**Федеральное государственное бюджетное образовательное учреждение высшего образования «Ивановский государственный энергетический университет имени В.И. Ленина»**

**Отчет по курсовой работе**

**по технологиям параллельного программирования**

**Выполнил: Кабанов К.О.**

**Проверила: Чернышева Л.П.**

**Иваново 2018**

**Задание №1 - Научная работа**

**Постановка задачи:**

Разработать программный комплекс на МВС для моделирования поведения частиц тела при ударе о неразрушимую стену.

**Актуальность задачи:**

Расчет изменения положения частиц тела после удара является очень важной задачей, которая находит применение в строительстве, автомобильной промышленности, металлургии, спорте и многих других областях, где необходимо точно рассчитать и учесть изменения, происходящие с телом после удара.

**Математическая модель:**

В задаче рассматривается тело, состоящее из N частиц. Из этих частиц выделим две: i и j, которые взаимодействуют друг с другом с силой, зависящей от расстояния между ними. Силы равны и противоположно направленны. В задаче не исследуется характер деформаций, которые имеют место при ударе, а определяется изменение скоростей частиц, вызванное уже свершившимся ударом. Будем рассматривать влияние на i –ю только ближайших частиц, входящих в ее окрестность. Для проверки на близость был выбран метод Эйлера.

Метод Эйлера предполагает, что исследуемая область с движущимися частицами покрывается сеткой, все пространство разбивается на квадратные ячейки со стороной *R* - радиус взаимодействия. Все частицы в соответствии с их координатами распределяются по ячейкам, по каждой из которых составляются списки входящих частиц. Для *i*-й частицы считается, что на нее действуют частицы из соприкасающихся ячеек и из той же ячейки, что и она сама. Затраты машинного времени пропорциональны *N* - числу частиц.

i = 1, 2, …, N; j = 1, 2, …, N.

где – константы; – константы, описывающие степени; - расстояние между частицами; – масса *i* – й частицы; – скорости *i* – й частицы по оси X и по оси Y соответственно; *t* – время; *i, j* – номера частиц; – координаты *i* – й и *j* – й частиц по оси X; – координаты *i* – й и *j* – й частиц по оси Y.

**Расчетные формулы:**

Для реализации задачи был выбран метод Рунге-Кутты второго порядка, были составлены такие уравнения:

1 этап:

2 этап:

**Алгоритм:**

На хосте:

1. Определяются начальные данные: положение точек тела внутри области (генерируется случайно или используется равномерное распределение), задаются начальные значения скоростей для каждой точки.
2. На всю расчетную область накладывается сетка, все пространство разбивается на квадратные ячейки со стороной *R* – радиус взаимодействия между частицами. Все частицы распределяются по ячейкам, по каждой из которых составляются списки входящих частиц.
3. На каждой итерации по времени вызывается функция девайса, выполняется синхронизация всех потоков, в массив старых значений записываются вычисленные на этой итерации.

На девайсе:

1. Каждый поток вычисляет значения скоростей, учитывая свое положение в полученной на хосте сетке. Для *i*-й частицы считается, что на нее действуют частицы из соприкасающихся ячеек и из той же ячейки, что и она сама.
2. Вычисляются новые координаты частицы.
3. Проверяются новые координаты частиц на столкновение со стеной, если частица столкнулась, то необходимо взять текущую скорость по координате *x* с обратным знаком и установить координаты точки в месте столкновения.
4. Полученные значения скоростей и координат записываются в массив с новыми значениями.

**Результаты:**

Результаты вычислений сохраняются в файл “OUT.xml”, после чего его можно открыть в специальной программе, написанной на языке C#, которая визуально изобразит положения точек в определенный момент времени.

Пример файла с результатами:

<out>

<settings tcount="100" tau="0.100000" zoneX="0.000000" zoneY="0.000000" zoneWIDTH="15.000000" zoneHEIGHT="10.000000" wallX="10.000000"/>

