# Паралелно програмиране с MPI – Част I *„Комуникации от точка в точка“*

## ***Цел:***

- Запознаване с примерни програми, използващи паралелно програмиране с програмен модел на системи с разпределена памет, компилиране, настройване и изпълнение на MPI програми.

## ***Очакван резултат:***

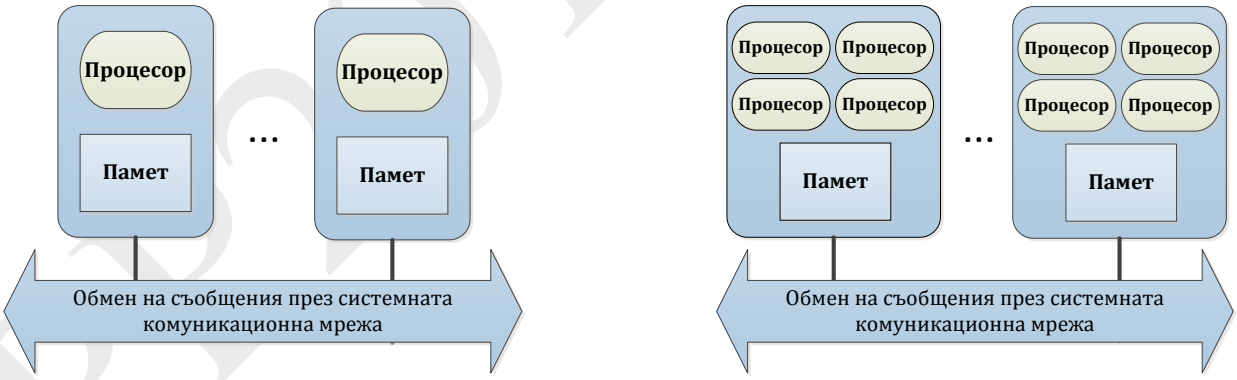
Писмено представяне на експерименталните резултати и анализи във вид на електронен протокол.

## ***Теоретична част:***

- **Въведение в паралелното програмиране с MPI**

MPI (Message Passing Interface) е стандарт за паралелно програмиране, базиран на обмен на съобщения между процеси, които могат да се изпълняват както на един и същи, така и на различни изчислителни възли в компютърния клъстер.

Моделът с обмен на съобщения (Фигура 3.1) е приложим за паралелни изчисления на компютърни клъстери, представляващи мултикомпютърни архитектури с множество процесори, всеки от които със собствена локална памет. Понастоящем MPI се използва в почти всички паралелни архитектури, включително масивно паралелни машини, SMP клъстери, PC клъстери и хетерогенни мрежи.




**Фиг. 3.1: Програмен модел с обмен на съобщения.**

Съществуват множество реализации на стандарта за почти всички операционни системи (Windows, Linux, Mac OS, HP-UX, AIX и др.), повечето от които са разработени като библиотеки. Някои MPI реализации са оптимизирани за конкретни високопроизводителни платформи: *LAM* (Local Area Multicomputer) – среда за стартиране и мониторинг на MPI приложения за клъстери под Unix; *openMPI* - свободна реализация с отворен код за мрежи от работни станции, клъстери, специализирани компютърни архитектури; *MPICH2* – високопроизводителна, преносима върху множество платформи Unix, Windows и MAC OC библиотека.

Паралелната програма обхваща множество процеси, всеки от който има идентификатор и завършва тогава, когато завърши изпълнението на всичките ѝ процеси.

Често използваният стил за паралелно програмиране в програми на MPI е **SPMD** (**Single Program Multiple Data**) - множество процеси изпълняват един и същи код над различни части от данните. Това предполага разпределянето на данните на приложението между наличните процесори. Този тип паралелизъм се нарича *паралелизъм по данни*. Указаният брой процеси се създават и разпределят на всеки процесорен възел на клъстера при стартиране на MPI програмата.

- **Основни функции в структурата на MPI програма**

  - ***MPI_Init()*** – инициализира MPI средата за изпълнение. Функцията трябва да бъде извиквана от всяка MPI програма еднократно. Може да бъде използвана за предаване на аргументите **argc** и **argv** от командния ред към всички процеси. Всички участващи процеси трябва да инициализират собствено копие на библиотеката.

```
int MPI_Init(
        int *argc,          // pointer to argc
        char ***argv);      // pointer to argv
```

  - ***MPI_Finalize()*** - терминира MPI средата за изпълнение. Всички процеси трябва да приключат изпълнението си посредством извикване на функцията.

```
int MPI_Finalize();
```

  - ***MPI_Comm_rank()*** - връща ранга на извикващия процес (**process ID**) в комуникатора **comm**.

