Лабораторная работа № 1

Параллельное вычисление суммы числового ряда

Цель работы

Для многопроцессорных вычислительных систем с распределенной памятью на примере задачи параллельного вычисления суммы числового ряда научиться программно реализовывать простейшие параллельные вычислительные алгоритмы и проводить анализ их эффективности.

Вычисление суммы числового ряда

Требуется вычислить сумму числового ряда

$$S = \sum_{n=1}^{N} a_n = a_1 + a_2 + \ldots + a_N \qquad a_n \in \mathbf{R},\ n \in \mathbf{N}.$$

В распоряжении - МВС с **P** процессорами (ядрами).

$$\underbrace{a_1 + a_2 + \ldots + a_{N_1}}_{S_1} + \underbrace{a_{N_1+1} + \ldots + a_{N_2}}_{S_2} + \ldots + \underbrace{a_{N_{P-1}+1} + \ldots + a_{N_P}}_{S_P}$$

process 0 process 1 process P-1

Использование базовых функций MPI

Создать параллельную версию написанной ранее программы вычисления суммы ряда с использованием базовых функций MPI

```
MPI_Init()
MPI_Comm_size()
MPI_Send()
MPI_Wtime()
MPI_Finalize()
MPI_Comm_rank()
MPI_Recv()
MPI_Barrier()
```

В программе предусмотреть следующее

а) Ввод-вывод данных должен осуществляться только через нулевой процесс, который рассылает остальным процессам число членов ряда, введенное через аргумент командной строки. Перед рассылкой фиксируется время начала выполнения параллельного участка программы.

б) Каждый процесс определяет количество членов, которое он суммирует. Для этого вычисляется

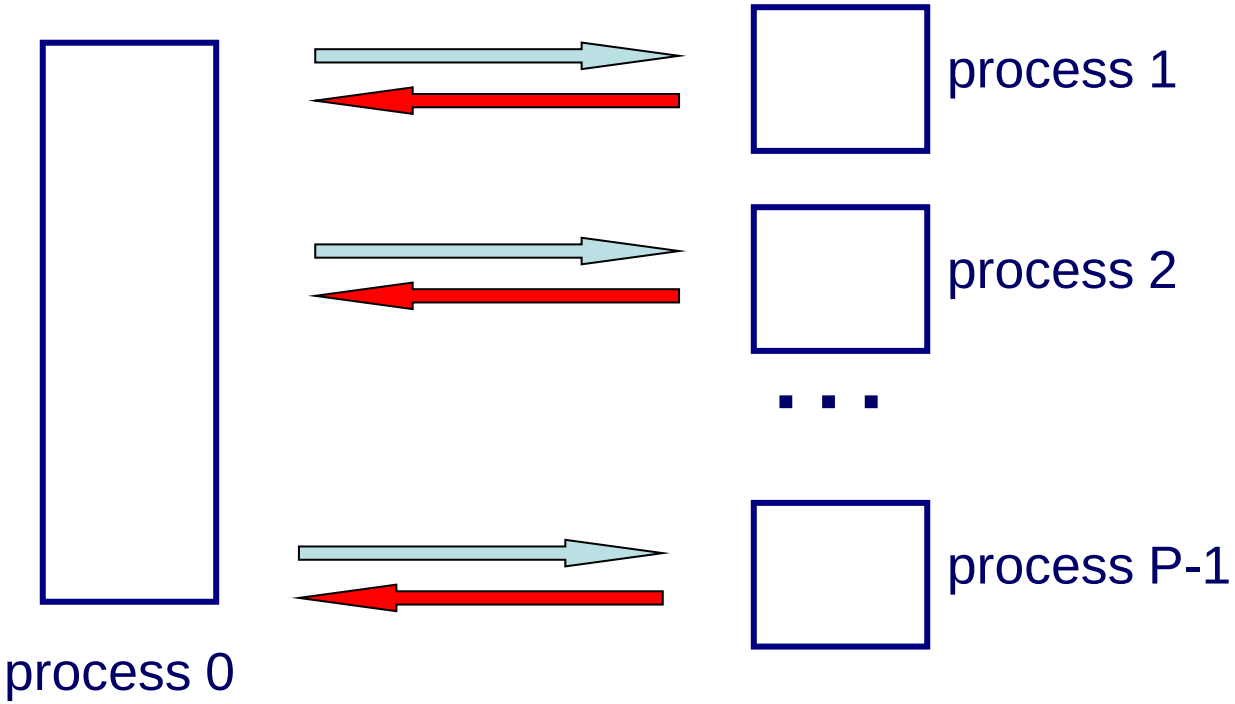
$$h = \left[\frac{N}{p} \right] \qquad m = N \bmod p.$$

