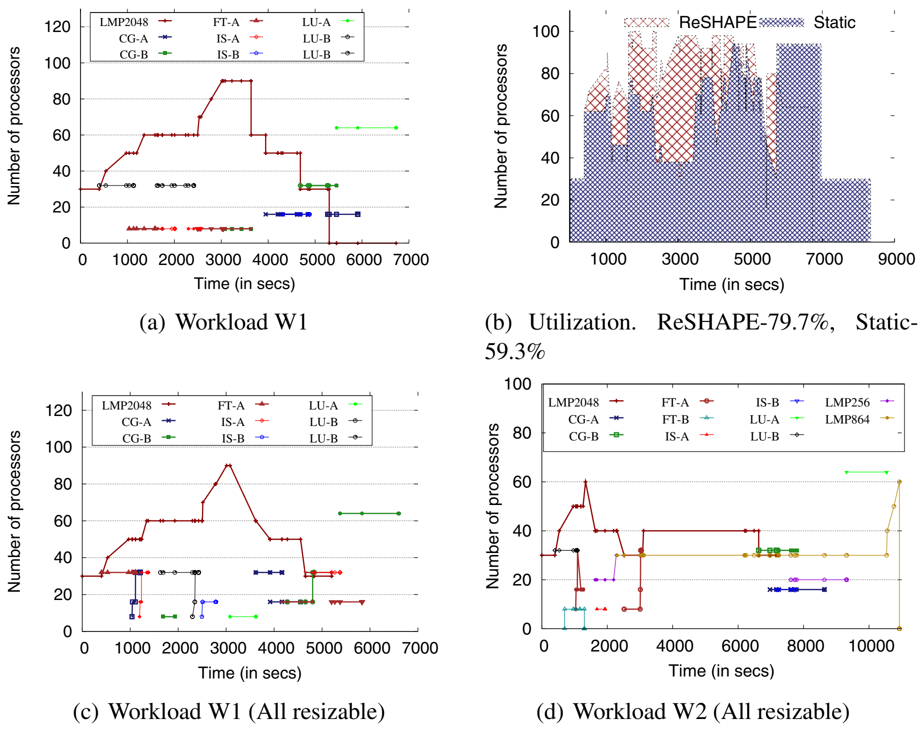
The second set of experiments involves concurrent scheduling of a variety of jobs on a cluster using the ReSHAPE framework. We can safely assume that when multiple jobs are concurrently scheduled in a system and are competing for resources, not all jobs can adaptively discover and run at their sweet spot for the entirety of their execution. In this experiment, we use a *FCFS + least-impact* policy to schedule and resize jobs on the cluster. Under this policy, arriving jobs are scheduled on a first-come, first-served basis. If there are not enough processors available to schedule a waiting job, the scheduler tries to contract one or more running jobs to accommodate the queued job. Jobs are selected for contraction if the ReSHAPE performance monitor predicts those jobs will suffer minimal performance degradation by being contracted. More sophisticated policies are available in ReSHAPE and are discussed in detail in [2].

We illustrate the performance benefits for a LAMMPS job using three different scenarios. The first scenario uses workload W1 and schedules a single resizable LAMMPS job with 16 static jobs. The second scenario also uses W1 but schedules all jobs as resizable jobs. The third scenario uses workload W2 to schedule 6 instances of LAMMPS with 11 instances of CG, FT, IS and LU as resizable jobs.

Figure 3(a) shows the processor allocation history for the first scenario. LMP2048 starts execution at t=0 seconds with 30 processors. At each resize point, LMP2048 tries to expand its processor size by 10 processors if there are no queued jobs. The granularity with which a resizable application expands at its resize point is set as a configuration parameter in ReSHAPE. At t=3017 seconds, LMP2048 expands to 90 processors and maintains that size until t=3636 seconds when a new static job, CG-B, arrives with a processor request of 32 processors. The scheduler immediately contracts LMP2048 at its next resize point to 60 processors to accommodate the new job. At t=3946 seconds, LMP2048 further contracts to 50 processors to accommodate a CG-A application. Finally, LMP2048 reduces to its starting processor size to accommodate another CG-B application at t=4686 seconds. Due to the lack of additional processors, LMP2048 maintains its starting processor size till it finishes its execution. Figure 3(b) compares system utilization using ReSHAPE with static scheduling. Static scheduling requires a total of 8837 seconds to complete the execution for all the jobs whereas ReSHAPE finishes the execution in 6728 seconds. An improvement of 20.4% in system utilization due to dynamic resizing translates into a 36.4% improvement in turn-around time for LMP2048.

