Automatic distributed workflow generation with GridMD library

I.V. Morozov a, b, *, I.A. Valuev a

a Joint Institute for High Temperatures of Russian Academy of Sciences, Izhorskaya 13, build. 2, Moscow 145412, Russia
b Moscow Institute of Physics and Technology, Institutskii per., 9, Dolgoprudny 141700, Moscow Region, Russia

Article info

ABSTRACT

GridMD is a C++ class library intended for constructing simulation applications and running them in distributed environments. The library abstracts away from details of distributed environments, so that almost no knowledge of distributed computing is required from a physicist working with the library. She or he just uses GridMD function calls inside the application C++ code to perform parameter sweeps or other tasks that can be distributed at run-time. In this paper we briefly review the GridMD architecture. We also describe the job manager component which submits jobs to a remote system. The C++ source code of our PBS job manager may be used as a standalone tool and it is freely available as well as the full library source code. As illustrative examples we use simple expression evaluation codes and the real application of Coulomb cluster explosion simulation by Molecular Dynamics.

Available online 20 January 2011

Accepted 11 January 2011

Received in revised form 7 January 2011

Article history:

© 2011 Elsevier B.V. All rights reserved.

1. Introduction

This paper describes the architecture details of a high-level C++ workflow library called GridMD. The abbreviation GridMD stands for Grid-enabled Molecular Dynamics, however the part of the library responsible for workflow definitions may be used in all kinds of numerical simulation, not limited to the Molecular Dynamics (MD) method. The problem of the workflow definition for large scientific applications is widely discussed [1–7]. For an application that can be split into separate stages logically linked between each other, one needs a scenario to specify the correct execution order and the dataflow. If the stages depend on each other only through input and output data, the application can be efficiently run in a distributed environment. Some workflow specifications from the application designer. Usually this is done by the explicit indication of the application subtasks as separate executable services and of the data flow between these subtasks [1, 3–5].

In many important cases workflow generation for an algorithm can be partly or fully automated, thus hiding implementation details from a programmer. The idea of using some standard code execution patterns is explored by the algorithmic skeletons community [8–11]. The approach of encapsulating workflow definitions into the application source code is also a basis of the GridMD architecture. Some algorithmic skeletons, for example Fork, are implemented in GridMD.

It should be noted that each stage of a distributed application is meant as a time consuming operation where the time required for the data transfer is negligible. It makes the application suitable for distributed environments with slow networks. In contrast to rather general standard algorithmic skeletons [11], GridMD offers specialized design patterns for the distributed parallelism. It is possible however to use general skeletons or basic libraries (MPI, OpenMP) in the code at the level of a single stage (subtask) execution.

In order to be specific we consider the special class of scientific applications: atomistic simulations by the methods of Molecular Dynamics (MD) and Monte Carlo (MC). In the simplest case MD and MC applications may have very straightforward scenarios, such as to strat from a system of particles in some initial state and to propagate it through a chain of other states solving Newton’s equations of particle motion or performing the temperature-conditioned Metropolis random process. This functionality is available in many MD/MC packages [12,13] having serial or parallel realization. However, the need for higher level processing like statistical averaging [14], parameter sweep, optimal parameter search, etc., makes the simulations very computationally demanding. Many of these problems can be efficiently solved by distributed computing.

The approach described in this paper can help programmers to easily create new or modify old serial programs by including distributed elements. It uses the traditional programming (C++), no special languages, code preprocessing, manual scripting tools or even the knowledge of Grid functionality and middleware are required to describe the GridMD workflow. From the existing Grid instrumentation tools [1,3–6], the Grid Superscalar [6] project follows the approach most similar to the one of GridMD [7,15]. Since there are significant differences, we will briefly indicate and dis-
cuss them in the following sections. In Sections 2–5 we describe the GridMD architecture using simple examples. Then we review the real MD application in Section 6.