<time tid="0">

<point id="0" x="7.598254" y="3.193426"/>

<point id="1" x="5.526048" y="4.093234"/>

<point id="2" x="6.936369" y="3.052431"/>

<point id="3" x="5.930113" y="2.133335"/>

<point id="4" x="5.473189" y="2.227576"/>

<point id="5" x="5.773858" y="1.314341"/>

<point id="6" x="7.332255" y="3.826991"/>

<point id="7" x="7.297708" y="2.222205"/>

<point id="8" x="6.835902" y="2.642994"/>

<point id="9" x="6.845180" y="3.663167"/>

</time>

…

<time tid="100">

<point id="0" x="2.499998" y="3.193426"/>

<point id="1" x="4.499996" y="4.093234"/>

<point id="2" x="3.099998" y="3.052431"/>

<point id="5" x="4.299996" y="1.314341"/>

<point id="6" x="2.699998" y="3.826991"/>

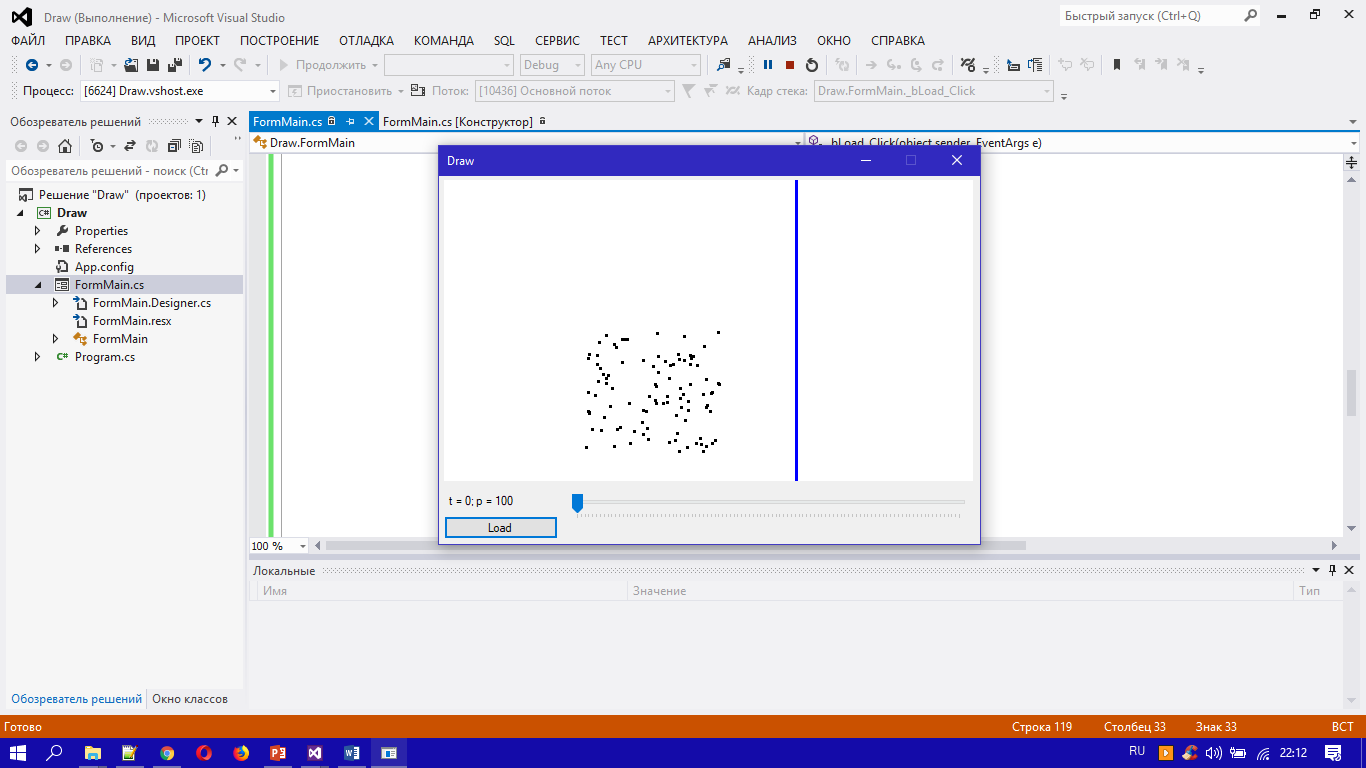
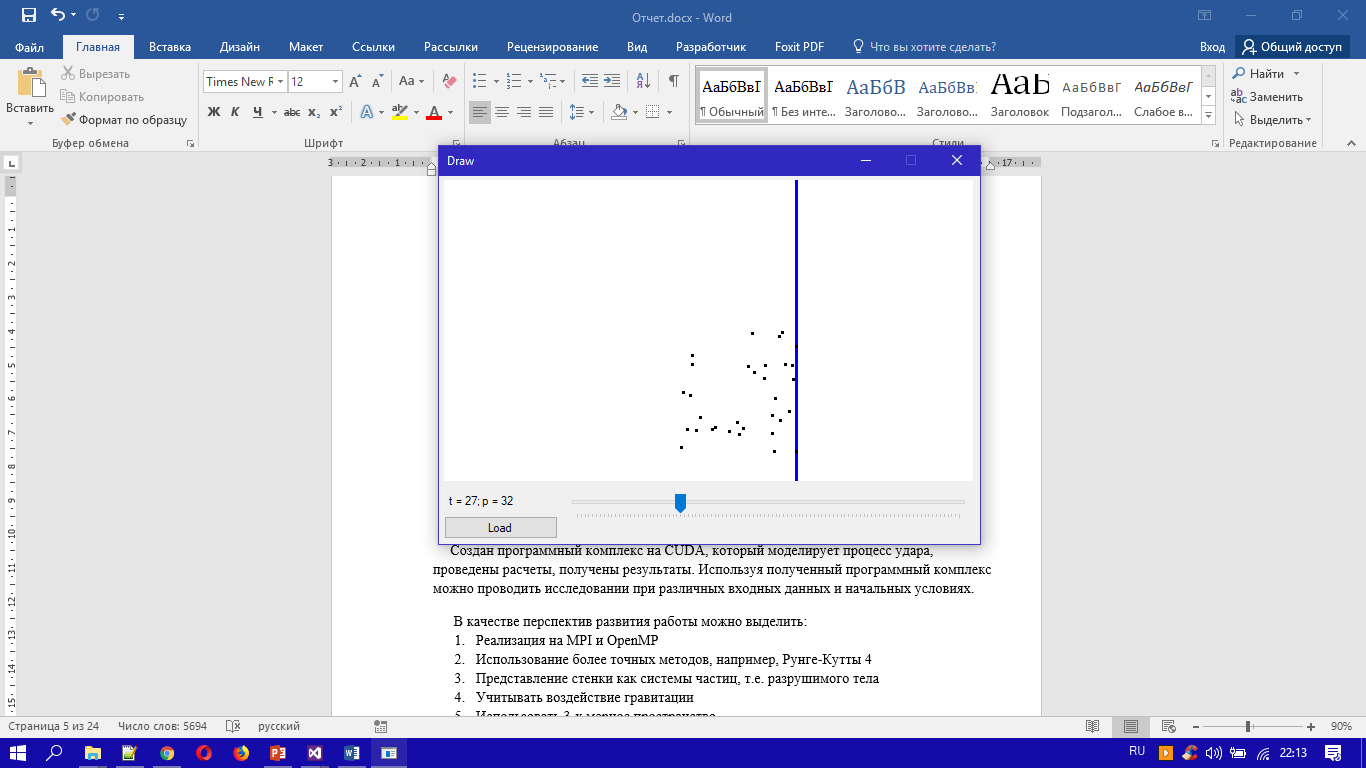
<point id="8" x="3.199997" y="2.642994"/>

<point id="9" x="3.199997" y="3.663167"/>

</time>

</out>

Пример работы программы визуализации:

**Вывод:**

Создан программный комплекс на CUDA, который моделирует процесс удара, проведены расчеты, получены результаты. Используя полученный программный комплекс можно проводить исследовании при различных входных данных и начальных условиях.

В качестве перспектив развития работы можно выделить:

1. Реализация на MPI и OpenMP
2. Использование более точных методов, например, Рунге-Кутты 4
3. Представление стенки как системы частиц, т.е. разрушимого тела
4. Учитывать воздействие гравитации
5. Использовать 3-х мерное пространство

**Текст программы:**

#include "cuda\_runtime.h"

#include <device\_launch\_parameters.h>

#include <stdio.h>

#include <windows.h>

#include <ctime>

#include <math.h>

#define N 100

#define Tau 0.1f

#define TMAX 10.0f

#define R 0.1f

#define m 1.0f

#define n1 6

#define n2 12

#define A1 1.0f

#define A2 2.0f

#define RectX 4.0f

#define RectY 1.0f

#define RectHEIGHT 5.0f

#define RectWIDTH 8.0f

#define WallX 10.0f

#define ZoneX 0.0f

#define ZoneY 0.0f

#define ZoneHEIGHT 10.0f

#define ZoneWIDTH 15.0f

struct U

{

bool active;

float Vx;

float Vy;

float X;

float Y;

};

struct xy

{

float X;

float Y;

