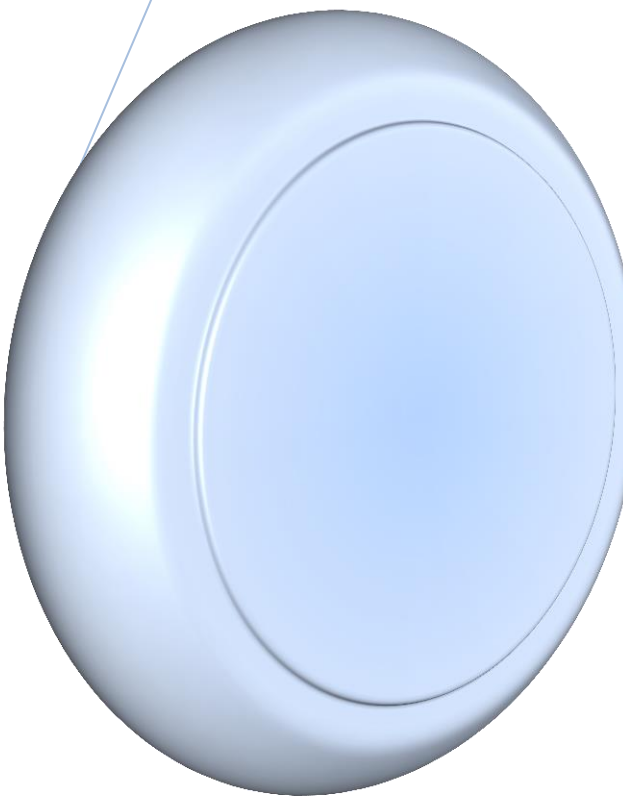
Тема 1: Въведение в паралелното програмиране с MPI и OpenMP

Въведение в паралелното програмиране с MPI и OpenMP

В тази тема са включени обяснения, задачи и примери имащи за цел придобиване на основните компетенции за работа с високопроизводителни системи.



1 Актуалност на проблема

Паралелното програмиране обхваща дейностите по създаването на компютърни програми, при които множество изчисления се изпълняват едновременно. То се базира на принципите за декомпозиране на изчислителния проблем на по-малки части, като обхваща различни програмни модели и технологии.

Техниките за паралелно програмиране, в областта на високопроизводителните изчисления, се разработват още от зората на компютрите, но поради физическите ограничение при увеличаване на честотата (достигане на термалната бариера) интересът към паралелизацията през последните години нараства значително, като основна парадигма за постигане на висока производителност и ниска консумация на енергия.

2 Цел и задачи

Целта на това упражнение е придобиване на основните компетенции за работа с многоядрени и мултипроцесорни компютърни системи включващи: осъществяване на отдалечен достъп до изчислителните ресурси, разработване и компилиране на паралелен код, стартиране на паралелни OpenMP и MPI програми, запознаване с основните техники за синхронизация и др.

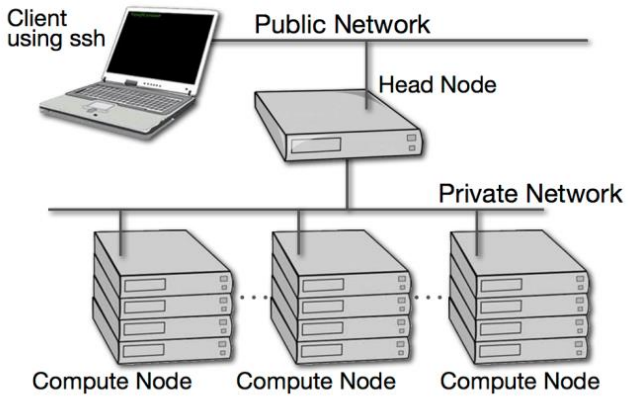
Цели се също така и запознаване с основните параметри за оценка на паралелната производителност, като ускорение и ефективност и използването на базови средства за анализ на комуникацията между процесите.

3 Експериментална платформа

За целите на упражненията студентите получават достъп до хетерогенен компютърен клъстер.

3.1 Достъп до изчислителните ресурси

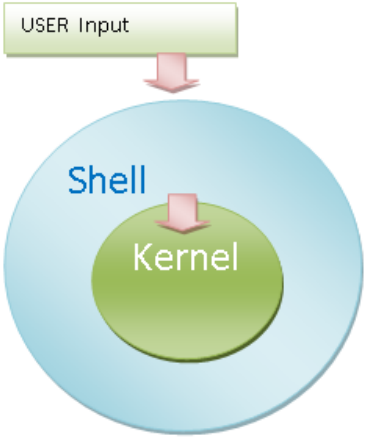
Компютърният клъстер се състои от свързани в мрежа компютри, които могат да работят по обща задача. Потребителят се свързва към входен възел през който може да пуска задачи на изчислителните възли, фиг. 2. Клъстерът разполага 10 изчислителни възела: 8 сървърни машини с AMD Opteron Dual Core процесори и 2 машини с по два Intel Xeon E5405 Quad Core процесори всяка.