2. GridMD scenarios and workflow graphs

In this section we illustrate the concepts of the GridMD workflow. Let us start from a serial application which may be logically viewed as a sequence of certain actions. We will denote these actions as nodes. The node is a portion of the application code and the data change associated with it. For example, if we want to split a loop into nodes, there will be different nodes representing each iteration rather than a single node ‘repeated’ several times. Data dependencies between nodes may be represented as directed links (edges of a graph). So, if the node A needs data from the node B there will be the edge from A to B. For simplicity, the edge data is assumed available when the edge source node is finished. Suppose we can choose the order of node execution, then the whole application run may be represented as a directed graph. The correct order of node execution is fully determined by the graph edges: a node may be executed if and only if all ‘parent’ nodes (those having outgoing edges to the node of interest) are finished.

In some cases the links between nodes can be explicitly formalized, so that all data attributed to a link can be packed into a file by some known procedure and transferred from one computer to another. We will call these edges data links and represent them as dashed lines graphically. In contrast, all other edges, for which the data transfer is not formalized and performed implicitly via internal variables will be called hard links and represented as solid lines (Fig. 1). If the node B has an incoming hard link from the node A, then B may be finished if and only if B is performed after A by the same process. This rule for hard links is a consequence of the implicit data transfer taking place when the program passes from A to B.

If the time required for the data transfer due to data links is small enough compared to the computational time of connected nodes, then the application is a good candidate for distributed execution. When the nodes are well separable by data links, portions of the application (nodes) may be concurrently run at different remote computational resources. GridMD is aimed at designing distributed applications by means of the execution graph construction, analysis and (possibly remote) execution of its nodes.

There is another basic idea of GridMD which makes it different from other workflow systems. GridMD uses the same code for designing and linking the graph elements, executing individual nodes and managing the job submission process. This makes GridMD applications very lightweight and compact, written in a single language (C++) and also highly portable. Software requirements for GridMD are quite modest: it depends on two free portable C++ libraries (wxWidgets and boost graph library [16]); for remote cross-platform execution the same application must be compiled for each of the available resources; when the managing GridMD application runs on the Windows system, an SSH-client for Windows (typically Putty [17]) is required to connect to remote compute resources. The compute servers may run a queueing system like PBS or be in a Globus-managed grid.

Let us briefly review the current version of the GridMD application programming interface. All distributed actions of a GridMD application must be placed inside the function int gridmd_main(int argc, char **argv) { There are two function calls that mark the beginning and the end of distributed sections in gridmd_main(): begin_distributed() and end_distributed(). The sections should not overlap. All graph nodes should be defined inside these sections.

In the following we discuss the node construction and execution mechanism of GridMD. Alternatively to defining a separate

Listing 1. Example for the basic GridMD functions.

```c
int gridmd_main(int argc, char* argv[]) {

    typedef double arg_t;  // argument type
    typedef double val_t;   // data type for the data transfer

    val_t (*A)(arg_t)=sin, (*B)(arg_t)=cos;  // test functions

    val_t arg_t=0.3;

    typedef double val_t;  // result type
    typedef double arg_t;  // argument type

    gmExperiment.init(argc,argv);  // initialization
    begin_distributed();           // 'start' node creation

    // Node 'A' with implicit hard link to the previous 'start' node
    if(node("A"))                   // node construction
        node_output<val_t>()=A(arg_t); // calculation of A(X)

    // Node 'B' with explicit hard link to 'start'
    if(node("B","get_currnode()"))
        node_output<val_t>()=B(x);    // calculation of B(X)

    // Node 'C=A*B' with val_t data links from A and B
    if(node("C=A*B","[A|B]","gmDataLink<val_t>()"))
        node_output<val_t>()=node_input<val_t>(0)*node_input<val_t>(1);

    // Node 'D=A+B' with val_t data link from A and B
    if(node("D=A+B","[A|B]","gmDataLink<val_t>()"))
        node_output<val_t>()=node_input<val_t>(0)+node_input<val_t>(1);

    // Node 'E=C/D' has val_t data links from 'C=A*B' and 'D=A*B'
    if(node("E=C/D","(C.*|D.*)","gmDataLink<val_t>()"))
        node_output<val_t>()=node_input<val_t>(0)/node_input<val_t>(1);

    // Finishes distributed section by adding final node and
    // linking it with nodes having no output links (actually 'D=C/D').
    end_distributed("finish",gmNODE_NOOUTPUTS,gmDataLink<val_t>())
    printf("The result is %g\n",result);
    return 0;
}
```