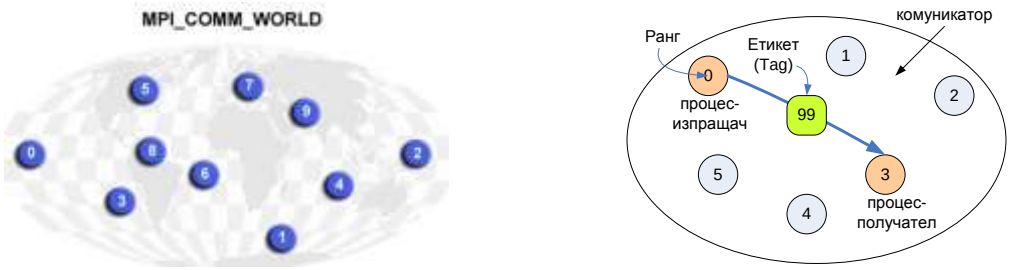
```
int MPI_Comm_rank(
        MPI_Comm comm,      // communicator
        int *rank);         // process ID (output)
```

  - ***MPI_Comm_size()*** - връща общия брой процеси, асоциирани с комуникатор **comm**.

```
int MPI_Comm_size(
        MPI_Comm comm,      // communicator
        int *size);         // number of process (output)
```

- **Комуникации от точка в точка (point-to-point)**

Базов механизъм за обмен между процеси в паралелно MPI приложение, при който процес-източник предава на процес-приемник. Всеки от двата процеса извиква съответната MPI операция. Комуникацията се осъществява в рамките на определен **комуникатор**, към който принадлежат и двата процеса. Комуникация в рамките на една група от процеси се нарича **интракомуникатор**, а тази между различни групи от процеси - **интеркомуникатор**; позволява логическо разделение на дейностите в рамките на една паралелна задача.




**Фиг. 3.2: Комуникации между процеси.**

- **Режими на комуникация**: *синхронен*, *буфериран*, *стандартен*, *режим с готовност*.

Режимите на комуникация се отнасят до операция „изпращане“. Приемането винаги е синхронно.

| Режим | Условия за приключване |
|---|---|
| Синхронно изпращане | Приключва след завършване на операцията по приемане. |
| Буферирано изпращане | Приключва винаги (ако не възникне грешка), независимо дали приемането е приключило. |
| Стандартно изпращане | Държи се като синхронно или буферирано според конкретната MPI реализация. |
| Изпращане с готовност | Приключва винаги (ако не възникне грешка), независимо дали приемането е успешно. |
| Приемане | Приключва след получаване на съобщението. |

- **Режими на изпращане**: блокиращ, неблокиращ (асинхронен).

*Режим на блокиране*: MPI извикването блокира и не връща управлението на програмата до приключване на операцията.

*Асинхронен режим*: MPI извикването връща управлението веднага и продължава да се изпълнява успоредно с изпълнението на програмата. Връща манипулатор на асинхронна операция, с който се тества или изчаква завършването на операцията.

| Операции | Блокираща форма | Неблокираща форма |
|---|---|---|
| Стандартно изпращане | MPI_Send() | MPI_Isend() |
| Синхронно изпращане | MPI_Ssend() | MPI_Issend() |
| Буферирано изпращане | MPI_Bsend() | MPI_Ibsend() |
| Изпращане с готовност | MPI_Rsend() | MPI_Irsend() |
| Приемане | MPI_Recv() | MPI_Irecv() |

- **Семантика на комуникациите в MPI**

o Изпратените от един до друг процес съобщения се приемат по реда на изпращането им, независимо от използвания режим на изпращане.

o При наличие на съвпадащи операции по изпращане и приемане, поне една от двете ще приключи нормално и няма да „зависне”.

o Блокиращите и неблокиращите операции могат да се смесват свободно и от двете страни.

