СОЗДАНИЕ РАСПРЕДЕЛЕННЫХ ПРОГРАММ МОЛЕКУЛЯРНО-ДИНАМИЧЕСКОГО МОДЕЛИРОВАНИЯ С ПОМОЩЬЮ БИБЛИОТЕКИ GRIDMD

И.В. Морозов, И.А. Валуев

Введение

В работе рассматривается архитектура и пример использования высокоуровневой библиотеки классов C++ “Grid-Enabled Molecular Dynamics” (GridMD), позволяющей включать инструкции по формулировке распределенного сценария выполнения в исходный код приложения на этапе его разработки. В первую очередь библиотека предназначена для создания программ моделирования методами молекулярной динамики (МД) и Монте-Карло (МК), однако может быть использована и для других приложений.

Проблема формулировки сценариев выполнения программ в распределенных средах (Грид) обсуждается достаточно давно [1-5]. Если вычислительная задача может быть разбита на отдельные этапы, зависящие друг от друга по входным и выходным данным, то приложения, реализующие эти этапы, могут быть запущены на различных узлах распределенной системы при условии, что будет обеспечена необходимая передача данных между узлами. Автоматизация процесса передачи данных обычно реализуется заданием потока исполнения (workflow), который формулируется на специальном языке [1] или с помощью графического интерфейса [2-5]. Однако формулировка сценария требует определенной квалификации и понимания того, как устроена распределенная вычислительная среда.

Идея создания библиотеки GridMD – обеспечить автоматическое разбиение на этапы, генерацию и обработку сценария, на основе инструкций, включаемых в исходный текст программы. Таким образом, GridMD является системой задания потоков исполнения (workflow system). Фактически структура потоков исполнения определяется добавлением в программу вызовов соответствующих функций GridMD. Следует отметить, что речь идет о достаточно крупных этапах, время выполнения которых существенно превышает расходы на передачу данных по возможно медленным каналам связи в распределенной среде. Этим развиваемый нами подход отличается от известных библиотек алгоритмических шаблонов (algorithmic skeletons) параллельного программирования [6-9]. В то же время выполнение одного этапа приложения GridMD можно дополнительно распараллелить в пределах одного узла или кластера с использованием указанных библиотек или более низкоуровневых средств (OpenMP, MPI и др.). Отметим тем не менее, что применение GridMD основано на использовании специальных конструкций в исходном коде приложения, поэтому GridMD может также быть классифицирована как система, предлагающая алгоритмические шаблоны, но ограниченные задачами распределенного исполнения.

В данной работе мы ограничимся приложениями МД/МК моделирования. Простейший сценарий для этих приложений состоит в том, чтобы: а) сгенерировать систему в начальном состоянии, б) последовательно переводить систему в другие состояния, используя ньютоновские уравнения движения (МД) или алгоритм Метрополиса (МК). Такая функциональность присутствует во всех доступных МД/МК пакетах [10,11], многие из которых имеют параллельные алгоритмы для расчета взаимодействий между частицами на каждом шаге по времени. На более высоком уровне возможен другой тип параллелизма, связанный с обработкой полученных данных о микроскопических состояниях системы для нахождения макроскопических величин. Такой обработкой может быть усреднение по микроскопическим состояниям и целым МД траекториям [12], получение зависимости от макроскопических параметров, поиск оптимальных параметров и др. Эти алгоритмы часто реализуются в виде отдельных программ на интерпретируемых языках (скриптов) и являются последовательными. В то же время указанные задачи могут эффективно решатся на распределенной вычислительной системе, а библиотека GridMD является инструментом облегчающим создание распределенных программ для разработчиков программ МД/МК моделирования, не являющих специалистами в области распределенных вычислений. Для определения сценария программист использует традиционные средства языка C++, не вникая в детали работы служб Грид и не прибегая к использованию специальных языков программирования.