Fig. 1. Execution graph automatically generated from the code presented in Listing 1. Hard links are represented by solid lines with arrows, data links by dashed lines with names of transferred files indicated. The numbers are node unique identifiers which are assigned to the nodes in the order of creation. Visualized by graphviz [18].
processing procedure for each node, we again start from the viewpoint of a serial application. In such application all nodes appear in a very definite order, so that we may use this ‘serial’ order to assemble the execution graph. To understand the node construction and execution process one can imagine that every distributed section of the GridMD application is executed at least twice: first, in the so-called construction mode and, second, in the worker mode.

The main aim of the construction mode is to list all nodes and to link them in a graph, bypassing actual nodal computations. This mode is switched on automatically after the call to begin_distributed(). The creation of a simple node is accomplished by the node() function call. The arguments of node() may contain information on the linking of the newly created node to the previously created ones. The linking (adding hard or data link to the graph) may also be performed after the node creation using the link() function call. The nodes are added one by one to the graph which starts with the obligatory ‘start’ node (added automatically by begin_distributed() function) and ends with the final node (added by end_distributed()). As a node is added, the global node Id counter, accessible by get_cur_node() in both the construction and the worker modes, is increased.

The worker mode is invoked when the GridMD engine starts the application on a compute resource with the command line parameter -w followed by a node list for processing. The same can be manually done by the user for testing. The aim of this mode is to perform computations associated with the selected nodes. To provide a mechanism of node selection from a single code, the node() functions return nonzero only in the worker mode and only when the node with the current node Id is requested for processing (listed in the command line). If one needs to separate the node definition and the node processing (i.e. if there are additional links to the current node added after node() call), the process_cur_node() function may be used to test the current node status. It returns the same value as node().

As a simple methodological example, let us consider a distributed evaluation of the expression \( R(x) := A(x) \ast B(x) / (A(x) + B(x)) \) for a given \( x \), where \( A(x) \) and \( B(x) \) are unary functions with arbitrary argument and return (value) data types and \(+, \ast, /, -\) operators defined for the value type. In the code example for simplicity we use double for both the argument and the value types and \( A \equiv \sin, B \equiv \cos \) (Listing 1). The algorithm is very straightforward: first we calculate \( A \) and \( B \) concurrently, then perform the operations \( C = A + B, D = A - B \) (also concurrently) and finally calculate the result \( R = C / D \). Each action is considered as a GridMD node with the data links requiring transfers of value type arguments. Obviously, it gives an advantage over the serial computation only if the data transfer is faster than the calculations.

The resulting graph is shown in Fig. 1. The arguments to node() functions are: the node name; a node selector to create input links (by default, it indicates the previously constructed node); and the link type (hard link by default). Note that the execution graph should reflect the programmed order of execution for the nodes linked with hard links, and this is a responsibility of the application programmer. This means that the control flow of the application in both the construction and the worker modes should be the same, with all GridMD function calls proceeding in the identical order.

The node input and output data are accessed by node_input<val_t>() and node_output<val_t>() functions in the worker mode, which return a reference to an object of val_t. There can be multiple input and output data ports of a node, each having unique number and associated with a certain data type. Output port numbers are indicated when creating data links, by default a new port is created for each new output link of a node. The input ports of a node are unique for each incoming link.