- **Функции за изпращане на съобщения при point-to-point комуникации**

o ***MPI_Send()***. Имплементира стандартно изпращане на данни към процес dest в режим на блокиране. Функцията приключва след започване на процеса на изпращане. На практика се реализира като синхронно или буферирано изпращане според реализацията. Изпратените и неполучени съобщения се натрупват в опашки и буфери, препълването на които може да причини проблеми. Не трябва да се разчита на преносимост на конкретното поведение на стандартното изпращане между различните реализации.

```
int MPI_Send(
         void *buf,              // send buffer
         int count,              // # of elements to send
         MPI_Datatype datatype,  // datatype of elements
         int dest,               // destination (receiver) process ID
         int tag,                // message tag
         MPI_Comm comm);         // communicator
```

- ***MPI_Isend()*.** Имплементира неблокиращо изпращане на съобщение, т.е. процесът, който прави обръщение към тази функция, веднага се връща към изпълнението на следващите оператори в програмата, без да се блокира, докато операцията по изпращане на съобщението се изпълни.

```
int MPI_Isend(
        void *buf,          // initial address of send buffer (choice)
        int count,          // number of elements in send buffer (integer)
        MPI_Datatype datatype, //datatype of each send buffer element (handle)
        int dest,           // rank of destination (integer)
        int tag,            // message tag (integer)
        MPI_Comm comm,      // communicator (handle)
        MPI_Request *request); // communication request (handle)
```

- ***MPI_Ssend()***. Имплементира синхронно изпращане в режим на блокиране. Операцията по изпращане приключва след започването на операцията по приемане от страна на приемащия процес. Това е най-бавната и сигурна операция по изпращане, улесняваща откриването на грешки в процеса на комуникация.

```
int MPI_Ssend(
        void* buf,          // send buffer
        int count,          // # of elements to send
        MPI_Datatype datatype, // datatype of elements
        int dest,           // destination (receiver) process ID
        int tag,            // message tag
        MPI_Comm comm);     // communicator
```

- ***MPI_Bsend()***. Имплементира буферирано изпращане на съобщения в режим на блокиране. Операцията по изпращане приключва след като съобщението бъде копирано в предварително прикачен буфер, зададен от потребителя. За прикачване на потребителски буфер с определен размер се използва функция **MPI_Buffer_attach(buf, size)**. След приключване на използването на буфера, той трябва да се отдели с **MPI_Buffer_deattach(buf, size)**. Предимствата на буферираното изпращане се изразяват в предсказуемостта на поведение на комуникационната система в условията на претоварване – грешка от препълване на буфера.

```
int MPI_Bsend(
        void* buf,          // send buffer
        int count,          // # of elements to send
        MPI_Datatype datatype, // datatype of elements
        int dest,           // destination (receiver) process ID
        int tag,            // message tag
        MPI_Comm comm);     // communicator
```

- ***MPI_Rsend()***. Имплементира изпращане на данни с готовност в режим на блокиране. Функцията приключва след започване на процеса по изпращането. Изпращането се счита за успешно след като получателят на съобщението вече е извикал **MPI_Recv** или **MPI_Irecv** (т.е. в приемащия буфер се съдържа новополученото съобщение) *преди* започването на изпращането. Поведението в останалите случаи не е дефинирано. В някои случаи използването на такъв тип комуникация спомага за премахването на част от комуникационния свръх товар, но от друга страна изисква специфична програмна логика и предварителна синхронизация.

```
int MPI_Rsend(
        void* buf,              // send buffer
        int count,              // # of elements to send
        MPI_Datatype datatype,  // datatype of elements
        int dest,               // destination (receiver) process ID
        int tag,                // message tag
        MPI_Comm comm);         // communicator
```

- ***MPI_Recv()*.** Имплементира получаване данни от процес **source**. Операцията приключва след получаването на съобщението. Приемащият буфер трябва да има капацитета да побере поне **count** елемента от тип **datatype**. Приемат се само съобщение от процес-източник (**source**) с маркер (**tag**) по реда на изпращането им. Не се прави проверка за съвпадение на типа на данните (**datatype**). Приемат се съобщения, изпратени във всеки един от четирите режима.

```
int MPI_Recv(
        void *buf,              // receive buffer
        int count,              // # of elements to receive
        MPI_Datatype datatype,  // datatype of elements
        int source,             // source (sender) process ID
        int tag,                // message tag (integer)
        MPI_Comm comm,          // communicator (handle)
        MPI_Status *status);    // status (output)
```

- ***MPI_Irecv()***. Имплементира неблокиращо приемане на съобщение, т.е процесът, който прави обръщение към тази функция, веднага се връща към изпълнение на следващите оператори в програмата, без да се блокира, докато операцията по получаване на съобщението не се изпълни.

```
int MPI_Irecv(
        void *buf,          // initial address of receive buffer (choice)
        int count,          // number of elements in receive buffer (integer)
        MPI_Datatype datatype,  // datatype of each receive buffer element
        int dest,               // rank of source (integer)
        int tag,                // message tag (integer)
        MPI_Comm comm,          // communicator (handle)
        MPI_Request *request);  // communication request (handle)
```

