Мутипроцесорски системи - вежбе

4. decembar 2014

Glava 1

MPI Hello World

MPI је библиотека функција за размену порука између процесора. Овим функцијама се може приступити из програмских језика C/C++ и Fortran. Једноставан програм који исписује "поздраве" са више процесора на терминал ће бити програм за упознавање са основним функцијама из MPI библиотеке.

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

const int MAX_STRING = 100;

int main(void) {
    char gret[MAX_STRING];
    int csize;
    int prank;

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &csize);
    MPI_Comm_rank(MPI_COMM_WORLD, &prank);

    if (prank != 0) {
        sprintf(gret, "Grets␣from␣process␣%d␣of␣%d!", prank, csize);
        MPI_Send(gret, strlen(gret)+1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    } else {
        printf("Grets␣from␣process␣%d␣of␣%d!\n", prank, csize);

        for (int q = 1; q < csize; q++) {
            MPI_Recv(gret, MAX_STRING, MPI_CHAR, q, 0,
                                        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("%s\n", gret);
        }
    }

    MPI_Finalize();
    return 0;
}
```

Да би се добио приступ MPI функцијама унутар C/C++ програма потребно је укључити датотеку `mpi.h`. Наравно, током развоја MPI програма морају

се користити и остале C/C++ библиотеке. У овом примеру то су `stdio.h` и `string.h`.

1.1 `MPI_Init` и `MPI_Finalize`

Секција програма у којој ће се извршавати MPI функције започиње са `MPI_Init` функцијом. Од линије у програму, када више MPI функције нису потребне, MPI библиотека треба да се затвори позивом `MPI_Finalize` функције.

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

Функција `MPI_Init` иницијализује MPI библиотеку. Као параметре прима исто што и main функција. У овом програму је main функција дефинисана без параметара па су зато у `MPI_Init` функцију прослеђене NULL вредности.

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

Функција `MPI_Finalize` затвара MPI библиотеку. Ова функција не прима ни један параметар. Након што је MPI библотека затворена не сме бити позива MPI функција.

1.2 `MPI_Comm_size` и `MPI_Comm_rank`

У MPI библиотеци communicator је скуп процеса који могу да размењују поруке. У C/C++ овај концепт је дефинисан типом `MPI_Comm`. Један од задатака који се обављају у току иницијализације библиотеке јесте и стварање communicator у коме се налазе сви процеси тренутног програма. Овај комуникатор је означен симболом `MPI_COMM_WORLD`. Овај симбол се користи као параметар у многим функцијама.

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

Функција `MPI_Comm_size` има два парамера. Први параметар је communicator. У другом параметру се враћа укупан број процеса који се користе за извршавање програма. Тај број се у овом програмз памти у променљивој `csize`.

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

Функција `MPI_Comm_rank` има два парамера. Први параметар је communicator. У другом параметру се враћа редни број процеса. Сваки процес има јединствен број, почевши од 0 па до укупног броја процеса умањеног за 1. Јединствени број процеса се памти у промнљивој `prank`.

1.3 MPI_Send и MPI_Recv

Слање порука у библиотеци MPI је имплементирано функцијом `MPI_Send`. Ова функција је сложена.

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

Прва три параметра одређују садржај који се шаље. Параметар `buf` је показивач на блок меморије у коме се налази садржај који треба да се пошаље. Параметар `count` је број елеменара који се шаљу. Параметар `datatype` је набројиви тип који одређује врсту послатих података. Неке од могућих вредности за овај параметар су: `MPI_BYTE`, `MPI_CHAR`, `MPI_INT`, `MPI_FLOAT`...

У примеру, променљива `gret` је показивач на блок меморије са подацима. Број података који се шаљу је израчунат функцијом `strlen` (стрингови у C/C++ су терминирани симболом `\0`, зато +1). Подаци који се шаљу су карактери па стога је 3. параметар `MPI_CHAR`.

Преостала три параметра функције `MPI_Send` одређују ка ком процесу се шаље порука. Параметар `dest` је јединствени број процеса коме се шаље порука. Параметар `tag` служи као додатак на основу кога се могу разликовати иначе исте поруке. Последњи параметар је communicator.

У примеру, сви процеси осим процеса 0, шаљу поруке ка процесу 0, стога је параметар `dest` једнак 0. Параметар `tag` није од важности па му је увек прослеђена 0. Параметар за communicator је као и до сада `MPI_COMM_WORLD`.

Примање порука у библиотеци MPI је имплементирано функцијом `MPI_Recv`. Ова функција има сличе параметре као и функција `MPI_Send`.

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

Првих 6 параметара имају исто значење као и параметри у случају функције `MPI_Send`. Последњи параметар је структура у којој се налазе додатни подаци: ко је пошиљаоц, који је `tag` поруке, као и да ли је дошло до неке грешке.