};

float randFloat() //return random float numer [0; 1]

{

return static\_cast <float> (rand()) / static\_cast <float> (RAND\_MAX);

}

float randFloat(float a) //return random float numer [0; a]

{

return static\_cast <float> (rand()) / (static\_cast <float> (RAND\_MAX/a));

}

float randFloat(float a, float b) //return random float numer [a; b]

{

return a + static\_cast <float> (rand()) /( static\_cast <float> (RAND\_MAX/(b-a)));

}

void getCells(xy \*cells, int sizeX, int sizeY, U \*u)

{

for(int x=0; x<sizeX; x++)

{

for(int y=0; y<sizeY; y++)

{

for(int i=0; i<N; i++)

{

if(u[i].active)

{

if(((u[i].X >= x\*R) && (u[i].X < (x+1)\*R)) && ((u[i].Y >= y\*R) && (u[i].Y < (y+1)\*R)))

{

cells[i].X = x;

cells[i].Y = y;

}

}

}

}

}

}

\_\_device\_\_ float r(float x1, float y1, float x2, float y2)

{

return pow(pow(x2 - x1, 2.0f) + pow(y2 - y1, 2.0f), 0.5f);

}

\_\_device\_\_ xy F(int i, xy \*cells, U \*u)

{

xy f;

f.X = 0.0f;

f.Y = 0.0f;

int cellX = cells[i].X;

int cellY = cells[i].Y;

for(int j = 0; j<N; j++)

{

if((i != j) && (u[j].active))

{

if(((cells[j].X == cellX+1) && (cells[j].Y == cellY)) || ((cells[j].X == cellX-1) && (cells[j].Y == cellY)) || ((cells[j].X == cellX) && (cells[j].Y == cellY+1)) || ((cells[j].X == cellX) && (cells[j].Y == cellY-1)) || ((cells[j].X == cellX) && (cells[j].Y == cellY)))

{

float rr = r(u[i].X, u[i].Y, u[j].X, u[j].Y);

float a = ((A1/pow(rr, n1+1)) - (A2/pow(rr, n2+1)));

f.X += a \* (u[j].X - u[i].X);

f.Y += a \* (u[j].Y - u[i].Y);

}

}

}

f.X \*= (1/m);

f.Y \*= (1/m);

return f;

}

\_\_global\_\_ U uhalf[N];

//First stage

\_\_global\_\_ void calcStageOne(xy \*cells, U \*u)

{

int i = threadIdx.x;

if(u[i].active)

{

xy f = F(i, cells, u);

uhalf[i].Vx = u[i].Vx + 0.5f \* f.X;

uhalf[i].X = u[i].X + 0.5f \* u[i].Vx;

uhalf[i].Vy = u[i].Vy + 0.5f \* f.Y;

uhalf[i].Y = u[i].Y + 0.5f \* u[i].Vy;

}

}

//Second stage

\_\_global\_\_ void calcStageTwo(xy \*cells, U \*u, U \*unew)

{

int i = threadIdx.x;

if(u[i].active)

{

xy f = F(i, cells, uhalf);

unew[i].Vx = u[i].Vx + Tau \* f.X;

unew[i].X = u[i].X + Tau \* uhalf[i].Vx;

unew[i].Vy = u[i].Vy + Tau \* f.Y;

unew[i].Y = u[i].Y + Tau \* uhalf[i].Vy;

//Wall collision

if(unew[i].X > WallX)

{

unew[i].Vx = -unew[i].Vx;

unew[i].X = WallX;

}

//Out of zone

if((unew[i].X < ZoneX) || (unew[i].X > ZoneWIDTH) || (unew[i].Y < ZoneY) || (unew[i].Y > ZoneHEIGHT))

{

unew[i].active = false;

}

}

}

int main()

{

srand(time(0));

U u[N], unew[N];

U \*u\_dev = NULL, \*unew\_dev = NULL;

int sizeU = N \* sizeof(U);

cudaMalloc((void\*)&u\_dev, sizeU);

cudaMalloc((void\*)&unew\_dev, sizeU);

int sizeX = ceil((ZoneWIDTH-ZoneX)/R);

int sizeY = ceil((ZoneHEIGHT-ZoneY)/R);

xy cells[N];

xy \*cells\_dev = NULL;

int sizeCells = N \* sizeof(xy);

cudaMalloc((void\*)&cells\_dev, sizeCells);

FILE \*f;

f = fopen("OUT.xml", "w");

fprintf(f, "<out>\n");

fprintf(f, "<settings tcount=\"%d\" tau=\"%f\" zoneX=\"%f\" zoneY=\"%f\" zoneWIDTH=\"%f\" zoneHEIGHT=\"%f\" wallX=\"%f\"\/>\n", (int)(TMAX/Tau), Tau, ZoneX, ZoneY, ZoneWIDTH, ZoneHEIGHT, WallX);

fprintf(f, "<time tid=\"%d\">\n", 0);

for(int i=0; i<N; i++)

{

u[i].active = true;

u[i].Vx = 1.0f;

u[i].Vy = 0.0f;

u[i].X = randFloat(RectX, RectWIDTH);

u[i].Y = randFloat(RectY, RectHEIGHT);

unew[i] = u[i];

fprintf(f, "<point id=\"%d\" x=\"%f\" y=\"%f\"\/>\n", i, u[i].X, u[i].Y);

}

fprintf(f, "<\/time>\n");

cudaMemcpy(u\_dev, u, sizeU, cudaMemcpyHostToDevice);

cudaMemcpy(unew\_dev, unew, sizeU, cudaMemcpyHostToDevice);

int tid = 1;

for(float t=Tau; t<TMAX+Tau; t+=Tau, tid++)

{

fprintf(f, "<time tid=\"%d\">\n", tid);

getCells(cells, sizeX, sizeY, u);

cudaMemcpy(cells\_dev, cells, sizeCells, cudaMemcpyHostToDevice);

//Stage one

calcStageOne <<<1,N >>> (cells\_dev, u\_dev);

cudaThreadSynchronize();

//Stage two

calcStageTwo <<<1,N >>> (cells\_dev, u\_dev, unew\_dev);

cudaThreadSynchronize();

cudaMemcpy(u\_dev, unew\_dev, sizeU, cudaMemcpyDeviceToDevice);

cudaMemcpy(u, u\_dev, sizeU, cudaMemcpyDeviceToHost);

for(int i=0; i<N; i++)

{

if(u[i].active) fprintf(f, "<point id=\"%d\" x=\"%f\" y=\"%f\"\/>\n", i, u[i].X, u[i].Y);

}

fprintf(f, "<\/time>\n");

}

fprintf(f, "<\/out>\n");

return 0;

}

**Задание №2 – Технология PVM**

**Описание:**

Parallel Virtual Machine (PVM) — общедоступный программный пакет, позволяющий объединять разнородный набор компьютеров в общий вычислительный ресурс («виртуальную параллельную машину») и предоставляющий возможности управления процессами с помощью механизма передачи сообщений. Имеет более расширенные возможности, чем её популярный аналог MPI, в плане контроля вычислений: присутствует специализированная консоль управления параллельной системой и её графический эквивалент XPVM, позволяющий наглядно продемонстрировать работу всей системы. Поддерживает программирование на языках Fortran, C и C++ путём предоставления специальных библиотек.

