Методические указания по проведению практических работ по курсу «Суперкомпьютерные технологии в атомистическом моделировании»

Составил к.ф.-м.н., доц. И.В. Морозов

Практическое занятие 1. Вводное занятие. Доступ к учебной вычислительной системе.

На вводном занятии озвучиваются общие требования к выполнению заданий, порядок проведения практических занятий и правила работы с учебными вычислительными системами.

1.1 Требования к оборудованию компьютерного класса для проведения занятий

Для проведения занятий необходим компьютерный класс с числом посадочных мест из расчета одно место на одного обучаемого, а также дополнительное рабочее место для преподавателя и проектор для демонстрации слайдов.

Все рабочие станции в компьютерном классе должны быть иметь возможность подключения к учебному вычислительному кластеру по локальной сети или по сети интернет. На компьютерах должна быть установлена операционная система Linux или Windows, а также прикладное программное обеспечение, позволяющее выполнять соединение с кластером по протоколу SSH, доступ к удаленной файлов системе по протоколу SCP, текстовый редактор.

Компиляция и запуск программ во время практических занятий осуществляются на учебном вычислительном кластере по удаленному доступу с применением технологии SSH. Обучаемым доступны компиляторы с языков C/C++ и Фортран (gcc, g++, icc, icpc, ifort). Допускается выполнение некоторых заданий на локальном компьютере в учебном классе или самостоятельная работа с использованием других вычислительных систем.

Визуализация результатов расчетов возможна, если в протоколе SSH включен форвардинг команд системы X-Windows. Для ОС Windows необходимо установить специальное ПО для отображения окон X-Windows (например, пакет Xming).

1.2 Требования к базовой подготовке слушателей курса. Правилами работы на учебном вычислительном кластере.

Для выполнения заданий обучаемые должны обладать общими навыками работы с персональным компьютером, уметь программировать на языке C, C++ или Фортран, уметь работать с интерпретатором командной строки операционной системы Linux.

Для допуска к работе на учебном вычислительном кластере обучаемые должны быть ознакомлены со следующими правилами работы на нем.

1. На управляющей машине и вычислительных узлах кластера разрешается хранить только те данные и выполнять только те действия, которые связанны с выполнением заданий по курсу «Суперкомпьютерные технологии в атомистическом моделировании».
2. Запрещается передавать кому-либо информацию о способе доступа к кластеру, а также свою персональную информацию: логин и пароль. Обучаемый несет ответственность за все действия на кластере, выполненные с использованием его логина.
3. Первоначально выданный пользователю пароль должен быть изменен при первом входе в систему.
4. Управляющая машина кластера предназначена для компиляции, тестовых запусков и обработки результатов расчетов. Запрещается запускать на управляющей машине программы с временем выполнения более одной минуты. Для таких программ необходимо использовать систему управления заданиями PBS.
5. Нарушение указанных выше правил, а также действия, направленные на изменение параметров или нарушение работы операционной системы и прикладного программного обеспечения узлов кластера, несанк-

ционированный доступ к данным других пользователей кластера, являются основанием для отстранения обучаемого от работы на кластере и от выполнения им практических заданий по курсу.

6. Необходимо регулярно копировать с кластера исходные тексты программ и полученные результаты расчетов, т.к. возможная потеря данных на кластере в результате аппаратного сбоя не будет рассматриваться как уважительная причина при отсутствии у обучаемого результатов практической работы.

1.3 Доступ к учебной вычислительной системе из ОС Linux.

В процессе вводного практического занятия необходимо выдать обучаемым логины и пароли, проверить возможность доступа к вычислительному кластеру, ознакомить обучаемых с настройками удаленной ОС кластера, отработать выполнение простейших команд.

Для удаленного доступа к кластеру из локального компьютера, работающего под управлением ОС Linux, выполняются команды

```
ssh -X -p 2200 username@hpc.mipt.ru
```

где 2200 – порт доступа, username – логин пользователя, hpc.mipt.ru – адрес головной машины вычислительного кластера.

Для копирования файлов используется команда scp, например,

```
scp prog.cpp username@hpc.mipt.ru:progs
```

1.4 Доступ к учебной вычислительной системе из ОС Windows.

Для удаленного доступа к кластеру из локального компьютера, работающего под управлением ОС Windows, используется свободно распространяемая программа PuTTY SSH или аналог. Схема настройки PuTTY SSH показана на рис. 1.1-1.3.

Для работы с удаленной файловой системой, копирования файлов между локальным компьютером и головной машиной кластера, редактирования файлов на кластере применяется свободно распространяемая программа WinSCP или ее аналог. Схема настройки WinSCP для той же учебной системы (hpc.mipt.ru) показана на рис. 1.4.

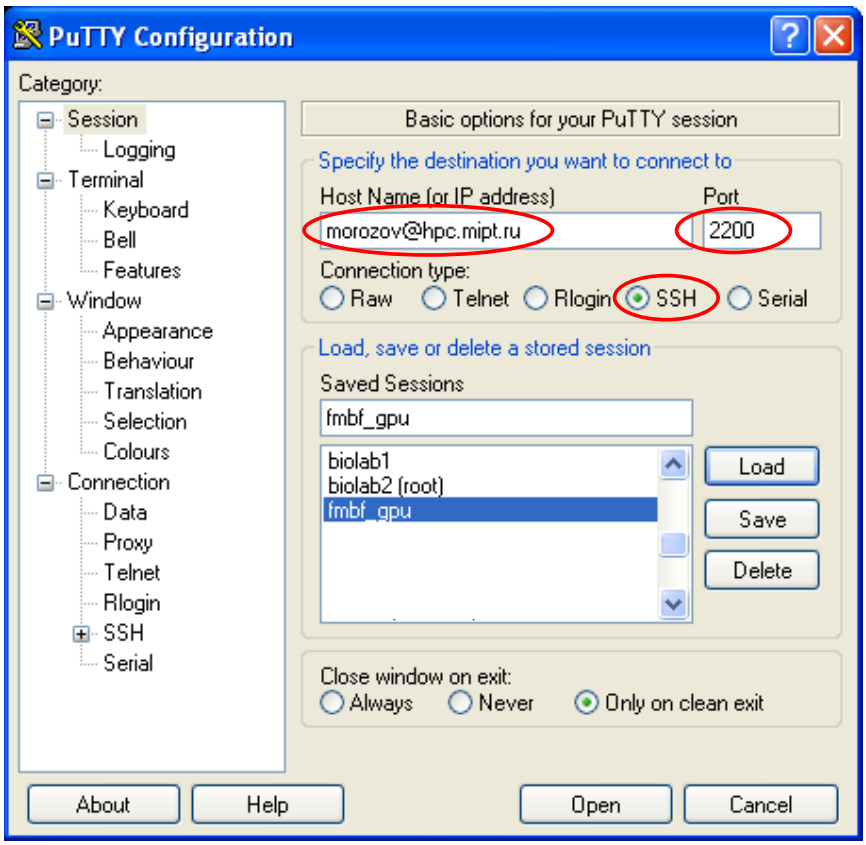

Рис. 1.1. Настройка PuTTY SSH для доступа к кластеру: задание данных о пользователе, порте и протоколе доступа.