```
struct MPI_Status {
    int MPI_SOURCE;
    int MPI_TAG;
    int MPI_ERROR;
};
```

Уколико се овај параметар не користи тада му се прослеђује вредност `MPI_STATUS_IGNORE`.

Да би се порука успешно послала и примила потребно је да у различитим процесима постоје упарене функције `MPI_Send` и `MPI_Recv`. Неопходно је да параметри `dest` и `source` буду исти као и `tag` параметри.

Уколико процес чека на поруку која не стиже тада је процес блокиран. Ово је веома чест узрок грешака у програмима. Уколико постоје неупарене функције `MPI_Send` и `MPI_Recv` тада скоро сигурно долази до тога да ће се програм блокирати.

1.4 Задатак

Написати програм коришћењем библиотеке MPI, где сваки процес шаље сваком другом процесу поруку од једног природног двоцифреног броја. Прва цифра је јединствени број тог процеса, а друга цифра је случајан број од 0 до 9. Сваки процес након што прими све поруке, исписује бројеве које је примио на терминалу.

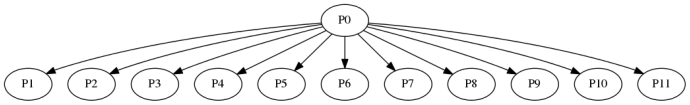
Коначни испис за случај са 6 процеса, за процеса 2, може да изгледа овако:

Process 2 received: 7 11 39 40 56

Glava 2

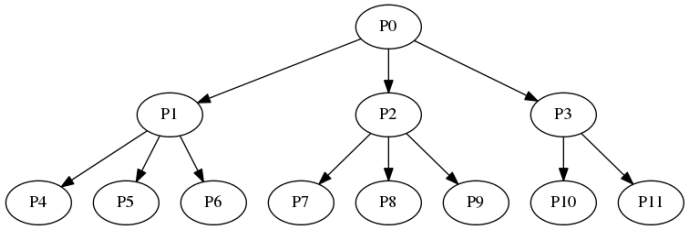
Колективна комуникација

У случајевима када је један податак потребно проследити са једног процеса до свих осталих процеса постоји више могућих решења. Једно од њих је да процес који шаље податке пошаље поруку сваком појединачном процесу директно (Слика 2.1).



Slika 2.1: Комуникација један ка свима

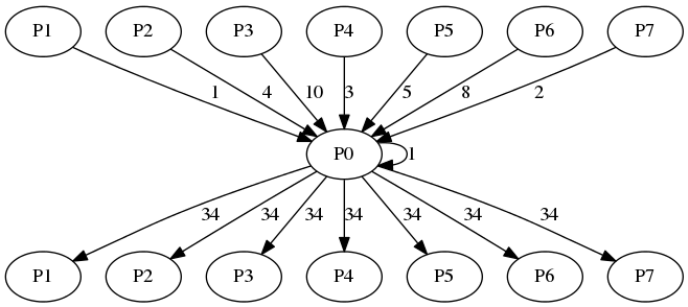
Други начин је да први процес посаље податак до посредника који затим прослеђују податак ка осталим процесима. Овај начин комуникације образује структуру стабла (Слика 2.2).



Slika 2.2: Комуникација са структуром стабла