**Понятие «задачи» (task):**

Программный интерфейс преднамеренно сделан «целевым», что позволяет доступ к простым программным структурам осуществлять интуитивным способом. Пользователь пишет свою программу в виде группы взаимосвязанных «задач». Задачи получают доступ к ресурсам PVM посредством библиотеки подпрограмм со стандартизированным интерфейсом. Эти подпрограммы позволяют инициировать и завершить задачу в сети, а также обеспечить связь между задачами и их синхронизацию.

Все задачи PVM идентифицируются посредством целочисленного «идентификатора задачи» (task identifier - TID). Сообщения передаются и принимаются с помощью идентификаторов задач, поэтому эти идентификаторы должны быть уникальными в пределах внутренней виртуальной машины. PVM содержит несколько подпрограмм, которые возвращают значения в TID, тем самым давая возможность пользовательскому приложению идентифицировать другие задачи в системе.

**Создание дочернего процесса:**

int pvm\_spawn (char \*task, char \*\*argv, int flag, char \*where, int ntask, int \*tids)

Подпрограмма pwm\_spawn запускает до ntask копий исполняемого файла task на виртуальной машине; argv - это указатель на массив аргументов для task, причем конец массива указывается с помощью NULL. Если задача не получает аргументов, то argv равняется NULL. Аргумент flag используется для указания опций, которые обобщены в табл. 1. Аргумент where указывает место запуска, ntask - число запускаемых копий программы, tids - массив значений tid для запущенных задач.

Таблица 1. Опции порождения потомков в PVM

|  |  |  |
| --- | --- | --- |
| **Значение** | **Опция** | **Cмысл** |
| 0 | PvmTaskDefault | Место порождения процессов выбирает сама PVM |
| 1 | PvmTaskHost | аргумент where - определенный хост для порождения |
| 2 | PvmTaskArch | аргумент where - PVM\_ARCH для порождения |
| 4 | PvmTaskDebug | запускает задачи под отладчиком |
| 8 | PvmTaskTrace | генерируются трассировочные данные |
| 16 | PvmMppFront | задачи запускаются в среде MPP |
| 32 | PvmHostCompl | where дополняет набор хостов |

**Обмен сообщениями:**

При посылке сообщения от одной задачи к другой внутри PVM нужно выполнить как минимум 3 действия: проинициализировать буфер для передаваемого сообщения, упаковать передаваемые данные в соответствии с форматами их представления и переслать сообщение в буфер получателя.

При приеме сообщения, во-первых, нужно определить алгоритм приема (блокированный или неблокированный), затем, приняв сообщение, его нужно распаковать, т.е. восстановить переданные фрагменты данных в соответствии с их типами.

При посылке сообщения ему присваивается идентификатор, принимающий целые неотрицательные значения и играющий для сообщений ту же роль, что и величина tid для задач в PVM. В то же время, в отличие от задач разные сообщения могут иметь одинаковые идентификаторы, поскольку есть гарантированная возможность различать их по другим признакам (tid отправителя или получателя, время прихода и т.д.).

Функции для работы с буферами:

int bufid = pvm\_initsend ( int encoding );

- возвращает значение, равное либо идентификатору буфера (bufid > 0), либо коду ошибки (bufid < 0). Параметр encoding принимает следующие значения:

PvmDataDefault - при кодировании данных в буфере автоматически учитываются различия в представлении данных на передающей и на приемной стороне;

PvmDataRaw - данные записываются в буфер без кодирования, что удобно, если их представления на передающей и на приемной стороне одинаковы;

PvmDataInPlace - данные будут упаковываться и передаваться в приемный буфер, минуя буфер посылающей задачи, в котором только хранится информация о формате передаваемых сообщений. Это бывает целесообразно, когда требуется передавать множество сообщений с одним и тем же сложным форматом.

Функцию pvm\_initsend() следует вызывать каждый раз перед упаковкой и передачей нового сообщения.

Следующие функции, касающиеся буферов для посылки и приема сообщений, позволяют одной задаче иметь несколько разных буферов как для передачи, так и для приема сообщений от других задач. При этом в каждый момент времени задаче доступен только один передающий и один приемный буферы, которые называются активными. Для работы с другим буфером задача должна сначала активизировать его, что автоматически делает неактивными остальные буферы того же назначения (передающие или приемные).

int bufid = pvm\_mkbuf ( int encoding );

- создает новый пустой передающий буфер с идентификатором bufid и со способом кодирования-упаковки данных, определяемым значением параметра encoding. При ошибке bufid < 0 - код ошибки.

int info = pvm\_freebuf ( int bufid );

- уничтожает буфер с идентификатором bufid. При ошибке info < 0.

int bufid = getsbuf ( void );

- возвращает идентификатор активного передающего буфера или код ошибки при bufid < 0.

int bufid = getrbuf ( void );

- возвращает идентификатор активного приемного буфера или код ошибки при bufid < 0.

int oldbuf = pvm\_setsbuf ( int bufid );

- переключает активность с передающего буфера oldbuf на передающий буфер bufid, сохраняя состояние буфера oldbuf.

int oldbuf = pvm\_setrbuf ( int bufid );

- переключает активность с приемного буфера oldbuf на приемный буфер bufid, сохраняя состояние буфера oldbuf.

Если в функциях pvm\_setsbuf(...) и pvm\_setrbuf(...) параметр bufid = 0, то состояние активного буфера сохраняется, а активность с него снимается, но не передается другому буферу. Это позволяет защитить временно неиспользуемый буфер от сообщений, которые могут порождаться некоторыми системными и библиотечными модулями, использующими тот же механизм обмена сообщениями.

В большинстве случаев достаточно для организации параллельных вычислений иметь в каждой выполняемой задаче один передающий и один приемный буфер, что позволяет ограничиться использованием только функции initsend(...).

