# Advanced Computer Architectures

## Parallel systems programming – Part II.

## OpenMP a MPI

Czech Technical University in Prague, Faculty of Electrical Engineering
Slides authors: Michal Štepanovský, update Pavel Píša

## Overview: OpenMP vs. MPI

**OpenMP**

(Open Multi-Processing)

- Only for SMS
- Simple parallelization (using directives)
- Implicit communication

**MPI**

(Message Passing Interface)

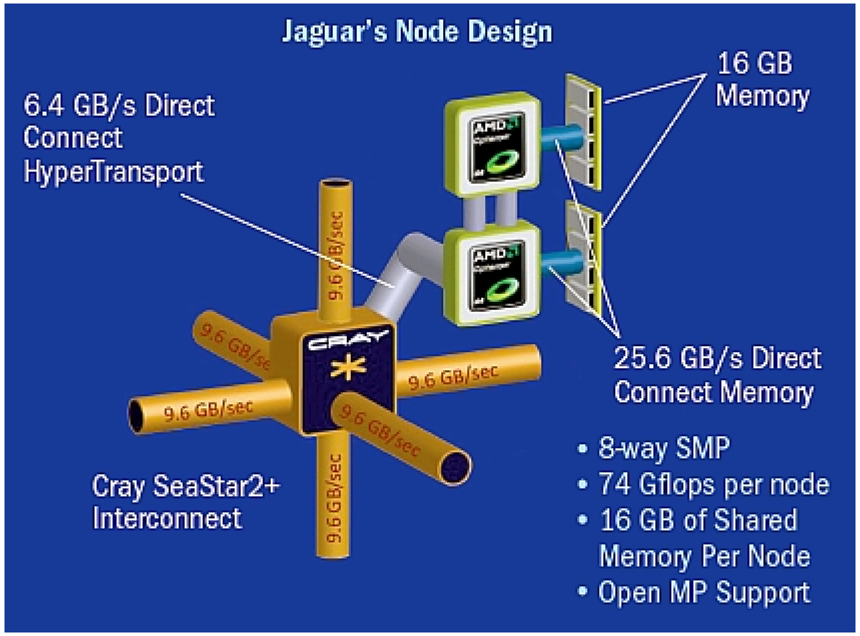
- For both SMS and DMS
- Demanding for programmer (parallelization, debugging)
- Explicit communication

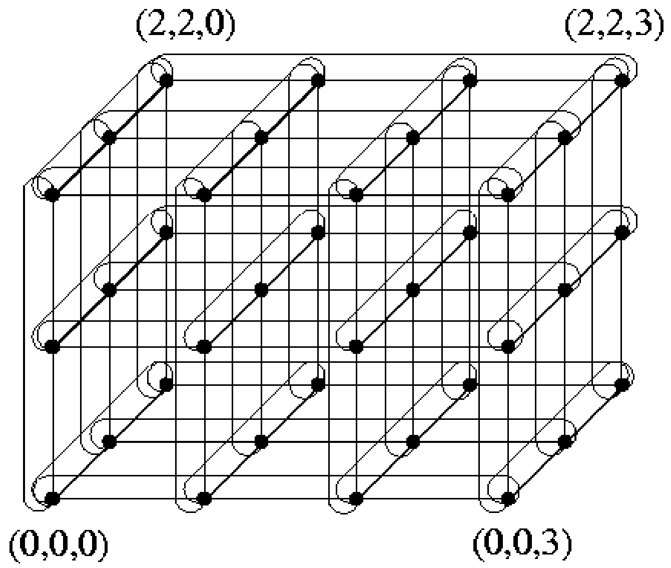
**Hybrid approach: OpenMP + MPI**

- the current software trend in parallel computing
- MPI across cluster nodes, OpenMP inside node (resource efficient, suppressing intra-node communication overload, dynamic load balancing)
- it is possible to achieve higher speedup than when using only OpenMP or MPI

# MPP – Cray XT5-HE

- ≈ 37 000 computation nodes (224 162 PE), service nodes for I/O
- One node – two 6-core processors – both have access to shared memory
- Each node – 25.6 GB/s into local memory (XT6: 85.3 GB/sec ), 6.4GB/s into network,
- Nodes are interconnected into 3D toroid
- 12 OpenMP or MPI tasks on node simultaneously, MPI between nodes