To select nodes and ports for creating links in node() and link() functions the gmSelector class is used. Nodes in GridMD are uniquely identified by their integer Ids, so gmSelector may be constructed from a plain integer. It is also constructible from a regular string expression (according to the standard regexp rules) listing all matching node names. Port numbers may be listed by adding the postfix \(<N>\) for the output port and \(>N\) for the input port with the number \( N \) to the selector expression. Note that there is no requirement from GridMD for the string node names to be unique. The advantage of this approach is that the same distributed portions of the code may be inserted at different locations of the source code without renaming all nodes. To deal with multiple identical names there is a default rule for gmSelector: between the nodes matching the selector expression and having the same string name, the one closest in Id to the current node is selected. This behavior may be changed however if one really needs the nodes with identical names in the selection. Another form of gmSelector is a pre-defined construct reflecting some frequently used patterns of linking. For example, gmNODE_PREV means the node preceding the current one and gmNODE_NOOUTPUTS lists all nodes having no output links.

The node and port numbers are used to direct a particular output from/to a node. When the linked nodes are executed by different application instances (locally or remotely) the data is transferred via creation of files. For example, the names of the output file (written by a source node) and the input file (read by a target node) for a link are depicted near the edges of the graph in Fig. 1. Note that a pair of files related to the same link contains the same data. File name change or copy is managed by GridMD.

The template argument val_t of node_input<val_t>() and node_output<val_t>() functions is the data type which has to be packed to a file for transfer. There are two global template functions dump_to_file<val_t>(FILE *, ...) and restore_from_file<val_t>(FILE *, ...) responsible for the file input and output. They are implemented by GridMD for simple plain data types and have to be specialized when the programmer wants to add his own complicated data type. Some nodes of GridMD may be executed locally (as a result of scheduling policy or by an explicit request), i.e. by the same application that executes the construction mode. If both linked nodes are local, no file creation is necessary and the data transfer is performed using memory.

Type matching between the arguments of node_input(), node_output() and the corresponding linked ports may only be checked at runtime, because these functions strictly belong to the worker part of the code. An error message is given in the worker mode if the data types do not match. For typed GridMD skeletons, like gmFork described in Section 5, the type check is implicitly performed by a C++ compiler. It increases type safety of the code and therefore the use of template skeletons is generally preferred to the explicit node creation.

There are two important subtypes of the GridMD data link: a file link and a status link. The first (gmFLink) is useful when the programmer wants to transfer his files without wrapping them into C++ data types. This type of data link is not bound to any of nodal input/output ports and the files must be created explicitly by the programmer. A status link (gmStatusLink) declares the precedence of node execution without requiring any data to be actually transferred. This type of link is useful for programs with intended side effects, not monitored by GridMD, for example, updating a server data base by a nodal computation.

At this stage we would like to compare the GridMD workflow functions with the mechanism used in the Grid Superscalar [6] project. It follows a similar approach where the program code is
used to produce a client binary, running on the user’s computer and distributing program subtasks (nodes) and server binaries running on remote computer(s) and calculating the subtasks. In Grid Superscalar all nodes are wrapped into functions having the input and output files as their arguments. These files are analogous to GridMD gmFileLinks, other link types are not supported. The programmer is responsible for proper creation of the input files at the client side, reading them and creating output files at the server side. The information about subtask functions of Grid Superscalar is placed in a definitions file, requiring a special pre-processor at compilation stage, when separate client and server versions of the code are generated. GridMD uses the same code both for the client and the server and requires a standard C++ compiler only. It assumes however recursive calls to gridmd_main function at the client side. Compared to rather general dependencies between subtasks allowed by GridMD, Grid Superscalar supports only the data file dependencies. The task graph is used by Grid Superscalar when determining the node precedence but remains hidden from the application programmer. We believe that portability, explicit use of the execution graph, flexible node markup mechanism, support for various link types and implicit data transfer are the main advantages of GridMD. Grid Superscalar allows the programmer to specify various task constraints and costs. This feature is not yet fully implemented in GridMD except for binding the nodes to specific compute resources. Extending GridMD with the load balancing and cost evaluation techniques is however planned for the future.