Сценарии и графы исполнения GridMD

В этой главе мы кратко описываем архитектуру GridMD, более подробно изложенную в работах [13,14]. Разработанный компонент библиотеки GridMD использует стандартный способ формулировки потока обработки (workflow), который мы также будем называть сценарием приложения. Основными элементами сценария являются узлы и связи, формирующие граф исполнения. С узлами ассоциируются определенные действия программы, а со связями – данные (либо файлы данных), являющиеся их результатами. Входящие и исходящие связи представляют зависимости между узлами. Считается, что система может исполнить все действия узла, если известны результаты всех входящих в узел связей. Связи в GridMD могут классифицироваться по следующим категориям:

- Жесткая логическая связь. Узлы, связанные данным типом связи должны исполняться в рамках одного процесса (иметь общее пространство данных). Данный тип связи предназначен в основном для отражения логических цепочек внутри приложения.
- Связь по данным. Такой тип связи между узлами А и Б подразумевает, что для выполнения узла Б, для которого связь по данным является входящей, требуется либо предварительное выполнения узла А, для которого связь по данным является исходящей, в рамках того же процесса, либо передача данных результата узла А на узел Б в виде файла с данными. Таким образом, появляется возможность выполнения узлов А и Б в рамках различных независимых вычислительных процессов.
- Связь по вычислениям. Данный тип связи между узлами А и Б подразумевает, что для выполнения узла Б, для которого связь по вычислениям является входящей, требуется выполнение узла А и некоторых вычислений В(АБ) в рамках одного процесса. Формально связь по вычислениям можно рассматривать как три узла А-В(АБ)-Б, связанных жесткой связью, где узел В производит вычисления. Выделение такой конструкции в отдельный тип связи по вычислениям обусловлено специальными правилами, применимыми к данному типу и тем, что этот тип является ключевым при итеративных расчетах. К специальным правилам для итеративных вычислений относятся возможность перезапуска или продолжения вычислений, получения их промежуточного результата и состояния и др.

Если известен граф исполнения определенной части программы, то определить какие из узлов могут быть исполнены в распределенном режиме (одновременно с другими) возможно с помощью специального алгоритма анализа графа исполнения. Приведем краткую формулировку этого алгоритма:

1. Отметить узел start как «выполненный».
2. Отметить все элементы, не отмеченные как «выполненные», как «невыполненные». Отметить все узлы, имеющие на входе связь по данным с «невыполненными» узлами, как «заблокированные». Если количество заблокированных узлов не меняется в процессе выполнения, программа диагностирует «тупик» и завершает работу с сообщением об ошибке.
3. Рекурсивно отметить все узлы, имеющие входные связи с заблокированными, также как «заблокированные».
4. Рекурсивно определить максимальный набор «незаблокированных», «невыполненных» узлов, связанных с «выполненными» входными связями.
5. Определить все узлы из шага 4, результат которых требуется для передачи в «заблокированные» узлы (т.е. их разблокировки) или совпадающие с узлом finish.
6. Для всех узлов из шага 5 найти наибольший подграф из узлов, связанных с ними входными логическими связями.
7. Для каждого подграфа из шага 6 определить полный список файлов (по входным связям по данным), необходимых для его выполнения.
8. Сформировать задания, состоящие из передачи всех файлов из шага 7 и выполнения узлов подграфов из шага 6.
9. (балансировка нагрузки) На этом этапе возможно слияние нескольких заданий в одно, если их подграфы пересекаются, и слияние приведет к эффективному уменьшению времени исполнения за счет сокращение расходов на запуск задания.
10. Выполнить задания, сформированные на шаге 8-9 и определить список выходных файлов, которые будут сгенерированы. На этом шаге задания передаются на выполнение соответствующему менеджеру ресурсов.
11. Дождаться выполнения хотя бы одного из запущенных заданий и отметить все узлы выполненных подграфов как «выполненные».
12. Повторить с шага 2, до тех пор, пока остается хотя бы один невыполненный элемент.

Отметим, что рекурсивный поиск по графу или получения максимального связанного подграфа может производиться стандартными алгоритмами для работы с графами. В GridMD для этого используется библиотека boost graph library [15].

Как видно из алгоритма обработки графа, источником параллелизма являются связи по данным. Действия, которые требуется исполнить для каждого узла графа исполнения, обычно оформляются в виде специальных подпрограмм [3,9]. Однако, в архитектуре GridMD предусмотрена принципиально новая возможность указания действий узла. Она реализована в неявном механизме задания графа исполнения, который и будет использован в примере приложения, описанном выше.