Фиг. 1. Достъп през ssh протокол

Всеки от съставляващите компютърния клъстер компютри използва операционна система Scientific Linux. Операционната система най-общо се състои от ядро (kernel), което се грижи за управлението на хардуера и обвивка (shell), чрез която се пускат потребителските програми, фиг. 2. Съществува rsh (Remote Shell) протокол за отдалечен достъп, но тъй като той не е сигурен се ползва ssh (Secure Shell) протокол.

Shell – позволява пускане на програми и т.н.
Rsh (remote shell) - несигурен метод за отдалечен достъп
Ssh (secure shell) – сигурен метод за отдалечен достъп. Каналът се криптира.



Фиг. 2. Архитектура на ОС Линукс

Ако връзката се осъществява от Linux, то за достъп до входния възел на клъстера е необходимо да се изпълни командата:

ssh username@example.com

Където username е потребителското име, а example.com адреса на отдалечената машина.

За Windows може да се използват различни ssh и sftp клиенти, като например Putty, PSFTP и Filezilla.

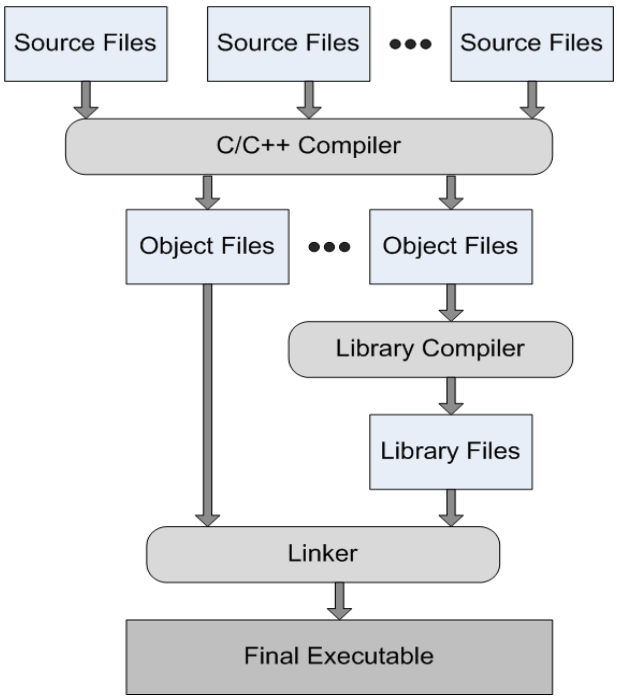
Забележка: Кратко описание за работа с клъстера може да се намери в файла info.txt намиращ се в началната директория на потребителя.

Необходимите стъпки, които трябва да се извършат са в следната последователност

- Осъществяване на връзка
- Копиране на файлове
- Компилиране на кода
- Стартиране на програмите

3.2 Компилиране и изпълнение паралелен код

При компилацията от сорс кодовете се получава изпълним файл, фиг. 3. При този процес компилатора транслира инструкциите на езика от високо ниво до обектен код, който най-общо представлява последователност от инструкции, които обаче нямат конкретни адреси я паметта. Линкерът комбинира обектните файлове и дава реални стойности на всички символни адреси, т.е. създава машинен код.



Фиг. 3. Илюстрационен процес на компилация

Например, за компилиране на файла Helloworld.cpp, съдържащ сорс код на примерна програма, до изпълнимият файла (програма) Helloword с компилатор gcc се изпълнява командата.
gcc Helloworld.cpp -o Helloworld
./Helloword Извиква програмата Helloworld за изпълнение.

Забележка: Ако не се специфицира името на изходния файл, чрез опцията „-o", то той ще бъде именуван по подразбиране "a.out"
При линукс текущата директория се означава с точка, а по-горната с две точки. От съображения за сигурност текущата директория не е в *Path* файла и е необходимо да бъде изрично упомената с „./" за да се стартира програма от нея. В горния пример ./Helloworld = Helloworld

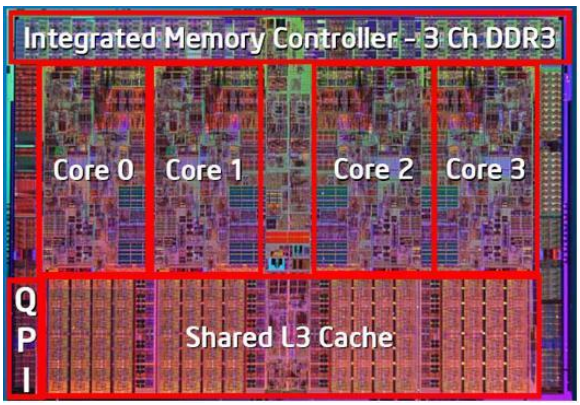
4 Методика за провеждане на упражнението

За да се акцентира върху същината на курса и повиши ефективността от обучението, са подготвени скелети на програми, които обучаемите трябва да модифицират, за да достигнат самостоятелно до решенията, които също са налични. За по-напреднали в програмирането студенти, които биха се правили по-бързо, са предвидени допълнителни примери.