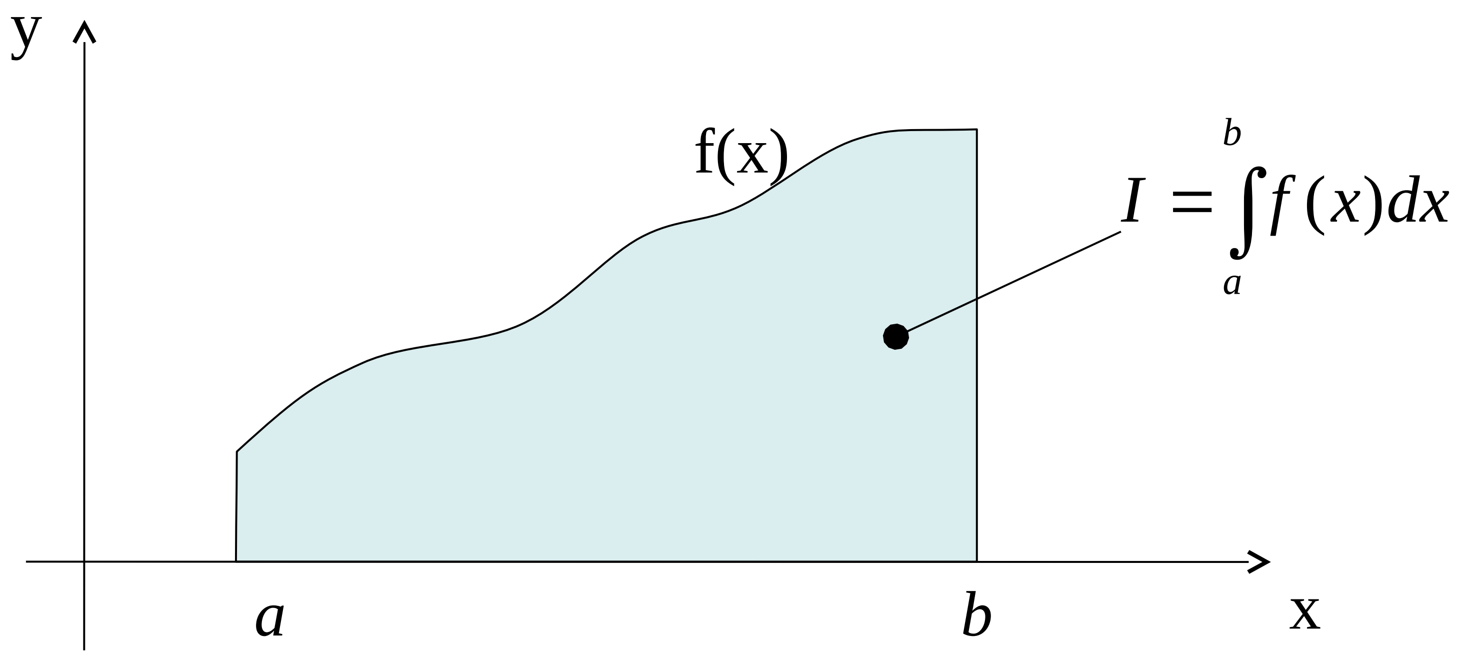




## Cluster

- provides computing power by connecting multiple computers (usually a medium price category)
- used by 85% of today's 500 most powerful systems
- the advantage is lower price, good scalability

## Example – calculation of a definite integral



$$I = \int_a^b f(x)dx$$

## Example – calculation of a definite integral

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

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

## Example – calculation of a definite integral – OpenMP simple

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

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

Speedup: 1,85 !!!

# OpenMP – overview

- All starts with: **#pragma omp directive [clause list]**
- program is executed sequentially until *parallel* directive is specified
- *clause list* specifies conditions for parallelization, number of threads, data access, sharing, partitioning and manipulation with them...
  - conditional parallelization: **if(scalar expression)**
  - degree of parallelization: **num_threads(integer expression)**
  - data access methods and rules: **shared(variable list), private(variable list), firstprivate(variable list)**
  - collection of private data at final stage of parallel execution: **reduction(operator: variable list)**, operators: +, *, -,&, |, ^, &&, ||
  - partitioning and scheduling: **schedule(scheduling_class[, parameter])**, where you can choose from: static, dynamic, guided, and runtime
- relaxed-consistency (if the shared variable must be seen identically by all threads, the programmer must take care of it – FLUSH)

## OpenMP – *for* loops parallelization

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

or combined in short:

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

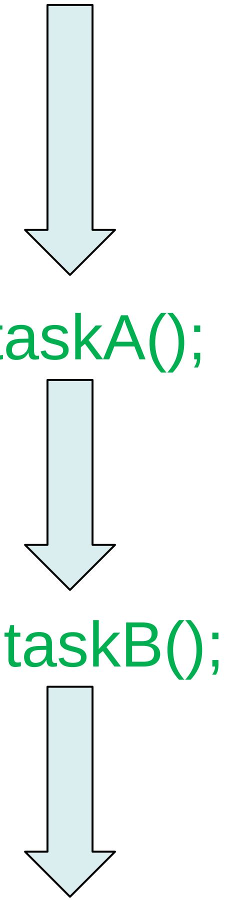
## OpenMP – *for* loops parallelization

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

?

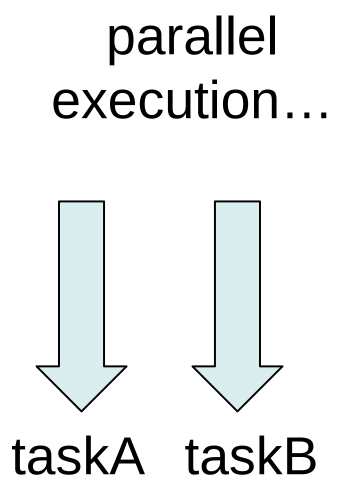
## OpenMP – each thread works on unique problem partition

Sequential execution…



## OpenMP – each thread works on unique problem partition

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            taskA();
        }
        #pragma omp section
         {
             taskB();
         }
     }
 }
```



## OpenMP – each thread works on unique problem partition

or in short:

```
#pragma omp parallel sections
{
    #pragma omp section
    {
        taskA();
    }
    #pragma omp section
    {
        taskB();
    }
}
```

## OpenMP – what it provides to programmer?

- nested parallelism (OMP_NESTED set to TRUE)
- explicit barriers specification (point of synchronization)
- mark code which should be executed by one thread only (arbitrarily selected or master only)
- mark critical section (only one thread can enter and execute at given time instant; others are required to wait or already have the given section done)
- ordered execution (as in serial execution)
- enforcing memory update and read (variable update …) – flush; when is flush used implicitly? 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 – what are other provided functionalities?

- 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);

**And many other functions…**

# OMP Directives and Control Overview

| OpenMP pragma, function, or clause | Concepts |
|---|---|
| `#pragma omp parallel` | Parallel region, teams of threads, structured block, interleaved execution across threads |
| `int omp_get_thread_num()`<br>`int omp_get_num_threads()` | Create threads with a parallel region and split up the work using the number of threads and thread ID |
| `double omp_get_wtime()` | Timing blocks of code |
| `setenv OMP_NUM_THREADS N`<br>`export OMP_NUM_THREADS=N` | Set the default number of threads with an environment variable |
| `#pragma omp barrier`<br>`#pragma omp critical`<br>`#pragma omp atomic` | Synchronization, critical sections |
| `#pragma omp for`<br>`#pragma omp parallel for` | Worksharing, parallel loops |
| `reduction(op:list)` | Reductions of values across a team of threads |
| `schedule(dynamic [,chunk])`<br>`schedule(static [,chunk])` | Loop schedules |
| `private(list), shared(list),`<br>`firstprivate(list)` | Data environment |
| `#pragma omp master`<br>`#pragma omp single` | Worksharing with a single thread |
| `#pragma omp task`<br>`#pragma omp taskwait` | Tasks including the data environment for tasks. |

## Legal forms for parallelizable *for* statements

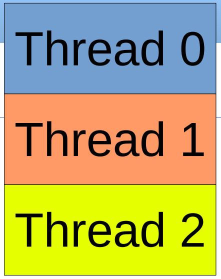
```
                                                  index++
                                                  ++index
                              index < end         index--
                              index <= end        --index
for   index = start  ;        index > end    ;    index += incr
                              index >= end        index -= incr
                                                  index = index + incr
                                                  index = incr + index
                                                  index = index - incr
```

- Variable index must have integer or pointer type (e.g., it can’t be a float)
- The expressions start, end, and incr must have a compatible type. For example, if index is a pointer, incr must have integer type
- The expressions start, end, and incr must not change during execution of the loop
- Variable index can only be modified by the “increment expression” in the “for” statement

## OMP – schedule(type, chunksize)

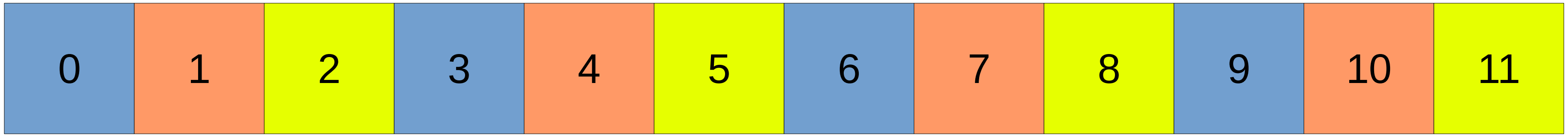
- **`type`** can be:
  - **`static`**: the iterations can be assigned to the threads before the loop is executed. If **`chunksize`** is not specified, iterations are evenly divided contiguously among threads
  - **`dynamic`** or **`guided`**: iterations are assigned to threads while the loop is executing. Default **`chunksize`** is 1
  - **`auto`**: the compiler and/or the run-time system determines the schedule
  - **`runtime`**: the schedule is determined at run-time using the **`OMP_SCHEDULE`** environment variable (e.g., **`export OMP_SCHEDULE="static,1"`**)
- Default schedule type is implementation dependent
  - GCC seems to use **`static`** by default

# OMP – schedule example

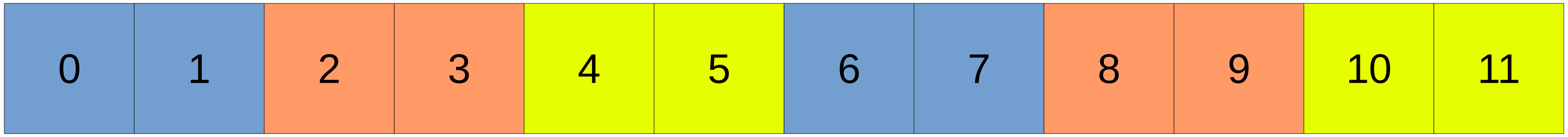


Twelve iterations 0, 1, … 11 and three threads

