



# *Computação em Larga Escala*

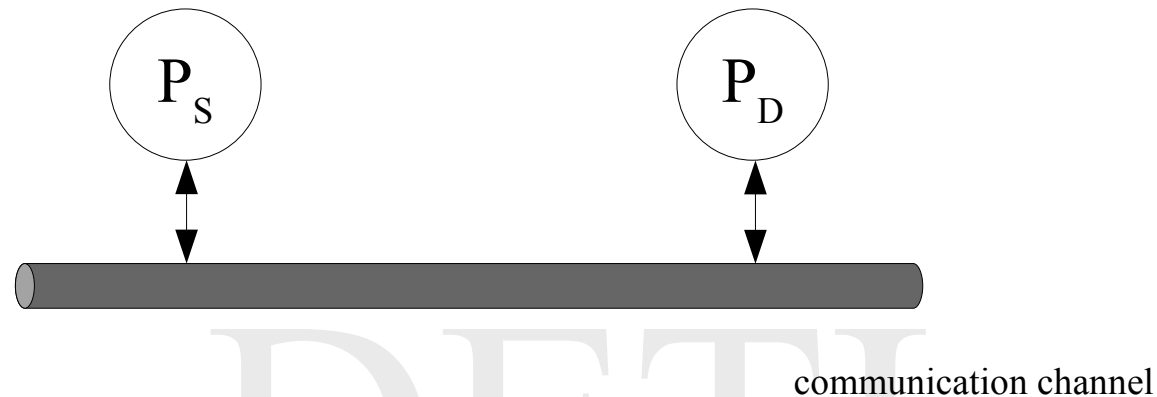
*Message Passing Interface (MPI) 2*

António Rui Borges

## *Summary*

- *Non-blocking communication communication*
  - *Send and receive primitives*
- *Collective synchronization*
  - *Barrier*
- *Example*
  - *Binary sorting*
- *Suggested reading*

## *Non-blocking communication - 1*



- although non-blocking communication is generic in MPI, it will be restricted here to *point-to-point communication* where a process, called the *source*, sends a message to another process, called the *destination*
- *send* and *receive* being non-blocking operations, are not synchronized: *send* forwards the message and returns immediately without checking if the message was or not delivered; *receive*, in turn, returns immediately without checking if a message was or not received
- new operations are needed for this purpose: *wait* and *test*; the former blocks until the transfer is completed, the latter checks the conclusion of the transfer

## *Non-blocking communication - 2*

```
int MPI_Isend (void *buf, int count, MPI_Datatype datatype, int dest, int tag,  
              MPI_Comm comm, MPI_Request *request);  
  
int MPI_Irecv (void *buf, int count, MPI_Datatype datatype, int source, int tag,  
              MPI_Comm comm, MPI_Request *request);
```

**buf** - pointer to the memory region where the information content of the message is, or will be, stored

**count** - number of elements of the array which represents the information content of the message

**datatype** - MPI information data type

**dest** - rank of the destination process (send primitive)

**source** - rank of the source process (receive primitive)  
it may be MPI\_ANY\_SOURCE so that a message from any source is received

**tag** - message tag, it may be used by the application programmer to distinguish different types of messages  
it may be MPI\_ANY\_TAG so that a message with any tag is received

**comm** - identification of the communication context (process group)

**request** - pointer to the handler of the transfer

## *Non-blocking communication - 3*

```
int MPI_Wait(MPI_Request *request, MPI_Status *status);  
int MPI_Test(MPI_Request *request, int *transComp, MPI_Status *status);
```

**request** - pointer to the handler of the transfer  
**transComp** - pointer to a boolean flag stating if the transfer is or not completed  
**status** - pointer to a structure which defines the operation status  
it contains at least the following fields: MPI\_TAG, and  
MPI\_ERROR; if the status is not important, the pointer constant  
MPI\_STATUS\_IGNORE may be used

