МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА»

Институт информатики, математики и электроники
Факультет информатики
Кафедра технической кибернетики

Параллельное программирование

Методические указания
к лабораторной работе №2

САМАРА 2017

Составитель: доц., Козлова Е.С.

УДК 519.681

Методические указания к лабораторным работам

Самарский государственный аэрокосмический университет имени академика С.П.КОРОЛЁВА (национальный исследовательский университет)

Составитель: Е.С. Козлова

Самара, 2018. 12 с.

Методические указания предназначены для бакалавров направления 010400.62 «Прикладная математика и информатика»

Печатается по решению редакционно-издательского совета Самарского университета

Рецензент:

1 Стандарт OpenMP

1.1 Модель программирования OpenMP

OpenMP (Open specifications for Multi–Processing) – стандарт для написания параллельных программ для многопроцессорных вычислительных систем с общей оперативной памятью. Программа представляется как набор нитей (threads), объединённых общей памятью, где проблема синхронизации решается введением критических секций и мониторов.

OpenMP используется модель параллельного выполнения “ветвление-слияние” (*fork-join*). Программа начинается выполнением одной нити, называемой начальной (*initial*) нитью. Начальная нить выполняется последовательно. Когда нить достигает директивы *parallel* она создает команду нитей, состоящую из неё самой и нуля или более дополнительных нитей, и становится хозяйкой (*master*) созданной команды. Все члены команды исполняют код структурной области, связанной с директивой *parallel* (параллельной области). В конце параллельной области размещается неявный барьер. Только нить-хозяйка продолжает выполнение после завершения параллельной области.

Число нитей в команде, выполняющихся параллельно, можно контролировать несколькими способами. Один из них – использование переменной окружения OMP_NUM_THREADS. Другой способ – вызов процедуры *omp_set_num_threads()*. Еще один способ – использование выражения *num_threads* в сочетании с директивой *parallel*.

1.2 Директивы OpenMP

В C/C++ директивы OpenMP определяются конструкциями *#pragma*, предусматривающимися стандартами C и C++, и используемых для задания дополнительных указаний компилятору. Использование специальной ключевой директивы “omp” указывает на то, что команды относятся к OpenMP и для того, чтобы исключить случайные совпадения имён директив OpenMP с другими именами. Таким образом директивы *#pragma* для работы с OpenMP имеют следующий формат:

#pragma omp директива<> опция[[[,] опция]...]

Объектом действия большинства директив является один оператор или блок, перед которым расположена директива в исходном тексте программы. Директивы – регистрозависимы, однако порядок опций в описании директивы несущественен, в одной директиве большинство опций может встречаться несколько раз. После некоторых опций может следовать список переменных, разделяемых запятыми.

Директива *parallel* создает параллельную область для следующего за ней структурированного блока, параллельная область задаётся при помощи записи:

#pragma omp parallel опция[[[,] опция]...]структурированный блок

Возможные опции:

if (условие) – выполнение параллельной области по условию. Вхождение в параллельную область осуществляется только при выполнении некоторого условия. Если условие не выполнено, то директива не срабатывает и продолжается обработка программы в прежнем режиме;

num_threads (целочисленное выражение) – явное задание количества нитей, которые будут выполнять параллельную область; по умолчанию выбирается последнее значение, установленное с помощью функции *omp_set_num_threads(),* или значение переменной OMP_NUM_THREADS;

default(shared|none) – всем переменным в параллельной области, которым явно не назначен класс, будет назначен класс *shared*; *none* означает, что всем переменным в параллельной области класс должен быть назначен явно;

private (список) – задаёт список переменных, для которых порождается локальная копия в каждой нити; начальное значение локальных копий переменных из списка не определено;

firstprivate (список) – задаёт список переменных, для которых порождается локальная копия в каждой нити; локальные копии переменных инициализируются значениями этих переменных в нити-мастере;

shared (список) – задаёт список переменных, общих для всех нитей;

copyin (список) – задаёт список переменных, объявленных как threadprivate, которые при входе в параллельную область инициализируются значениями соответствующих переменных в нити-мастере;

reduction (оператор:список) – задаёт оператор и список общих переменных; для каждой переменной создаются локальные копии в каждой нити; локальные копии инициализируются соответственно типу оператора (для аддитивных операций – 0 или его аналоги, для мультипликативных операций – 1 или её аналоги); над локальными копиями переменных после выполнения всех операторов параллельной области выполняется заданный оператор; оператор это: +, *, -, &, |, ^, &&, ||; порядок выполнения операторов не определён, поэтому результат может отличаться от запуска к запуску.