```
schedule(static, 1) num_threads(3)
```

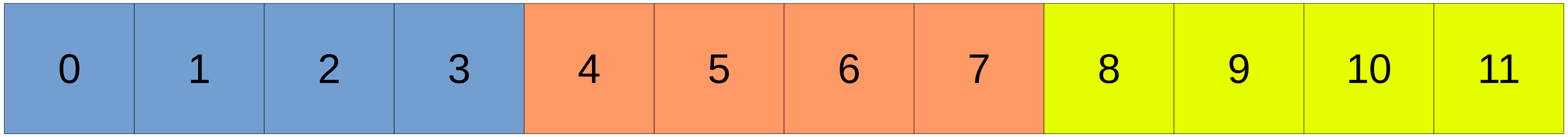


```
schedule(static, 2) num_threads(3)
```



```
schedule(static, 4)
```

Default chunksize in this case



## Choosing a schedule clause

| Schedule clause | When to use | Note |
|---|---|---|
| `static` | Pre-determined and predictable work per iteration | Least work at runtime: scheduling done at compile-time |
| `dynamic` | Unpredictable, highly variable work per iteration | Most work at runtime: complex scheduling logic used at run-time |

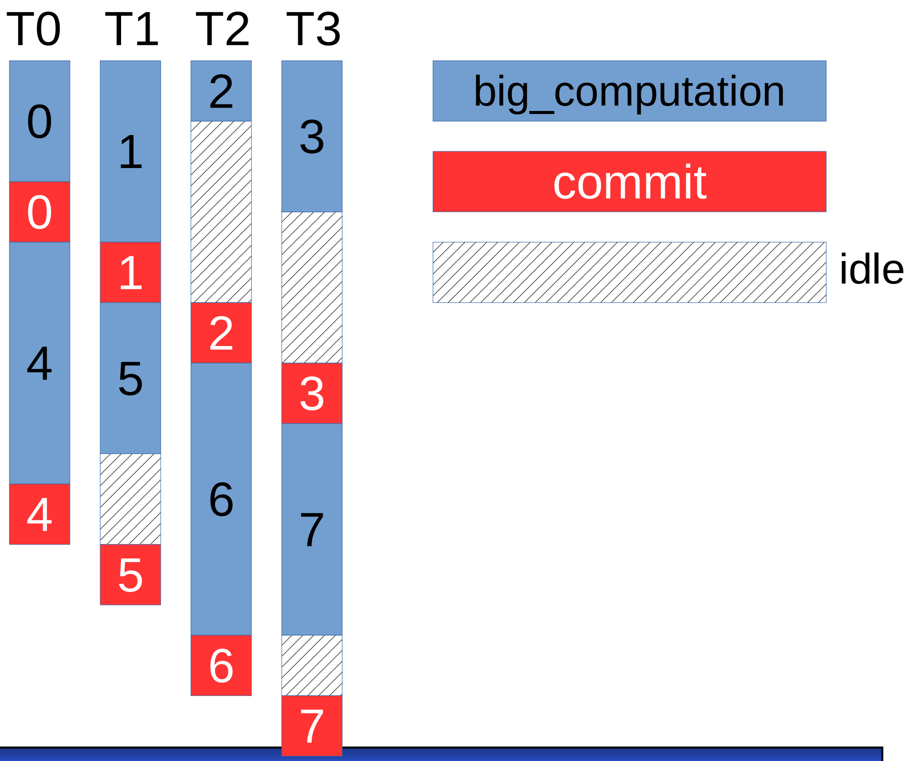
collapse(2) makes x and y private by default