3. Extracting jobs from a scenario

After the execution graph is created, the following iterative algorithm is used to determine which branches (subgraphs) may be used concurrently:

1. Specify ‘start’ root node as ‘processed’ and all other nodes as unprocessed.
2. Specify all nodes having incoming data links with unprocessed sources as ‘blocked’. Exit with deadlock indication if the number of blocked nodes is nonzero and did not change since the previous iteration.
3. Specify all nodes connected with blocked nodes by recursively following outgoing hard links as blocked.
4. Find the maximal set of unprocessed nonblocked nodes connected with processed nodes by recursively following all outgoing links.
5. Find all nodes from the set of step 3 that have outgoing data links to blocked nodes, have no output links or coincide with the final graph node.
6. For each node found in step 4 find the maximal subgraph connected with it by incoming hard links.
7. For each subgraph of step 5 list all input files corresponding to incoming data links entering the subgraph.
9. Execute the worker tasks of step 7, collect all files corresponding to outgoing data links of the subgraphs’ nodes (or put these tasks in the submission script). Wait until at least one of the subgraph tasks finishes.
10. Mark all nodes in all finished subgraphs of step 7 as ‘processed’.
11. Finish if there are no unprocessed nodes, otherwise go to step 2.

We use the standard graph analysis techniques from the boost graph library [16] to implement this algorithm. Note that the submitted combination of minimal worker subgraphs formed at step 7 may be a subject to optimization and workload balancing which are planned for the future GridMD releases.

Applied to the graph shown in Fig. 1, the graph analysis algorithm will produce the following results. At first, two concurrent sets of nodes are scheduled for execution: (0,1) and (0,2). Note that the sets have an overlapping element 0 representing the hard-linked ‘start’ node responsible for initialization. The original GridMD application then sleeps till the sets are finished. When all submitted sets are finished, the nodes 3 and 4 are scheduled concurrently, then the node 5. The node 6 transferring the result to the local machine, finalizes the distributed section. Note that the final nodes of distributed sections are always performed locally (by the same process that originally started the GridMD application).

An important feature of a GridMD application is the possibility to interrupt the client code when it is waiting for a job completion and to restart it by a request (specifying -r command line option). After restart, the program resumes in the construction mode retrieving the saved remote job information and continuing from its last saved state. This provides a ready to use mechanism of application checkpointing based on the execution graph information.

When working with GridMD the programmer and the user may face various types of errors. The first type are logical construction errors, for example execution graph looping. These errors are reported in construction time and the execution terminates. There are also logical errors that may be detected at the worker mode only, for example the absence of a data needed for a link. These errors are reported in course of the application run and also cause program termination. However, in most cases the data already calculated for the correct nodes may be reused after the errors are corrected and the application is restarted. Another kind of errors is connected to the interaction with remote compute resources, which may be unavailable, improperly configured or may fail. In this case GridMD makes several attempts to reschedule the jobs to another resource before reporting a failure. The error report in this case contains a complete human-readable log of interaction with the failed remote resource.

4. Job scheduling and submission

In course of execution a GridMD application submits jobs to external resource managers. Usually these jobs imply invocation of the same GridMD application in the worker mode. Deployment of the application to a remote site (source code transfer and compilation) should be performed manually in the current GridMD version (2.1). In the future releases this procedure will be automated. It is also possible to run any other application installed on a compute server as a GridMD node. This is configured by setting the corresponding GridMD node properties. In this case the programmer is responsible for proper naming and placement of the input and output files on the server side.

Job creation and submission are governed by the GridMD scheduling and job management components. One job can represent multiple graph nodes as the nodes may be hard-linked or the scheduler may pack them in a cluster to minimize file transfers. The last is a part of possible load-balancing which can be performed based on the execution graph information (planned for the future GridMD releases). The programmer can also explicitly mark a node to be bound to a particular resource or even to be executed locally.