Приложение GridMD может работать в двух главных режимах – менеджера и исполнителя, задаваемых через параметры командной строки. Приложение обычно запускается в режиме менеджера, а потом передает свои вычислительные узлы на исполнение по мере их активации алгоритмом анализа графа. Передача узлов на исполнение формально является запуском того же приложения в режиме исполнителя и с параметрами командной строки, указывающими, какие именно узлы необходимо исполнить в рамках запускаемого процесса. При неявном механизме создания узлов приложение в режиме исполнителя формально проходит те же стадии, что и в режиме менеджера. Таким образом, в любом случае обходятся все узлы графа, однако в режиме исполнителя функции node_define(…), отвечающие за определение узла, возвращают 1 только при проходе узлов из списка, переданного приложению через параметры командной строки. Анализ возвращаемого значения

дает возможность выполнить действия, связанные с данным узлом, не создавая для этого специальной процедуры или класса.

Запуск исполнителя производится компонентой JobManager и может осуществляться внутри основного вычислительного потока приложения (рекурсивным вызовом главной функции), путем организации нового потока, путем системного вызова внутри локальной операционной системы, путем передачи MPI-сообщения в случае реализации на параллельных системах, либо путем передачи задачи системе запуска приложений на удаленных узлах. Последний способ запуска является ключевым для работы с распределенными системами. В данной работе будет подробно описан способ запуска задач с помощью ssh-канала связи с удаленным ресурсом, на котором установлена система очередей PBS.

Пример использования GridMD

Для иллюстрации функциональности GridMD рассмотрим реальную задачу МД моделирования кулоновского разлета нанометровых твердотельных кластеров. При облучении кластера коротким лазерным импульсом высокой интенсивности происходит полная ионизация атомов кластера и дальнейший разлет ионов за счет кулоновского отталкивания. При этом формируются ионы с нестандартным (неравновсным) распределением по скоростям. Изучение этого состояния необходимо для понимания процессов термоядерного синтеза, происходящих при столкновении ионов и фиксируемых в реальных экспериментах.

Listing 1 (common.h):

```
// number of generated trajectories (usually large)
const int NUM_TRJ = 3;

// bit flags to encode 3 execution phases
enum {PHASE_A=1, PHASE_B=2, PHASE_C=4};

// storage for physical parameters of the task
class process_params { ... };

// task functions:
// get parameters from program arguments
process_params get_parameters(int argc, char **argv);

// get execution phase from program arguments
int get_phase(int argc, char **argv);

// calculate ith trajectory of count, using input
// parameters and saving output to outfile
bool calculate_trajectory(int i, int count,
  process_params par, const char *outfile);

// calculate velocity distributions using trajectory
// file as input and saving them to distrfile
bool process_trajectory(const char *trjfile,
  process_params par, const char *distrfile);

// average distribution results over count files,
// using the file pattern as input
bool avarage_results(const char *filepattern, int count,
  process_params par, const char *outfile);
```

Listing 2 (main.cpp):

```
#include "common.h"

void main(int argc, char **argv){
  // get parameters from program arguments
  process_params par = get_parameters(argc, argv);
  // get requested execution phase(s)
  // form program arguments
  int phase = get_phase(argc, argv);

  if(phase & PHASE_A){ // calculation phase
    for(int i=0; i<NUM_TRJ; i++)
      calculate_trajectory(i, NUM_TRJ, par,
        fmt("traj%02d.t",i) );
  }
  if(phase & PHASE_B){
    for(int i=0; i<NUM_TRJ; i++)
      process_trajectory( fmt("traj%02d.t",i),
        par, fmt("distr%02d.dat",i) );
  }
  if(phase & PHASE_C){
    average_results("distr%02d.dat",
      NUM_TRJ, par, "main_data");
  }
}
```