Все порождённые нити исполняют один и тот же код, соответствующий параллельной области. Предполагается, что в SMP-системе нити будут распределены по различным процессорам (однако это, как правило, находится в ведении операционной системы). Нить может узнать свой номер с помощью вызова библиотечной функции *omp_get_thread_num*.

Директива *for* служит для распределения итераций цикла между различными нитями можно использовать директиву *for*:

#pragma omp for опция[[[,] опция] ...]цикл for

Эта директива относится к идущему следом за данной директивой блоку, включающему оператор for. Если в параллельной области встретился оператор цикла без директивы, то, согласно общему правилу, он будет выполнен всеми нитями текущей группы, то есть каждая нить выполнит все итерации данного цикла.

Возможные опции:

private;

firstprivate;

lastprivate (список) – переменным, перечисленным в списке, присваивается результат с последнего витка цикла;

reduction;

schedule(type[, chunk]) – опция задаёт, каким образом итерации цикла распределяются между нитями;

collapse(n) – опция указывает, что n последовательных тесновложенных циклов ассоциируется с данной директивой; для циклов образуется общее пространство итераций, которое делится между нитями; если опция *collapse* не задана, то директива относится только к одному непосредственно следующему за ней циклу;

ordered – опция, говорящая о том, что в цикле могут встречаться директивы ordered; в этом случае определяется блок внутри тела цикла, который должен выполняться в том порядке, в котором итерации идут в последовательном цикле;

nowait – в конце параллельного цикла происходит неявная барьерная синхронизация параллельно работающих нитей: их дальнейшее выполнение происходит только тогда, когда все они достигнут данной точки; если в подобной задержке нет необходимости, опция *nowait* позволяет нитям, уже дошедшим до конца цикла, продолжить выполнение без синхронизации с остальными.

На вид параллельных циклов накладываются достаточно жёсткие ограничения. В частности, предполагается, что корректная программа не должна зависеть от того, какая именно нить какую итерацию параллельного цикла выполнит. Нельзя использовать побочный выход из параллельного цикла. Размер блока итераций, указанный в опции schedule, не должен изменяться в рамках цикла.

Эти требования введены для того, чтобы OpenMP мог при входе в цикл точно определить число итераций. Если директива параллельного выполнения стоит перед гнездом циклов, завершающихся одним оператором, то директива действует только на самый внешний цикл. Итеративная переменная распределяемого цикла по смыслу должна быть локальной, поэтому в случае, если она специфицирована общей, то она неявно делается локальной при входе в цикл. После завершения цикла значение итеративной переменной цикла не определено, если она не указана в опции *lastprivate*.

Директива *critical* оформляет критическую секцию программы:

#pragma omp critical имя[()]структурированный блок

Критическая секция запрещает одновременное исполнение структурированного блока более чем одним потоком. В каждый момент времени в критической секции может находиться не более одной нити. Если критическая секция уже выполняется какой-либо нитью, то все другие нити, выполнившие директиву для секции с данным именем, будут заблокированы, пока вошедшая нить не закончит выполнение данной критической секции. Как только работавшая нить выйдет из критической секции, одна из заблокированных на входе нитей войдет в неё. Если на входе в критическую секцию стояло несколько нитей, то случайным образом выбирается одна из них, а остальные заблокированные нити продолжают ожидание.