```
#pragma omp parallel for collapse(2)
   for ( y = 0; y < ysize; y++ ) {
      for ( x = 0; x < xsize; x++ ) {
         drawpixel( x, y );
      }
   }
```

## The ordered directive

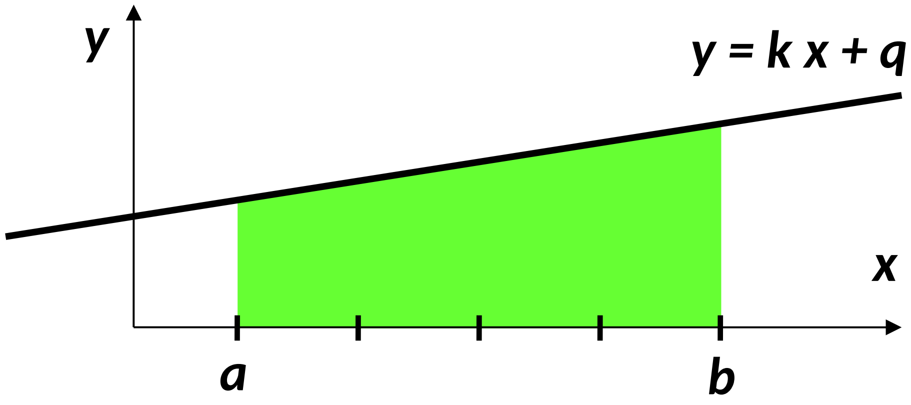
- Used when part of the loop must execute in serial order
  - **`ordered`** clause plus an **`ordered`** directive
- Example
  - **`big_computation(i)`** may be invoked concurrently in any order
  - **`commit(i)`** is invoked as if the loop were executed serially
- See omp-mandelbrot-ordered.c

```
#pragma omp parallel for ordered
for (i=0; i<n; i++) {
        big_computation(i);
#pragma omp ordered
        commit(i);
}
```



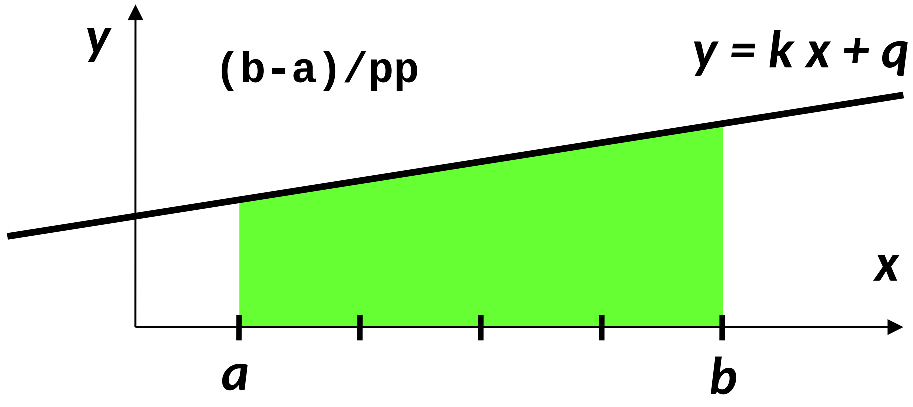
## Example – calculation of a definite integral – serial

```
#include <stdio.h>


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

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

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






        res = integral(a,  b,  polynomial,  N);