## *Non-blocking communication - 4*

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

int main (int argc, char *argv[])
{
    int rank, size;
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    if (rank == 0)
    {
        int i;
        char *sndData, **recData;
        bool allMsgRec, recVal, msgRec[size];
        MPI_Request reqSnd[size], reqRec[size];
        sndData = malloc (100);
        recData = malloc (size * sizeof (char *));
        for (i = (rank + 1) % size; i < size; i++)
            recData[i] = malloc (100);
        sprintf (sndData, "Start working!");
        printf ("I, %d, am going to transmit the message: %s to all other processes in the group\n",
            rank, sndData);
        for (i = (rank + 1) % size; i < size; i++)
            MPI_Isend (sndData, strlen (sndData) + 1, MPI_CHAR, i, 0, MPI_COMM_WORLD, &reqSnd[i]);
        printf ("I, %d, am going to receive a message from all other processes in the group\n",
            rank);
        for (i = (rank + 1) % size; i < size; i++)
        {
            MPI_Irecv (recData[i], 100, MPI_CHAR, i, 0, MPI_COMM_WORLD, &reqRec[i]);
            msgRec[i] = false;
        }
    }
}
```

## *Non-blocking communication - 5*

```
do
{ allMsgRec = true;
  for (i = (rank + 1) % size; i < size; i++)
    if (!msgRec[i])
    { recVal = false;
      MPI_Test(&reqRec[i], (int *) &recVal, MPI_STATUS_IGNORE);
      if (recVal)
      { printf ("I, %d, received the message: %s\n", rank, recData[i]);
        msgRec[i] = true;
      }
      else allMsgRec = false;
    }
  } while (!allMsgRec);
}
else { char *sndData, *recData;
      recData = malloc (100);
      MPI_Recv (recData, 100, MPI_CHAR, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      sndData = malloc (100);
      sprintf (sndData, "I am working (%d)!", rank);
      MPI_Send (sndData, strlen (sndData) + 1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    }
MPI_Finalize ();

return EXIT_SUCCESS;
}
```

## ***Non-blocking communication - 6***

```
[ruib@ruib-laptop basic3]$ mpiexec -n 4 ./nonBlockComm
I, 0, am going to transmit the message: Start working! to all other processes in the group
I, 0, am going to receive a message from all other processes in the group
I, 0, received the message: I am working (1)!
I, 0, received the message: I am working (3)!
I, 0, received the message: I am working (2)!
[ruib@ruib-laptop basic3]$
```

```
[ruib@ruib-laptop basic3]$ mpiexec -n 16 ./nonBlockComm
I, 0, am going to transmit the message: Start working! to all other processes in the group
I, 0, am going to receive a message from all other processes in the group
I, 0, received the message: I am working (1)!
I, 0, received the message: I am working (3)!
I, 0, received the message: I am working (7)!
I, 0, received the message: I am working (10)!
I, 0, received the message: I am working (11)!
I, 0, received the message: I am working (12)!
I, 0, received the message: I am working (15)!
I, 0, received the message: I am working (4)!
I, 0, received the message: I am working (6)!
I, 0, received the message: I am working (8)!
I, 0, received the message: I am working (9)!
I, 0, received the message: I am working (5)!
I, 0, received the message: I am working (13)!
I, 0, received the message: I am working (14)!
I, 0, received the message: I am working (2)!
[ruib@ruib-laptop basic3]$
```



## *Collective synchronization*

*Collective synchronization* addresses the problem of ensuring that multiple processes, organized in a group, block waiting for one another to reach the same point of program execution.

There is a single type of collective synchronization

- *barrier* – works, as the name says, like a barrier which is only lifted when all processes reach the same point of program execution; all block, except the last one to reach that point, which wakes up the rest and allows all of them to proceed.

## *Barrier - 1*

```
int MPI_Barrier (MPI_Comm comm);
```

**comm** - identification of the communication context (process group)

DETI

## *Barrier - 2*

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

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

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    printf ("I, process %d, am going to block waiting for the others!\n", rank);
    MPI_Barrier (MPI_COMM_WORLD);
    printf ("I, process %d, am going to proceed!\n", rank);
    MPI_Finalize ();
    return EXIT_SUCCESS;
}
```