4.1 Паралелно програмиране с OpenMP

Този раздел цели запознаване с основните техники за паралелно програмиране при използването на програмния модел с обща памет OpenMP.

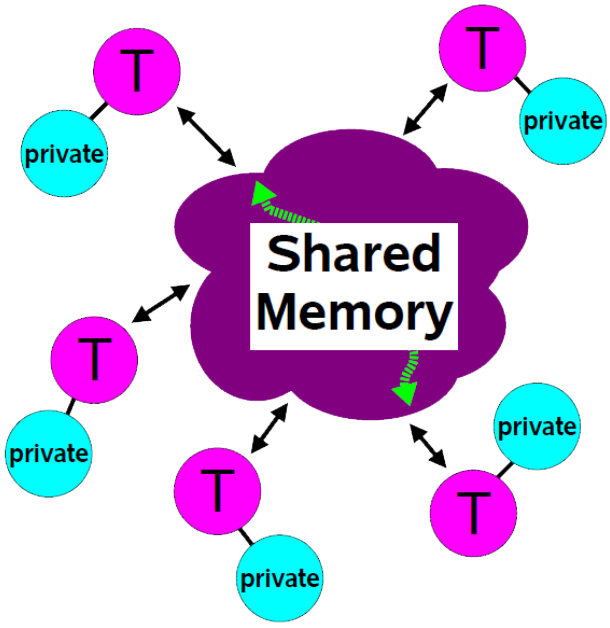
OpenMP се базира на програмен модел с обща памет, до която всички нишки имат асинхронен достъп и е лесно приложим за многоядрени архитектури, където няколко физически ядра споделят обща памет, фиг. 4.



Фиг. 4. Разположение на физически ядра в многоядрен процесо

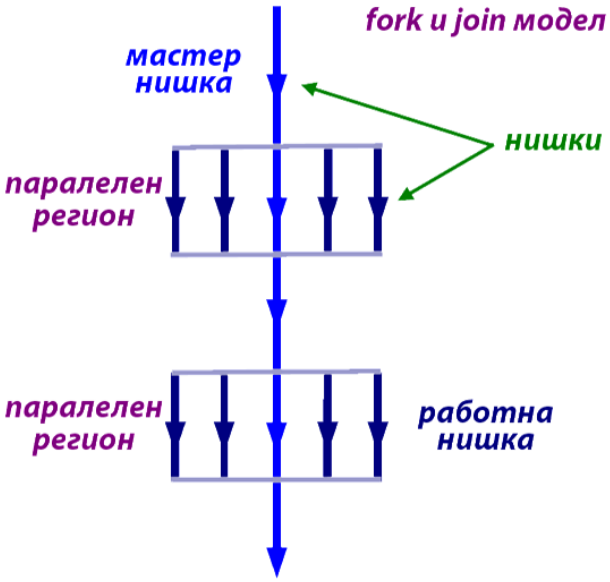
Променливите могат да бъдат **shared** или **private**, фиг 5.

- ***shared променливи***
  - Видими са за всички нишки.
  - Ако една нишка промени shared променлива, то всички нишки ще видят промяната.
  - Всички променливи създадени преди създаването на нишките са shared.
- ***private променливи***
  - Видими са само за нишката, която ги притежава.
  - Промяна направена в private променлива се вижда само от нишката, която притежава променливата.
  - Локално създадените променливи са винаги private.



Фиг. 5. Достъп на нишките до Private и Public променливите

При стартиране на програмата само една нишка (master thread) е активна. Главната нишка изпълнява последователните участъци от програмата, а при наличие на разклонение, разпростира допълнителни нишки. В паралелните секции главната нишка и останалите се изпълняват паралелно (fork), а в края на паралелния код разпространените нишки се терминират и управлението се връща на главната (join), фиг. 6.



Фиг. 6. Паралелизация при OpenMP

OpenMP използва два основни подхода за разпределяне на обработката между множеството нишки.

- Паралелизъм на ниво цикли: всяка нишка получава уникален обхват от стойностите на индексираната променлива.
- Паралелизъм на ниво участъци: паралелизират се участъци, които могат да бъдат произволни редове от програмния код.

OpenMP синтаксис: Основни функции

- ***void omp_set_num_threads(int num_threads)*** - задава с колко нишки да се изпълняват паралелните региони.
- ***int omp_get_num_procs()*** - връща колко процесора (ядра) има системата.
- ***int omp_in_parallel()*** – връща резултат различен от нула ако се извика от паралелен регион.
- ***int omp_get_thread_num()*** – връща номера на нишката, която изпълнява паралелния регион.
- ***int omp_get_num_threads***() – връща броя на нишките, които изпълняват паралелния регион.
- ***double omp_get_wtime()*** – връща секундите от даден постоянен момент в миналото.

Забележка: За да се използват OpenMP функциите трябва да се добави OpenMP хедъра #include <omp.h>
Оптималният брой нишки е равен на броя физически процесори (ядра), който може да се получи чрез функцията omp_get_num_procs()

OpenMP осигурява широк спектър от т.нар. *прагми*, които представляват директиви за компилатора, необходими за паралелизацията и организацията на програмата.