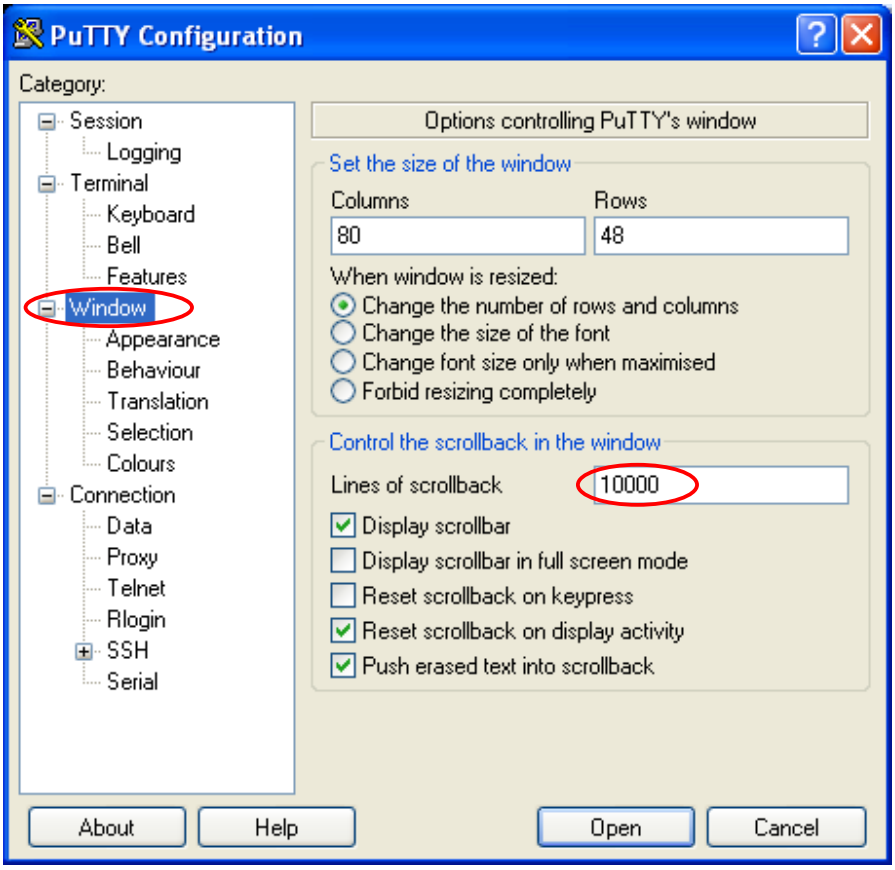

Рис. 1.2. Настройка PuTTY SSH для доступа к кластеру: задание оптимальных параметров консоли.

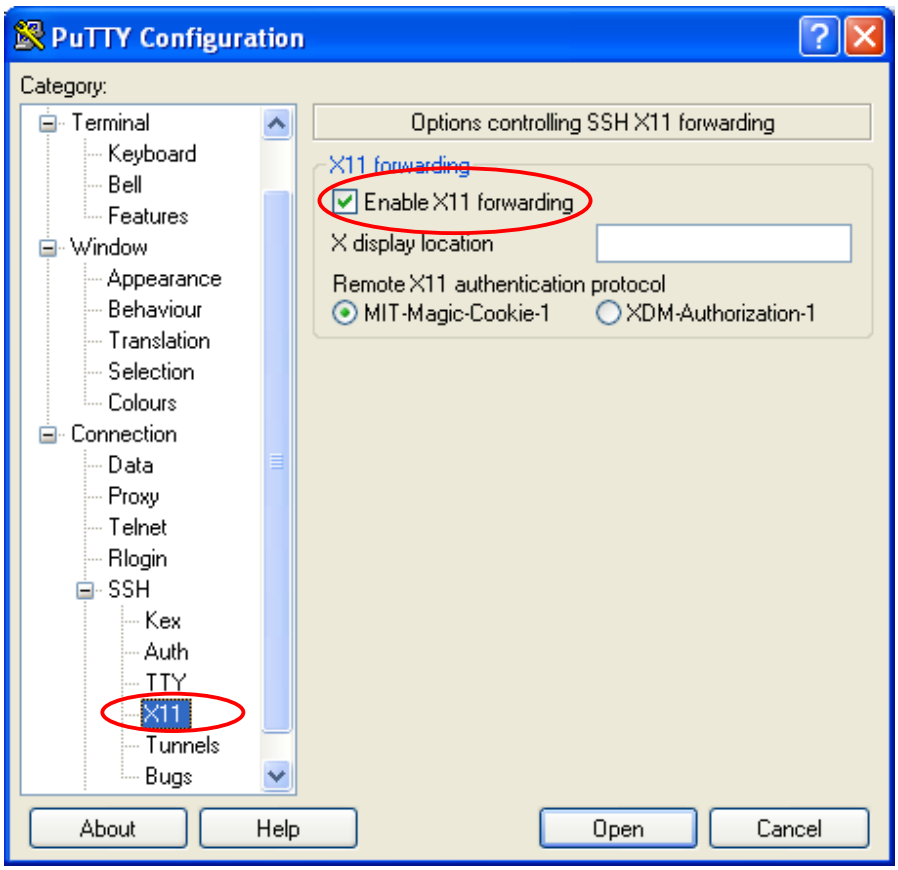

Рис. 1.3. Настройка PuTTY SSH для доступа к кластеру: включение форвардинга команд X-Windows для работы с программами визуализации.