Директива *atomic* блокирует доступ к переменной всем запущенным в данный момент нитям, кроме нити, выполняющей операцию:

#pragma omp atomic [read | write | update | capture]оператор

или

#pragma omp atomic capture структурированный блок

Частым случаем использования критических секций на практике является обновление общих переменных. Например, если переменная sum является общей и оператор вида *sum = sum + expr* находится в параллельной области программы, то при одновременном выполнении данного оператора несколькими нитями можно получить некорректный результат. Чтобы избежать такой ситуации можно воспользоваться

механизмом критических секций или специально предусмотренной для таких случаев директивой *atomic*.

2 Стандарт MPI

2.1 Модель программирования MPI

Под параллельной программой в рамках MPI понимается множество одновременно выполняемых процессов. Все процессы порождаются один раз, образуя параллельную часть программы. Каждый процесс работает в своем адресном пространстве, никаких общих переменных или данных в MPI нет. Процессы могут выполняться на разных процессорах, но на одном процессоре могут располагаться и несколько процессов. В предельном случае для выполнения параллельной программы может использоваться один процессор – как правило, такой способ применяется для начальной проверки правильности параллельной программы.

Каждый процесс параллельной программы порождается на основе копии одного и того же программного кода (модель SPMP - Single Programm Multiple Processes). Количество процессов и число используемых процессоров определяется в момент запуска параллельной программы средствами среды исполнения MPI-программ и в ходе вычислений меняться не может (в стандарте MPI-2 предусматривается возможность динамического изменения количества процессов). Все процессы программы последовательно перенумерованы от 0 до p-1, где p есть общее количество процессов. Номер процесса именуется рангом процесса. При этом для того, чтобы избежать идентичности вычислений на разных процессорах, можно, во-первых, подставлять разные данные для программы на разных процессорах, а во-вторых, с помощью средств для идентификации процесса, на котором выполняется программа, организовать различия в вычислениях в зависимости от используемого программой процессора. Это позволяет загружать ту или иную подзадачу в зависимости от “номера” процессора.

Основу MPI составляют операции передачи сообщений. Сообщение – это набор данных некоторого типа. Каждое сообщение имеет атрибуты, такие как: номер процесса – отправителя, номер процесса – получателя, идентификатор сообщения и другие. Одним из важных атрибутов сообщения является его идентификатор или тэг. По идентификатору процесс, принимающий сообщение, например, может различить два сообщения, пришедшие к нему от одного и того же процесса. Сам идентификатор сообщения является целым неотрицательным числом. Среди предусмотренных в составе MPI функций различаются парные (point-to-point) операции между двумя процессами и коллективные (collective) коммуникационные действия для одновременного взаимодействия нескольких процессов.

2.2 Обязательные функции MPI

Для инициализации и завершения среды MPI, а так же организации параллельной работы в коде программы используется базовый набор функций.

Функция *MPI_Init* устанавливает “среду” (environment) MPI, предшествует всем другим вызовам MPI:

int MPI_Init (int argc, char*** argv)*

где *argc* – указатель на количество параметров командной строки; *argv* – параметры командной строки. Допускается только одно обращение к *MPI_Init*.

Функция *MPI_Comm_size* определяет число процессов в области связи:

*int MPI_Comm_size(MPI_Comm comm, int *size)*

где *comm* – коммуникатор; *size* – число процессов в области связи коммуникатора *comm*.

Каким способом пользователь запускает эти процессы – зависит от реализации, но любая программа может определить число запущенных процессов с помощью данного вызова. Значение *procs_count* – это, по сути, размер группы, связанной с коммуникатором *MPI_COMM_WORLD*.

Функция *MPI_Comm_rank* определения номера процесса:

*int MPI_Comm_rank(MPI_Comm comm, int *rank)*

где *comm* – коммуникатор, *rank* – номер процесса, вызвавшего функцию.

Процессы каждой группы пронумерованы целыми числами, начиная с 0, которые называются рангами (*rank*). Каждый процесс определяет свой номер в группе, связанной с данным коммуникатором, с помощью *MPI_Comm_rank*. Таким образом, каждый процесс получает одно и то же число в *procs_count*, но разные числа в *rank*.