У неким случајевим је потребно обрадити податке од којих се сваки податак налази у посебном процесу. На пример, потребно је направити суму бројева. Сваки појединии број се налази у посебном процесу. Коначна сума треба да се нађе на свим процесима на крају прорачуна.

Најједноставније решење је да сви процеси пошаљу своје податке до само једног процеса. Тај процес одради сумирање и потом врати резултат свим остали процесима (Слика 2.3).



Slika 2.3: Сумирање један ка свима

Као и код слања података и овде је могуће рачунање са структураом стабла (Слика 2.4).

У овом случају је могућа и да процеси размењују парцијалне резултате током рачунања. На тај начин се може добити структура лептира. Закључак је да од случаја до случаја и од тога колико процеса учествује у прорачуну, другачије структуре могу бити оптималне.

У оквиру MPI библиотеке ова комплексност одабира одговарајуће структуре за глобално слање података или за глобални прорачун је сакривено од кориснока. Сама имплементација библиотеке брине о детаљима. На кориснику је да одабере одговарајућу функцију из MPI библиотеке.

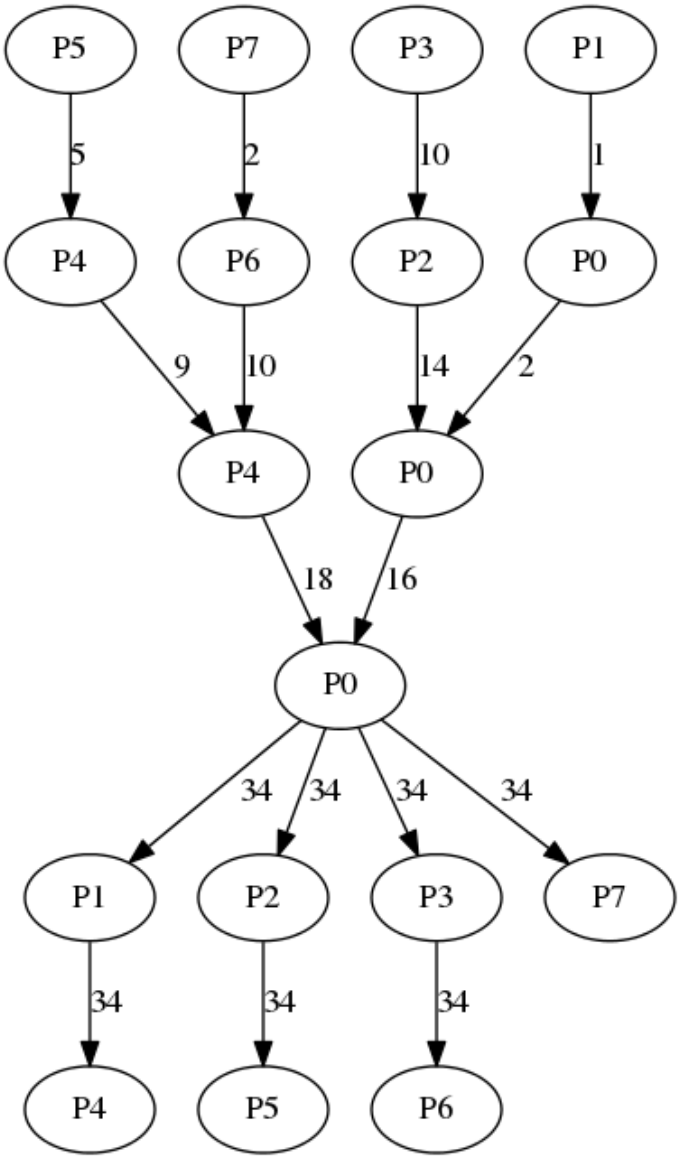
2.1 MPI_Bcast и MPI_Reduce

Наредни пример програма, који представља имплементацију сумирања првих N природних бројева, садржи употребу две функције за колективну комуникацију. Функција `MPI_Bcast` омогућава једном процесу да пошаље податак свим осталим процесима, док се функција `MPI_Reduce` користи за срачунавање колективне суме бројева.

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

double getInput()
{
        int res;
        printf("Number:␣");
        fflush(stdout);
        scanf("%d", &res);
        return double(res);
}

int main(int argc, char* argv[])
{
        double n;
        double sum = 0;
        int csize, prank;
```