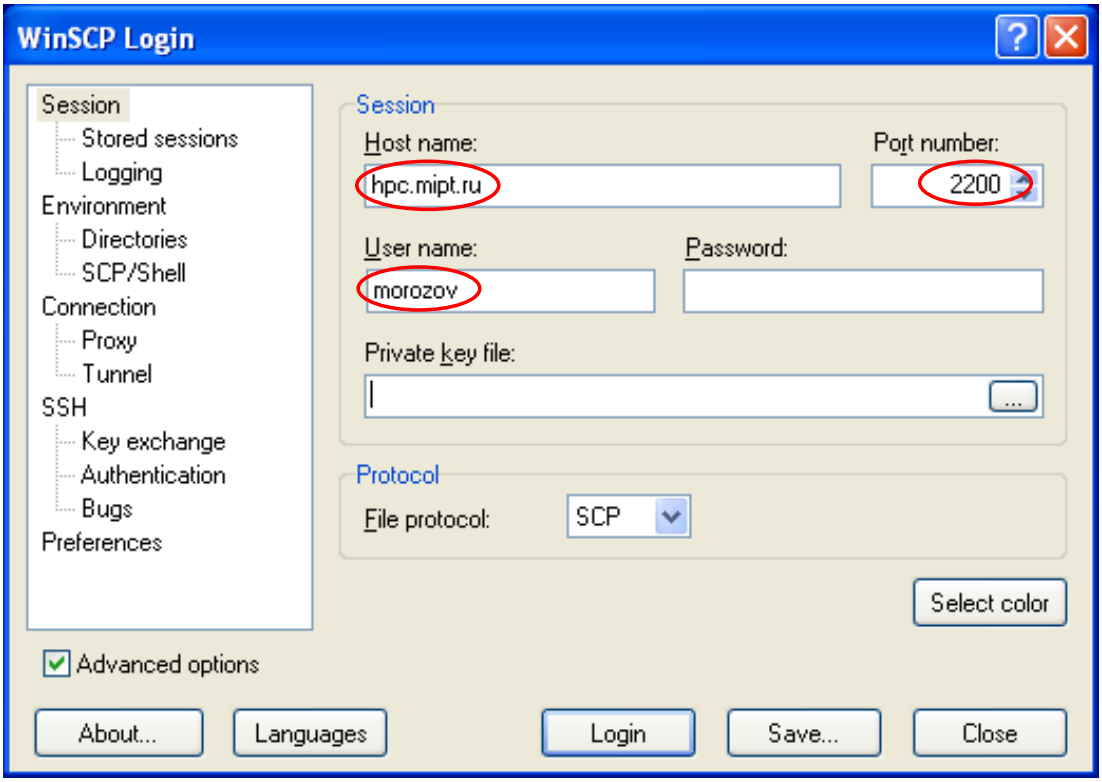

Рис. 1.4. Настройка WinSCP для доступа к кластеру: задание данных о пользователе, порте и протоколе доступа.

Практическое занятие 2. Запуск задач с помощью системы очередей PBS

Для запуска задач на кластере должна использоваться система очередей. На учебном вычислительном кластере установлена одна из разновидностей свободно распространяемой системы очередей PBS (Portable Batch System), состоящая из менеджера ресурсов (TORQUE Resource Manager) и планировщика задач (Maui Cluster Scheduler).

2.1 Задание 1. Создание и проверка работы скрипта

Для запуска задачи в системе очередей PBS необходимо создать скрипт (программу на языке командного интерпретатора Linux), из которого будут запущены основные программы моделирования. В данном задании изучаются основы написания скриптов для распространенного интерпретатора Bash shell.

Набор скриптов можно выполнять на локальном компьютере или удаленно с использованием текстовых редакторов vi, nano, joe, mc, и др. Рассмотрим простейший скрипт, который выводит название вычислительного узла и текущей директории. Для увеличения времени исполнения в него добавлена задержка 10 секунд:

```
vi test.sh
```

```
#!/bin/bash
echo Script starts on ...
sleep 10
hostname
echo Current directory: `pwd`
```

Следующие команды устанавливают атрибут, необходимый для запуска, и проверяют работу скрипта в интерактивном режиме.

```
chmod u+x test.sh
./test.sh
```

Часто в скрипте необходимо использовать переменные окружения, как показано на следующем примере

```
vi test.sh
```

```
#!/bin/bash
echo Script starts on ...
sleep 2
hostname
echo Current directory: `pwd`
echo "Testvar = ${TESTVAR}"
```

```
export TESTVAR=1234
./test.sh
```

2.2 Задание 2. Использование основных команд системы очередей

Систем очередей PBS имеет следующие основные команды:

- **qsub** – (queue submit) постановка задачи в очередь на исполнение. Эта команда поставит программу в очередь задач PBS и выведет уникальный идентификатор задачи, например: 1891.head.
- **qstat** – (queue statistics) получение информации о текущем состоянии системы очередей.
- **qselect** – (queue select) выбор номеров заданий по заданному критерию.
- **qdel** – (queue delete) удаление задачи из очереди.

Как уже было сказано, при использовании PBS для каждой программы необходимо создать отдельный скрипт — текстовый файл с последовательностью команд для ее запуска. В этом скрипте могут присутствовать специальные комментарии, начинающиеся с #PBS, с опциями для команды qsub, например:

```
#!/bin/bash
#PBS -N hello_test
cd /home/username/tests
./test.sh
```

В этом скрипте могут изменяться следующие позиции:

- lammps – имя задачи в очереди (любое);
- /home/username/tests – каталог, из которого запускается задача;

- ./test.sh – исполняемая программа с необходимыми опциями.

В отчете команды qstat указывается: идентификатор задачи, имя задачи (см. параметр -N), имя пользователя, а в последних трех строках: запрошенное пользователем время, состояние задачи ("Q" - задача ожидает освобождения очереди, "R" – выполняется, "E" – завершается и др.), текущее время выполнения. Пример такого отчета показан на рис. 2.1.

Расширенную диагностику с указанием имен, на которых задача фактически была запущена, можно получить с использованием ключа “-n” (см. рис. 2.2).

Статистика по состоянию всех вычислительных узлов выдается по команде “ pbsnodes –a ” (см. рис. 2.3).

```
[imstud30@head ~]$ qstat
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
208.head                   cpmd             mukhanov         00:00:03 R cpu
241.head                   Qnexus           msu              03:59:38 R gpu
```

Рис. 2.1. Пример вывода команды qstat.

```
[imstud30@head ~]$ qstat -n

head.fmbf.ru:
                                                                  Req'd  Req'd   Elap
Job ID                  Username Queue    Jobname          SessID NDS   TSK Memory Time  S Time
----------------------- -------- -------- ---------------- ------ ----- --- ------ ----- - -----
208.head.fmbf.ru        mukhanov cpu      cpmd              22366     2  24    --  80:00 R 48:31
   node02/11+node02/10+node02/9+node02/8+node02/7+node02/6+node02/5+node02/4
   +node02/3+node02/2+node02/1+node02/0+node01/11+node01/10+node01/9+node01/8
   +node01/7+node01/6+node01/5+node01/4+node01/3+node01/2+node01/1+node01/0
241.head.fmbf.ru        msu      gpu      Qnexus            29019     1   2    --  80:00 R 04:00
   node04/1+node04/0
```

Рис. 2.2. Пример вывода команды qstat -n.

```
[imstud30@head ~]$ pbsnodes -a
n01.fmbf.ru
     state = free
     np = 12
     properties = C2050
     ntype = cluster
     jobs = 0/4264.head.fmbf.ru, 1/4264.head.fmbf.ru, 2/4264.head.fmbf.ru,
3/4264.head.fmbf.ru, 4/4264.head.fmbf.ru, 5/4264.head.fmbf.ru, 6/4264.head.fmbf.ru,
7/4264.head.fmbf.ru, 8/4264.head.fmbf.ru, 9/4264.head.fmbf.ru
     status =
rectime=1320701683,varattr=,jobs=4264.head.fmbf.ru,state=free,netload=4034996285420,gres=,loa
dave=9.99,ncpus=12,physmem=16431188kb,availmem=47694704kb,totmem=48431180kb,idletime=1145618,
nusers=1,nsessions=2,sessions=27785 32022,uname=Linux n01.fmbf.ru 2.6.18-194.32.1.el5 #1 SMP
Wed Jan 5 17:52:25 EST 2011 x86_64,opsys=linux
     gpus = 2
...
```

Рис. 2.3. Пример вывода команды pbsnodes –a.