в) Если остаток равен нулю, то каждый процесс суммирует одинаковое количество членов ряда. Если количество членов ряда не кратно числу процессов, то остаток от деления равномерно распределяется между всеми процессами.

г) Каждый процесс считает сумму, отведенной ему части ряда, и после окончания расчета пересылает ее нулевому процессу.

д) Нулевой процесс находит искомую сумму, как сумму своей частичной суммы и всех присланных. Фиксируется время окончания параллельного участка программы и результаты выводятся на экран (сумма ряда и затраченное время).

Использование базовых функций MPI



Рассылка N — MPI_Send → MPI_Recv

Прием частичных сумм S_p — MPI_Recv ← MPI_Send

Компиляция и запуск программы

1. Зайти
   Пуск -> Favorites ->
   -> Terminal Programming (Konsole)
2. Появляется консольное окошко, в котором, например, можно вызвать **midnight commander**, набрав **mc**
3. Открываем папку, где лежат написанные программы
4. Набираем в командной строке (для удобства можно нажать Ctrl+o) команду, отвечающую за компиляцию программы

   mpiicc имя_файла.c -lm

   или **mpiicc -o имя имя_файла.c -lm**

   В первом случае создается файл **a.out**, во-втором **имя**
5. Запуск программы

   mpirun -np 4 ./a.out

На кластере

Компиляция осуществляется такими же командами как и ранее. Расчет осуществляется при числе процессов `p = 1, 2, 4, 8, 16, 32, 64, 128`, а число членов ряда подбирается таким образом, чтобы время работы программы при `p = 1` составляло порядка 150 и 300 с. Время работы параллельного участка кода занести в таблицу

p	T_p, с	
	N_1	N_2
1 2 .. 64 128		

Анализ полученных результатов

О п р е д е л е н и е. Отношение времени выполнения параллельной программы на одном процессоре (ядре) T_1^* ко времени выполнения параллельной программы на p процессорах T_p называется *ускорением* при использовании p процессоров:

$$S_p^* = \frac{T_1^*}{T_p}$$

О п р е д е л е н и е. Отношение ускорения S_p^* к количеству процессоров p называется *эффективностью* при использовании p процессоров:

$$E_p^* = \frac{S_p^*}{p}$$

Запуск программы

Вычислить ускорение и эффективность по данным формулам, полученные значения занести в таблицу

p	S_p^*		E_p^*	
	N_1	N_2	N_1	N_2
1 2 .. 64 128				

По данным таблицы построить графики зависимостей ускорения и эффективности от числа процессов при различном числе членов ряда.

Базовые функции MPI

Базовые функции MPI

УГАТУ

Функция инициализации MPI

```
int MPI_Init(int *argc, char **argv[]);
```

Возвращает предопределенные константы

MPI_SUCCESS - возвращается в случае успешного выполнения,

MPI_ERR_ARG - ошибка неправильного задания аргумента,

MPI_ERR_INTERN - внутренняя ошибка (нехватка памяти),

MPI_ERR_UNKNOWN - неизвестная ошибка.

Функция завершения работы с MPI

```
int MPI_Finalize(void);
```

Базовые функции MPI

УГАТУ

Функция определения количества процессов **size** в коммуникационной группе с коммуникатором **comm**

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

Функция, возвращающая номер **rank** вызвавшего ее процесса, входящего в коммуникационную группу с коммуникатором **comm**

```
int MPI_Comm_rank(MPI_comm comm, int* rank);
```

Базовые функции MPI

УГАТУ

Пример программы «Hello, World!»

