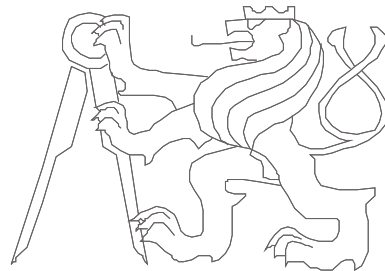


Pokročilé architektury počítačů

03

Programování paralelních systémů – Část II.

OpenMP a MPI



České vysoké učení technické, fakulta elektrotechnická

Přehled: OpenMP vs. MPI

OpenMP

(Open Multi-Processing)

- jenom pro SMS
- jednoduchá paralelizace (pomocí direktiv)
- implicitní komunikace

MPI

(Message Passing Interface)

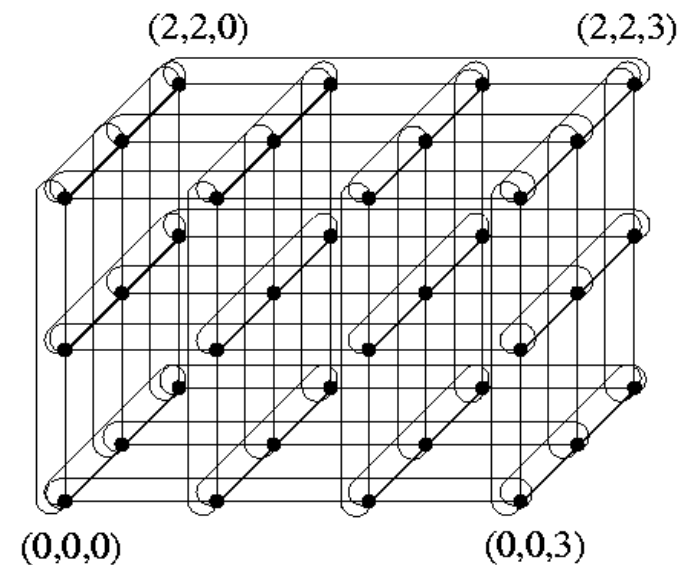
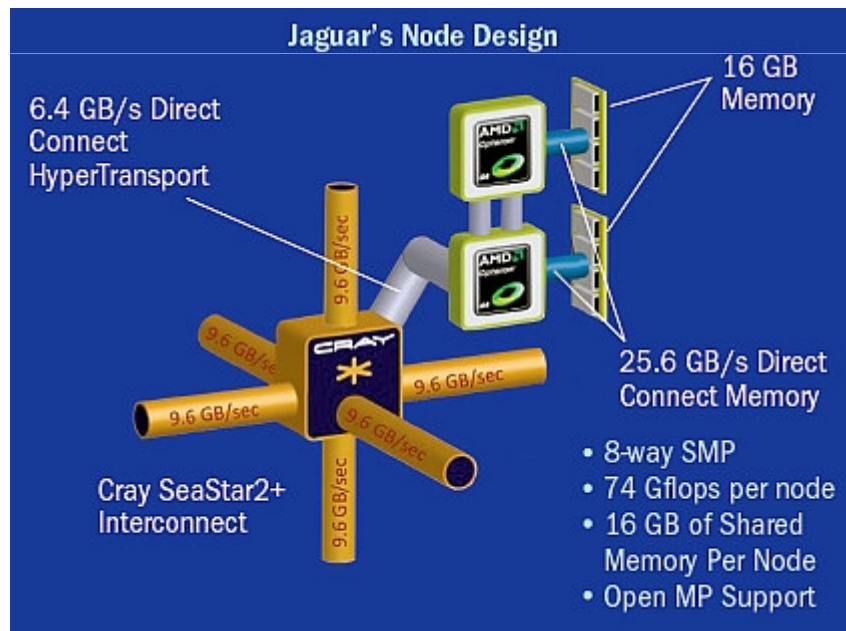
- jak pro SMS tak DMS
- náročné pro programátora (paralelizace, debugging)
- explicitní komunikace