Listing 3 (main_gridmd.cpp):

```
#include "gridmd.h" #include "common.h"

void main(int argc, char **argv){
  // init and analyze startup flags, manager enters
  // construction mode
  gmdExperiment.init(argc,argv);
  process_params par=get_parameters(argc,argv);
  gmdSweep<> select("select"), traj("trj"), distr("distr");

  int phase = gmdGetMode()==gmdMODE_MANAGER ?
    PHASE_A | PHASE_B | PHASE_C : get_phase(argc,argv);

  select.mark_begin();
  if(phase & PHASE_A){
    select.mark_node_define("phase a)");
    traj.mark_begin();
    for(int i=0; i<NUM_TRJ; i++){
      traj.( fmt("traj%02d",i) );
      if(traj.mark_node_calculate())
        calculate_trajectory(i, NUM_TRJ, par);
    }
    traj.mark_end();
  }
  if(phase & PHASE_B){
    select.mark_node_define("phase b)");
    distr.mark_begin();
    // current node and all dependent nodes must be
    // rebuilt when the program is restarted
    set_node_property(gmdPROP_REBUILD);
    for(int i=0; i<NUM_TRJ; i++){
      distr.mark_node_init( fmt("proc%02d",i) );
      if(distr.mark_node_calculate())
        process_trajectory(i, NUM_TRJ, par);
      distr.mark_node_finalize( fmt("distr%02d",i) );
    }
    distr.mark_end();
  }
  if(phase & PHASE_C){
    select.mark_node_init("average");
    select.set_branch_label("phase c)");
    if(select.mark_node_calculate())
      avarage_results(NUM_TRJ, par);
  }
  select.mark_end("",gmdLINK_HARD);
  // making additional data links: using labels
  // to find the proper nodes
  for(int i=0; i<NUM_TRJ; i++){
    node_link<gmdBinFile>( fmt("traj%02d",i),
     fmt("proc%02d",i), fmt("traj%02d.t",i) );
    node_link<gmdBinFile>( fmt("distr%02d",i),
     "average", fmt("distr%02d.dat",i) );
  }
  // re-linking with a data link instead of hard link
  node_relink<gmdBinFile>( "average",
   select.get_bottom(), "main_data" );
  gmdExperiment.execute();
}
```

Рис 1. Листинги исходной (последовательной) и результирующей (распределенной с использованием GridMD) программ расчета. Листинг 1: определения (константы, расчетные и служебные подпрограммы), Листинг 2: последовательное приложение, Листинг 3: GridMD-приложение.

Для расчета распределения ионов по скоростям в условиях неравновесного расширения кластера используется усреднение по начальным микросостояниям. В целом численный эксперимент состоит из трех этапов: a) расчет траекторий ионов (координат и скоростей) в процессе разлета кластера при различных начальных микросостояниях; b) вычисление распределений ионов по скоростям в выбранные моменты времени, используя сохраненные файлы с траекториями; c) усреднение по траекториям распределений ионов, соответствующих одним и тем же моментам времени. Наиболее затратным по времени является этап a, поэтому после генерации траекторий движения ионов, эти данные разумно сохранить в виде файлов. Этапы b и c представляют собой один из вариантов последующей обработки траекторий (наряду с возможным расчетом других макроскопических величин), которая, как правило, требует меньше времени. Обработка может быть выполнена даже если траектории рассчитаны лишь частично, т.е. этапы b, c можно запускать не дожидаясь окончания этапа a. Периодически выполняя обработку исследователь получает результат для все более поздних этапов разлета кластера и может определить время, послед которого расчет траектории следует остановить. Кроме того, обработку траекторий (этапы b и c) часто приходится выполнять повторно для подбора оптимальных параметров. Если в процессе расчета окажется, что точность недостаточна, исследователь должен иметь возможность рассчитать дополнительные траектории и тем самым уменьшить статистическую погрешность.

С точки зрения распараллеливания, расчет и обработка отдельных траекторий (этапы a, b) могут выполняться независимо на узлах (кластерах) распределенной вычислительной системы. Этап c требуется синхронизации для операции “reduction”, однако он требует минимального количества времени и может выполняться последовательно.