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



# Example – calculation of a definite integral – preparation

```
#include <stdio.h>


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

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

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

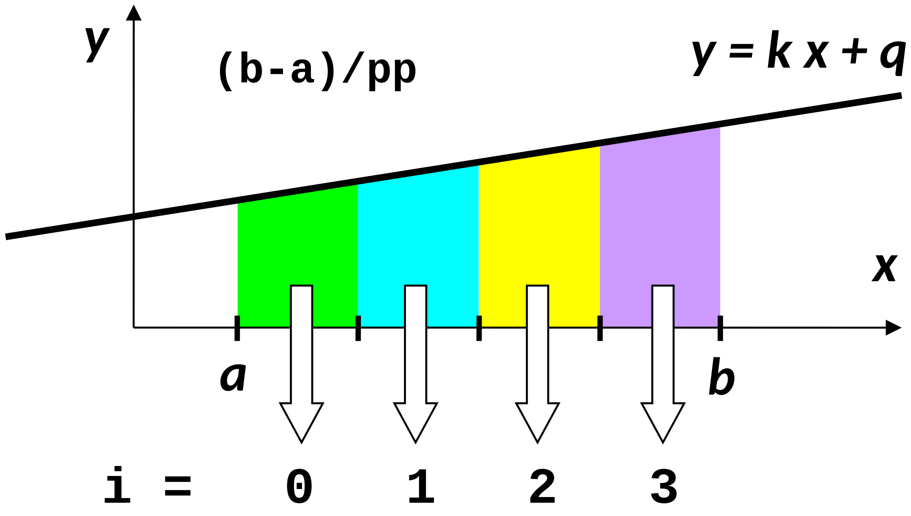


$$\texttt{res = integral(}\overbrace{\texttt{a+((b-a)/pp)*i}}^{\texttt{a}}\texttt{, }\overbrace{\texttt{a+((b-a)/pp)*(i+1)}}^{\texttt{b}}\texttt{, polynomial, }\overbrace{\texttt{N/pp}}^{\texttt{N}}\texttt{);}$$

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

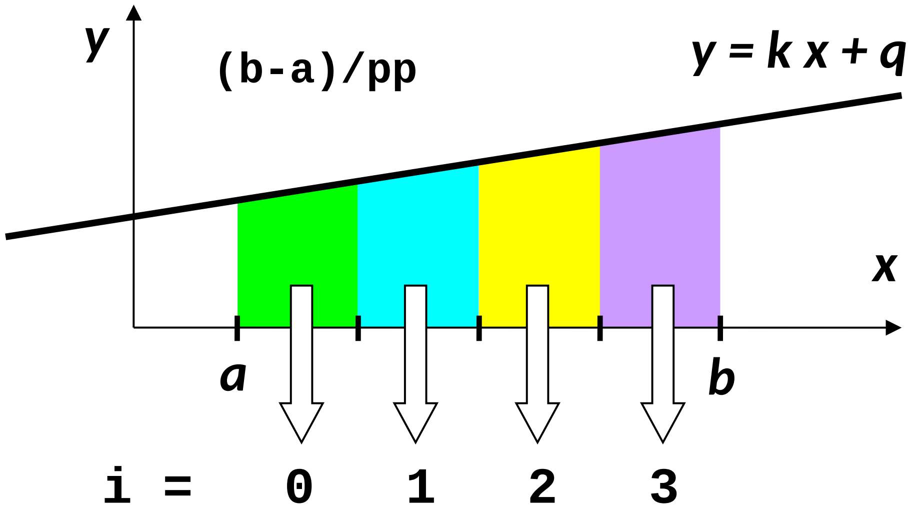
# Example – calculation of a definite integral – parallel

```
#include <stdio.h>


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

double polynomial(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),  polynomial,  N/pp);