## *Barrier - 3*

```
[ruib@ruib-laptop basic3]$ mpicc -Wall -o barrier barrier.c
```

```
[ruib@ruib-laptop basic3]$ mpiexec -n 10 ./barrier
I, process 2, am going to block waiting for the others!
I, process 5, am going to block waiting for the others!
I, process 4, am going to block waiting for the others!
I, process 7, am going to block waiting for the others!
I, process 8, am going to block waiting for the others!
I, process 1, am going to block waiting for the others!
I, process 3, am going to block waiting for the others!
I, process 0, am going to block waiting for the others!
I, process 9, am going to block waiting for the others!
I, process 6, am going to block waiting for the others!
I, process 0, am going to proceed!
I, process 1, am going to proceed!
I, process 2, am going to proceed!
I, process 3, am going to proceed!
I, process 4, am going to proceed!
I, process 5, am going to proceed!
I, process 8, am going to proceed!
I, process 9, am going to proceed!
I, process 6, am going to proceed!
I, process 7, am going to proceed!
[ruib@ruib-laptop basic3]$
```

## *Binary sorting - 1*

The approach to be followed stems from the fact that it is less complex to sort a somewhat presorted sequence than a complete random one.

Thus, suppose the sequence of  $N$  values `val`, with  $N \geq 2$ , is such that both its halves, from 0 to  $N/2-1$  and  $N/2$  to  $N-1$ , respectively, are already sorted. In order to sort the whole sequence one may use merge sorting

```
if ((N % 2) == 1)
{ for (m = 0; m < N/2 + 1; m++)
    for (n = (m == 0) ? 1 : 0; (m + n) < N/2 + 1; n++)
        CAPS (val[m+n-1], val[N/2+n]);
}
else { for (m = 0; m < N/2; m++)
    for (n = 0; (m + n) < N/2; n++)
        CAPS (val[m+n], val[N/2+n]);
}
```

where CAPS stands for *compare and possible swap the values*.

The merge sorting operation costs  $N(N+2)/8$  CAPS, when  $N$  is even  
and  $N(N+2)/8-1$  CAPS, when  $N$  is odd.

## Binary sorting - 2

### Example using an 7-valued sequence

|          |          |          |          |          |          |          |     |   |
|----------|----------|----------|----------|----------|----------|----------|-----|---|
| 5        | 6        | 9        | 2        | 4        | 7        | 8        | --- | initial situation                                 |
| 4        | 6        | 8        | 2        | 5        | 7        | <b>9</b> | --- | iteration 1 (3 CAPS): first value positioned      |
| <b>2</b> | 5        | 7        | 4        | 6        | <b>8</b> | <b>9</b> | --- | iteration 2 (3 CAPS): another 2 values positioned |
| <b>2</b> | <b>4</b> | 6        | 5        | <b>7</b> | <b>8</b> | <b>9</b> | --- | iteration 3 (2 CAPS): another 2 values positioned |
| <b>2</b> | <b>4</b> | <b>5</b> | <b>6</b> | <b>7</b> | <b>8</b> | <b>9</b> | --- | iteration 4 (1 CAPS): last 2 values positioned    |

### Example using an 8-valued sequence

|          |          |          |          |          |          |          |          |     |   |
|----------|----------|----------|----------|----------|----------|----------|----------|-----|---|
| 1        | 5        | 6        | 9        | 2        | 4        | 7        | 8        | --- | initial situation                                 |
| <b>1</b> | 4        | 6        | 8        | 2        | 5        | 7        | <b>9</b> | --- | iteration 1 (4 CAPS): first 2 values positioned   |
| <b>1</b> | <b>2</b> | 5        | 7        | 4        | 6        | <b>8</b> | <b>9</b> | --- | iteration 2 (3 CAPS): another 2 values positioned |
| <b>1</b> | <b>2</b> | <b>4</b> | 6        | 5        | <b>7</b> | <b>8</b> | <b>9</b> | --- | iteration 3 (2 CAPS): another 2 values positioned |
| <b>1</b> | <b>2</b> | <b>4</b> | <b>5</b> | <b>6</b> | <b>7</b> | <b>8</b> | <b>9</b> | --- | iteration 4 (1 CAPS): last 2 values positioned    |