Удаление задачи из очереди выполняется командой:

```
qdel <идентификатор задачи>
```

Удаление всех задач пользователя из очереди:

```
qdel `qselect username`
```

В результате выполнения задачи в текущем каталоге появляются два файла с именами типа <имя задачи>.o< идентификатор задачи> и <имя задачи>.e< идентификатор задачи>. Например, для приведенного выше случая в каталоге /home/username/ это будут файлы test.sh.o1891 и test.sh.e1891. В файл test.sh.o1891 система очередей сохраняет вывод исполняемого процесса на консоль (например, вывод программы в поток cout в C++), а в файл test.sh.e1891 – сообщения об ошибках (например, вывод в поток cerr в C++).

2.3 Задание 3. Запуск простейшей программы с помощью системы очередей.

Для закрепления навыков работы с системой очередей предлагается создать простейшую программу на языке C и запустить ее с применением системы очередей PBS. Текст программы приведен ниже.

```
vi hello.cpp
```

```
// PBS test program
#include <stdio.h>
#include <unistd.h>
int main(void) {
```

```
  puts("Hello world!");
  sleep(10);
  puts("Goodbye world!");
  return 0;
}
```

Для ее компиляции может использовать любой компилятор языка C, например:

```
g++ hello.cpp -o hello
```

Далее следует проверить работу программы:

```
./hello
```

После этого можно создать скрипт PBS:

```
vi hello.sh
```

```
#!/bin/bash
#PBS -N hello_test
cd /home/username/tests
./hello
```

```
chmod u+x hello.sh
```

И запустить его с помощью системы очередей:

```
qsub hello.sh
qstat
```

По окончании работы программы можно посмотреть выходные файлы

```
cat hello.sh.oXXXX
cat hello.sh.eXXXX
```

При запуске программ выполняющихся достаточно длительное время удобно перенаправлять их вывод в лог-файл:

```
vi hello.sh
```

```
#!/bin/bash
#PBS -N hello_test
cd /home/username/tests
./hello >log
```

В этом случае, не дожидаясь завершения задачи, можно посмотреть промежуточную диагностику командами

```
cat log
```

или

```
tail -f log
```

Во время выполнения программы можно использовать временную директорию $PBS_O_WORKDIR для хранения промежуточных данных. По окончании задачи все файлы в этой директории будут уничтожены.

Практическое занятие 3. Создание многопоточных программ с использованием функций POSIX threads

Занятие посвящено созданию параллельных программ для систем с общей памятью с применением библиотеки POSIX threads, входящей в состав ОС Linux.

3.1 Задание 1. Создание и уничтожение потоков. Компиляция и запуск простейшей программы.

Компиляция программ, в которых используется библиотека POSIX threads, выполняется с применением ключа “-pthread”, например:

```
gcc –pthread [–lm] prog.cpp
icc –pthread prog.cpp
```

Простейшая программа, создающая один дополнительный поток и ожидающая его завершения в главном потоке, приведена ниже

```
// pthread basic test program
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

void* testfn(void *data) {
    printf("Thread is started, data = %s\n", data);
    getchar();
    printf("Thread is finished\n");
```

```
    return NULL;
}

int main() {
  pthread_t tid;
  if(pthread_create(&tid, NULL, testfn,
                    (void*)"Some data")) {
    puts("Error executing thread!");
    return 1;
  }

  for(int i=0; i<10; i++) {
    printf("Main thread, i=%d\n", i);
    usleep(1000000);  // 1sec
  }

  pthread_join(tid, NULL);
  return 0;
}
```

Следует запустить и проверить работу этой программы на одном из вычислительных узлов кластера.

3.2 Задание 2. Программа для вычисления числа π.

Задание заключается в том, чтобы написать программу, вычисляющую число в виде суммы ряда. Один из потоков должен выполнять вычисления, а другой выводить промежуточный результат и число выполненных итераций на консоль с периодичностью 0.25 секунд.

Пример результата выполнения задания приведен ниже.

```
// Calculation of PI
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
```

```
double volatile p = 0.;
int done = 0;

void* do_loop(void *data) {
    int i, iter = *(int*)data, s = 1;
    double a = 1.;

    for(i=0; i<iter; i++) {
      s = -s;
      p += s/a;
      a += 2.;
    }
    done = 1;
    return (void*)&p;
}

int main(int argc, char* argv[]) {
  pthread_t  tid;
  int iter = 100000000, i;
  void *retval;
  p = 0;

  if(pthread_create(&tid, NULL, do_loop, &iter)) {
    printf("Error executing thread!\n");
    return 1;
  }
  printf("Thread started: p_thread = %d\n", tid);

  for(int i=0; i<3; i++)  {
    usleep(250000);
    printf("pi = %.15f\n", 4.*p);
  }
```

```
  pthread_join(tid, &retval);
  printf("Thread ended with retval = %f\n",
         *(double*)retval*4.);
  return 0;
}
```

Следует обратить внимание на использование модификатора volatile в строке

```
double volatile p = 0.;
```

Отсутствие этого модификатора является типично ошибкой.

Практическое занятие 4. Синхронизация потоков в системах с общей памятью

4.1 Задание 1. Поиск потенциальных ошибок в программе из-за отсутствия синхронизации потоков

Синхронизация потоков является одной из основных проблем параллельного программирования. В приведенном ниже примере возможно возникновение ситуации «гонок» (race condition) между потоками. Обучаемый должен найти ошибку и попытаться ее исправить.

```
// Race condition
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

const int NT = 3;

void* threadfn(void *data) {
    usleep(100);
    int x = *(int*)data;
    int y = x*x;
    return (void*)y;
}
```

```
int main() {
  pthread_t tid[NT];
  int x[NT] = {1, 2, 3}, x2;

  for(int i=0; i<NT; i++) {
    x2 = x[i] * 2;
    pthread_create( &tid[i], NULL,
                    threadfn, (void*)&x2 );
  }

  for(int i=0; i<NT; i++) {
    int result;
    pthread_join(tid[i], (void**)&result);
    printf("Result of thread %d: %d\n", i, result);
  }

  return 0;
}
```

Одно из наиболее простых решений заключается в замене переменной x2 на массив:

```
...
int x[NT] = {1, 2, 3}, x2[NT];

  for(int i=0; i<NT; i++) {
    x2[i] = x[i] * 2;
    pthread_create( &tid[i], NULL,
                    threadfn, (void*)&x2[i] );
  }
...
```

4.2 Задание 2. Использование объекта синхронизации «взаимное исключение»