Figure 3(c) shows the processor allocation history for the second scenario where all the jobs are resizable. Similar to the first scenario, LMP2048 starts execution at t=0 seconds and gradually grows to 90 processors. Due to contention for resources among jobs, all CG, LU, FT-B and IS-B jobs run at their starting processor configuration. IS-A and FT-A are short running jobs and hence they are able to resize quickly and grow up to 32 processors. Although LMP2048 is a long running job, it is able to expand because of its small and flexible processor reconfiguration requirement at its resize point. LMP2048 contracts to 60 processors at t=3618 seconds and further to 50 processors at t=3916 seconds to accommodate two CG applications. It contracts to its starting processor size at t=4655 seconds and maintains the size till it finishes execution. For the same workload W1, ReSHAPE finishes the execution of all the jobs in 6588 seconds with an overall system utilization of 75.3%.The turn-around time for LMP2048 improves by 36.8%.

Figure 3(d) illustrates a third job mix scenario where multiple instances of the LAMMPS application and applications from the NAS benchmark suite are scheduled



(a) Workload W1

(b) Utilization. ReSHAPE-79.7%, Static-59.3%

(c) Workload W1 (All resizable)

(d) Workload W2 (All resizable)

Fig. 3. Processor allocation and overall system utilization for a job mix of static and resizable LAMMPS jobs

Table 3. Performance results of LAMMPS jobs for jobmix experiment. Time in secs.

	LAMMPS jobs	Job completion time			
		ReSHAPE	Static	Improvement	%
Scenario 1	LMP2048	5301	8337	3036	36.4
Scenario 2	LMP2048	5268	8337	3069	36.8
Scenario 3	LMP2048	7239	8469	1257	15.0
	LMP864	3904	3726	-178	-5.0
	LMP864	3892	3763	-129	-3.0
	LMP864	3756	3744	-12	-0.3
	LMP256	1370	1592	222	14.0
	LMP256	1704	1594	-110	-7.0

together as resizable applications. LMP2048 increases from 30 to 60 processors at 1337 seconds and contracts immediately to its starting processor size to allow scheduling of other queued jobs. LMP2048 expands again and maintains its processor size at 40 till it finishes execution. LMP256 expands from its starting processor configuration to 30 processors and maintains its size at 30 till completion. LMP864 starts its execution at t=7173 seconds, and executes at its starting processor size due to unavailability of additional processors. It expands to 40 and 50 processors at t=10556 seconds and t=10760 seconds, respectively, and finishes with 60 processors at t=10929 seconds. The remaining LAMMPS jobs execute at their starting processor configuration. Static scheduling finishes the execution of W3 in 10997 seconds whereas ReSHAPE requires only 10929 seconds. With ReSHAPE, the overall utilization is 88.1%, an improvement of 9.3% compared to static scheduling.

Table 3 summarizes the performance improvements for LAMMPS jobs in all three scenarios. The performance of LMP2048 improved by 36.4% and 36.8% in scenarios 1 and 2, respectively, whereas in scenario 3 the performance improved by only 15%. We observe that LMP864 and LMP256 jobs perform better with static scheduling than with ReSHAPE. The degradation in performance is due to the data redistribution overhead incurred by the application at each resize point. In the third scenario, all three LMP864 and one LMP256 jobs execute either at their starting processor configuration or close to it. In our current implementation, LAMMPS considers each run to the resize point as an independent execution step. It requires a restart file to resume the execution after resizing. The restart file is required even when the application does not resize at its resize point. Thus, an additional overhead cost is incurred in writing and reading the restart files resulting in performance degradation. LMP256 suffers a maximum performance degradation of 7% whereas LMP864 loses performance by 5% compared to static scheduling. However, note that the statically scheduled LAMMPS jobs did not write any restart files, which in practice is unusual.