Hybridní přístup: OpenMP + MPI

- současný softwarový trend v paralelním počítání
- MPI napříč uzly clusteru, OpenMP uvnitř uzlu (efektivní využití prostředků; potlačení komunikačního přetížení uvnitř uzlu; dynamické vyvažování zátěže)
- je možné dosáhnout lepší speedup než při použití jenom OpenMP nebo jenom MPI

MPP – Cray XT5-HE

- $\approx 37\,000$ výpočetních uzlů (224 162 VE celkem), servisní uzly pro I/O
- Jeden uzel – dva 6-jádrové procesory – oba přístup do sdílené paměti
- Každý uzel – 25.6 GB/s do lokální paměti (XT6: 85.3 GB/sec), 6.4GB/s do sítě,
- Uzly jsou pospojovány do 3D toroidu
- 12 OpenMP nebo MPI úloh v uzlu, MPI mezi uzly

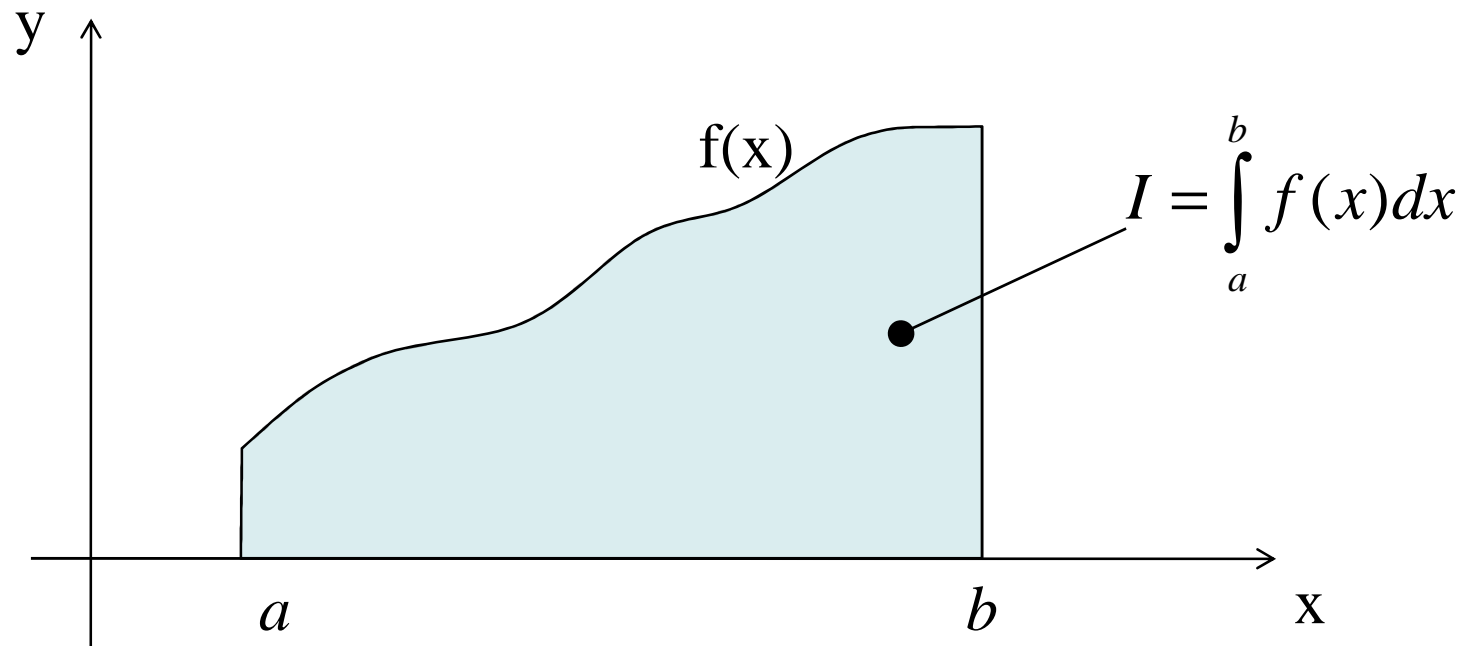


Cluster

- poskytuje výpočetní výkon spojením vícero počítačů (obvykle střední cenové kategorie)
- patří sem 85 % systémů z dnes 500 nejvýkonnějších
- výhodou je nižší cena, dobrá škálovatelnost