The job management component is designed to provide interfaces to various external resource managers. Currently the interfaces exist for Portable Batch System (PBS), Grid manager of Joint Supercomputer Center of Russian Academy of Sciences [19] based on Globus 4 [20] and for the DEISA Grid environment [21] based on UNICORE [1]. In the future we plan to add interfaces for other workflow middleware such as Nimrod [4] and Kepler [5].
In order to use the specific resource manager the application user has to specify both the manager type and the communication protocol. Generally GridMD communicates with a resource manager using command line tools. It executes (locally or remotely) the conventional commands like ‘qsub’, ‘qstat’, etc. for PBS, ‘grid-*’ and ‘mpirun’ for JSCC Grid service, ‘deshl’ for DEISA shell tool. Then it analyzes their output and changes the job state accordingly or raises an error. If the resource manager is installed on the same computer where the GridMD application is running, the commands are just passed through the Unix shell. For remote execution the SSH connection is used.

The information about each external resource manager is located in the corresponding ‘resource’ section of the GridMD configuration XML file which is loaded at the application startup. The parameters for each manager include the host name, user credentials, paths to the certificate files, additional command arguments, etc. To avoid interactive authorization procedures the user is encouraged to use the key-based SSH authorization or to provide a pre-initialized Grid proxy certificate.

As jobs are created they are submitted to an external resource manager. If more than one manager is available the current GridMD version uses a simple round-robin algorithm for work distribution. It should be noted that GridMD is not meant as a replacement for job scheduling middleware. However it is expected that the future GridMD versions will provide higher level load balancing tools based on particular job properties and data links.

Typically GridMD creates and handles compute jobs automatically. However the programmer can create the jobs manually and use the job management component directly and even independently of the entire library. In this case the interfaces of gmJob and gmJobManager classes should be used.

The gmJob class object represents a single job and stores the information about input and output files and the command to be executed on the compute resource. The input and output files can be located either locally or in a persistent directory of the remote system. These files are transferred by GridMD to and from the job working directory on the remote resource automatically or upon request. Submission links the job with the resource and assigns the unique job id. Later using the gmJob class object or the job id the user can retrieve the job status, fetch the intermediate or final output files, manage files in the working directory, stop the job and wait for its completion. GridMD takes care of the text file conversion when the local and remote hosts are running different operating systems (Windows and Unix).

5. Algorithmic skeletons in GridMD

One of the concepts of GridMD enhancement is addition of higher level instrumentation tools that help to easily implement frequently used patterns for distributed computation. In this section we briefly describe the gmFork skeleton which represents code branching into several concurrent execution paths. This skeleton is also used in the real MD application described in Section 6.

An example of a distributed loop (calculating items of the ln(x) series) is shown in Listing 2 and the corresponding graph is shown in Fig. 2. gmFork skeleton consists of four categories of nodes: ‘begin’ node (where the branches fork), ‘split’ (starting node for each concurrent branch), ‘merge’ (final node for each branch), ‘end’ node (where all branches join). All node categories are marked by corresponding gmFork member functions. Appearing in two flavors: the programmer may create a new node or use the current node to serve as a fork part. The second flavor markup function names end with _here() postfix, i.e. the first node of the graph in Fig. 2 is also the ‘begin’ node of fork1. There are also three categories of links forming the node connection pattern, the data type for each link category (‘begin’ → ‘split’, ‘split’ → ‘merge’, ‘merge’ → ‘end’) is a template parameter of gmFork<i>. Hard links are represented by the void (default) type. The links are added automatically when using the skeleton markup functions. Arbitrary number of nodes may be inserted between ‘split’ and ‘merge’ nodes of each branch, thus allowing nesting of forks. These nodes are added in the usual way, for example by node() functions or nested skeletons, and in this case the link ‘split’→’merge’ is not added automatically for the branch.

One of the advantages of skeleton construction is a strict type control. Instead of using node_input() and node_output() functions which accept an arbitrary data type at compilation stage, the programmer may use properly typed data access functions of gmFork, corresponding to each branch. For example, in Listing 2 the functions fork1.vsplit_out() and fork1.vmerge_in() accept the val_t data type only.