4 Conclusion

In this paper we describe the steps required to resize a well known molecular dynamics application and evaluate resulting performance benefits. We present a case study using LAMMPS and identify the minor modifications required to execute it with ReSHAPE. Experimental results show that dynamic resizing significantly improves LAMMPS execution turn-around time as well as overall cluster utilization under typical workloads.

Although data redistribution using file-based checkpointing is expensive (compared with message-passing based redistribution available for some common distributed data structures [3]), we find that it is a realistic approach for production scientific codes such as LAMMPS. This approach takes advantage of checkpoint and recovery mechanisms already available in the code; deep understanding of distributed data structures is not required. The extra overhead is relatively modest for long-running applications as long as resizing is not done too often, and in fact, may not be any added cost at all since the usual (statically scheduled) case would write checkpoint files periodically anyway. We also note that this approach requires no changes to the ReSHAPE framework. The ReSHAPE runtime environment treats resizable LAMMPS jobs like any other resizable job, irrespective of how data redistribution is accomplished.

References

1. Sudarsan, R., Ribbens, C.: ReSHAPE: A Framework for Dynamic Resizing and Scheduling of Homogeneous Applications in a Parallel Environment. In: Proceedings of the 2007 International Conference on Parallel Processing, Xian, China, pp. 44–54 (2007)

2. Sudarsan, R., Ribbens, C.: Scheduling Resizable Parallel Applications. In: Proceedings of 23rd IEEE International Parallel and Distributed Processing Symposium (to appear)
3. Sudarsan, R., Ribbens, C.: Efficient Multidimensional Data Redistribution for Resizable Parallel Computations. In: Proceedings of the International Symposium of Parallel and Distributed Processing and Applications, Niagara falls, ON, Canada, pp. 182–194 (2007)
4. Plimpton, S.: Fast parallel algorithms for short-range molecular dynamics. Journal of Computational Physics 117, 1–19 (1995)
5. Plimpton, S., Pollock, R., Stevens, M.: Particle-Mesh Ewald and rRESPA for Parallel Molecular Dynamics Simulations. In: Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing, pp. 8–21 (1997)
6. Farkas, D., Mohanty, S., Monk, J.: Linear grain growth kinetics and rotation in nanocrystalline ni. Physical Review Letters 98, 165502 (2007)
7. Huedo, E., Montero, R., Llorente, I.: A Framework for Adaptive Execution in Grids. Software Practice and Experience 34, 631–651 (2004)
8. Buisson, J., Sonmez, O., Mohamed, H., Lammers, W., Epema, D.: Scheduling malleable applications in multicluster systems. In: Proceedings of the 2007 IEEE International Conference on Cluster Computing, Austin, USA, pp. 372–381 (2007)
9. Vadhiyar, S., Dongarra, J.: SRS - A framework for developing malleable and migratable parallel applications for distributed systems. Parallel Processing Letters 13, 291–312 (2003)
10. Moriera, J.E., Naik, V.K.: Dynamic resource management on distributed systems using reconfigurable applications. IBM Journal of Research and Development 41, 303–330 (1997)
11. Cirne, W., Berman, F.: Using Moldability to Improve the Performance of Supercomputer Jobs. Journal of Parallel and Distributed Computing 62, 1571–1602 (2002)
12. Open MPI v1.3 (2009), `http://www.open-mpi.org`
13. Van der Wijngaart, R., Wong, P.: NAS Parallel Benchmarks Version 2.4. NASA Ames Research Center: NAS Technical Report NAS-02-007 (2002)
14. Daw, M.S., Baskes, M.I.: Embedded-atom method: Derivation and application to impurities, surfaces, and other defects in metals. Phys. Rev. B 29, 6443–6453 (1984)