Директивите имат следния синтаксис:

#pragma omp <тип> [clause[[,] clause]...]

Типове прагми:

#pragma omp parallel – Определя паралелен регион, който ще се изпълни от няколко нишки. Тук компилатора създава съответните нишки.

#pragma omp for – Указва на компилатора да изпълни итерациите на даден for цикъл паралелно.

```
#pragma omp parallel
{
        #pragma omp for
        for (i = 0; i < N; i++)
        {
        InputMatrix[i][j] = InputMatrix[i][j] + InputMatrix[i][j - 1];
        }
}
```

#pragma omp parallel for – Комбинация от предходните две прагми.

#pragma omp single – Определя секция от кода, която трябва да се изпълни само от една нишка.

```
#pragma omp single
{
        //What needs to be done
}
```

#pragma omp master – Синхорнизираща прагма, която указва на компилатора, че само мастер нишката трябва да изпълни този код.

#pragma omp critical – Определя, част от кода, която трябва да се изпълни само от една нишка в даден момент.

#pragma omp atomic – определя област от паметта, която може да се промени само от една нишка в даден момент.

Клаузите са опционални модификатори на действието на директивите, например клаузите за #pragma omp parallel са:

- **if (израз)** - Паралелния регион ще се изпълни само ако израза в скобите е верен.
- **private (променлива1, …)** – декларира, че променливите в скобите ще са private за всяка нишка.
- **firstprivate (променлива1, …)** – както при предишната клауза, но променливите ще бъдат инициализирани със стойността им по подразбиране.
- **copyin (променлива1, …)** - както при предишната клауза, но първоначалната стойност на променливите ще се вземе от променливата в общата памет.
- **num_threads (int)** – указва колко нишки да изпълнят паралелния регион.
- **shared (променлива1, …)** – декларира кой променливи ще са общи за всички нишки.

4.1.1 Задачи

Задача 1: Компилирайте и изпълнете примерен код на една от 8 ядрените машини в клъстера, чрез следните команди, където опцията -fopenmp включва библиотеката openmp, а опцията -o специфицира изходния файл.

$g++ -fopenmp omp_hello_world.c -o omp_hello_world
$./omp_hello_world

Сорс кодът е във файла *omp_hello_world.c*

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

/* Main Program */

int main()
{
  int threadID;
  /* Set the number of threads */
  omp_set_num_threads(2);