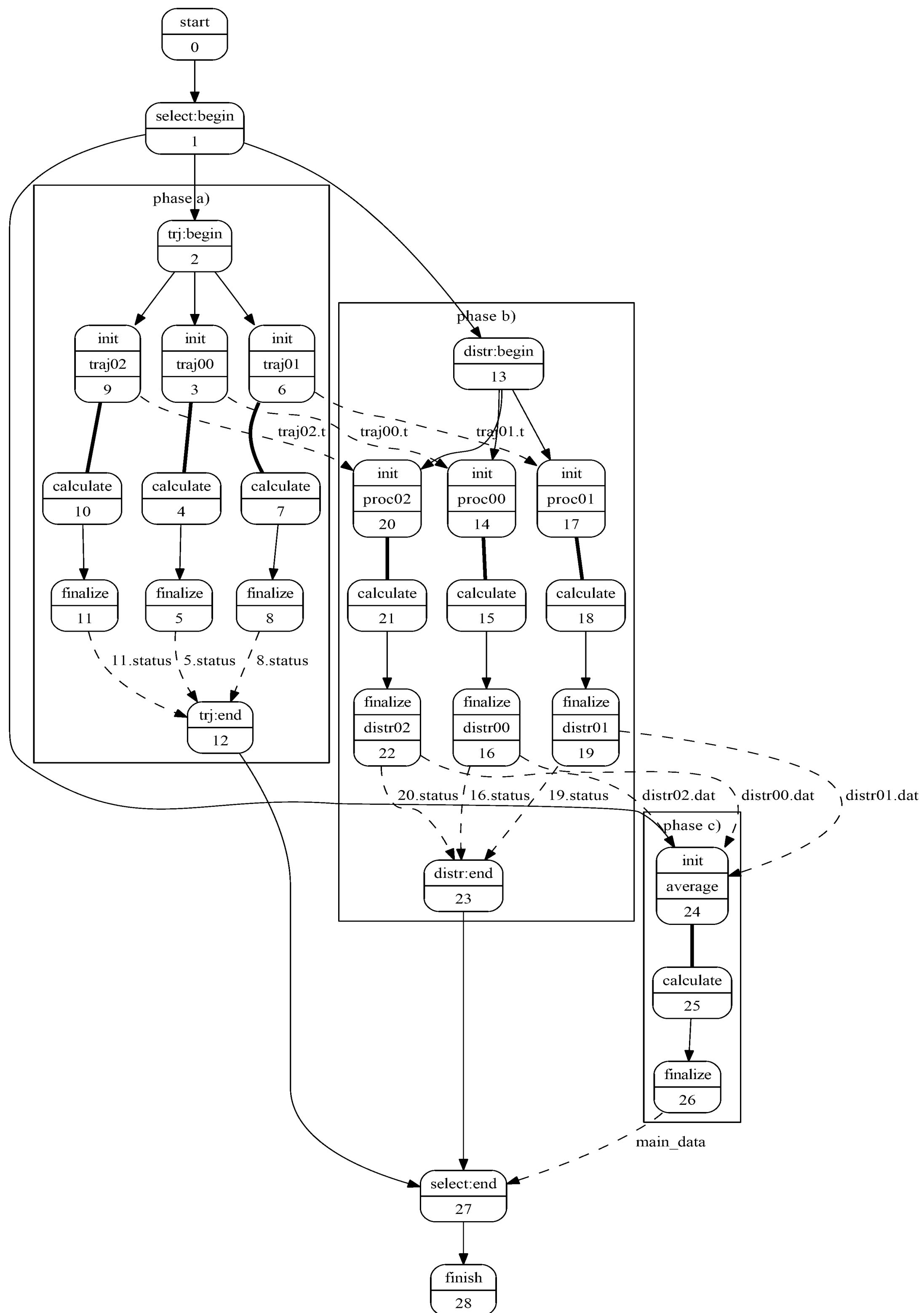

Рис 2 Граф исполнения, автоматически сгенерированный для программы, представленной на Листинге 3. Логические связи изображены сплошными линиями со стрелками, связи по данным - пунктирными линиями с указанием имен передаваемых файлов, связи по вычислениям показаны жирными линиями. Номера, показанные для каждого узла, являются уникальными идентификаторами узлов.