Объект «взаимное исключение» (mutex) предоставляет простейший механизм синхронизации доступа к данным при выполнении параллельных программ. Для изучения методов его использования предлагается написать параллельный аналог приведенной ниже программы численного интегрирования.

```
// Calculation of definite integrals: serial version
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <math.h>

double inline f(double x) {
  return x*exp(x);
}

const double a=0., b=1.;
const int nsteps = 1000000000;

int main() {
  double result = 0.;
  double h = (b - a)/nsteps;

  for(int i=1; i<nsteps; i++) result += f(a+i*h);
  result = (result + .5*(f(a) + f(b)))*h;

  printf("Final result = %.15f\n", result);
  return 0;
}
```

Результат распараллеливания данной программы с использованием объекта синхронизации «взаимное исключение» приведен на следующем листинге.

```
// Calculation of definite integrals: parallel version
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <stdlib.h>
#include <math.h>

double f(double x) {
  return x*exp(x);
}

const double a=0., b=1.;
const int nsteps = 1000000000;
double result = 0.;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
const int NTHREADS=4;

void* calc(void *data) {
    int iproc = *(int*)data;
    int n = (nsteps - 1) / NTHREADS + 1;
    int ibegin = iproc*n, iend = (iproc+1)*n;
    if(iend > nsteps) iend = nsteps;
    double h = (b - a)/nsteps;
    double res = .5*(f(a+ibegin*h) + f(a+iend*h));

    for(int i=ibegin+1; i<iend; i++)
      res += f(a+i*h);

    pthread_mutex_lock(&mut);
```

```
    result += res*h;
    pthread_mutex_unlock(&mut);

    return NULL;
}

int main(int argc, char* argv[]) {
  pthread_t tid[NTHREADS];
  int iproc[NTHREADS];
  void *retval;
  int i;

  for(i=0; i<NTHREADS; i++)   {
    iproc[i] = i;
    if(pthread_create(tid+i, NULL, calc, iproc+i)) {
      puts("Thread start failed!");
      return 1;
    }
    printf("Thread %d started.\n", tid[i]);
  }

  for(i=0; i<NTHREADS; i++) pthread_join(tid[i], NULL);
  printf("Final result = %.15f\n", result);
  return 0;
}
```

Следует обратить внимание на использование локальной переменной res для сохранения частичной суммы, что значительно повышает быстродействие.

4.3 Задание 3. Использование объекта синхронизации «условная переменная»

Объект «условная переменная» (conditional variable) применяется для барьерной синхронизации, в задачах типа «производитель-потребитель» для

сигнализации о завершении обработки данных или помещении новых данных в очередь, в других более сложных параллельных алгоритмах.

В данном задании предлагается модифицировать программу, рассмотренную в задании 1, обеспечив синхронизацию потоков с помощью объекта «условная переменная». Требуемый результат выполнения задания приведен ниже.

```
// Race condition: fixed by condvar
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

const int NT = 3;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;

void* threadfn(void *data) {
    usleep(100);
    int x = *(int*)data;
    pthread_cond_signal(&cv);
    int y = x*x;
    return (void*)(x*x);
}

int main() {
  pthread_t tid[NT];
  int x[NT] = {1, 2, 3}, x2;

  pthread_mutex_lock(&mut);
  for(int i=0; i<NT; i++) {
    x2 = x[i] * 2;
    pthread_create( &tid[i], NULL,
                    threadfn, (void*)&x2 );
```

```
    pthread_cond_wait(&cv, &mut);
  }

  pthread_mutex_unlock(&mut);

  for(int i=0; i<NT; i++) {
    int result;
    pthread_join(tid[i], (void**)&result);
    printf("Result of thread %d: %d\n", i, result);
  }
  return 0;
}
```

Практическое занятие 5. Программирование с применением OpenMP и TBB

5.1 Задание 1. Распараллеливание программы вычисления определенного интеграла с использованием технологии OpenMP

Предлагается распараллелить программу вычисления определенного интеграла, рассмотренную на предыдущей лекции, с использованием технологии OpenMP. Требуемый результат выполнения задания приведен ниже.

```
// Calculation of definite integrals using OpenMP

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

double f(double x) {
  return 1.+exp(x);
}

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

```
  const double a=0., b=1.;
  int nsteps = 50000000, nproc = 0;
  int i;

  if(argc > 1) nproc = atoi(argv[1]);
  if(!nproc) nproc = 1;

  double h = (b - a)/nsteps;
  double result = 0.;

  #pragma omp parallel num_threads(nproc)
                       reduction(+:result)
  {
    #pragma omp for schedule(static) private(i)
    for(i=1; i<nsteps; i++)
      result += f(a+i*h);
  }
  result = (result + .5*(f(a) + f(b)))*h;

  printf("Final result = %.15f\n", result);
  return 0;
}
```

5.2 Задание 2. Распараллеливание программы вычисления определенного интеграла с использованием библиотеки TBB

Предлагается распараллелить программу вычисления определенного интеграла, рассмотренную на предыдущей лекции, с использованием библиотеки Intel Threading Building Blocks (TBB). Требуемый результат выполнения задания приведен ниже.

```
// Calculation of definite integrals
```

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <math.h>
#include "tbb/task_scheduler_init.h"
#include "tbb/blocked_range.h"
#include "tbb/parallel_reduce.h"

using namespace tbb;

double inline f(double x) {
  return x*exp(x);
}

class Integrator {
public:
  double res, a, h;

 void operator()( const blocked_range<int>& r ) {
    double result = res;
      int end = r.end();
    for(int i=r.begin(); i<end; i++) result += f(a+i*h);
      res = result;
  }

 void join( const Integrator& x ) { res += x.res; }

 Integrator( Integrator& x, split ) : a(x.a), h(x.h),
res(0) {}

 Integrator( double a_, double h_ ) : a(a_), h(h_),
res(0) {}
};

int main() {
  double a = 0., b = 1., result = 0.;
```

```
  int nsteps = 1000000000;
  double h = (b - a)/nsteps;

  task_scheduler_init init;
  Integrator integr(a, h);

  parallel_reduce(blocked_range<int>(0,nsteps,1000000),
                  integr, auto_partitioner() );

  result = (integr.res + .5*(f(a) + f(b)))*h;
  printf("Final result = %.15f\n", result);
  return 0;
}
```

Практическое занятие 6. Отладка параллельных программ

6.1 Задание 1. Отладка программы с использованием утилиты Intel Thread Checker

Ручной поиск ошибок, связанных с синхронизацией потоков, бывает достаточно проблематичным. То, что ошибка может проявляться на одной системе и не проявляться на другой вносит дополнительные трудности.

В настоящее время существуют ряд инструментов для проверки корректности параллельных программ. В данном задании рассматривается один из таких инструментов – Intel Thread Checker.

В качестве примера выбрана программа, рассмотренная в задании № 1 практического занятия № 4, посвященного синхронизации потоков в системах с общей памятью. Предполагая, что исходный файл называется pthread_race2.cpp выполним его компиляцию и запуск:

```
[pthreads]$ gcc -pthread pthread_race2.cpp
[pthreads]$ ./a.out
Result of thread 0: 36
```

```
Result of thread 1: 36
Result of thread 2: 36
```

Неверный результат выполнения программы указывает на отсутствие должной синхронизации потоков. Выполним теперь ту же программу под управлением Intel Thread Checker:

```
[pthreads]$ tcheck_cl ./a.out
...
```

В результате будет выведена некоторая отладочная информация, результат выполнения программы, показанный выше, а также таблица со списком потенциальных проблем показанная на рис. 6.1.