  /* OpenMP Parallel Directive */
  #pragma omp parallel private(threadID)
  {
  threadID = omp_get_thread_num();
  printf("\nHello World is being  printed by the thread id %d\n", threadID);
  }
}
```

Задача 2: Да се промени примера, така че основната нишка да принтира броя на всички пуснати нишки.

Задача 3: Да се пусне предишния пример на клъстера на университета.

Задача 4: Да се модифицира примера така, че да се използва клаузата редукция.
#pragma omp parallel for reduction(+ : sum)

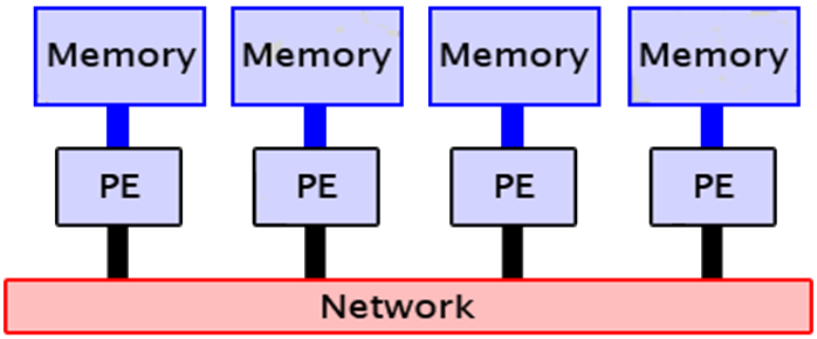
Задача 5: Да се паралелизира последователната имплементация на програмата за изчисление на PI.

Задача 6: Да се създаде паралелна част от програмата, която намира максимален елемент в масив.

Забележка: Серийните имплементации на програмите са в папка *omp_exercises*, а готовите решения на задачите в папката *solutions*.

4.2 Паралелно програмиране с MPI

Message Passing Interface е стандарт за обмен на съобщения между процеси, които могат да са на една и съща машина или на различни. Моделът с обмен на съобщения, фиг. 7, предполага мултикомпютърна архитектура с множество процесори, всеки с локална памет. Дори процесите да се изпълняват на една единствена машина те са изолирани един от друг.



Фиг. 7. Модел с обмен на съобщения.

Съществуват множество MPI имплементации за почти всички операционни системи (Windows, Linux, Mac OS, HP-UX, AIX и др.), повечето от които са разработени като библиотеки;

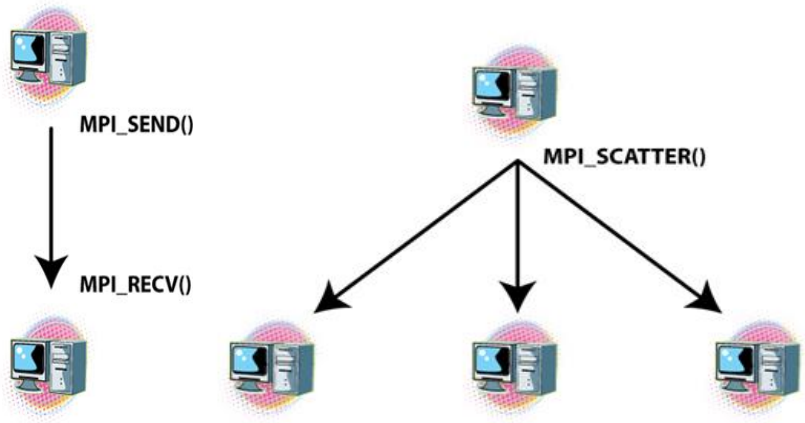
Някои от по-известните са:

- **MPICH2** - http://www.mcs.anl.gov/research/projects/mpich2/
- **OpenMPI** – http://www.open-mpi.org/

MPI типове данни

MPI_INT - signed int
MPI_SHORT - signed short int
MPI_LONG - signed long int
MPI_UNSIGNED - unsigned int
MPI_UNSIGNED_SHORT - unsigned short int
MPI_UNSIGNED_LONG - unsigned long int
MPI_BYTE - 8 binary digits
MPI_CHAR – char
MPI_UNSIGNED_CHAR - unsigned char
MPI_FLOAT – float
MPI_DOUBLE – double
MPI_LONG_DOUBLE - long double
MPI_PACKED – данни, които могат да се пакетират/разпакетират с *MPI_Pack()* и *MPI_Unpack()*

В MPI съществуват 2 основни типа комуникации: колективна и от точка до точка, фиг. 8.



Фиг. 8 Основни типове комуникация в MPI

Комуникацията от точка до точка бива:

- Синхронна (**MPI_SEND**, **MPI_RECV**);
- Асинхронна (**MPI_ISEND**, **MPI_IRECV**, **MPI_WAIT**);
- Буферирана синхронна (**MPI_BSEND, MPI_BRECV**);
- Съкратен запис (**MPI_ISENDRECV**, **MPI_SENDRECV**);

Примерни функции при синхронната комуникация са:

int MPI_Send (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

- ✓ *buf* – адрес на буфера за изпращане;

- ✓ *count* – брой елементи в буфера