Функции для упаковки сообщений:

Для упаковки в передающий буфер данных различного типа служит набор функций с однотипным интерфейсом. Первый параметр функции является указателем соответствующего типа на первый элемент массива, подлежащего упаковке, второй параметр nitem равен длине этого массива, а третий параметр stride указывает, сколько элементов указанного типа составляют один элемент массива (всего упаковывается (nitem \* stride) элементов указанного типа).

int info = pvm\_pkbyte ( char \*xp, int nitem, int stride );

int info = pvm\_pkcplx ( float \*cp, int nitem, int stride );

int info = pvm\_pkdcplx ( double \*zp, int nitem, int stride );

int info = pvm\_pkdouble ( double \*dp, int nitem, int stride );

int info = pvm\_pkfloat ( float \*fp, int nitem, int stride );

int info = pvm\_pkint ( int \*ip, int nitem, int stride );

int info = pvm\_pkuint ( unsigned int \*ip, int nitem, int stride );

int info = pvm\_pkshort ( short \*ip, int nitem, int stride );

int info = pvm\_pkushort ( unsigned short \*ip, int nitem, int stride );

int info = pvm\_pklong ( long \*ip, int nitem, int stride );

int info = pvm\_pkulong ( unsigned long \*ip, int nitem, int stride );

int info = pvm\_pkstr ( char \*cp, int nitem, int stride );

Для перечисленных функций возвращаемое значение info либо равно 0, либо ( при info < 0 ) является кодом ошибки. Функция pvmpkstr(...) упаковывает строку символов, начиная с адреса cp и кончая первым встреченным символом NULL ('\0').

Последовательное применение указанных функций позволяет упаковывать в передающем буфере сообщение, содержащее несколько массивов разных типов. Для упаковки данных сложного типа (например, типа struct) нужно последовательно упаковать все фрагменты, принадлежащие к указанным типам. К сожалению, функции типа sprintf(), которая принимала бы строковый параметр с описанием упаковываемых типов, и позволяла бы за один прием упаковывать структуры и юнионы, в 3 редакции PVM нет.

Функции для передачи и приема сообщений:

При посылке сообщения ему обязательно присваивается идентификатор ( обозначаемый здесь msgtag ), который облегчает задачу приема сообщения и поиска его в PVM. Кроме того, во всех функциях посылки сообщений присутствует обязательный параметр tid - идентификатор принимающей задачи, и этот же параметр tid в функциях приема сообщений является идентификатором посылающей задачи.

int info = pvm\_send ( int tid, int msgtag );

- присваивает сообщению идентификатор msgtag и посылает сообщение задаче с идентификатором tid.

int info = pvm\_mcast ( int \*tids, int ntask, int msgtag );

- присваивает сообщению идентификатор msgtag и посылает его задачам, идентификаторы которых перечислены в ntask элементах массива tids.

int info = pvm\_psend ( int tid, int msgtag, void \*vp, int cnt, int type )

- упаковывает cnt элементов массива vd типа type в сообщение, присваивает ему идентификатор msgtag и посылает его задаче с идентификатором tid. Эта функция позволяет обходиться без функций упаковки, если сообщение содержит лишь один массив простого типа. Для значений параметра type используются следующие символические имена: PVM\_STR, PVM\_BYTE, PVM\_SHORT, PVM\_INT, PVM\_LONG, PVM\_USHORT, PVM\_UINT, PVM\_ULONG, PVM\_FLOAT, PVM\_DOUBLE, PVM\_CPLX, PVM\_DCPLX.

Перечисленные функции возвращают либо значение info = 0, либо код ошибки info < 0.

Для функций приема сообщений возможны следующие вариации значений параметров. Если tid = -1, то принимается любое сообщение с идентификатором msgtag, направленное к данной задаче. Если msgtag = -1, то принимается любое сообщение, направленное к данной задаче задачей с идентификатором tid. Если tid = msgtag = -1, то принимается любое сообщение от любой задачи, направленное к данной задаче.

int bufid = pvm\_recv ( int tid, int msgtag );

- осуществляет блокированный прием, т.е. ожидает поступления сообщения с идентификатором msgtag от задачи с идентификатором tid, после чего создает новый приемный буфер с идентификатором bufid, помещает в него принятое сообщение и завершает работу.

int bufid = pvm\_nrecv ( int tid, int msgtag );

- осуществляет неблокированный прием, т.е. при отсутствии сообщения завершает работу, возвращая значение bufid = 0. Если сообщение с заданными значениями tid и msgtag поступило, то оно помещается в созданный для него приемный буфер, идентификатор которого является возвращаемым значением. В процессе ожидания сообщения эту функцию можно вызывать несколько раз, заполняя промежутки времени между вызовами какой-либо полезной работой.

int bufid = pvm\_trecv ( int tid, int msgtag, struct timeval \*tmout );

- осуществляет блокированный прием сообщения, если время его ожидания не больше величины tmout, в противном случае через время tmout функция возвращает значение bufid = 0. В структуре типа timeval имеется два целочисленных поля, в которые записываются число секунд и миллисекунд.

Когда заданное время ожидания tmout = 0, функция pvm\_trecv(...) осуществляет неблокированный прием, а когда указатель на tmout равен NULL (соответствует бесконечному времени ожидания), осуществляется блокированный прием, в остальных же случаях реализуется промежуточный алгоритм приема сообщений.

int bufid = pvm\_probe ( int tid, int msgtag );

- проверяет, поступило ли ожидаемое сообщение. Если оно поступило, то возвращается идентификатор приемного буфера bufid > 0, если сообщение еще не поступило, то возвращается значение bufid = 0.

int info = pvm\_bufinfo ( int bufid, int nbytes, int msgtag, int tid );

- возвращает значения tid, msgtag и длину в байтах nbytes для сообщения в буфере с идентификатором bufid. Эту функцию целесообразно использовать в сочетании с приемом сообщений при неопределенных значениях tid = -1 и/или msgtag = -1.

Распаковка данных из принятого сообщения:

Функции распаковки данных, записанных в приемном буфере, применяются в той же последовательности, в какой применялись функции упаковки данных в посылаемое сообщение. Пусть, например, нужно передать в сообщении следующие данные

char cp[5]; int np[3]; double dp[7]; string s[];

Тогда сообщение упаковывается следующим образом:

info = pvm\_pkbyte ( cp, 5, 1 );

info = pvm\_pkint ( np, 3, 1 );

info = pvm\_pkdouble ( dp, 7, 1 );

info = pvm\_pkstr ( s );

и после приема в другой задаче делается следующая распаковка

char cp[]; int np[]; double dp[]; string s[];

info = pvm\_upkbyte ( cp, 5, 1 );

info = pvm\_upkint ( np, 3, 1 );

info = pvm\_upkdouble ( dp, 7, 1 );

info = pvm\_upkstr ( s );

при условии, что для массивов cp, np, dp, s выделена память.