ID	Short Description	Severity	Count	Context[Best]	Description	1st Access[Best]	2nd Access[Best]
1	Write -> Read data-race	Error	2	[a.out, 0x448]	Memory read at [a.out, 0x45e] conflicts with a prior memory write at [a.out, 0x4a9] (flow dependence)	[a.out, 0x4a9]	[a.out, 0x45e]
2	Thread termination	Information	1	WholeProgram	Thread termination at [a.out, 0x4c4] - includes stack allocation of 1,996 MB and use of 1,969 KB	[a.out, 0x4c4]	[a.out, 0x4c4]
3	Thread termination	Information	1	WholeProgram	Thread termination at [a.out, 0x4c4] - includes stack allocation of 1,996 MB and use of 1,969 KB	[a.out, 0x4c4]	[a.out, 0x4c4]
4	Thread termination	Information	1	WholeProgram	Thread termination at [a.out, 0x4c4] - includes stack allocation of 1,996 MB and use of 1,969 KB	[a.out, 0x4c4]	[a.out, 0x4c4]
5	Thread termination	Information	1	WholeProgram	Thread termination at [a.out, 0x46c] - includes stack allocation of 32 KB and use of 8,637 KB	[a.out, 0x46c]	[a.out, 0x46c]

Рис. 6.1. Диагностическая таблица программы Intel Thread Checker без отладочной информации.

6.2 Задание 2. Получение более полной отладочной информации

Более полная отладочная информация о потенциально «опасном» месте в исходном коде может быть получена с применением компилятора Intel в совокупности с утилитой Intel Thread Checker:

```
[pthreads]$ icc -pthread -tcheck pthread_race2.cpp
icc: warning: thread checking compilation enables debug
information
icc: warning: thread checking compilation disables
optimization
[pthreads]$ tcheck_cl ./a.out
...
```

ID	Short Description	Severity[Best] Name	C[o]n[t]	Context[Best]	Description	1st Access[Best]	2nd Access[Best]
1	Write -> Read data-race	Error	2	"pthread_race2.cpp":11	Memory read of *data at "pthread_race2.cpp":24 (flow dependence) conflicts with a prior memory write of xi at "pthread_race2.cpp":24	"pthread_race2.cpp":24	"pthread_race2.cpp":13
2	Thread termination	Information	1	1	WholeProgram "pthread_race2.cpp":24 - includes stack allocation of 1,996 MB and use of 1,84 KB	"pthread_race2.cpp":25	"pthread_race2.cpp":25
3	Thread termination	Information	1	2	WholeProgram "pthread_race2.cpp":25 - includes stack allocation of 1,996 MB and use of 1,84 KB	"pthread_race2.cpp":25	"pthread_race2.cpp":25
4	Thread termination	Information	1	3	WholeProgram "pthread_race2.cpp":25 - includes stack allocation of 1,996 MB and use of 1,84 KB	"pthread_race2.cpp":25	"pthread_race2.cpp":25
5	Thread termination	Information	1	4	WholeProgram "pthread_race2.cpp":18 - includes stack allocation of 28 KB and use of 7,887 KB	"pthread_race2.cpp":18	"pthread_race2.cpp":18

Рис. 6.2. Диагностическая таблица программы Intel Thread Checker со включенной отладочной информацией.

Отладочная информация, полученная в этом случае (см. рис. 6.2), является более полной и указывает на конкретные номера строк в тексте программы, где возможен одновременный доступ к данным. Также показаны названия переменных, являющихся причиной конфликта по данным.

Практическое занятие 7. Компиляция и запуск MPI-программ

7.1 Задание 1. Компиляция программ с использованием MPI

Простейшая программа на языке C, использующая базовые функции библиотеки MPI приведена на следующем листинге.

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char* argv[])
{
  int size, rank;
  if( MPI_Init( &argc, &argv ) ) return -1;
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  char str[MPI_MAX_PROCESSOR_NAME];
  int len;
  MPI_Get_processor_name(str, &len);
  printf("Process %d/%d  on %s\n",rank,size,str);
  MPI_Finalize();
  return 0;
}
```

Данная программа выводит на консоль номера всех процессов, их общее число и название узла, на котором каждый из процессов был запущен.

Пример команды для компиляции этой программы показа ниже:

```
mpicc prog1.c -o prog1 -lm -O2
```

где “prog1.c -o prog1 -lm -O2” – стандартные опции компилятора C.

Для компиляции программ на языках C, C++, Фортран-77 и Фортран-90 в большинстве реализаций MPI используются команды

```
mpicc prog.c ...
mpic++ prog.cpp ...
mpif77 prog.f77 ...
mpif90 prog.f90 ...
```

7.2 Задание 2. Запуск MPI-программ в интерактивном режиме

Порядок запуска программ несколько отличается в зависимости от версии MPI. В любом случае необходимо предварительно создать файл, указывающий, на каких узлах будет произведен запуск. Это файл имеет формат для LAM MPI:

```
hpc.mipt.ru schedule=no
node02 cpu=2
node03 cpu=2
node04 cpu=2
...
```

Для MPICH:

```
node02:2
node03:2
node04:2
…
```

Команды запуска приложения на 5 ядрах выглядят следующим образом для LAM MPI:

```
lamboot nodes
mpirun -np 5 prog
lamhalt
```

для MPICH и Open MPI:

```
mpirun -np 3 -machinefile nodes prog
```

где “nodes” – имя файла с описанием узлов.

7.3 Задание 3. Запуск MPI-программ с использованием системы очередей

При использовании системы очередей PBS требуемое количество ядер запрашивается в аргументах команды qsub или в специальных комментариях скрипта, начинающихся с “#PBS”.

Пример скрипта для запуска программы, скомпилированной с помощью пакета LAM MPI:

```
#!/bin/sh
#PBS -l nodes=3:ppn=2,walltime=10:00:00
#PBS –N test
cd /home/morozov/programs/test
lamboot
mpiexec prog1
lamhalt
```

Для MPICH аналогичный скрипт выглядит немного иначе:

```
#!/bin/sh
#PBS -l nodes=3:ppn=2,walltime=10:00:00
#PBS –N test
cd  /home/morozov/programs/test
mpiexec -boot  prog1
```

Следует обратить внимание на использование команды mpiexec вместо mpirun. Эта команда автоматически получает список узлов от PBS и позволяется контролировать (более корректно уничтожать) задачу при удалении ее из очереди.

Строка параметров

```
#PBS -l nodes=3:ppn=2,walltime=10:00:00
```

означает, что задаче будет выделено по два ядра на трех узлах (всего 6 ядер), а максимально время выполнения задачи составит 10 часов.

Для постановки задачи в очередь следует использовать стандартную инструкцию:

```
qsub pbsrun.sh
```

Практическое занятие 8. Основы программирования с использованием MPI

8.1 Задание 1. Использование функций двухточечного обмена сообщениями.

Простейшая программа с использованием MPI уже была рассмотрена на предыдущей лекции. Однако в этой программе не использовались функции явной передачи данных между параллельно выполняющимися процессами. В приведенном ниже примере реализуется отправка и прием одного сообщения с числом типа double:

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char* argv[])
{
  int size, rank;
  double data;
  if( MPI_Init( &argc, &argv ) ) return -1;
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );

  if (rank == 0){
    for (int i=1; i<size; i++){
      MPI_Status status;
      MPI_Recv(&data, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
               MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      printf("Received '%g' from process %d\n",
        data, status.MPI_SOURCE);
    }
  }
  else {
    data = rank*rank;
    MPI_Send(&data, 1, MPI_DOUBLE, 0, 1234,
```

```
            MPI_COMM_WORLD);
  }

  MPI_Finalize();
  return 0;
}
```

8.2 Задание 2. Объединение запросов на пересылку данных.

Пересылка разнотипных данных или данных, расположенных не в последовательных областях памяти, с помощью отдельных функций Senv, Recv является неэффективной. В данном задании предлагается реализовать отправку разнотипных данных многократно (т.е. в цикле) с применением функций объединения запросов на пересылку данных. Пример результата выполнения задания показан ниже

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char* argv[]) {
  const int n_iter = 3;
  int data_i;
  double data_d;
  MPI_Request req[2];
  MPI_Status stat[2];

  if( MPI_Init( &argc, &argv ) ) return -1;
  int size, rank;
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );

  if (rank == 0){
    // ----------------- rank == 0 -----------------
    MPI_Recv_init( &data_i, 1, MPI_INT, 1,
```

```
        MPI_ANY_TAG, MPI_COMM_WORLD, &req[0] );
    MPI_Recv_init( &data_d, 1, MPI_DOUBLE, 1,
        MPI_ANY_TAG, MPI_COMM_WORLD, &req[1] );

   for(int i=0; i<n_iter; i++){
      MPI_Startall(2, req);  // фактический прием данных
      MPI_Waitall(2, req, stat);
      printf("iter=%d, received: data_i=%d,data_d=%f\n",
             i, data_i, data_d);
     }
 }
  else if (rank == 1) {
   // ------------------ rank == 1 ------------------
   MPI_Send_init( &data_i, 1, MPI_INT, 0,
        0, MPI_COMM_WORLD, &req[0] );
   MPI_Send_init( &data_d, 1, MPI_DOUBLE, 0,
        0, MPI_COMM_WORLD, &req[1] );

   for(int i=0; i<n_iter; i++) {
       data_i = 1000*rank + i;
       data_d = 1000*rank + 0.1*i;
      // фактическая отправка данных
       MPI_Startall(2, req);
       MPI_Waitall(2, req, stat);
    }
 }

  MPI_Finalize();
  return 0;
}
```

8.3 Задание 3. Использование функций одностороннего обмена сообщениями.

Одним из преимуществ MPI-2 является возможность использование функций одностороннего обмена сообщениями. Ниже приведена программа, иллюстрирующая эту возможность.

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char* argv[]) {
 int size, rank;
 if( MPI_Init( &argc, &argv ) ) return -1;
 MPI_Comm_size( MPI_COMM_WORLD, &size );
 MPI_Comm_rank( MPI_COMM_WORLD, &rank );

 // Массив для заполнения
 double *data;
 if(rank == 0) data = new double[size];

 // Инициализация общей области памяти ("окна")
 MPI_Win win;
 MPI_Win_create(data, sizeof(data), sizeof(data),
                MPI_INFO_NULL, MPI_COMM_WORLD, &win);
 MPI_Win_fence(0, win);  // Начало работы с окном

 // Односторонние коммуникации
 double ldata = rank*rank;
 MPI_Put(&ldata, 1, MPI_DOUBLE, 0, rank,
                   1, MPI_DOUBLE, win); // rank > 0

 // Синхронизация, завершение работы с окном
 MPI_Win_fence(0, win);
 MPI_Win_free(&win);
```

```
 // Вывод результатов на нулевом процессе
 if(rank == 0)
   for(int i=0; i<size; i++)
        printf("data[%d] = %g\n", i, data[i]);

  if(rank == 0) delete data;
  MPI_Finalize();
  return 0;
}
```

Практическое занятие 9. Использование пакета молекулярно-динамического моделирования LAMMPS

9.1 Общие сведения о пакете программ LAMMPS.

LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator) является пакетом программ молекулярно-динамического моделирования с открытым исходным кодом, распространяемым под лицензией GPL. Пакет разрабатывается Сандийской национальной лаборатории США. Его исходный код и документация доступны на сайте http://lammps.sandia.gov.

9.2 Задание 1. Компиляция и запуск LAMMPS.

Для компиляции следует перейти в поддиректорию src, открыть в текстовом редакторе один из фалов MAKE/Makefile.serial (для компиляции последовательной версии) или MAKE/Makefile.linux (для компиляции параллельной версии с поддержкой MPI) и изменить настройку переменных окружения и параметры компиляции в соответствии с настойками вычислительной системы. Далее из командной строки необходимо выполнить команды:

```
make serial
```

или

```
make linux
```

соответственно. При необходимости LAMMPS можно скомпилировать как библиотеку подпрограмм для встраивания отдельных модулей МД моделирования в ваше приложение.

На учебном кластере установлена уже скомпилированная версия LAMMPS, запуск последовательно версии которой осуществляется командой

```
lmp_serial [ -echo none | screen | log | both ]
           [ –partition NxCPUs ... ]
           [-in file]
           [ -log file ] [ -screen file ]
           [ -var name value ]
```

9.3 Задание 2. Выполнение тестового расчета.

Для выполнения простейшего примера и проверки работоспособности программы рекомендуется выполнить следующие действия.

```
cp -R /usr/local/lammps/examples/melt .
cd melt
ls
lmp_serial –in in.melt
```

В результате на экран должен быть выведен протокол работы программы (см. пример на рис. 9.1), в файл log.lammps – полная диагностика, а в dump.melt – МД траектория движения частиц (см. пример на рис. 9.2). Просмотреть их можно командами

```
less log.lammps
less dump.melt
```

```
LAMMPS (5 Oct 2007)
Lattice spacing in x,y,z = 1.6796 1.6796 1.6796
Created orthogonal box = (0 0 0) to (16.796 16.796 16.796)
  1 by 1 by 1 processor grid
Created 4000 atoms
Setting up run ...
Memory usage per processor = 2.1554 Mbytes
Step Temp E_pair E_mol TotEng Press
       0            3   -6.7733681            0   -2.2744931   -3.7033504
...
     250    1.6275257   -4.7224992            0    -2.281821    5.9567365
Loop time of 4.00001 on 1 procs for 250 steps with 4000 atoms

Pair  time (%) = 2.93709 (73.427)
Neigh time (%) = 0.314858 (7.87144)
Comm  time (%) = 0.241926 (6.04813)
Outpt time (%) = 0.0869321 (2.1733)
Other time (%) = 0.419206 (10.4801)

Nlocal:    4000 ave 4000 max 4000 min
Histogram: 1 0 0 0 0 0 0 0 0 0
Nghost:    5499 ave 5499 max 5499 min
Histogram: 1 0 0 0 0 0 0 0 0 0
Neighs:    151513 ave 151513 max 151513 min
Histogram: 1 0 0 0 0 0 0 0 0 0

Total # of neighbors = 151513
Ave neighs/atom = 37.8783
Neighbor list builds = 12
Dangerous builds = 0
```