- ✓ *datatype* – тип на данните в буфера;
- ✓ *dest* – дестинация;
- ✓ *tag* – таг на съобщение;
- ✓ *comm* – комуникатор.

int MPI_Recv (void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

- ✓ *buf* – адрес за съхранение на получената стойност;
- ✓ *status* – статус;
- ✓ *count* – брой елементи за получаване;
- ✓ *datatype* – тип на елементите;
- ✓ *source* – от къде се получава съобщението;
- ✓ *tag* – таг на съобщението;
- ✓ *comm* – комуникатор.

4.2.1 ЗАДАЧИ

Задача 1: Да се компилира и изпълни на клъстера примерен „helloworld" код, който се намира във файл mpi_hello_world.cpp. Да се снеме и анализира диаграмата на Ганд на паралелната комуникация.

За да компилирате файла mpi_hello_world.c до изпълним файл mpi_hello_world, е необходимо да изпълните следната команда, където mpicxx е обвивката (wripper) на GCC C++ компилатора с включени MPI библиотеки.

$mpicxx mpi_hello_world.c –o mpi_hello_world

Забележка: Обикновено MPI програми се извикват с командите *mpirun* или *mpiexec*, като се специфицира броят процеси, но тъй като на клъстера има инсталирам ресурсен мениджър Slurm, то в случая MPI програмите се изпълняват с srum [–n X] <program>, където X е броя процеси които да се стартират. Конкретно за стартиране на горния пример с 8 процеса, е необходимо да се изпълни следната команда.

$srun –n 8 ./*mpi_hello_world*

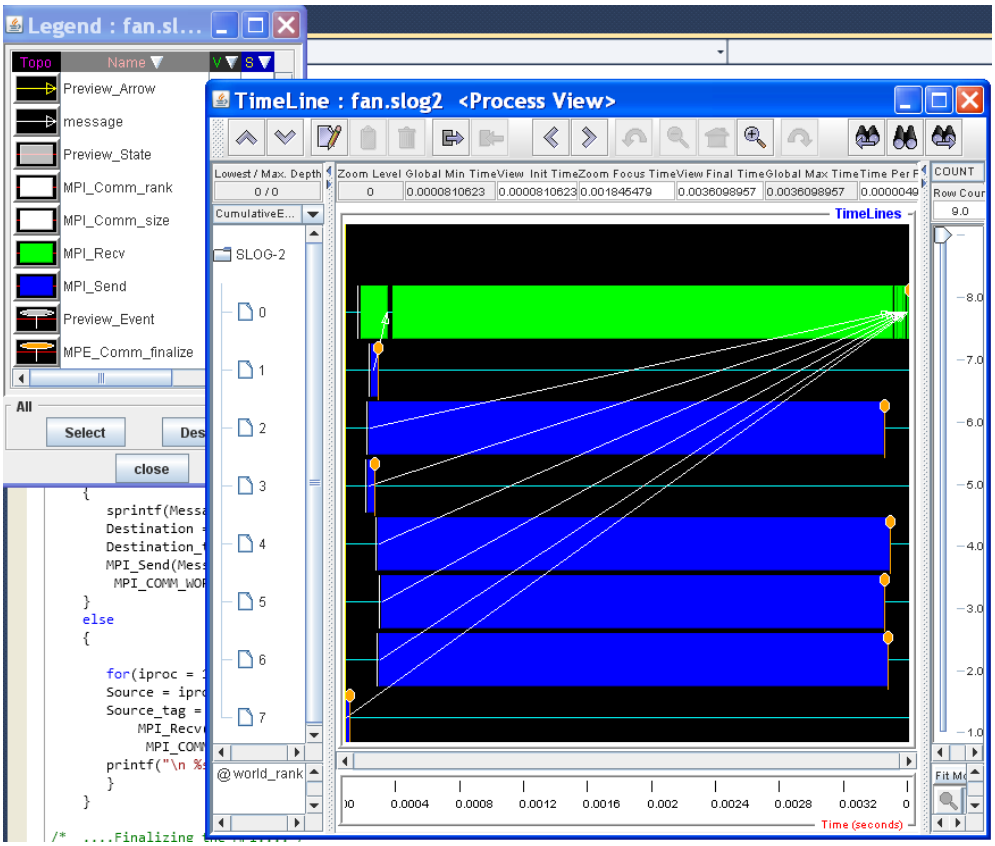
Стартирана по този начин програмата ще създаде 8 процеса, като задава по два на изчислител възел.

Удобно средство за анализ на паралелните програми и изобщо за разбирането на принципите при паралелното програмиране предоставя библиотеката MPE (MPI Parallel Environment). За да се добави MPE поддръжка се добавя опцията „-mpe=mpilog".

mpicxx -mpe=mpilog *mpi_hello_world.c –o mpi_hello_world*

След като е програмата компилирана по този начин бъде изпълнена, че се създаде clog2 файл съдържащ информация за комуникацията между процесите. Този файл може да бъде визуализиран на локалната машина с програмата Jumpshot, която е част от пакета mpich2. Фигура 9 показва диаграма на Ганд за

комуникация от тип P2P, при която процесът с ранг 0 получава съобщения от останалите процеси, чрез функцията MPI_Recv.



Фиг. 9. Диаграма на Ганд на паралелната комуникация

Задача 2: Да се разгледа, компилира и изпълни програма намираща сумата от цели числа, използваща блокираща комуникация от тип точка до точка. Да се представи диаграма на Ганд на паралелната комуникация.

Задача 3: Д се преработи кода от задача 2, така че комуникацията да е в линейна топология. Да се представи диаграма на Ганд на паралелната комуникация.