Перечень функций для распаковки данных в принятом сообщении:

int info = pvm\_upkbyte ( char \*xp, int nitem, int stride );

int info = pvm\_upkcplx ( float \*cp, int nitem, int stride );

int info = pvm\_upkdcplx ( double \*zp, int nitem, int stride );

int info = pvm\_upkdouble ( double \*dp, int nitem, int stride );

int info = pvm\_upkfloat ( float \*fp, int nitem, int stride );

int info = pvm\_upkint ( int \*ip, int nitem, int stride );

int info = pvm\_upkshort ( short \*ip, int nitem, int stride );

int info = pvm\_upklong ( long \*ip, int nitem, int stride );

int info = pvm\_upkstr ( char \*cp, int nitem, int stride );

Перечисленные функции возвращают значение либо info = 0, либо код ошибки info < 0.

**Задание №3 – Технология Linda**

**Описание:**

В Linda параллельная программа есть множество параллельных процессов, и каждый процесс работает согласно обычной последовательной программе. Все процессы имеют доступ к общей памяти, единицей хранения в которой является кортеж. Отсюда происходит и специальное название для общей памяти - пространство кортежей. Каждый кортеж — это упорядоченная последовательность значений.

Примеры кортежей:

(“master”, 1, 2.3), (13, 284.56)

Все процессы работают с пространством кортежей по принципу: поместить кортеж, забрать, скопировать. В отличие от традиционной памяти, процесс может забрать кортеж из пространства кортежей, после чего данный кортеж станет недоступным остальным процессам. Если в пространство кортежей положить два кортежа с одним и тем же именем, то не произойдет привычного для нас "обновления" значения переменной - в пространстве кортежей окажется два кортежа с одним и тем же именем. Изменить кортеж непосредственно в пространстве нельзя. Для изменения значений элементов кортежа, его нужно сначала оттуда изъять, затем процесс, изъявший кортеж, может изменить значения его элементов и вновь добавить измененный кортеж в память. В отличие от других систем программирования, процессы в системе Linda никогда не взаимодействуют друг с другом явно, и все общение всегда идет через пространство кортежей.

**Функции:**

1. **OUT** - помещает кортеж в пространство кортежей.

Например, out ("GoProcess", 5) - помещает в пространство кортежей кортеж ("GoProcess", 5). Если такой кортеж уже есть в пространстве кортежей, то появится второй, что, в принципе, позволяет иметь сколь угодно много экземпляров одинаковых кортежей. По этой же причине с помощью функции out нельзя изменить кортеж, уже находящийся в пространстве. Для этого кортеж должен быть сначала оттуда изъят, затем изменен и после этого помещен назад. Функция out никогда не блокирует выполнивший ее процесс.

1. **IN** - ищет подходящий кортеж в пространстве кортежей, присваивает значения его элементов элементам своего параметра-кортежа и удаляет найденный кортеж из пространства кортежей.

Например, in ("P", int i, FALSE) - этой функции соответствует любой кортеж, который состоит из трех элементов: значением первого элемента является "P", второй элемент может быть любым целым числом, а третий должен иметь значение FALSE. Подходящими кортежами могут быть ("P", 5, FALSE) или ("P", 135, FALSE) и т.п., но не ("P", 7.2, FALSE) или ("Proc", 5, FALSE). Если параметру функции in соответствуют несколько кортежей, то случайным образом выбирается один из них. После нахождения кортеж удаляется из пространства кортежей, а неопределенным формальным элементам параметра-кортежа, содержащимся в вызове данной функции, присваиваются соответствующие значения.

1. **READ** - отличается от функции in лишь тем, что выбранный кортеж не удаляется из пространства кортежей. Все остальное точно так же, как и у функции in. Этой функцией удобно пользоваться в том случае, когда значения переменных менять не нужно, но к ним необходим параллельный доступ из нескольких процессов.
2. **EVAL** - похожа на функцию out. Разница заключается лишь в том, что дополнительным элементом кортежа у eval является функция пользователя. Для вычисления значения этой функции система Linda порождает параллельный процесс, на основе работы которого она формирует кортеж и помещает его в пространство кортежей.

Например, eval ("hello", funct (z), TRUE, 3.1415) - при обработке данного вызова система создаст новый процесс для вычисления функции funct (z). Когда процесс закончится и будет получено значение w = funct (z), в пространство кортежей будет добавлен кортеж ("hello", w, TRUE, 3.1415). Функция, вызвавшая eval, не ожидает завершения порожденного параллельного процесса и продолжает свою работу дальше. Следует отметить и то, что пользователь не может явно управлять размещением порожденных параллельных процессов на доступных ему процессорных устройствах - это Linda делает самостоятельно.

Параллельная программа в системе Linda считается завершенной, если все порожденные процессы завершились или все они заблокированы функциями in и read.

**Синхронизация:**

Не имея в системе Linda никаких явных средств для синхронизации процессов, совсем не сложно их смоделировать самому. Предположим, что в некоторой точке нужно выполнить барьерную синхронизацию N процессов. Какой-то один процесс, например, стартовый, заранее помещает в пространство кортеж ("ForBarrier", N). Подходя к точке синхронизации, каждый процесс выполняет следующий фрагмент, который и будет выполнять функции барьера:

in ( "ForBarrier", formal Bar);

Bar = Bar - 1;

if( Bar != 0 ) {

out( "ForBarrier", Bar);

read( "Barrier" );

} else

out( "Barrier" );

Если кортеж с именем "ForBarrier" есть в пространстве, то процесс его изымает, в противном случае блокируется до его появления. Анализируя второй элемент данного кортежа, процесс выполняет одно из двух действий. Если есть процессы, которые еще не дошли до данной точки, то он возвращает кортеж в пространство с уменьшенным на единицу вторым элементом и встает на ожидание кортежа "Barrier". В противном случае он сам помещает кортеж "Barrier" в пространство, который для всех является сигналом к продолжению работы.

**Задание №4 – Технология PARIX**

**Описание:**

PARIX (Parallel extensions to UNIX) является надстройкой над ОС AIX. При обращении к ОС PARIX, последняя захватывает ресурсы всех процессоров в указанном разделе и запускает на них вашу прикладную задачу.

PARIX - это однозадачная система: если один пользователь запустил PARIX-приложение на разделе, то никакой другой пользователь уже не сможет запустить своего PARIX-приложения на данном разделе, покуда первое приложение не закончит свою работу и не освободит раздел.