Příklad – výpočet určitého integrálu



Příklad – výpočet určitého integrálu

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

```
double integral(double a, double b, double (*f)(double), unsigned long int N)
```

```
{
```

```
    double sum=0, dx=(b-a)/N, x=a+dx;
```

```
    long int i;
```

```
    for(i=1; i<N; i++ )
```

```
        sum += f(x+i*dx);
```

```
    sum += (f(a)+f(b))/2;
```

```
    return sum*dx;
```

```
}
```

```
double polynom(double x) { return 2*x+1; }
```

```
void main() {
```

```
    long int N=5e8;
```

```
    double res, a=4, b=5;
```

```
    res = integral(a, b, polynom, N);
```

```
    printf("I = %lf ",res);
```

```
}
```

Příklad – výpočet určitého integrálu

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

```
double integral(double a, double b, double (*f)(double), unsigned long int N)
{
    double sum=0, dx=(b-a)/N, x=a+dx;
    long int i;
    #pragma omp parallel for shared(x,N,dx,f) private(i) reduction(+:sum)
    for(i=1; i<N; i++ )
        sum += f(x+i*dx);
    sum += (f(a)+f(b))/2;
    return sum*dx;
}
```

```
double polynom(double x) { return 2*x+1; }
```

Speedup: 1,85 !!!

```
void main() {
    long int N=5e8;
    double res, a=4, b=5;
    res = integral(a, b, polynom, N);
    printf("I = %lf ",res);
}
```

OpenMP - přehled

- vše začíná: **#pragma omp directive [clause list]**
- program běží sekvenčně pokud nenarazí na *parallel* direktivu
- *clause list* specifikuje podmínky paralelizace, počet vláken, přístup k datům a následnou manipulaci s nimi, rozčleňování..
 - podmíněná paralelizace: **if(scalar expression)**
 - stupeň paralelizace: **num_threads(integer expression)**
 - přístup k datům: **shared(variable list), private(variable list), firstprivate(variable list)**
 - sběr privátních dat na konci paralelního vykonávání: **reduction(operator: variable list)**, možné operátory: +, *, -, &, |, ^, &&, ||
 - rozčleňování a rozvrhování: **schedule(scheduling_class[, parameter])**, přičemž je možné volit z: static, dynamic, guided, a runtime

OpenMP – paralelizace *for* cyklů

```
#pragma omp parallel default (private) shared (n)
{
    #pragma omp for
    for (int i = 0; i < n; i++) {
        /* tělo cyklu*/
    }
}
```

nebo zkráceně:

```
#pragma omp parallel for default (private) shared (n)
    for (int i = 0; i < n; i++) {
        /* tělo cyklu*/
    }
```

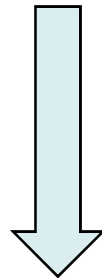
OpenMP – paralelizace *for* cyklů

```
#pragma omp parallel default (private) shared (n)
{
    #pragma omp for
    for (i = 1; i < n; i++) {
        a[i] = i + a[i-1];
    }
}
```

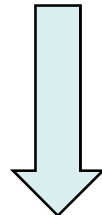
?

OpenMP – každé vlákno dělá něco jiného

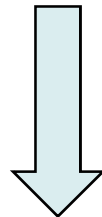
sekvenční vykonávání...



`taskA();`



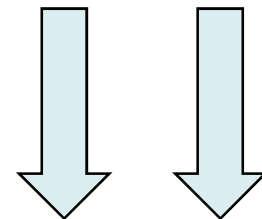
`taskB();`



OpenMP – každé vlákno dělá něco jiného

```
1  #pragma omp parallel
2  {
3      #pragma omp sections
4      {
5          #pragma omp section
6          {
7              taskA();
8          }
9          #pragma omp section
10         {
11             taskB();
12         }
13     }
14 }
```

paralelní
vykonávání...



taskA taskB

OpenMP – každé vlákno dělá něco jiného

nebo zkráceně:

```
1      #pragma omp parallel sections
2      {
3          #pragma omp section
4          {
5              taskA();
6          }
7          #pragma omp section
8          {
9              taskB();
10         }
11     }
```

OpenMP – co dále umožňuje?