Задача 4: Д се преработи кода от задача 3, така че комуникацията да е в кръгова топология. Да се представи диаграма на Ганд на паралелната комуникация.

Задача 5: Да се преработи задача 3, така че да се използва функцията MPI_Reduce. Да се представи диаграма на Ганд на паралелната комуникация.

Задача 6: Да се преработи задача 5, така че:
Процес 0 да разпредели елементите от едномерен масив с 1000 елемента между всички процеси (MPI_Scatter);

- Всеки процес да сумира своята част от елементите;
- Да се сумират временните резултати с MPI_Reduce();
- Да се извежда резултата.
- Да се представи диаграма на Ганд на паралелната комуникация.

Задача 7: Да се оцени паралелната производителност на програма намираща прости числа (Сито на Ератостен).

Съществуват два основни подхода за програмен анаиз: статичен и динамичен.
Едно от популярните средства за статичен анализ Doxygen. представлява инструмент който генерира документация от анотиран (annotated) C++ сорс код, н също съдържа други популярни програмни езици като C, Objective C, C#, PHP, Python, Fortran, VHD, Tcl и др.

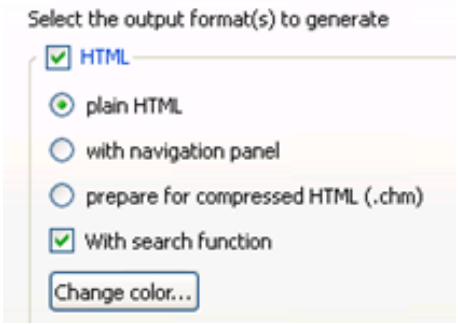
С какво може да ни помогне Doxygen?

1. *Може да генерира доументация и да я визуализира в браузър (HTML) или под формата на офлайн ръководство (Latex). Текста на документацията се извича директно от сорс кода, което го прави много по-лесно да се държи консистентноста на документацията заедно с кода.*

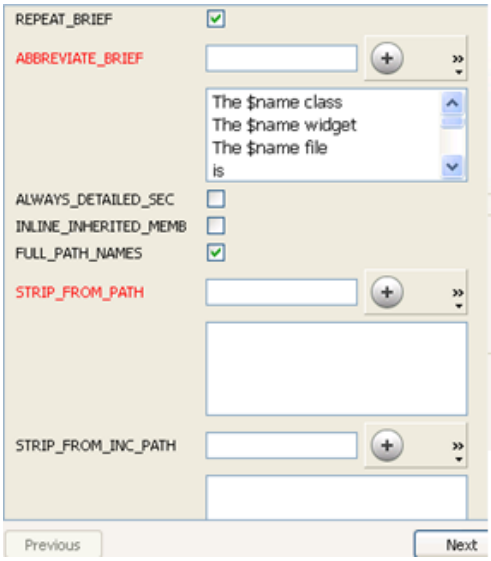
2. *Doxygen може да се конфигурира да извлича структурата на кода от недокументирани сорс файлове. Това е много удобно особено когато трябва да се ориентираме в големи файлове с доументация. Doxygen може също да визуализира връзките между различните елементи под формата на графики, диаграми на наследяванията и диаграми на съвместната работа на процесите - всичко това се генерира автоматично.*

Употреба** чрез Windows **:

1. *Стартираме **Doxywizard***
   *Слагаме отметка на полето **Scan Recursively***
2. *От таба **Expert**, избираме **DOT** от листа Topics*
   *От листа в средата на екрана изберете **HAVE_DOT** и укажете **DOT_PATH***
3. *Укажете изхода (**output**) в таб **Wizard***



4. *Пълният път може да се пропусне (не е задължителен)*



5. *Стартирайте Doxygen*

При динамичния анализ дадена програмата се оценява в резултат от нейното изпълнение.

Налични на клъстера средства за динамичен анализ, освен вече разгледаното MPE, са Scalasca, GProf и др.

Сито на Ератостен е алгоритъм за намиране на всички прости числа в интервала [1, n], където n е произволно естествено число.

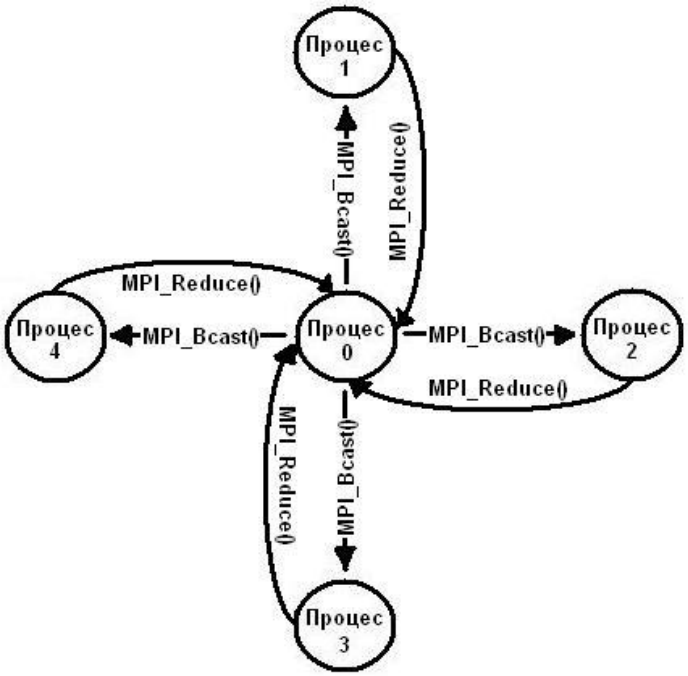
Псевдокод на алгоритъмът :