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



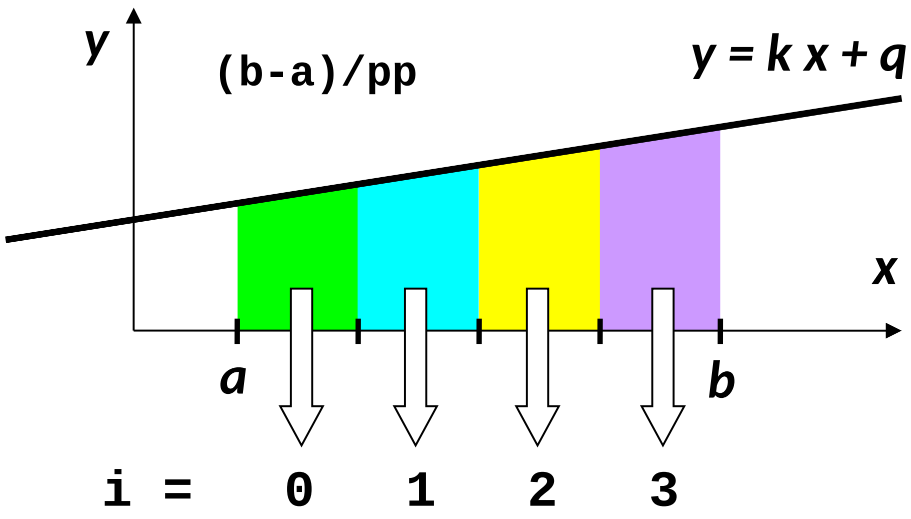
## Example – calculation of a definite integral – parallel

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

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

double polynomial(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),  polynomial,  N/pp);
        }
        printf("I = %lf ",res);
}
```



# Example – calculation of a definite integral – parallel

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

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

double polynomial(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),  polynomial,  N/pp);
        }
        printf("I = %lf ",res);
}
```



## OpenMP – another example

```
#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

**Explicit** communication, that is why:

- Most of MPI functions require communicator as argument
- **communicator** can be thought of as a handle (object) to a group of processes which can communicate with each other
- communicator MPI_COMM_WORLD includes all processes of given program
- processes belonging to the given communicator have assigned their unique identification number – RANK, which is integer in (0, N-1) – N is number of processes
- communication is either collective (all the communicator processes are involved) or point-to-point

## MPI – Message Passing Interface

Collective communication:

- is always blocking
- uses predefined MPI types (MPI_CHAR, MPI_INT, MPI_LONG,…)
- division according to purpose:
  - synchronization (processes are waiting at a given location)
  - collective computation (reduction) – when one process collects data from all the other processes and final result is computed by mutual operation between partial results
  - data movement (Broadcast, Scatter, Gather, All-to-All)

## MPI Custom Data Type – Fixed Length Array

```
int MPI_Type_contiguous(int count, MPI_Datatype
   old_type, MPI_Datatype *new_type);

MPI_Datatype dt_point;
MPI_Type_contiguous(3, MPI_DOUBLE, &dt_point);
MPI_Type_commit(&dt_point);
```

Release type

```
int MPI_Type_free(MPI_Datatype *datatype)
```

## MPI Custom Data Type – Structure

```
int MPI_Type_create_struct(int count, const int block_length[], const
   MPI_Aint displacement[], const MPI_Datatype types[], MPI_Datatype
   *new_type);

struct MyStruct {
  int a; double b; char c[10]; float d;
};

MPI_Datatype custom_dt;
MPI_Datatype custom_types[4] = {MPI_INT, MPI_DOUBLE, MPI_CHAR,
   MPI_FLOAT};
int custom_blocklen[3] = { 1, 1, 10, 1};
MPI_Aint custom_displacements[4] = {offsetof(MyStruct, a),
   offsetof(MyStruct, b), offsetof(MyStruct, c), offsetof(MyStruct, d)};

MPI_Type_create_struct(4, custom_blocklen, custom_displacements,
   custom_types, &custom_dt);
MPI_Type_commit(&custom_dt);
```

## MPI – almost minimal 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();
}
```

## Example – calculation of a definite integral – MPI

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

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

int main(int argc, char *argv[])
{
  int myid, numprocs;
  unsigned long int i, n = 0;
  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);  // obtain number
                                             // of processes
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);  // obtain process number
```

## Example – calculation of a definite integral – MPI

```
do {
    if (myid == 0) {
        printf("\n Enter number of intervals (0 to exit): "); fflush(stdout);
        scanf("%ld", &n);
        startwtime = MPI_Wtime();
    }