## *Binary sorting - 3*

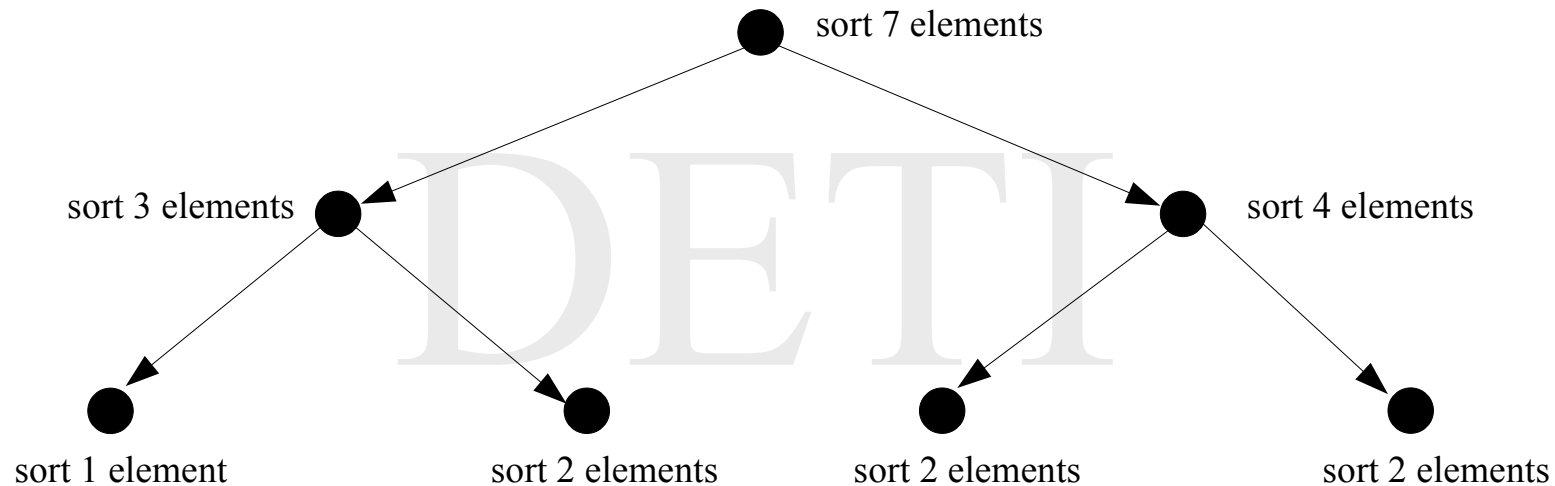
### **Top-down (or recursive) decomposition**

```
void binSort (unsigned int nElem, unsigned int val[])
{
    unsigned int m, n;

    if (nElem == 1) return; // trivial situation
    if (nElem > 2)
    {
        binSort (nElem/2, val);
        binSort (nElem-nElem/2, &val[nElem/2]);
    }
    if ((nElem % 2) == 1)
    {
        for (m = 0; m < nElem/2 + 1; m++)
            for (n = (m == 0) ? 1 : 0; (m + n) < nElem/2 + 1; n++)
                CAPS (val[m+n-1], val[nElem/2+n]);
    }
    else {
        for (m = 0; m < nElem/2; m++)
            for (n = 0; (m + n) < nElem/2; n++)
                CAPS (val[m+n], val[nElem/2+n]);
    }
}
```

## *Binary sorting - 4*

### **Example using an 7-valued sequence**



The total number of CAPS required to sort a  $N$ -valued sequence is of the order of magnitude of  $N \log_2(N)$ , or  $O[N \log_2(N)]$ .