- vnořený paralelizmus (OMP_NESTED nastavit na TRUE)
- explicitně specifikovat bariéry (bod synchronizace)
- označit kód, který má vykonávat pouze jedno vlákno (buď libovolné nebo master)
- označit kritickou sekci (může vstoupit a vykonávat pouze jedno vlákno v daném čase; ostatní buď čekají, nebo jsou již za touto sekcí)
- uspořádané vykonávání (jako při sériovém vykonávání)
- vynucení zápisu do paměti (aktualizace proměnných..) – flush; kdy se flush použije implicitně? A flush is implied at a barrier, at the entry and exit of critical, ordered, parallel, parallel for, and parallel sections blocks and at the exit of for, sections, and single blocks. A flush is not implied if a nowait clause is present. It is also not implied at the entry of for, sections, and single blocks and at entry or exit of a master block.

OpenMP – co dále poskytuje?

- `void omp_set_num_threads (int num_threads);`
- `int omp_get_num_threads ();`
- `int omp_get_max_threads ();`
- `int omp_get_thread_num ();`
- `int omp_get_num_procs ();`
- `int omp_in_parallel();`

- `void omp_init_lock (omp_lock_t *lock);`
- `void omp_destroy_lock (omp_lock_t *lock);`
- `void omp_set_lock (omp_lock_t *lock);`
- `void omp_unset_lock (omp_lock_t *lock);`
- `int omp_test_lock (omp_lock_t *lock);`

A mnoho jiných funkcí...

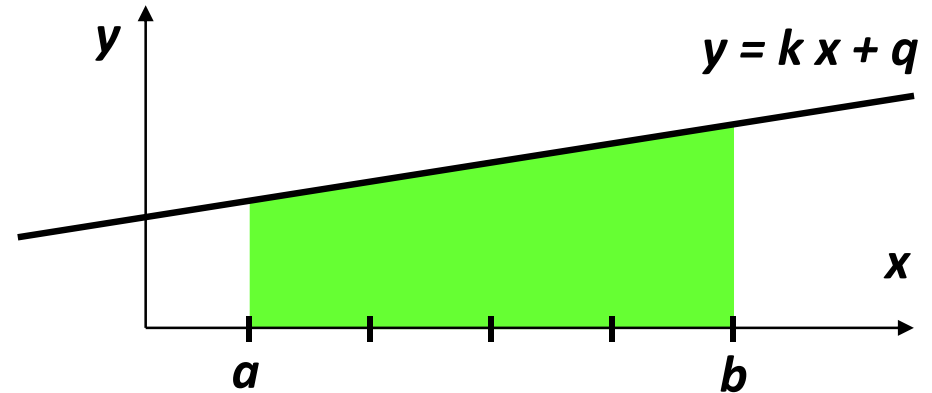
Příklad – výpočet určitého integrálu jinak

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

```
double integral(double a, double b, double (*f)(double), long N);
```

```
double polynom(double x) {  
    return 2*x+1;  
}
```

```
void main() {  
    long int N=5e8;  
    double res, a=4, b=5;
```



```
    res = integral(a, b, polynom, N);
```

```
    printf("I = %lf ", res);
```

```
}
```