Slika 2.4: Сумирање са структураом стабла

```
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &csize);
    MPI_Comm_rank(MPI_COMM_WORLD, &prank);

        if (prank == 0)
        {
                n = getInput();
        }
        //MPI_Barrier(MPI_COMM_WORLD);
        MPI_Bcast(&n, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

        double s = MPI_Wtime();
        double i = (double)prank;
        double ds = (double)csize;
        while (i <= n)
        {
                sum += i;
                i += ds;
        }
        double tsum;
        MPI_Reduce(&sum, &tsum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

        double e = MPI_Wtime();
        double d = e - s;
        double mind;
        MPI_Reduce(&d, &mind, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
        if (prank == 0)
        {
                printf("Sum␣first␣%f␣integer␣is␣%f\n", n, tsum);
                printf("Elapsed␣time:␣%f\n", d);
        }

    MPI_Finalize();

        return 0;
}
```

Процес 0 преко терминала прима податак од корисника за који број првих N природних бројева се срачунава сума. Добијени податак се потом функцијом MPI_Bcast прослеђује свим осталим процесима. Прототип ове функције је:

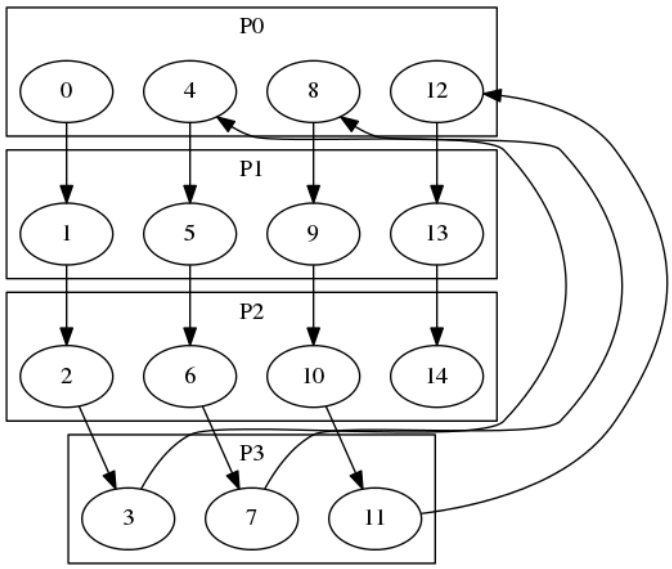
```
int MPI_Bcast(
    void *buffer,
        int count,
        MPI_Datatype datatype,
        int root,
        MPI_Comm comm
);
```

Прва три параметра ове функције су иста као параметри функције MPI_Send. То значи да се у параметар buffer смешта показивач на податке који се шаљу. Параметар count је број података типа параметра datatype који се налазе у параметру buffer. Параметар root је број процеса са кога се подаци шаљу. То значи да ће се након позива ове функције, подаци на које показује buffer процеса root бити послани ка свим осталим

процесима. У свим осталим процесима у buffer ће завршити подаци из процеса root. За root процес buffer је улазни параметар, док је за све остале излани. За параметар comm се прослеђује MPI_COMM_WORLD.

У примеру се са процеса 0, податак који се налази у променљивој n шаље ка свим осталим процесима, исто у променљиву n. Ова променљива је локална сваком процесу и након ове функције је иста на свим процесима.

У овом примеру више процеса заједно раде на истом задатку. Постоји три главна начина на која процеси могу да поделе заједнички задатак: блоковски, циклично и комбиновано. Стратегије поделе посла ће бити илустроване за сумирање бројева од 0 до 14 на 4 процеса.

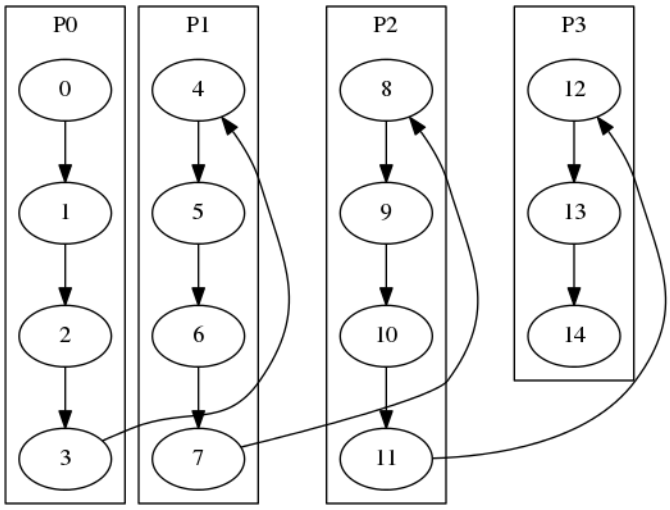


Slika 2.5: Циклична расподела послова

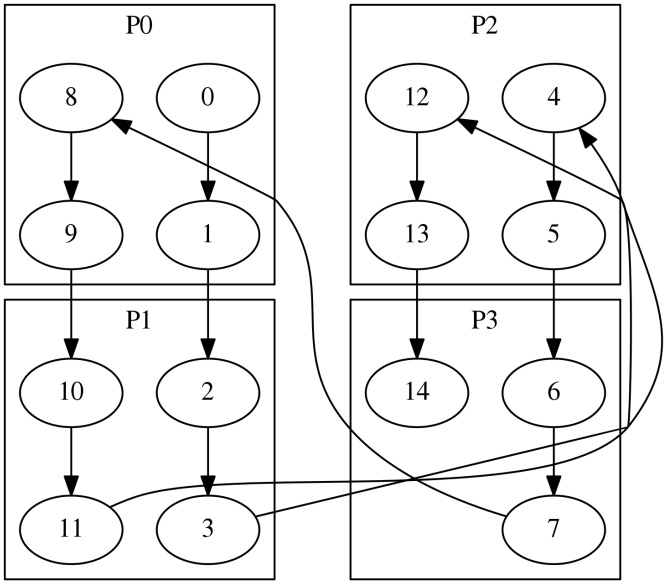
У случају да се користи циклична подела процес 0 ће сабрати бројеве 0, 4, 8 и 12. Процес 1 ће сабрати бројеве 1, 5, 9 и 13. Процес 2 ће сабрати бројеве 2, 6, 10 и 14. И на крају процес 3 ће сабрати бројеве 3, 7 и 11. Оваквом стратегијом послови у низу за обраду циклично мењају процесе на којима се обрађују (Слика 2.5).