6. Example of MD application

In this section we consider (without going into details) the real MD application as an example illustrating various aspects of GridMD functionality. This application is a numerical simulation of the Coulomb explosion of a nanosize cluster after its complete ionization by a short and intense laser pulse [22,23]. Statistical averaging over an ensemble of initial microscopic states is performed [14]. The application is organized in three phases: (a) cal-

### Listing 2. Example for the GridMD fork skeleton.

```c
begin_distributed();
// Define the skeleton and internal data link types
gmFork<void, val_t, void> fork1("loop");
fork1.begin_here(); // marks the loop ‘begin’ node

int nterms = 3; // number of terms in the series
val_t sum = 0.; // ‘sum’ accumulates the result
for(int i=0; i<nterms; i++){
    // Create a new ‘split’ node and define its output
    if(fork1.split())
        fork1.vsplit_out() = pow(x, (val_t)i) / i;
    // Define the action of the ‘merge’ node
    if(fork1.merge())
        sum += fork1.vmerge_in(); // accumulation of the terms
}
if(!end_distributed()) // loop ‘end’ node is optional
    printf("The result is \%\n\", sum);
```

![Execution graph generated from the code presented in Listing 2.](image-url)
calculation of cluster expansion trajectories (coordinates and velocities) with different initial conditions and saving them onto a disk; (b) analysis of particle motion using the saved trajectories to extract useful information such as mean energies, one- and two-time distribution functions, etc.; (c) averaging the analysis results over all trajectories (initial conditions).

Storage of the trajectories is needed as their calculation is the most time consuming operation. Then the analysis phases may be repeated for different parameters. The phases (b) and (c) may be performed even if partial trajectory data (unfinished trajectories or smaller number of them) exists. The phases (a) and (b) are performed in a distributed way by concurrent jobs, while the phase (c) is performed by a single job.

The C++ source code of the main distributed application loop is presented in Listing 3. Three working phases are represented by the subroutines (C++ functions) calculate_trajectory, process_trajectory and average_results. All application parameters are transferred from the command line to the process_params class object and then used in working subroutines. The required working phase(s) are also read from the command line parameters and then combined into bit flags switching the execution on or off in the corresponding branches. GridMD library allows one to fully specify the workflow required for the application described above. The corresponding graph is presented in Fig. 3. In the construction mode the program passes over all execution branches to encounter all possible node markup functions. File links are used to represent data dependencies.

The calculation phases (a) and (b) are best performed at the same compute resource since the files ‘traj*.t’ are very large. To use this data, generated at the phase (a), for different kinds of analysis the restart mechanism of GridMD is utilized. The phases (b) and (c) may be run as long as the first trajectory data has appeared (see ‘begin_calc’ nodes in Fig. 3). To repeat the phases (b) and (c)
using the existing data from the phase (a), the nodes of phase (b) are forced to restart each time the program is restarted by setting the corresponding node property.

7. Conclusion

In this work we described the design of GridMD workflow patterns and job management system. GridMD is a developing molecular simulation library with growing number of different tools and features. The intended users of the library are simulation code developers, who need to implement complicated numerical experiments for routine use in the applied science. Keeping the code compact, readable and portable is very important for atomistic simulations which usually consist of multiple independent runs. The workflow concepts of GridMD library can be helpful for transforming existing sequential applications into Grid-aware programs. Flexible design allows one to use different distributed execution mechanisms, also keeping the possibility to run the code on a single computer. The GridMD source code, documentation and examples are available at the web page [24].

Acknowledgements

This work is supported by the Programs of Fundamental Research of the Presidium of RAS Nos. 2, 13 and 14, Russian Federal Agency for Science and Innovations (contract No. 02.740.11.0236). I. Morozov acknowledges support from the Dynasty foundation and by the Grant of President of Russian Federation No. MK-64941.2010.8.

References