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

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

{ int MyID, NumProc;

  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &NumProc);

  MPI_Comm_rank(MPI_COMM_WORLD, &MyID);

  fprintf(stdout,"Process %d of %d:\t",MyID, NumProc);

  fprintf(stdout,"Hello, World!\n");

  MPI_Finalize();

  return 0;

 }
```

Пример программы «Hello, World!»

После запуска программы, например, на трех процессах

```
mpirun  -np 3 ./a.out
```

на экране появится следующее сообщение:

```
Process 0 of 3:  Hello, World!
Process 1 of 3:  Hello, World!
Process 2 of 3:  Hello, World!
```

Базовые функции MPI

УГАТУ

Блокирующая функция передачи данных.

```
int MPI_Send(void* sbuf, int count,
             MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm);
```

Входные параметры:

sbuf	– адрес в памяти, начиная с которого размещаются передаваемые данные;
count	– количество передаваемых элементов;
datatype	– тип передаваемых элементов;
dest	– номер процесса-получателя сообщения;
tag	– метка передаваемого сообщения;
comm	– коммуникатор.

Базовые функции MPI

УГАТУ

Блокирующая функция приема данных.

```
int MPI_Recv(void* rbuf, int count,
             MPI_Datatype datatype, int source,
             int tag, MPI_comm comm,
             MPI_Status *status);
```

Входные параметры:

`count`	– количество получаемых элементов;
`datatype`	– тип получаемых элементов;
`source`	– номер процесса-отправителя сообщения;
`tag`	– метка принимаемого сообщения;
`comm`	– коммуникатор.

Выходные параметры:

`rbuf`	– адрес в памяти, начиная с которого размещаются принимаемые данные;
`status`	– структура, содержащая информацию о принятом сообщении.

Базовые функции MPI

Структура **status** имеет три поля.

Status.MPI_SOURCE - номер процесса-отправителя;

Status.MPI_TAG - метка принимаемого сообщения;

Status.MPI_ERROR - код завершения приема сообщения.

Тип данных **datatype** может быть одной из предопределенных констант.

MPI_CHAR	**MPI_UNSIGNED_CHAR**
MPI_SHORT	**MPI_UNSIGNED_SHORT**
MPI_INT	**MPI_UNSIGNED_INT**
MPI_LONG	**MPI_UNSIGNED_LONG**
MPI_FLOAT	
MPI_DOUBLE	**MPI_LONG_DOUBLE**
MPI_BYTE	**MPI_PACKED**

УГАТУ

Пример передачи сообщения

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{ int MyID, NumProc, ierror;
  double a, b;
  MPI_Status status;
  ierror=MPI_Init(&argc, &argv);
   if(ierror!=MPI_SUCCESS)
      printf("MPI initialization error!");
  MPI_Comm_size(MPI_COMM_WORLD, &NumProc);
  MPI_Comm_rank(MPI_COMM_WORLD, &MyID);
  if(MyID==0)
  {   a=NumProc;
     for(i=1;i<=NumProc-1;i++)
         MPI_Send(&a,1,MPI_DOUBLE,i,i+777,MPI_COMM_WORLD);
  }
```

Базовые функции MPI

УГАТУ

```
if(MyID>0)
{   MPI_Recv(&a,1,MPI_DOUBLE,0,MyID+777,
        MPI_COMM_WORLD,&status);
    b=a+MyID;
    fprintf(stdout,"Process %d: b=%lf\n",MyID, b);
}
MPI_Finalize();
return 0;
}
```

Пусть **NumProc**=4, тогда на экране появится

```
Process 1: b=5

Process 2: b=6

Process 3: b=7
```

Функции определения времени

Определение времени выполнения параллельной программы

```
double  MPI_Wtime();
```

Пример

```
double tstart, tfinish;
. . .
MPI_Barrier(MPI_COMM_WORLD);
tstart = MPI_Wtime();
. . . /* Участок кода, время выполнения которого
         измеряется */
MPI_Barrier(MPI_COMM_WORLD);
tfinish = MPI_Wtime()-tstart;
. . .
```