У случају да се користила блоковска подела задатка, тада би процес 0 добио да сабере бројеве до 0 до 3, процес 1 би добио да сабере бројеве од 4 до 7, процес 2 би добио да сабере бројеве од 8 до 11 и процес 3 би добио да сабере бројеве од 12 до 14. Када се примени блоковкса расподела послова, група послова која је једна до другог у низу за обраду се распоређују на исти процес (Слика 2.6).

Кобинована подела је мешање цикличног и блоковског приступа. Код ове стратегије се прво послови распореде у мање блокове. Ти блокови се затим циклично распоређују по процесима. Ако би у овом примеру



Slika 2.6: Блоковска расподела послова



Slika 2.7: Комбинована расподела послова

блокови били величине 2 тада би бројеви 0, 1, 8 и 9 сумирали на процесу 0. Бројеви 2, 3, 10 и 11 би се сумирали на процесу 1. Бројеви 4, 5, 12 и 13 би се сумирали на процесу 2. Бројеви 5, 7 и 14 би се сумирали на процесу 3 (Слика 2.7).

У овом примеру процеси су поделили задатак циклично. Сваки од процеса рачуна свој парцијални део суме. Нако што сви процеси заврше рачунање све парцијалне суме је потребно сабрати да би се добила коначна сума. Да би се што ефикасније сабрале парцијалне суме користи се функција MPI_Reduce.

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

Функција MPI_Reduce има 7 параметара. Параметар sendbuf је показивач на меморију где се налазе подаци које треба обрадити. Параметар recvbuf је показивач на меморију где ће се сместити резултат након колективног израчунавања. Параметар count је број података типа параметра datatype који требају да се обраде и које се налазе у меморији на коју показује sendbuf. Параметар op је операција која треба да се уради над подацима. Овај параметар може да има више вредности од којих су неке: MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD... Параметар root је број процеса у чијој меморији ће се сместити резултат срачунавања. Иако сви процеси прослеђују неку вредност за параметар recvbuf, тај параметар ће се користити само за процес са јединственим бројем root. За параметар comm се прослеђује MPI_COMM_WORLD.

У примеру се све парцијалне суме сабирају и резултат се шаље на процес 0. Променљива tsum ће саджљати коначан резултат сумирања само на процесу 0. На осталим процесима ова променљива има произвољну вредност.

MPI функција MPI_Wtime враћа протекло време од почетка стартовања програма на сваком процесу. Уз помоћ ове функције могуће је мерити време потребно да се програм заврши, а на основу тога може се утврдити и убрзање које се добија извршавањем програма на више од једног процесора.

2.2 Задатак

Написати програм коришћењем библиотеке MPI за сумирање првих N природних бројева употребом блоковске стратегије.

2.3 Задатак

Написати програм коришћењем библиотеке MPI за множење два вектора. Векторе треба да генерише програм на случајан начин.

Glava 3

Множење матрице и вектора

Наредни програм илуструје имплементацију множења матрице и вектора. Ако је величина вектора n, тада је величина матрице n x n. Димензија n, за овај пример мора бити умножак броја процесора на којем ће се пример покренуту. У супоротном програм не може да ради.

Матрица и вектор се учитавају из датотека које се прослеђују програму као параметри преко командне линије. Учитавање се ради само на процесу 0. Потом се подаци прослеђују свим осталим процесима. Да би се израчунала вредност једног елемента резултата потребна је само једна врста матрице. Овде пример илуструје како се податак може дистрибуирати по процесима.