На рис. 1 приведены исходные коды последовательной (листинги 1, 2) и параллельной (листинги 1, 3) версий программы. Указанные выше этапы моделирования представлены в виде функций calculate_trajectory, process_trajectory и average_results. Предполагается, что входные параметры расчета передаются через командную строку и после обработки сохраняются в структуре process_params. Битовые поля в переменной

phase определят какую фазу необходимо выполнять при данном запуске программы. Такая структура соответствует основному принципу GridMD, согласно которому один и то же исполняемый файл используется для выполнения различных этапов в зависимости от передаваемых параметров.

Проиллюстрируем работу алгоритма анализа графа исполнения на примере графа, изображенного на рис. 2. Если с помощью параметров командной строки запрашивается исполнение этапа a), следующие подграфы узлов передаются на распределенное (независимое) исполнение: (0,1,2,3,4,5), (0,1,2,9,10,11) и (0,1,2,6,7,8). У этих подграфов есть пересекающиеся узлы (0,1,2), представляющие части кода, ответственные за инициализацию. Приложение-менеджер после запуска этих 3 заданий находится в спящем режиме, ожидая их завершения, однако оно может быть остановлено и перезапущено, и это не повлияет на результат. После перезапуска программа возобновляет исполнение в режиме менеджера, восстанавливая текущее состояние графа исполнения из сохраненных данных. Когда все три упомянутых задания будут выполнены, будет исполнен набор узлов (12,27,28), завершающий этап a. Если требуется исполнение этапа b, то формируются задания, состоящие из наборов узлов (0,1,13,14,15,16), (0,1,13,17,18,19) и (0,1,13,20,21,22). Они анализируют данные траекторий, записываемых узлами 3,6 и 9 и могут быть запущены, только если эти узлы завершены. В противном случае программа войдет в спящий режим, периодически проверяя состояние блокирующих узлов. Набор (23,27,28) исполняется при завершении этапа b. На этапе c исполняется набор узлов (0,1,24,25,26) только в случае снятия блокировок этапа b. Конечный результат (файл “main_data”) формируется заданием из улов (27,28) на этапе c. Таким образом, приложение автоматически гарантирует правильную последовательность выполнения всех фаз и возможность перезапуска менеджера, задача которого проверять готовность узлов к исполнению и передавать их на удаленный ресурс. Отметим, что для число ветвей (траекторий) для данного приложение должно быть большим (100-1000), и мы использовали только три траектории для упрощения нашего примера.

Управление отдельными заданиями

Компонент GridMD, называемый “менеджером заданий” (Job Manager), отвечает за выполнение отдельного узла графа на локальной или удаленной машине. Под удаленной системой предполагается головная машина кластера или Грид-системы, на которую менеджер заданий должен скопировать входные данные, после чего передать задачу удаленной системе очередей, осуществить контроль за ее выполнением и получить результат. В обычном режиме внутренние функции GridMD обращаются к менеджеру заданий автоматически, однако программист может использовать его интерфейс напрямую или даже отдельно от остальных функций GridMD. В настоящее время менеджер заданий поддерживает различные типы систем очередей для кластеров под управлением Portable Batch System (PBS), и Грид-системы UNICORE. В дальнейшем планируется поддержка других Грид-систем (ССРВ МСЦ РАН), а также пакетов управление сценариями (Kepler, Nimrod и др.).

При инициализации менеджера заданий необходимо указать тип удаленной системы очередей, а также способ доступа к ней. В качестве способов доступа в настоящее время поддерживаются протокол SSH и система Deisa Shell (DESHL). Если приложение GridMD выполняется по MS Windows, то для доступа используются утилиты plink и pscp из свободно распространяемого пакета PuTTY. Классы, реализующие протоколы доступа, предоставляют функции передачи и преобразования файлов, управления каталогами пользователя, выполнения отдельных команд на удаленной системе.