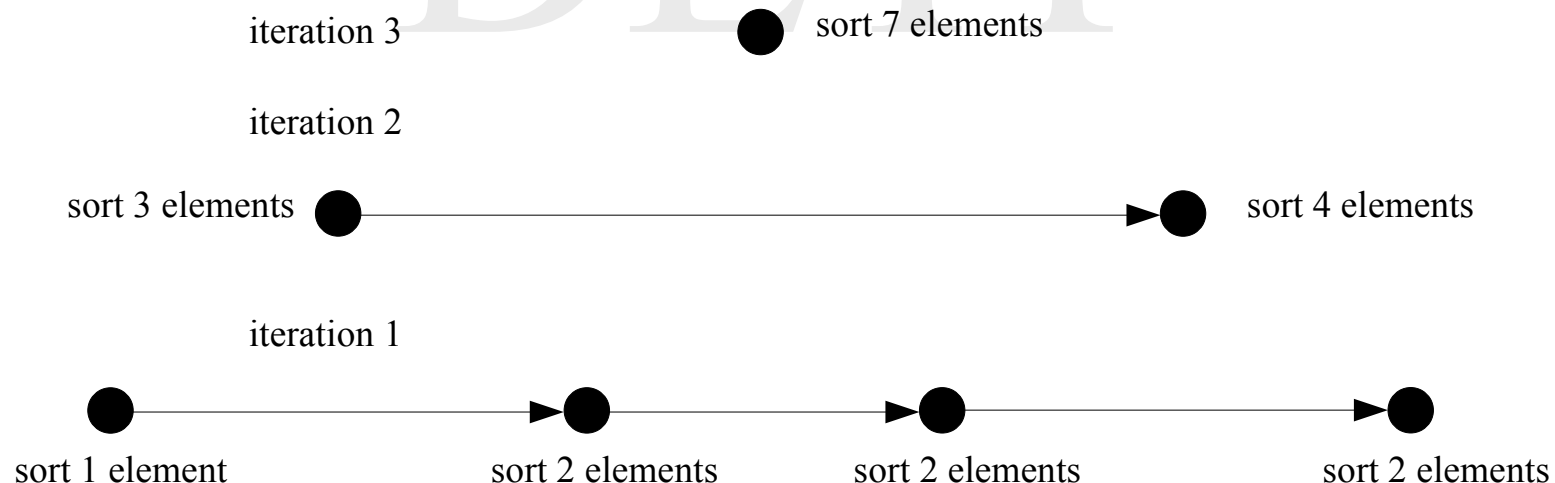


## *Binary sorting - 5*

### **Bottom-up (or iterative) decomposition**

The bottom-up decomposition parses the spanning from the bottom leaves, as the first iterative step, and proceeds up parsing the next layer until the root is reached.

#### **Example using an 7-valued sequence**



## *Binary sorting - 6*

So, for a given  $N$ , one has first to determine how the sequence will be partitioned at the lowest level so that the sorting may start.

A way of doing this is through the successive application of the following operations

```
void split (unsigned int N, unsigned int ord[])
{
    if (N > 2)
    {
        ord[0] = N/2;
        ord[N/2] = N - N/2;
        split (N/2, ord);
        split (N - N/2, ord + N/2);
    }
    else if (N == 2) ord[0] = 2;
}

void merge (unsigned int n, unsigned int ord[], unsigned int *pSize)
{
    unsigned int k, m;
    for (k = 0, m = 0; k < n; k++)
        if (ord[k] > 0)
        {
            ord[m] = ord[k];
            m += 1;
        }
    *pSize = m;
}
```