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

int returnSize(char* fname)
{
        FILE* f = fopen(fname, "r");
        int dim = 0;
        double tmp;
        while(fscanf(f, "%lf", &tmp) != EOF)
                dim++;
        fclose(f);
        return dim;
}

double* loadVec(char* fname, int n)
{
        FILE* f = fopen(fname, "r");
        double* res = new double[n];
        double* it = res;
        while(fscanf(f, "%lf", it++) != EOF);
        fclose(f);
        return res;
}

double* loadMat(char* fname, int n)
{
        FILE* f = fopen(fname, "r");
```

```
        double* res = new double[n*n];
        double* it = res;
        while(fscanf(f, "%lf", it++) != EOF);
        fclose(f);
        return res;
}

void logRes(const char* fname, double* res, int n)
{
        FILE* f = fopen(fname, "w");
        for (int i = 0; i != n; ++i)
                fprintf(f, "%lf␣", res[i]);
        fclose(f);
}

int main(int argc, char* argv[])
{
        int csize;
        int prank;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &csize);
        MPI_Comm_rank(MPI_COMM_WORLD, &prank);

        char* vfname = argv[1];
        char* mfname = argv[2];
        int dim;
        double* mat;
        double* vec;
        double* tmat;
        double* lres;
        double* res;

        if (prank == 0)
                dim = returnSize(vfname);

        MPI_Bcast(&dim, 1, MPI_INT, 0, MPI_COMM_WORLD);

        if (prank == 0)
                vec = loadVec(vfname, dim);
        else
                vec = new double[dim];

        MPI_Bcast(vec, dim, MPI_DOUBLE, 0, MPI_COMM_WORLD);

        if (prank == 0)
                tmat = loadMat(mfname, dim);

        int msize = dim * dim / csize;
        mat = new double[msize];
        MPI_Scatter(tmat, msize, MPI_DOUBLE,
                                mat, msize, MPI_DOUBLE,
                                0, MPI_COMM_WORLD);

        int to = dim / csize;
        lres = new double[to];
        for (int i = 0; i != to; ++i)
```

```
        {
                double s = 0;
                for (int j = 0; j != dim; ++j)
                        s += mat[i*dim+j] * vec[j];
                lres[i] = s;
        }

        if (prank == 0)
                res = new double[dim];

        MPI_Gather(lres, to, MPI_DOUBLE,
                           res, to, MPI_DOUBLE,
                           0, MPI_COMM_WORLD);

        if (prank == 0) {
                logRes("res.txt", res, dim);
        }

        if (prank == 0)
        {
                delete [] tmat;
                delete [] res;
        }
        delete [] vec;
        delete [] mat;
        delete [] lres;

        MPI_Finalize();

        return 0;
}
```

Подаци у датотекама су релани бројеви раздвојени белим симболима. Примери улазних датотека за вектор и матрицу су излистани.

```
2 2 2 2 4 4 4 4 1 1 1 1 3 3 3 3


2 0 0 0 1 1 1 1 2 0 0 0 1 1 1 1
0 2 0 0 1 1 1 1 0 2 0 0 1 1 1 1
0 0 2 0 1 1 1 1 0 0 2 0 1 1 1 1
0 0 0 2 1 1 1 1 0 0 0 2 1 1 1 1
1 1 1 1 2 0 0 0 1 1 1 1 2 0 0 0
1 1 1 1 0 2 0 0 1 1 1 1 0 2 0 0
1 1 1 1 0 0 2 0 1 1 1 1 0 0 2 0
1 1 1 1 0 0 0 2 1 1 1 1 0 0 0 2
2 0 0 0 1 1 1 1 2 0 0 0 1 1 1 1
0 2 0 0 1 1 1 1 0 2 0 0 1 1 1 1
0 0 2 0 1 1 1 1 0 0 2 0 1 1 1 1
0 0 0 2 1 1 1 1 0 0 0 2 1 1 1 1
1 1 1 1 2 0 0 0 1 1 1 1 2 0 0 0
1 1 1 1 0 2 0 0 1 1 1 1 0 2 0 0
1 1 1 1 0 0 2 0 1 1 1 1 0 0 2 0
1 1 1 1 0 0 0 2 1 1 1 1 0 0 0 2
```

Функција returnSize враћа велину n. Функција loadVec учитава вектор из датотеке. Слично, loadMat учитава матрицу из датотеке. Коначно, резултат множења се смешта у датотеку функцијом logRes. У свим овим

функцијама датотекама се манипулише стандардним ANSI C функцијама укљученим у програм библиотеком stdio.h.

Подаци се учитавају само на процесу 0. Димензија n и вектор vec се прослеђују функцијом MPI_Bcast до осталих процеса. Матрица се учита цела на процесу 0. Делови матрице се потом прослеђују до осталих процеса функцијом MPI_Scatter. Матрица је у овом примеру смештена у један низ. У том низу је прво стављена врста 1, потом врста 2 итд.