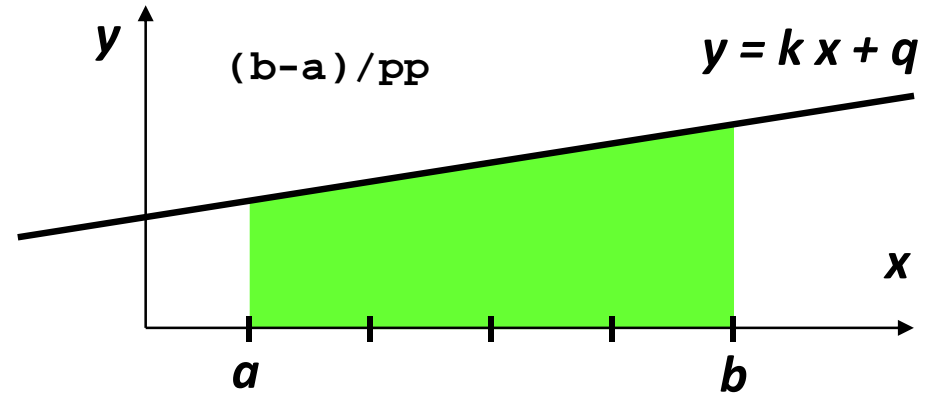

Příklad – výpočet určitého integrálu jinak

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

```
double integral(double a, double b, double (*f)(double), long N);
```

```
double polynom(double x) {  
    return 2*x+1;  
}
```

```
void main() {  
    long int N=5e8;  
    double res, a=4, b=5;  
    int i=0, pp=1;
```



```
res = integral(a+((b-a)/pp)*i, a+((b-a)/pp)*(i+1), polynom, N/pp);
```

```
printf("I = %lf ",res);
```

```
}
```

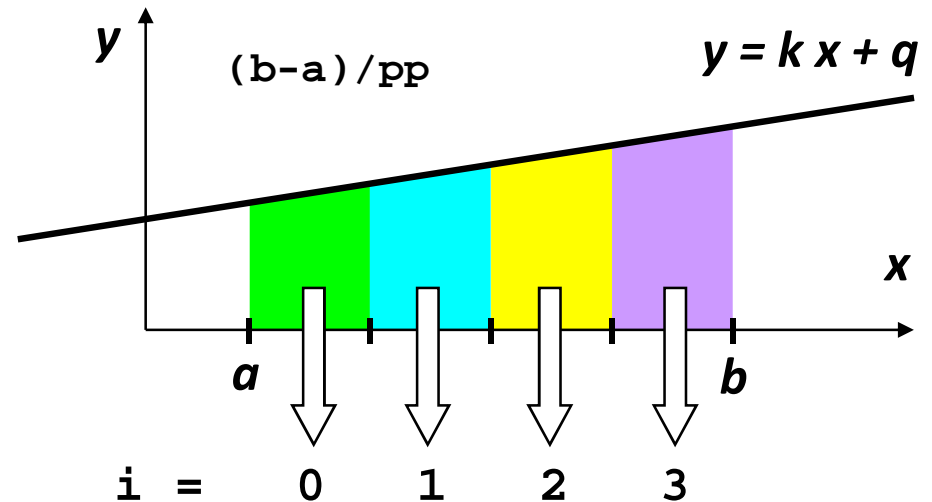
Příklad – výpočet určitého integrálu jinak

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

```
double integral(double a, double b, double (*f)(double), long N);
```

```
double polynom(double x) {  
    return 2*x+1;  
}
```

```
void main() {  
    long int N=5e8;  
    double res, a=4, b=5;  
    int i=???, pp=4;
```



```
    res = integral(a+((b-a)/pp)*i, a+((b-a)/pp)*(i+1), polynom, N/pp);
```

```
    printf("I = %lf ",res);
```

```
}
```