```
масивът S се попълва със стойности 'да'
за всяко i от 2 до n
  ако S[i] е 'да', тогава
    j = i+i; //с оптимизация 1. от долу: j = i*i;
    докато j <= n
      S[j] = 'не'; //"задраскване"
      j = j + i;
```

След изпълняването на този алгоритъм, всяко число i, за което S[i] има стойност "да", е просто. Последователният алгоритъм, предложен от Ератостен, е практически неефективен, тъй като има сложност (*n ln ln n*), където *n* е броят на целите числа в масива.

Масивът от цели числа се разделя на *n/p* блока (, където *p* е броят на процесорите) и всеки процесор отговаря за пресяването на *n/p* цели числа.

Паралелните изчисления се организират, както следва: главният процес разпръсква текущото просто число за пресяване останалите процеси. След завършване на пресяването на текущото число се осъществява колективна комуникация от тим редукция, посредством която резултатите от пресяването на всички процеси се изпращат към главният процес, както е показано на фиг.10.



Фиг.10 Паралелен модел за ситото на Ератостен

Забележка: Конкретната имплементация на алгоритъма изисква подаване на аргумент от командния ред.

Примерни опитни резултати:

За по-точни резултати всеки опит е направен 5 пъти, и като стойност за опита е взета средноаретмитичната от 5-те опита. Резултатите от опитите са представени спрямо размерът на масивът от цели числа – *n*.

Ускорението е изчислено по формулата:

$$S_p=T_s/T_{par}$$

където *p* е броят на процесорите, S_p - ускорението на *p*-процесорна, T_s – времето за последователно изпълнение на дадената задача и T_{par} – времето за паралелно изпълнение са същата задача.

Ефективността е изчислена по формулата:

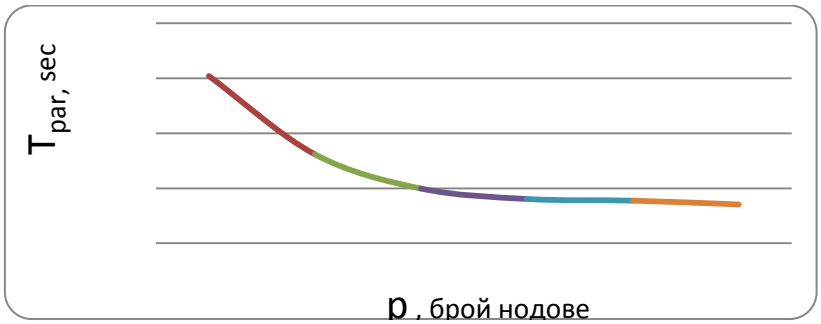
$$E_{p=}S_p/p$$

където *p* е броят на процесорите, S_p - ускорението на *p*-процесорна и E_p – ефективността на паралелната обработка.

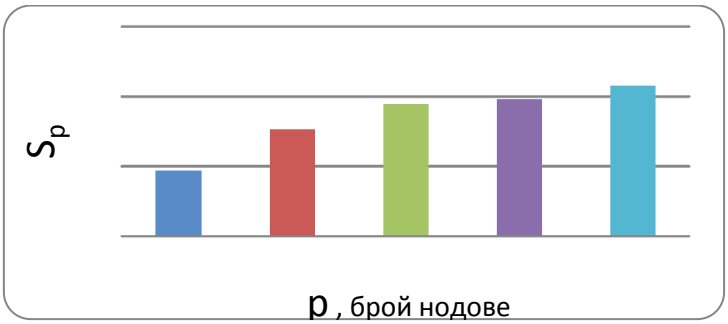
- n=1000000 :

брой нодове	време, сек.	Ускорение	Ефективност	n=1000000
1	0.0608444	0.00	0.00	
2	0.0323594	1.88	0.94	
4	0.019913	3.06	0.76	
6	0.0161118	3.78	0.63	
8	0.015498	3.93	0.49	
10	0.0141302	4.31	0.43	

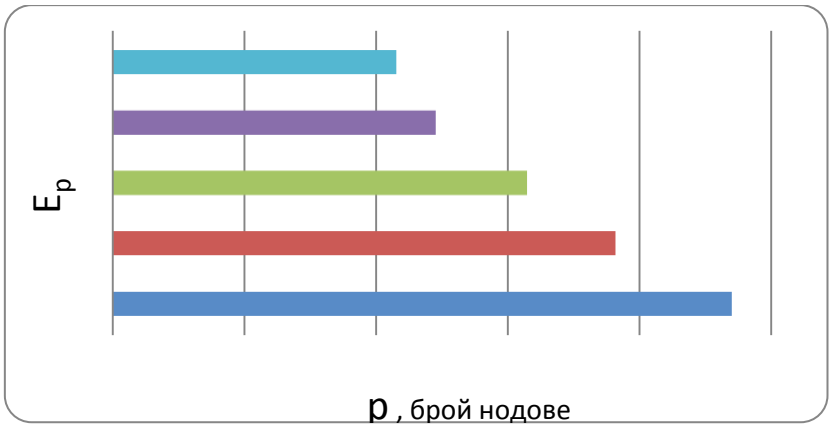
Табл.1 Таблица на примерните опитни резултати



Граф.1 Графика на времето за изпълнение на задачата



Граф.2 Графика за ускорението при изпълнение на задачата върху p нода



Граф.3 Графика за ефективността при изпълнение на задачата върху p нода

Забележка: Серийните имплементации на програмите са в папка mpi_exercises, а готовите решения на задачите в папката solutions.

4.2.2 Хибриден програмен модел

Разгледаните два програмни модела (MPI и OpenMP) могат да бъдат комбинирани чрез програмиране с обмен на съобщения и многонишкова обработка. Грубият грануляритет (Coarse-grained domain decomposition) се реализира с обмен на съобщения, а финия (Fine-grained), с многонишкова обработка.

Компилирането и изпълнението на примерен хибриден код, който се намира във файл hybrid.cpp е аналогично с MPI кодовете, но е необходимо да се добавят библиотеки за OpenMP.

$mpicxx -fopenmp hello-hybrid.c –o hello-hybrid
$srun –n 5 ./ hello-hybrid

5 Експериментални резултати и оформяне на протокола

За всички задачи от точка 4.2.1 да се снемат диаграми на Ганд и да се представи кратко обяснение по тях. За задача 6 от точка 4.1.1 и за задачи 6 и 7 от точка 4.2.1 с MPI да се оцени паралелната производителност, като се представят графики на ускорението и ефективността.