```
int MPI_Scatter(
        void *sendbuf, int sendcount, MPI_Datatype sendtype,
        void *recvbuf, int recvcount, MPI_Datatype recvtype,
        int root, MPI_Comm comm
);
```

Параметар sednbuf је бафер са свим подацима који се шаљу са процеса root. Параметар sendcount је број података типа sendtype који се шаљу. Слично важи и за парамере recvbuf, recvcount и recvtype, само се они користе на процесима који примају податке.

У примеру, учитана матрица се налази у променљивој tmat. Меморијски простор на који показује ова променљива је валидан и резервисан само на процесу 0. Делови матрице које ће обрађивати појединачни процеси се налазе у меморији на коју показује променљива mat.

На исти начин са којим се манипулише са функцијом MPI_Scatter се манипулише и са функцијом MPI_Gather само се подаци скушљају на један од процеса.

```
int MPI_Gather(
        void *sendbuf, int sendcount, MPI_Datatype sendtype,
        void *recvbuf, int recvcount, MPI_Datatype recvtype,
        int root, MPI_Comm comm
);
```

3.1 Задатак

Уопштити пример множења матрице и вектора тако да могу да се помноже матрица и вектор произвољне величине n. Другим речима, треба уклонити ограничење да n мора да буде умножак броја процеса на којима се програм извршава.

Glava 4

OpenMP основни концепти

4.1 Компајлирање

Сви примери који ће илустровати начин рада са библиотеком OpenMP ће бити намењени за компајлирање уз помоћ GCC C++ компајлера. Да би се комапјлирала C++ датотека и повезала са OpenMP библиотеком могуће је употребити следећу команду:

```
g++ -fopenmp -Wall -o exe file.cpp
```

Компајлер GCC C++ се позива командом g++. Параметар -fopenmp повезује бинарну датотеку намењену извршавању bin са библиотеком OpenMP. Параметар -Wall подешава компајлер тако да исписује сва упозорења током компајлирања и повезивања. Параметар file.cpp је C++ датотека која се компајлира.

4.2 Директива `parallel` и функције `omp_get_thread_num`, `omp_get_num_threads`

OpenMP је библиотека функција намењена писању програма за извршњавање на више процесора који деле заједничку меморију. OpenMP се састоји од функција и компајлерских директива. Намењена је за коришћење са програмским језиком C/C++. Део кода који се извршава на више процесора који деле заједничку меморију се назива нит.

Једноставан програм који исписује ‘‘поздраве’’ са више процесора на терминал ће бити програм за упознавање са основним функцијама и директивама из OpenMP библиотеке.

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

int main(int argc, char* argv[])
{
    #pragma omp parallel // num_threads(3)
        {
                int trank = omp_get_thread_num();
```

```
                int tc = omp_get_num_threads();
                printf("Hello␣from␣thread␣%d␣of␣%d\n", trank, tc);
        }
        return 0;
}
```

Da bi se dobio pristup OpenMP funkcijama potrebno je uključiti datoteku omp.h. Izlistani primer koristi jednu direktivu i dve funkcije iz OpenMP biblioteke. Direktive za kompajler iz OpenMP biblioteke započinju sa #pragma omp. U primeru se koristi direktiva parallel koja označava da se naredni blok komandi treba da izvrši paralelno na više procesora. Dodatni parametar za ovu direktivu, num_threads, je zakomentarisan. U tom slučaju će se program izvršiti sa onim brojem nити kolики je broj procesora u sistemu. Ukolико bi se ovaj parametar uključio u primetu, tada bi se program izvršio korišćenjem 3 nити.

Dve funkcije koje su upotrebljene u primeru su omp_get_thread_num i omp_get_num_threads. Funkcija omp_get_threads vraća ukupan broj nити koji se koriste trenutno u programu. Tokom rada programa broj nити se može mењati. Funkcija omp_get_thread_num vraća jedinstveni broj nити. Jedinstveni brojevi nити se nalaze u opsegu od 0 do ukupnog broja nити umanjenog za 1.

4.3 Direktiva barrier

Direktiva barrier se koristi da bi se nити sinhronizovale. Ukolико prilikom izvršavanja nit naiđe na ovu direktivu, tada ta nit čekda da i sve ostale nити dosegnu tu istu tačku u izvršavanju programa.