Příklad – výpočet určitého integrálu jinak

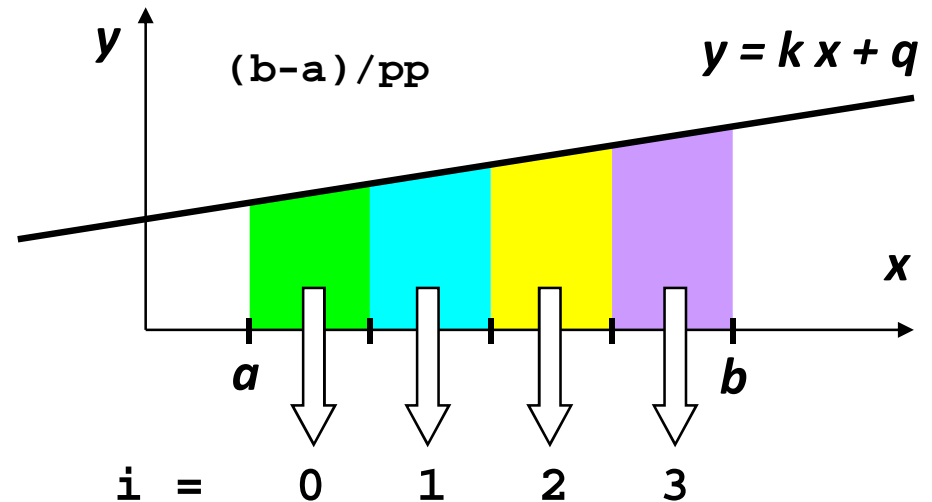
```
#include <stdio.h>
```

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

```
double integral(double a, double b, double (*f)(double), long N);
```

```
double polynom(double x) {  
    return 2*x+1;  
}
```

```
void main() {  
    long int N=5e8;  
    double res, a=4, b=5;  
    int i, pp=4;
```



```
#pragma omp parallel private(i) reduction(+:res) num_threads(pp)
```

```
{
```

```
    i = omp_get_thread_num();
```

```
    res = integral(a+((b-a)/pp)*i, a+((b-a)/pp)*(i+1), polynom, N/pp);
```

```
}
```

```
    printf("I = %lf ", res);
```

```
}
```

Příklad – výpočet určitého integrálu jinak

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

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

```
double integral(double a, double b, double (*f)(double), long N);
```

```
double polynom(double x) {  
    return 2*x+1;  
}
```

```
void main() {  
    long int N=5e8;  
    double res, a=4, b=5;
```

```
    int i, pp;
```

```
    pp = omp_get_num_procs();
```

```
    #pragma omp parallel private(i) reduction(+:res) num_threads(pp)
```

```
    {
```

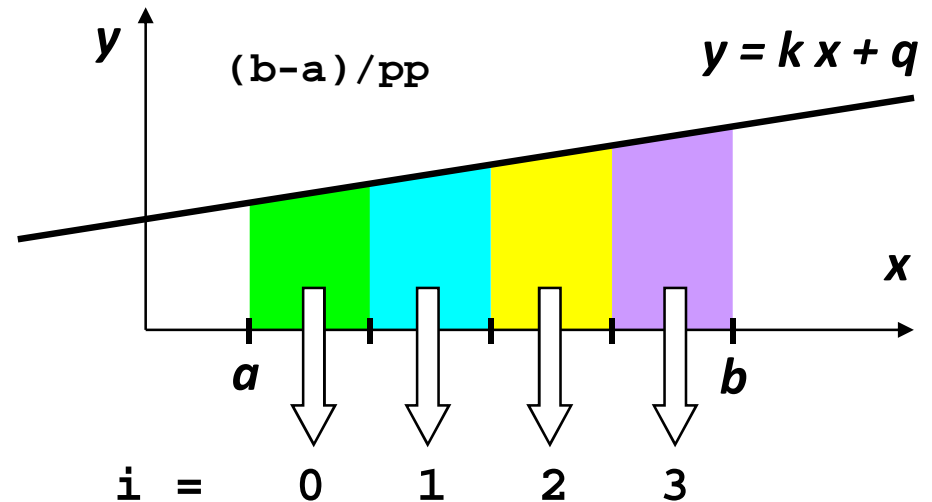
```
        i = omp_get_thread_num();
```

```
        res = integral(a+((b-a)/pp)*i, a+((b-a)/pp)*(i+1), polynom, N/pp);
```

```
    }
```

```
    printf("I = %lf ",res);
```

```
}
```



OpenMP – další příklad

```
#include <omp.h>
int a, b, i, tid;
#pragma omp threadprivate(a)

main () {
    omp_set_dynamic(0); /* Explicitly turn off dynamic threads */

    #pragma omp parallel private(b,tid)
    {
        tid = omp_get_thread_num();
        a = b = tid;
        printf("Thread %d:  a,b,%d %d \n",tid,a,b);
    } /* end of parallel section */

    #pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num();
        printf("Thread %d:  a,b,%d %d \n",tid,a,b);
    } /* end of parallel section */
}
```