Рис. 9.1. Образец вывод программы LAMMPS на экран при выполнении тестового расчета.

```
ITEM: TIMESTEP
0
ITEM: NUMBER OF ATOMS
4000
ITEM: BOX BOUNDS
0 16.796
0 16.796
0 16.796
ITEM: ATOMS
1 1 0 0 0
2 1 0.05 0.05 0
3 1 0.05 0 0.05
4 1 0 0.05 0.05
5 1 0.1 0 0
6 1 0.15 0.05 0
7 1 0.15 0 0.05
8 1 0.1 0.05 0.05
9 1 0.2 0 0
10 1 0.25 0.05 0
11 1 0.25 0 0.05
12 1 0.2 0.05 0.05
13 1 0.3 0 0
14 1 0.35 0.05 0
15 1 0.35 0 0.05
16 1 0.3 0.05 0.05
17 1 0.4 0 0
18 1 0.45 0.05 0
19 1 0.45 0 0.05
20 1 0.4 0.05 0.05
...
```

Рис. 9.2. Структура dump-файла LAMMPS с траекторией движения частиц.

Для выполнения того же примера с применением системы очередей PBS, обсуждавшейся в одной из предыдущих лекций, необходимо создать скрипт (текстовый файл) pbsrun.sh следующего вида

```
#!/bin/bash
### Job name
#PBS -N lammps
cd $HOME/melt
/usr/local/bin/lmp_serial -in in.melt
```

Полученный скрипт необходимо сделать исполняемым и поставить в очередь на исполнение командами

```
chmod u+x pbsrun.sh
qsub pbsrun.sh
```

В процессе выполнения программы можно использовать следующую команду для получения информации о текущем состоянии

```
tail –f log.melt
```

9.4 Задание 3. Модификация входного файла с параметрами.

Входного файла с параметрами является текстовым файлом специального формата и его имя задается в опции –in командой строки. Инструкции во входном файле выполняются последовательно, поэтому их порядок важен. Например, указанные две последовательности приводят к разным результатам

```
run 100            timestep 0.5
timestep 0.5       run 100
```

Рассмотрим основные параметры, задаваемые во входном файле LAMMPS на примере, приведенном в предыдущем задании. Входной файл параметров в данном примере называется in.melt. Ниже приведено его содержимое с комментариями относительно назначения тех или иных параметров.

```
# 3d Lennard-Jones melt
# Символ "#" в начале строки означает комментарий
```

```
units           lj
# Единицы измерения: lj или real или metal или si или
cgs

atom_style      atomic
# Определяет, какие атрибуты используются для описания
состояния каждого атома: angle или atomic или bond или
charge или dipole или dpd или ellipsoid или full или
granular или molecular или peri или hybrid

lattice         fcc 0.8442
# Типы кристаллических решеток: sc или bcc или fcc или
hcp или diamond или sq или sq2 или hex или custom
# Для L-J: приведенная плотность, для не L-J –
постоянная решетки

region          box block 0 10 0 10 0 10
# Тип геометрии: block или cylinder или prism или sphere
или union или intersect
# Координаты в постоянных решетки (4 атома на эл. объем
для fcc)

create_box      1 box
# Создает ячейку в соответствии с областью ‘box’, 1 –
кол-во типов атомов

create_atoms    1 box
# Помещает атомы в ячейку или область, 1 – тип атомов

mass            1 1.0
# Масса для атомов типа 1

velocity        all create 3.0 87287
```

```
# Скорости для атомов типа all.
# Второй параметр: create или set или scale или ramp или
zero
# Третий – температура, четвертый – random seed

pair_style      lj/cut 2.5
pair_coeff      1 1 1.0 1.0 2.5
# Выбор и настройка потенциала: L-J с отсечкой 2.5
# Аргументы команды pair_coeff: I J epsilon sigma
cutoff1 cutoff2

neighbor        0.3 bin
neigh_modify    every 20 delay 0 check no
# Настройка списка ближайших соседей

fix             1 all nve
# Схема интегрирования уравнений движения (Верле второго
порядка), формат команды: fix ID group-ID style args

dump            id all atom 50 dump.melt
# Настройка вывода МД траекории
# dump ID group-ID style N file args
# N – частота вывода

thermo          50
# Частота вывода диагностики

run             250
# Выполнение 250 шагов МД моделирования
```

Для выполнения задания по ознакомлению с возможностями LAMMPS рекомендуется следующая модификация входного файла:

1. Отключить вывод траектории.

2. Уменьшить в 10 раз плотность.
3. Изменить радиус отсечки потенциала.
4. Вставить timestep 0.01 (по умолчанию 0.005 для L-J), timestep 0.05, timestep 0.001.
5. Использовать термостат Нозе-Хувера: fix 1 all nvt 3. 3. 0.1 (оставить шаг 0.001).
6. Использовать термостат Ланжевена: fix 1 all nve; fix 2 all langevin 3. 3. 0.1 1111

9.5 Задание 4. Визуализация результата расчета.

Для визуализации используется МД траектория в файле dump.melt, полученная при выполнении предыдущих заданий.

Простейшая визуализация возможна с помощью встроенного в пакет LAMMPS приложения xmovie:

```
xmovie –scale dump.melt
```

Для ознакомления рекомендуется сделать вывод в траекторию на каждом шаге, использовать всего 50 шагов.

Более мощные возможности для визуализации предоставляется пакет AtomEye. Для его работы необходимо создать конфигурационный файл dump2cfgs.melt(конечный файл dump2cfgs.melt не должен содержать комментариев)

```
vi dump2cfgs.melt
```

```
1           # число типов атомов в dump-файле
dump.melt   # имя dump-файла
0           # номер первого шага интегрирования
250         # номер последнего шага
1           # частота перебора шагов
1 1.0 Ar    # номер типа атома, масса, двухсимвольное
обозначение
```

Далее следует выполнить команды

```
dump2cfgs dump2cfgs.melt
AtomEye dump.melt_ts=000000.cfg
```

В результате выполнения этих команд буде открыто окно с трехмерным отображением расчетной ячейки. Обучаемый имеет возможность настраивать параметры отображения атомов, изменять угол зрения, масштаб и другие параметры.