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

int main(int argc, char* argv[])
{
    #pragma omp parallel
    {
        printf("Printf␣1␣of␣%d␣thread\n", omp_get_thread_num());
        # pragma omp barrier
        printf("Printf␣2␣of␣%d␣thread\n", omp_get_thread_num());
    }

    return 0;
}
```

U izlistanom primeru, postoje dva ispisa na standardni izlaz koje će svaka nit da uradi. Zavaљujући direktivi barrier prvo će biti izvršeni 1. ispisi svih nити. To će se desiti stoga kada neka nit uradi 1. ispis ona će čekati ostale nити na direktivi barrier pa će tek potom nastaviti dalje sa izvršavanjem. Ukolико bi se ova direktiva uklonila, tada bi neke nити uradile ispis 2. pre nego što bi neke druge nити izvršile ispis 1.

4.4 Директиве parallel for и reduction

Директива parallel for се користи за паралелизацију for петљи. Следећи пример сумира првих n природних бројева.

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

int main(int argc, char* argv[])
{
    int tc = strtol(argv[1], NULL, 10);
    double n;
    double sum = 0;

    printf("Number:␣");
    scanf("%lf", &n);

        double s = omp_get_wtime();
    #pragma omp parallel for num_threads(tc) reduction(+: sum)
    for (int i = 1; i <= (int)n; i++)
        sum += (double)i;

        s = omp_get_wtime() - s;
    printf("\nSum␣is␣%lf\n", sum);
        printf("Executed␣for␣%lf␣s\n", s);
    return 0;
}
```

У овом примеру као први параметар програма се прослеђује број нити који ће се користити за извршавање програма. Број нити се затим прослеђују директиви parallel for помоћу параметра num_threads.

Варијабла sum се помоћу директиве reduction проглашава дељеном вариаблом на коју ће све нити надодавати вредност (због знака +).

Директива parallel for се може користити само на специјалним for петљама. Нека од ограничења постављених на петљама су:

- Вариабла за итерацију мора бити целобројна.
- Петља не сме да има break.

Петље код којих су итерације међусобно зависне се могу имплементирати директивом parallel for али се може добити погрешан резултат.

Функција omp_get_wtime се користи у програмима за мерење времена извршавања програма.

4.5 Задатак

Написати програм за сумирање првих n природних бројева тако што свака нит прво рачуна своју парцијалну суму. Потом се коначна сума рачуна сумирањем тако добијених парцијалних сума. Упоредити време извршавања такве имплементације са решеним примером.

Glava 5

OpenMP напредни концепти

5.1 Стратегије дељења посла

Постоји више стратегија како се посао може поделити у блоку кода који се паралелно извршава на више нити. Уколико се стратегија не наведе користи се подразумевана стратегија. У већини имплементација OpenMP библиотеке подразумевана стратегија је блоковска подела.

Стратегија се може променити кључном речју schedule у parallel директиви. У следећем коду је експлицитно речено која стратегија треба да се користи.

```
sum = 0;
#pragma omp parallel for reduction(+:sum) schedule(static, 1)
for (int i = 0; i != n; ++i)
{
    sum += i;
}
```

```
schedule(<type> [, chunksize])
```

Тип стратегије type може бити:

- static
- dynamic
- guided
- auto
- runtime

Када се користи static стратегија тада се посао дели пре саме петље. Свакој нити се додељује број итерација chunksize и тако у круг. Уколио је chunksize једнак 1 добија се циклична подела посла. Уколико је chunksize највећи број тако да свака нит добије отприлике исто посла, тада се добија блоковкса подела. За све остале вредности добија се комбинована подела.

Стратегија dynamic исто дели посао на мање делове величине chunksize али се додела послова не ради унапред већ се послови додељују нитима у току рада система.

Стратегија guided додљује послове нитима у току рада система. Величина делова је у старту већа а како се преостали део посла смањује и сами делови посла се смањуjу. Параметар chunksize нема утицај.

Стратегија runtime нема одређену стратегију док се програм не стартује. Када се програм стартује варијабла окружења OMP_SCHEDULE одређује која ће се стратегија користити.

5.2 Задатак

Одличан пример који илуструје колико коришћена стратегија може да утиче на брзину извшавања програма је алгоритам за тражење простих бројева Ератостеново сито. Потребно је релизовати имплементацију алгоритма коришћењем библиотеке OpenMP. Улаз у програм је број до ког је потребно тражити просте бројеве. Узлаз из програма су сви прости бројеви до унесеног простог броја исписани у неки фајл. Упоредити времена извршавања за различите стратегије.