MPI – Message Passing Interface

komunikace je **explicitní**, proto:

- většina MPI funkcí požaduje komunikátor jako argument,
- **komunikátor** je objekt definující skupinu procesů, které mohou vzájemně mezi sebou komunikovat,
- komunikátor `MPI_COMM_WORLD` zahrnuje všechny procesy,
- uvnitř komunikátoru pak mají všechny procesy přiděleno své unikátní identifikační číslo – `RANK`,
- komunikace je buď kolektivní (zúčastňují se všechny procesy komunikátoru), nebo point-to-point

MPI – Message Passing Interface

kolektivní komunikace:

- je vždy blokující,
- používá předdefinované datové typy MPI (MPI_CHAR, MPI_INT, MPI_LONG,...)
- se dále dělí na:
 - synchronizaci (procesy se počkají v daném bodě)
 - kolektivní výpočet (redukci) – kdy jeden proces sesbírá data od všech ostatních a vykoná nad nimi zadanou operaci
 - přesuny dat (Broadcast, Scatter, Gather, All-to-All)

MPI – téměř minimální program

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

void main(int argc, char *argv[]) {
    int numprocs, rank, namelen;
    char proc_name[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(proc_name, &namelen);

    printf("Process %d on %s out of %d\n", rank, proc_name,
        numprocs);

    MPI_Finalize();
}
```


Příklad – výpočet určitého integrálu ještě jinak

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

```
#include "mpi.h"
```

```
inline double f(double x) { return 2*x+1; }
```

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

```
{
```

```
    int myid, numprocs, volba=0;
```

```
    unsigned long int i, n;
```

```
    double startwtime, endwtime, a = 4.0, b = 5.0;
```

```
    double total, integral = 0.0;
```

```
    MPI_Init(&argc, &argv);
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs); // zjistí počet procesů
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &myid); // zjistí číslo procesu
```

Příklad – výpočet určitého integrálu ještě jinak

```
do {
    if (myid == 0) {
        printf("\n Zadej pocet intervalu (0 pro ukonceni): "); fflush(stdout);
        scanf("%ld", &n);
        if(n==0) break;
        startwtime = MPI_Wtime();
    }

    // pošle proměnou n, délky 1, typu int, z procesu #0 ostatním
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Příklad – výpočet určitého integrálu ještě jinak

```
if (n != 0) {  
    double h = (b - a) / n;  
    double i1 = myid*(n/ numprocs);  
    double i2 = (myid+1)*(n/numprocs);  
  
    integral= ( f(a+i1*h) + f(a+i2*h) ) / 2;  
  
    for( i=i1+1 ; i<i2 ; i++ )  
        integral += f(a+i*h);  
}  
  
// proměnou integral ze všech procesů převede do total procesu 0,  
// délka 1, typ double, operace sčítání  
MPI_Reduce(&integral, &total, 1, MPI_DOUBLE, MPI_SUM, 0,  
MPI_COMM_WORLD);
```