    // send variable n, with vector length 1, type int, from process #0 to others
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // finish test if zero is broadcasted or received
    if(n==0) break;
```

## Example – calculation of a definite integral – MPI

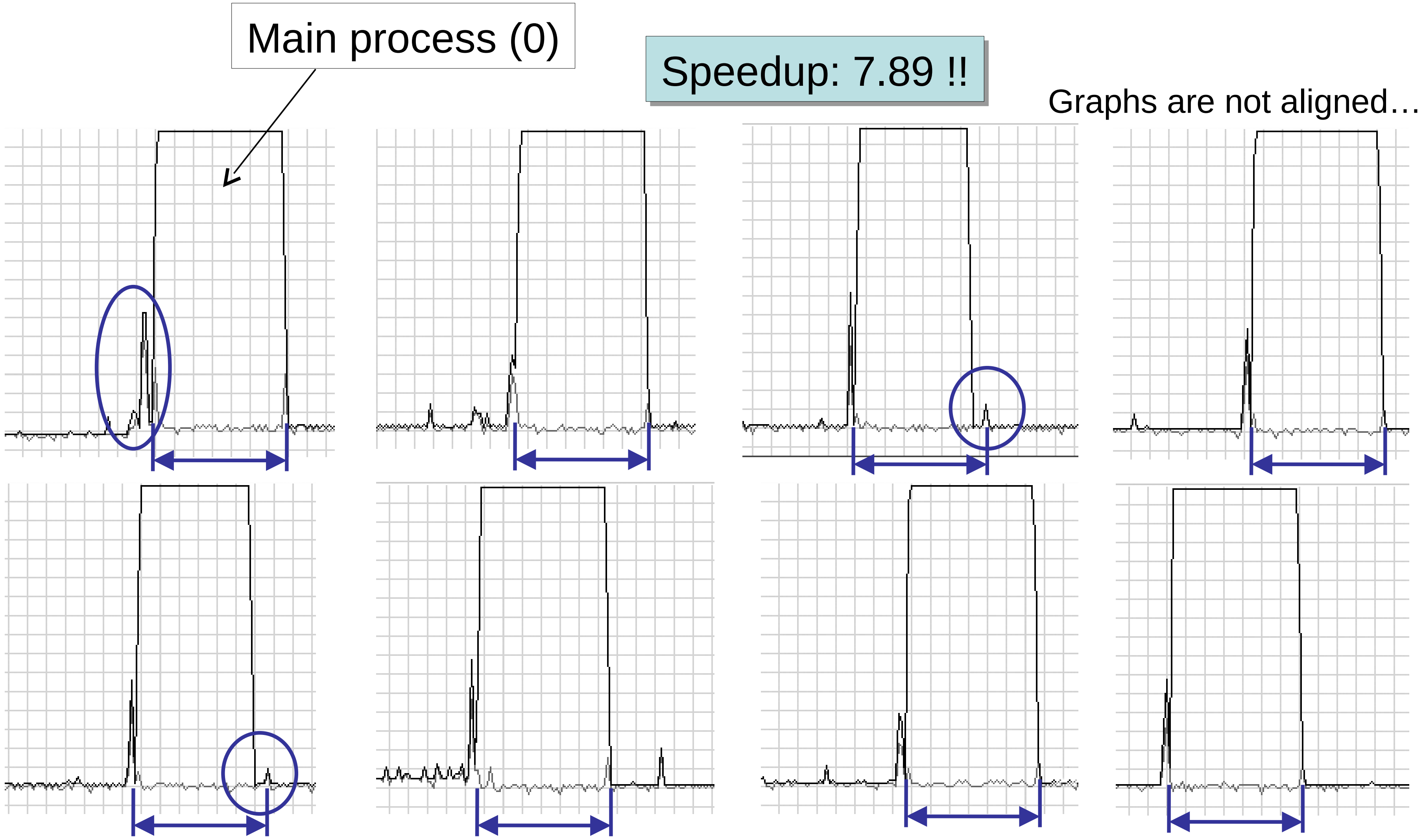
```
double h = (b - a) / n;
double i1 = myid*(n/ numprocs);
double i2 = (myid+1)*(n/numprocs);

integral= ( f(a+i1*h) + f(a+i2*h) ) / 2;

for( i=i1+1 ; i<i2 ; i++ )
    integral += f(a+i*h);

// summarize integral variable from all processes to total in process 0,
// length 1, type double, operation sum/addition
MPI_Reduce(&integral, &total, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
```

## Example – calculation of a definite integral – MPI

```
    if (myid == 0) {
            endwtime = MPI_Wtime();
            printf("I= %f\n", total);
            printf("spent time: %f s\n", endwtime - startwtime); fflush(stdout);
        }
    } while (1);

    MPI_Finalize();
    return 0;
}
```

## Example – calculation of a definite integral – MPI

Computation on 8 single-core processors:



## Combination of OpenMP a MPI – almost minimal 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
     %s\n", iam, np, rank, numprocs, processor_name);
  }
    MPI_Finalize();
    return 0;
}
```