После инициализации пользовать может создавать объекты класса gmJob, описывающие отдельную задачу, и передавать их на выполнение тому или иному менеджеру заданий. Объект gmJob хранит информацию о входных и выходных файлах, командах, исполняемых на удаленной системе, требуемом числе процессоров и других ресурсов. Описание задачи не зависит от типа менеджера заданий, таким образом задача может быть передана на выполнение любому менеджеру, однако, после запуска в объекте gmJob сохраняется информация о том, где задача была запущена. Используя функции-члены класса gmJob пользователь может управлять заданием, копировать файлы во временный каталог задачи на удаленной системе и получать промежуточные/конечные файлы с результатами, определять текущее состояние задания. Предполагается, что задание может находится в одном из следующих состояний:

`JOB_INIT` - инициализация (состояние по умолчанию после создания объекта gmJob);

`JOB_PREPARED` - задание подготовлено к выполнению: временный каталог создан, входные файлы в скопированы, осуществлена привязка к менеджеру заданий;

`JOB_SUBMITTED` - задание передано системе управления заданиями;

`JOB_QUEUED` - задание поставлено в очередь;

`JOB_RUNNING` - задание выполняется;

JOB_SUSPENDED - задание приостановлено;

JOB_EXITING - выполнение окончено и происходит удаление из очереди;

JOB_COMPLETED - задание полностью выполнено и выходные данные доступны для копирования;

JOB_HAVERESULT - выходные данные скопированы на локальную машину;

JOB_FAILED - при выполнении одной из операций произошла ошибка.

При завершении приложения все задачи по умолчанию останавливаются, однако, если необходимо, пользователь может указать, что задача должна продолжить выполнение на удаленной системе. При следующем запуске программы информация о задаче может быть восстановлена.

Заключение

В работе изложена общая архитектура высокоуровневой библиотеки классов GridMD с подробным рассмотрением компонент библиотеки, отвечающих за обработку сценариев и управление отдельными заданиями. Для иллюстрации функциональности библиотеки приведен пример ее использования для МД моделирования в физике плазмы. GridMD является развивающимся проектом с постоянно увеличивающимся набором функций. Потенциальными потребителями библиотеки являются разработчики и пользователи программ численного моделирования, которым приходится регулярно выполнять масштабные численные эксперименты с нетривиальным сценарием. Основная концепция, заложенная при создании GridMD, состоит в том, что для разработки распределенных программ используется единственный язык программирования (C++), и разработчик может придерживаться стандартной структуры кода, определяемого решаемой задачей, не прибегая к разбиению его на отдельные части или реорганизации под формат библиотеки. Сохранение структуры и читабельности кода особенно важно для научных программ, в которые часто приходится вносить изменения. Гибкость применяемого подхода позволят выполнять приложение на различных типах распределенных системах, оставляя возможность запуска в последовательном режиме на одном процессоре.

ЛИТЕРАТУРА:

1. W.M.P. van der Aalst // The Journal of Circuits, Systems and Computers, V. 8, No. 1, P. 21 (1998).
2. Pytlinski, J., Skorwider, L., Benedyczak, K., et al // LNCS, V. 2658, P. 307 (2003).
3. Kepler project web site: http://www.kepler-project.org
4. Pegasus project web site: http://pegasus.isi.edu
5. Sudholt, W., Baldridge, K., Abramson, D., Enticott, C., Garic, S. // New Generation Computing, V. 22, P. 125 (2004).
6. Fastflow project web site: http://mc-fastflow.sourceforge.net
7. Intel Threading Building Blocks library web site: http://www.threadingbuildingblocks.org
8. The Edinburgh Skeleton Library web site: http://homepages.inf.ed.ac.uk/abenoit1/eSkel
9. Skandium project website http://skandium.niclabs.cl
10. LAMMPS web site: http://www.ks.uiuc.edu/Research/namd
11. NAMD web site: http://lammps.sandia.gov
12. Kuksin, A.Yu., Morozov, I.V., Norman, G.E., Stegailov, V.V., Valuev, I.A. // Mol. Simul., V. 31, P. 1005 (2005).
13. I.A. Valuev. GridMD: Program Architecture for Distributed Molecular Simulation // LNCS , V. 3719, P. 309 (2005).
14. Morozov., I. Valuev, Distributed Applications From Scratch: Using GridMD Workflow Patterns // LNCS, V. 4489, P. 199 (2007).
15. Boost library web site: www.boost.org