Функция *MPI_Finalize* завершает MPI программу:

int MPI_Finalize(void)

Эта функция должна быть выполнена каждым процессом MPI и приводит к ликвидации “среды” MPI. Никакие вызовы MPI не могут быть осуществлены процессом после вызова MPI_Finalize (повторный MPI_Init также невозможен).

2.3 Функции передачи сообщений MPI

Начнем рассмотрение средства взаимодействия в MPI с так называемых точечных обменов. Это название происходит от английского “point-to-point communications”, которое означает, что у взаимодействия есть две точки: процесс-отправитель и процесс-получатель сообщения. Точечные, так же как и обмены других типов, всегда производятся в пределах одного коммуникатора, который указывается в качестве параметра в вызовах функций обмена. Ранги процессов, принимающих участие в обменах, вычисляются по отношению к указанному коммуникатору.

Функция *MPI_Send* отправляет сообщение:

int MPI_Send(void **buf*, int *count*, MPI_Data*type type*, int *dest*, int *tag*, MPI_*Comm com m*);

где *buf* – адрес начала расположения пересылаемых данных; *count* – число пересылаемых элементов; *type* – MPI-тип посылаемых элементов; *dest* – номер процесса-получателя в группе, связанной с коммуникатором *comm*; *tag* – идентификатор сообщения (аналог типа сообщения функций nread и nwrite PSE nCUBE2); *comm* – коммуникатор области связи.

Функция *MPI_Recv* принимает сообщение:

*int MPI_Recv(void *buf, int count, MPI_Datatype type, int source, int tag, MPI_Comm c omm, MPI_Status *status);*

где *buf* – адрес начала расположения памяти для получаемых данных; *count* – число получаемых элементов; *type* – MPI-тип получаемых элементов; *source* – ранг процесса, от которого осуществляется прием сообщения; *tag* – ожидаемый идентификатор сообщения; *comm* - коммуникатор, в рамках которого выполняется прием данных; *status* – атрибуты принятого сообщения.

Одной из наиболее привлекательных сторон MPI является наличие широкого набора коллективных операций, которые берут на себя выполнение наиболее часто встречающихся при программировании действий. Например, часто возникает потребность разослать некоторую переменную или массив из одного процессора всем остальным. Каждый программист может написать такую процедуру с использованием операций *Send*/*Recv*, однако гораздо удобнее воспользоваться коллективной операцией *MPI_Bcast*. Причем гарантировано, что эта операция будет выполняться гораздо эффективнее, поскольку MPI – функция реализована с использованием внутренних возможностей коммуникационной среды. Главное отличие коллективных операций от операций типа "точка-точка" состоит в том, что в них всегда участвуют все процессы, связанные с некоторым коммуникатором. Несоблюдение этого правила приводит либо к аварийному завершению задачи, либо к еще более неприятному зависанию задачи.

Функция *MPI_Bcast* осуществляет широковещательную рассылку данных:

int MPI_Bcast(void buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)*

где *buf* – адрес начала расположения в памяти рассылаемых данных, *count* – число посылаемых элементов, *datatype* – тип посылаемых элементов, *root* – номер процесса-отправителя, *comm* – коммуникатор.

В параллельном программировании математические операции над блоками данных, распределенных по процессорам, называют глобальными операциями редукции. В общем случае операцией редукции называется операция, аргументом которой является вектор, а результатом – скалярная величина, полученная применением некоторой математической операции ко всем компонентам вектора. В частности, если компоненты вектора расположены в адресных пространствах процессов, выполняющихся на различных процессорах, то в этом случае говорят о глобальной (параллельной) редукции. Например, пусть в адресном пространстве всех процессов некоторой группы процессов имеются копии переменной var (необязательно имеющие одно и то же значение), тогда применение к ней операции вычисления глобальной суммы или, другими словами, операции редукции SUM возвратит одно значение, которое будет содержать сумму всех локальных значений этой переменной. Использование этих операций является одним из основных средств организации распределенных вычислений.