С одной стороны, такой подход несомненно ограничивает пользователей, заставляя их строго координировать свои расчетные задачи. С другой же стороны, такая схема обеспечивает наиболее благоприятный режим для запущенного приложения.

В PARIX принята следующая модель представления многопроцессорной системы: все процессоры “заполняют” ячейки в некоем трехмерном “ящике”. Каждому процессору приписаны уникальные координаты (тройка целых положительных чисел). Так что с точки зрения пользователя процессор кроме своего порядкового номера имеет “координаты”.

В трехмерном ящике с узлами не может быть дыр: если существует процессор с координатами (0,0,0) и существует (0,0,2), значит обязан существовать и (0,0,1). Кстати, узел (0,0,0) существует всегда (если в системе ненулевое количество процессоров).

Конечная трехмерная решетка, включающая в себя все процессоры данной машины, называется основным разделом. Обычно основной раздел делится на несколько более мелких (и, возможно, перекрывающихся) разделов. Самым мелким, понятно, является раздел, состоящий из одного узла. Важнейшей характеристикой раздела является его размер, т.е. размер трехмерного ящика с узлами, входящими в данный раздел.

**Топологии процессоров:**

В PARIX имеются синхронные и асинхронные операции обмена сообщениями, работающие с понятием виртуальной топологии - топологии сети, образуемой виртуальными каналами. PARIX предлагает богатый набор типов топологий - конвейер (Pipe), кольцо (Ring), двух/трех (2D/3D) мерные решетки (Grid) и торы (Torus), гиперкуб (HCube), клика (Clique), дерево (Tree и RTree). Более того, пользователь может создавать свои собственные топологии сетей, образованных виртуальными каналами.

Для каждого стандартного типа XXX топологии имеются функции, обеспечивающие создание, удаление коммуникационной сети с соответствующей топологией XXX.

int MakeXXX (int RequestId,

dims . . .,

int xmin, int xmax,

int ymin, int ymax,

int zmin, int zmax),

int FreeXXX (int topId),

XXXData\_t \* GetXXX\_Data(int topId)

MakeXXX создает коммуникационную сеть с соответствующей топологией XXX размерностью dim, которая отображается на подмножество процессоров, описываемое параметрами xmin, xmax, ymin, ymax, zmin. Только процессоры, имеющие в разделе координаты (x, y, z) такие, что xmin <= x <= xmax, ymin <= y <= ymax и zmin <= z <= zmax будут использоваться в этой сети. Сети с топологией XXX (а именно ее каналам) необходимо сопоставить уникальный идентификатор канала, передаваемый через параметр RequestId и используемый ОС для того, чтобы различать сообщения, передаваемые по каналам разных сетей. Функция возвращает идентификатор сети (топологии). Каждый узел сети имеет каналы, связывающие его с соседними узлами. Количество каналов зависит от вида топологии сети. Для указания каналов используются целые значения от 0 до <число соседних узлов>-1, называемые логическими каналами. Например, все узлы сети с топологией кольцо имеют два канала для двух направлений - вперед, назад, которым сопоставлены логические каналы 1 и 0 соответственно. В сети с топологией клика каждый узел имеет каналы, связывающие его со всеми остальными узлами сети, - логический канал i представляет виртуальный канал, ведущий из узла с номером n к узлу с номером, получающимся из x инвертированием i-го бита (x ^ (1<<i)).

Функция FreeXXX удаляет сеть, имеющую указанный идентификатор. Функция GetXXX\_Data возвращает указатель на дескриптор сети с указанным идентификатором. Дескриптор имеет следующий вид:

struct XXXData\_t {

char type; /\* содержит имя топологии вида: XXX\_TYPE (например. RING\_TYPE) \*/

int status; /\* указывает статус процессора, характеризующий его отношение к сети, его положение в сети. Например, для конвейера это значения PIPE\_HEAD, PIPE\_IN и PIPE\_TAIL. Если процессор не входит в соответствующую сеть, то он имеет статус XXX\_NONE (например. PIPE\_NONE). \*/

xxx\_t id; /\* идентифицирует положение процессора в топологии. Например, для конвейера это значение в диапазоне 0 .. <длина\_конвейера>-1, причем процессор со статусом PIPE\_HEAD имеет id =0 , а процессор со статусом PIPE\_TAIL - id = <длина\_конвейера>-1 \*/

xxx\_t dims; /\* хранит размерности топологии. Например, для 2D решетки это ее ширина и высота \*/

};

Функции

int Send (int TopId, int LogLinkId, void \*Data, int Size)

int Recv (int TopId, int LogLinkId, void \*Data, int Size)

реализуют синхронный обмен сообщениями в сети с топологией, имеющей идентификатор TopId. Параметр LogLinkId указывает, какой логический канал топологии следует использовать для приема/передачи сообщения.

Используя функции

int NewTop (int nLinks),

int AddTop (int TopId, LinkCB\_t \*Link),

пользователь может создавать свои собственные топологии сетей виртуальных каналов. Функция NewTop, с вызова которой начинается процедура формирования топологии, создает в памяти вызвавшего ее процессора некоторую внутреннюю таблицу под предполагаемое количество (nLinks) каналов, и возвращает идентификатор топологии. Идентификатор с этого момента представляет вновь образованную топологию. Внутренняя таблица будет хранить указатели на дескрипторы виртуальных каналов топологии.

Функция AddTop добавляет новый виртуальный канал Link к топологии TopId и возвращает идентификатор представляющего его логического канала (LogLinkId). С этого момента логический канал LinkId топологии TopId можно использовать для обмена сообщениями. Функция

LinkCB\_t \*GetLinkCB (int TopId, int LogLinkId, int \*Error)

позволяет получить указатель на виртуальный канал логического канала LinkId топологии TopId. Удалить топологию, е§ каналы можно с помощью функции

int FreeTop (int TopId),

которая удаляет внутренние структуры топологии, неявно освобождает каждый канал сети и внутренние данные, используемые при асинхронных коммуникациях.