## ***Изпълнение на експерименталната част:***

Изчислителният ресурс, използван за провеждане на лабораторните експерименти, е хетерогенен компютърен клъстер. Той се състои от свързани в мрежа многоядрени сървъри, които могат да работят съвместно по обща задача. Потребителят се свързва към входен възел, през който се подават задачи за изпълнение на изчислителните възли. Клъстерът разполага с 10 изчислителни възела: 8 двуядрени сървъра с процесори AMD Opteron Dual Core и 2 сървъра с по два Intel Xeon E5405 Quad Core процесори всеки. Всеки от съставляващите компютърния клъстер сървъри използва операционна система Scientific Linux.

## A. Създаване, компилиране и изпълнение на MPI приложения

A1. Създайте показания програмен код на C с MPI. Съхранете го във файл под име „**MPI_HelloWorld.c**“.

```c
// File: MPI HelloWorld.c
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[]){
      int ntasks, taskid, len, rc, version, subversion;
      char hostname[MPI_MAX_PROCESSOR_NAME];

      rc = MPI_Init(&argc,&argv);
      if (rc != MPI_SUCCESS) {
            printf ("Error starting MPI program. Terminating.\n");
            MPI_Abort(MPI_COMM_WORLD, rc);
      }
      MPI_Comm_size(MPI_COMM_WORLD,&ntasks);
      MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
      MPI_Get_processor_name(hostname,&len);
      MPI_Get_version(&version,&subversion);
      printf("MPI standard version: %d, subversion: %d\n", version, subversion);
      printf("%d processes are running on %s\n",ntasks, hostname);

      MPI_Finalize();
      return 0;
}
```

A2. Осъществете достъп до клъстера с помощта на програма за отдалечен достъп. Въведете потребителско име и парола.

A3. В работната директория на входния възел на клъстера създайте директория „**PP_MPI**“. С помощта на програмата за файлов трансфер, прехвърлете файла „**MPI _HelloWorld.c**“. Компилирайте и изпълнете кода.

A4. Модифицирайте горната програма така, че: 1) процес 0 да бъде идентифициран като „главен“ (master); 2) да се изведат идентификационните номера на всички останали процеси, заедно с името на хоста, на който се обслужват; 3) главният процес да изведе информация за броя на всички процеси в комуникатора. Съхранете кода на модифицираната програма във файл „**mpi_HelloWorld_v1.c**“. Компилирайте кода на клъстера и го изпълнете. Проведете експерименти с вариране на броя изчислителни възли и задачи. Анализирайте получените резултати.

*Документирайте изпълнението на заданието и получените резултати в отчета си.*

## B. Комуникации между процеси

- **Блокиращи и неблокиращи комуникации между процеси**

B1. Създайте показания програмен код на C с MPI. Съхранете го във файл под име „**MPI_p2p.c**“.

```c
// File: MPI_p2p.c
#include "mpi.h"
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[])
{
      char message[20];
      int myrank;
      MPI_Status status;
      MPI_Init( &argc, &argv );
      MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
      if (myrank == 0)
      {
            strcpy(message,"Hello, there!");
            MPI_Send(message, strlen(message), MPI_CHAR, 1, 99, MPI_COMM_WORLD);
      }
```

```
	else
	{
		MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
		printf("Received :%s:\n", message);
	}
	MPI_Finalize();
	return 0;
}
```

B2. Прехвърлете и компилирайте кода на клъстера. Анализирайте резултатите от неговото изпълнение за два процеса.

B3. Модифицирайте горния пример така, че: 1) всеки от процесите да върне обратно съобщението към неговия подател и 2) всеки от процесите да изведе идентификационния си номер, съдържанието на полученото съобщение, идентификационния номер на подателя на съобщението, заедно с тага на съобщението, както е показано:

**Process** *<procss_id>*: **Received** *<message_text>* **with tag** *<message_tag>* **from process** *<process_id>*.

Съхранете кода на модифицираната програма във файл „**MPI_p2p_v1.c**“. Компилирайте кода на клъстера и го изпълнете. Анализирайте получените резултати.