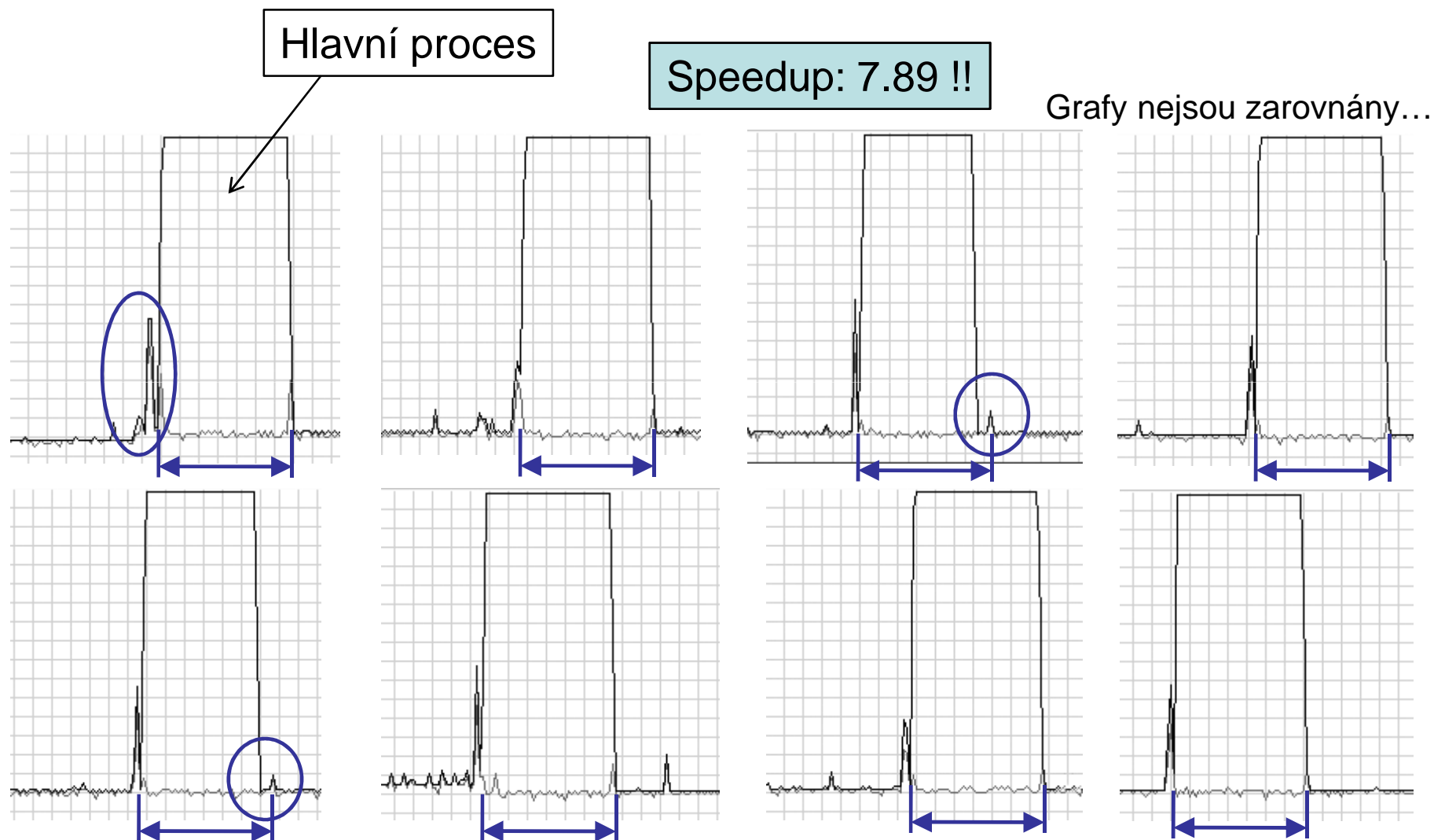
Příklad – výpočet určitého integrálu ještě jinak

```
if (myid == 0) {  
    endwtime = MPI_Wtime();  
    printf("I= %f\n", total);  
    printf("trvani vypoctu: %f s\n", endwtime - startwtime); fflush(stdout);  
}  
} while (n != 0);
```

```
MPI_Finalize();  
return 0;  
}
```

Příklad – výpočet určitého integrálu ještě jinak

Výpočet na osmi jedno-jádrových procesorech:



Kombinace OpenMP a MPI – téměř minimální program

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

int main(int argc, char *argv[]) {
    int numprocs, rank, namelen, iam = 0, np;
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(processor_name, &namelen);

    #pragma omp parallel default(shared) private(iam, np)
    {
        np = omp_get_num_threads();
        iam = omp_get_thread_num();
        printf("Hello from thread %d out of %d from process %d out of %d on\n", iam, np, rank, numprocs, processor_name);
    }

    MPI_Finalize();
    return 0;
}
```