Каждая топология имеет свои ей присущие свойства такие, как размерности dim\_x, dim\_y, dim\_z трехмерных реш§тки и тора, глубина дерева и число потомком у его вершин. Функция

int AddTop\_Data (int TopId, void \*Data)

позволяет связать с топологией е§ атрибуты, хранящиеся в дескрипторе Data. Затем их можно будет извлечь с помощью функции

void \*GetTopData (int TopId, int \*Error)

Следующий набор функций, реализует асинхронные обмены сообщениями по каналам виртуальных топологий.

int AInit (int TopId, int Threads, int Size)

int ASend (int TopId, int LogLinkId, byte \*Data, int Size, int \*Result)

int ARecv (int TopId, int LogLinkId, byte \*Data, int Size, int \*Result)

int ASync (int TopId, int LogLinkId)

int AExit (int TopId)

**Каналы связей:**

В PARIX введено понятие виртуального канала. Процессор может иметь необходимое число виртуальных каналов, которые могут связывать его с произвольными процессорами, расположенными в произвольном месте сети. PARIX сам осуществляет рассылку между транспьютерами сообщений, передаваемых по виртуальным каналам. Можно создать виртуальные каналы между транспьютером и внешним компьютером.

Чтобы создать виртуальный канал, два процесса, которые хотят установить связь с помощью этого канала, должны вызвать функцию:

LinkCB\_t \*ConnectLink(int Processor, int RequestId, int \*Error).

Каждый процесс, используя параметр Processor, должен указать номер процессора, с процессом которого он хочет установить связь. Каналу необходимо сопоставить уникальный идентификатор канала, передаваемый через параметр RequestId. Это значение используется ОС для того, чтобы различать сообщения, передаваемые по разным каналам. Функция возвращает указатель на дескриптор созданного канала.

Функция

int BreakLink (LinkCB\_t \*Link)

удаляет виртуальный канал. Каждый из процессов партнеров должен удалить свою часть канала.

Пересылка сообщений по раннее созданному каналу осуществляется с помощью синхронных операций:

int SendLink (LinkCB\_t \*Link, void \*Buf, int Size),

int RecvLink (LinkCB\_t \*Link, void \*Buf, int Size)

**Обмен данными:**

Осуществляется с помощью каналов, виртуальных топологий или используя синхронные операции обмена:

int SendNode (int Processor, int RequestId, byte \*Buffer, int Size) и

int RecvNode (int Processor, int RequestId, byte \*Buffer, int Size),

позволяют обмениваться сообщениями между процессами без создания виртуальных каналов. Для каждой операции каждый процесс-партнер, используя параметр Processor, должен указать номер процессора, процессу которого он хочет передать сообщение или принять сообщение от него. Операции обмена необходимо сопоставить уникальный идентификатор запроса, передаваемый через параметр RequestId. Значение -1, используемое в качестве Processor или RequestId, позволяет RecvNode получать сообщения от произвольного процессора или с произвольным идентификатором запроса.

**Задание №5 – задание на MPI Fortran**

**Текст задания:**

Написать программу на Fortran, которая решает систему дифференциальных уравнений методом Рунге-Кутты 4:

y1’ = y2, y10 = 2.0;

y2’ = (1 – y12) y2 – y1, y20 = 0.0;

**Текст программы:**

program RK4

include 'mpif.h'

integer ierror, msgtag, rank, rank, size

integer status (MPI\_STATUS\_SIZE)

parameter (m = 4)

parameter (n = 2)

real y(n), yy(n), time, tmax, tau, r(m), ff, tn, tk, tmp

msgtag = 12

y(1) = 2.0

y(2) = 0.0

yy(1) = 0.0

yy(2) = 0.0

time = 0.0

tmax = 0.1

tau = 0.01

ff = 0.0

call MPI\_Init(ierror)

call MPI\_Comm\_size(MPI\_COMM\_WORLD, size, ierror)

call MPI\_Comm\_rank(MPI\_COMM\_WORLD, rank, ierror)

if(rank.EQ.0) then

open(1, file='RK4.res', status = 'new')

write(1, \*) 'time = ', time

do i=1, n

write(1, \*) ' y[', i, '] = ', y(i)

end do

close(1)

end if

tn = MPI\_Wtime()

do

if(rank.EQ.0) then

do i=1, m

do j=1, n

call MPI\_Recv(tmp, 1, MPI\_REAL, MPI\_ANY\_SOURCE, msgtag, MPI\_COMM\_WORLD, status, ierror)

yy(status(MPI\_SOURCE)) = tmp

end do

do j=1, size-1

call MPI\_Send(yy, n, MPI\_REAL, j, msgtag, MPI\_COMM\_WORLD, ierror)

end do

end do

else

r(1) = tau \* func(y(1), y(2), time, rank)

ff = y(rank) + 0.5 \* r(1)

call MPI\_Send(ff, 1, MPI\_REAL, 0, msgtag, MPI\_COMM\_WORLD, ierror)

call MPI\_Recv(yy, n, MPI\_REAL, 0, msgtag, MPI\_COMM\_WORLD, status, ierror)

r(2) = tau \* func(yy(1), yy(2), time + 0.5 \* tau, rank)

ff = y(rank) + 0.5 \* r(2)

call MPI\_Send(ff, 1, MPI\_REAL, 0, msgtag, MPI\_COMM\_WORLD, ierror)

call MPI\_Recv(yy, n, MPI\_REAL, 0, msgtag, MPI\_COMM\_WORLD, status, ierror)

r(3) = tau \* func(yy(1), yy(2), time + 0.5 \* tau, rank)

ff = y(rank) + r(3)

call MPI\_Send(ff, 1, MPI\_REAL, 0, msgtag, MPI\_COMM\_WORLD, ierror)

call MPI\_Recv(yy, n, MPI\_REAL, 0, msgtag, MPI\_COMM\_WORLD, status, ierror)

r(4) = tau \* func(yy(1), yy(2), time + tau, rank)

y(rank) = y(rank) + (r(1) + 2.0 \* r(2) + 2.0 \* r(3) + r(4)) / 6.0;

call MPI\_Send(y[rank], 1, MPI\_REAL, 0, msgtag, MPI\_COMM\_WORLD, ierror)

call MPI\_Recv(y, n, MPI\_REAL, 0, msgtag, MPI\_COMM\_WORLD, status, ierror)

end if

time = time + tau;

MPI\_Barrier(MPI\_COMM\_WORLD);

if(rank.EQ.0) then

open(1, file='RK4.res', status = 'old')

write(1, \*) 'time = ', time

do i=1, n

write(1, \*) ' y[', i, '] = ', yy(i)

end do

close(1)

end if

if(time.GT.tmax) exit

end do

tk = MPI\_Wtime()

if(rank.EQ.0) then

open(1, file='RK4.res', status = 'old')

write(1, \*) 'time\_end = ', tk-tn

close(1)

end if

call MPI\_Finalize(ierror)

stop

end

function func(real x1, real x2, real t, integer rank)

real func

if(rank.EQ.1) then

func = x2

end if

if(rank.EQ.2) then

func = (1 - x1\*x1)\*x2-x1

end if

end function func