*Документирайте изпълнението на заданието и получените резултати в отчета си. Приложете разработения програмен код.*

B4. Модифицирайте горната програма така, че: 1) при изпълнението си да позволява да бъдат стартирани само четен брой процеси; 2) главният процес (master) да извежда информация за броя на стартираните процеси; 3) всеки процес да изведе информация за името на възела (хоста), на който се обслужва; 4) процесите да бъдат организирани по двойки, които да разменят помежду си информация за идентификационните си номера; 5) всеки процес да изведе кой е неговият комуникационен партньор. Съхранете кода на модифицираната програма във файл „**MPI_p2p_v2.c**“. Компилирайте кода на клъстера и го изпълнете. Проведете експерименти с вариране на: 1) броя изчислителни възли и 2) броя изчислителни задачи. Анализирайте получените резултати.

*Документирайте изпълнението на заданието и получените резултати в отчета си. Приложете разработения програмен код.*

B5. Преработете програма от задание B4, като за осъществяване на комуникациите използвайте вместо блокиращи, неблокиращи комуникации с **MPI_Isend()** и **MPI_Irecv()**. Съхранете кода на модифицираната програма във файл „**MPI_p2p_v2_nonblock.c**“. Компилирайте кода на клъстера и го изпълнете. Проведете експерименти за същия брой изчислителни възли и задачи, както в задание B4. Анализирайте получените резултати, като ги съпоставите с тези, получени при изпълнение на задание B4.

- В какъв момент от време настъпват комуникациите между задачите при използване на неблокиращи комуникации?
- Има ли ситуации, в които настъпва „мъртва хватка“? Предложете и реализирайте вариант за справяне с проблема.

*Документирайте изпълнението на заданието и получените резултати в отчета си. Приложете разработения програмен код.*

- **Комуникационна производителност на блокиращ и неблокиращ режими**

B6. Анализирайте приложените към упражнението програмни кодове на C с MPI „**MPI_p2p_bandwidth_block.c**“ и „**MPI_p2p_bandwidth_nonblock.c**“. Компилирайте всеки от кодовете

на клъстера. Анализирайте изпълнението на всеки от тях поотделно, като проведете два вида експериментални изследвания: 1) с две задачи, стартирани на един и същи възел и 2) с две задачи, стартирани на два различни възела. Съпоставете получените експериментални резултати.

- Как се променя комуникационната производителност в зависимост от размера на обменяните съобщения и вида на комуникационния режим?
- Как се променя комуникационната производителност в зависимост от това дали двете задачи споделят изпълнението си върху един или върху два различни изчислителни възли?

*Документирайте изпълнението на заданието и получените резултати в отчета си.*

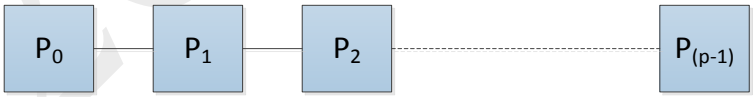
## C. Задачи за самостоятелна работа

**C1**: Да се разработи програма на C с MPI, която да изчислява сумата на *n* цели числа. Програмата да бъде реализирана така, че: (а) всеки процес да използва ранга си като входна стойност за сумиране; (б) всеки процес с ранг, по-голям от 0, трябва да изпраща стойността на ранга си на процес с ранг 0; (в) процес с ранг 0 да получава стойностите от останалите процеси и да изчислява сумата, след което да я извежда на екрана.

*Документирайте изпълнението на заданието и получените резултати и анализи в отчета си. Приложете разработения програмен код.*

**C2**: Преработете програмата от задание C1, така че обменът на съобщения да се реализира при линейна топология[1] на мрежата за междупроцесорни комуникации (Фигура 3.3). Проведете експериментални изследвания с вариране на броя на генерираните процеси. Анализирайте получените резултати.

[1]**Линейна топология** на мрежата за междупроцесорни комуникации означава, че всеки процес, освен първия и последния, има директни комуникационни канали за връзка с непосредствените си съседи. Прост начин за разпространяване на съобщение в този случай е повтарящо се изпращане на съобщението към процесора, който е непосредствен съсед от ляво или отдясно, докато не достигне до целта си, т.е. последния процесор в линейния масив. Всички процеси участват в комуникациите. Процес с ранг *k* (*k* е по-голямо от 0) получава акумулираната или частична сума от предишния процес с ранг *k-1*. Накрая процес с ранг *p-1* извежда крайната сума.




**Фиг. 3.3: Процеси, свързани в линейна топология.**

*Документирайте изпълнението на заданието и получените резултати и анализи в отчета си. Приложете разработения програмен код.*