Функция *MPI_Reduce* выполняет операцию глобальной редукции, указанную параметром op, над первыми элементами входного буфера, и результат посылает в первый элемент буфера приема процесса *root*, затем то же самое делает для вторых элементов буфера и т.д.:

int MPI_Reduce(void sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)*

где *sendbuf* – адрес начала входного буфера, *recvbuf* – адрес начала буфера результатов (используется только в процессе-получателе *root*), *count* – число элементов во входном буфере, *datatype* – тип элементов во входном буфере, *op* – операция, по которой выполняется редукция, *root* – номер процесса-получателя результата операции, *comm* – коммуникатор.

3 Задание на лабораторную работу

3.1 Лабораторная работа 2

Произвести запуск программ для поиска суммы элементов, которые используют технологии MPI и OpenMP, на различном количестве процессоров (потоков). Для корректной работы программ в шаблоны для запуска, представленные ниже, добавить код инициализации входных данных.

На основе технологии OpenMP реализуйте три варианта с использованием:

1) опции *reduction*;
2) директивы *critical*;
3) директивы *atomic*.

На основе технологии MPI реализуйте два варианта сбора информации на 0-ой процессор с использованием:

1) операций "точка-точка";
2) коллективной операции.

В программе с использованием технологией MPI для корректной декомпозиции входных данных необходимо использовать широковещательную рассылку исходного массива, который генерируется только 0-ым процессом.

Для упрощения процесса написания программы допускается использование размерности вектора, кратной количеству процессоров.

В ходе анализа работы программы оцените время ее выполнения на различном количестве исполняющих нитей (процессов). Оцените влияние различных функций и директив на скорость работы приложений.

Ниже приведен шаблоны программ для запуска.

Пример 1. Код OpenMP

```
#include <omp.h>
#define CHUNK 100
#define NMAX 1000000
#include "stdio.h"

int main(int argc, char* argv[]){
      omp_set_num_threads(2);
      int i;
      double a[NMAX], sum;
      //заполнение массива
      double st_time, end_time;
      st_time = omp_get_wtime();
      sum = 0;
#pragma omp parallel for shared(a) private(i)
      for (i=0; i<NMAX; i++){
            sum = sum + a[i];
      }
      end_time = omp_get_wtime ();
```

```
     end_time = end_time - st_time;
     printf("\nTotal Sum = %10.2f",sum);
     printf("\nTIME OF WORK IS %f ", end_time);
     return 0;
}
```

Пример 2. Код MPI передачи по звезде

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

int main(int argc, char* argv[])
{
     double x[1000000], TotalSum, ProcSum = 0.0;
     int ProcRank, ProcNum, N=1000000,i;
     MPI_Status Status;

     double st_time, end_time;

     MPI_Init(&argc,&argv);
     MPI_Comm_size(MPI_COMM_WORLD,&ProcNum);
     MPI_Comm_rank(MPI_COMM_WORLD,&ProcRank);
     if (ProcRank==0)
     {
          //заполнение массива
     }
     // подготовка данных, рассылка 0-ым процессом по всем
остальным
     st_time = MPI_Wtime();

     int k = N / ProcNum;
     int i1 = k * ProcRank;
     int i2 = k * ( ProcRank + 1 );

     if ( ProcRank == ProcNum-1 ) i2 = N;
     for ( i = i1; i < i2; i++ )
          ProcSum = ProcSum + x[i];

     if ( ProcRank == 0 )
     {
          TotalSum = ProcSum;
          for ( i = 1; i < ProcNum; i++ )
          {
               MPI_Recv(&ProcSum,    1,    MPI_DOUBLE,    i,    0,
MPI_COMM_WORLD,&Status);
               TotalSum = TotalSum + ProcSum;
```

```
            }
        }
        else
            MPI_Send(&ProcSum,     1,     MPI_DOUBLE,     0,     0,
MPI_COMM_WORLD);

        MPI_Barrier(MPI_COMM_WORLD);

        end_time = MPI_Wtime();
        end_time = end_time - st_time;

        if ( ProcRank == 0 )
        {
            printf("\nTotal Sum = %10.2f",TotalSum);
            printf("\nTIME OF WORK IS %f ", end_time);
        }

        MPI_Finalize();
        return 0;
}
```