## *Binary sorting - 7*

### **Example using an 7-valued sequence**

6 9 5 8 4 7 2 --- initial situation

6 9 5 8 4 7 2 --- iteration step 1

6 5 9 4 8 2 7

6 5 9 4 8 2 7 --- iteration step 2

5 6 9 2 4 7 8

5 6 9 2 4 7 8 --- iteration step 3

2 4 5 6 7 8 9

### **Evolution of the contents of ord during the iteration process**

1 2 2 2 --- iteration step 1

0 3 0 4 --- iteration step 2

0 0 0 7 --- iteration step 3

## *Binary sorting - 8*

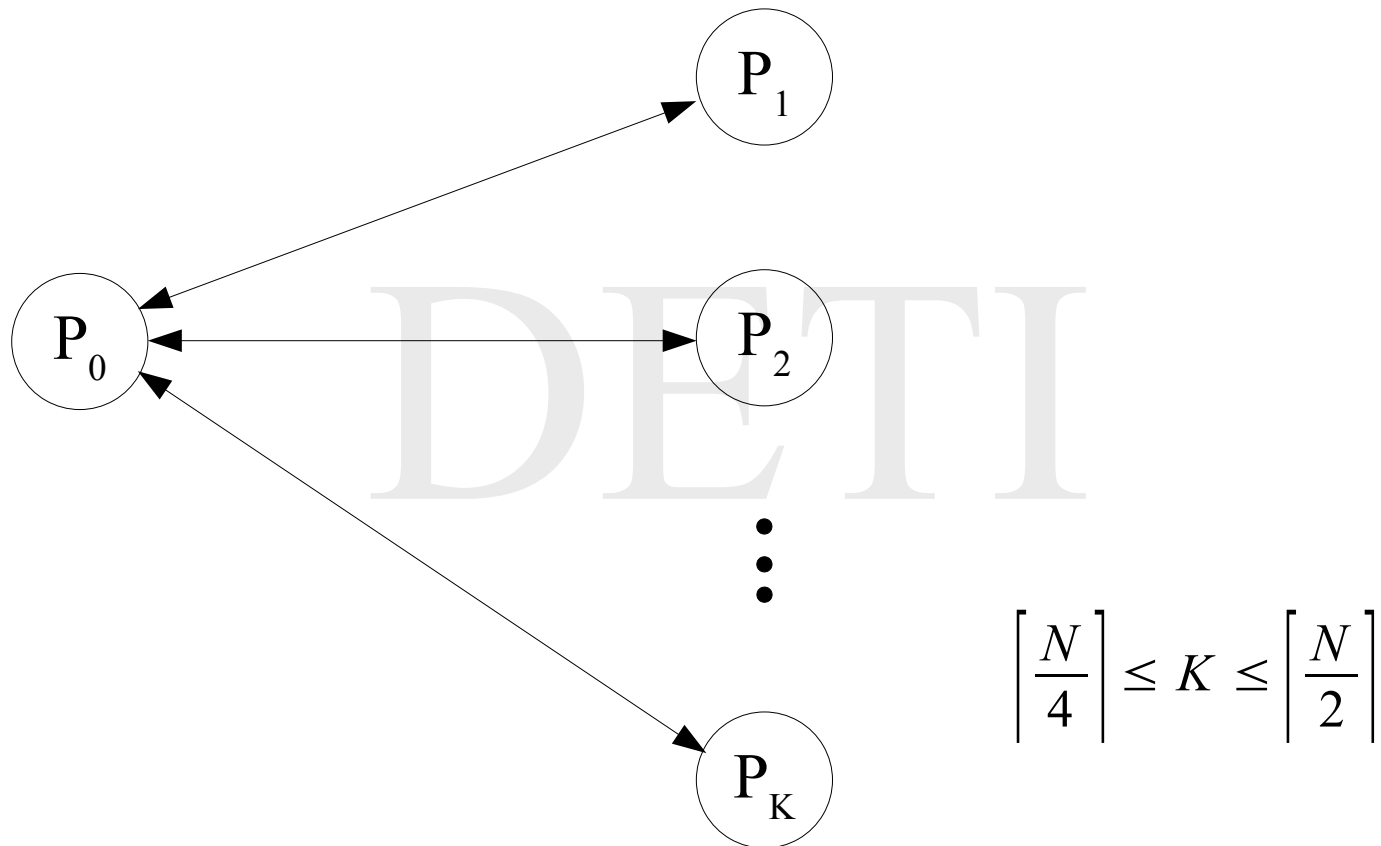
### **Bottom-up (or iterative) decomposition**

```
unsigned int k, r, m, n, t, nBase, incNBase, size, iters, base;  
unsigned int val[N], ord[N];  
  
iters = (unsigned int) ceil (log2 ( (double) N ));  
for (n = 0; n < N; n++)  
    ord[n] = 0;  
split (N, ord);  
merge (N, ord, &size);  
base = 2;
```

## Binary sorting - 9

```
for (k = 0, base = 2; k < iters; k++, base *= 2)
{ nBase = 0;
  for (r = 0; r < size; r++)
  { incNBase = ord[r];
    if ((ord[r] > base/2) && (ord[r] <= base))
    { if ((ord[r] % 2) == 0)
      { for (m = 0; m < ord[r]/2; m++)
        for (n = nBase; (m + n) < nBase + ord[r]/2; n++)
          CAPS (val + m+n, val + ord[r]/2+n);
      }
      else if (ord[r] > 1)
      { for (m = 0; m < ord[r]/2 + 1; m++)
        for (n = nBase + ((m == 0) ? 1 : 0);
              (m + n) < nBase + ord[r]/2 + 1; n++)
          CAPS (val + m+n-1, val + ord[r]/2+n);
        }
      }
    t = r - base/2;
    while (t < (r - base/4))
    { if ((r > (base/2-1)) && (ord[t] != 0) && (ord[t] <= ord[r]))
      { ord[r] += ord[t];
        ord[t] = 0;
        break;
      }
      t += 1;
    }
    nBase += incNBase;
  }
}
```

## *Mapping binary sorting to independent processes - 1*



## *Mapping binary sorting to independent processes - 2*

- process 0 has the role of the *dispatcher*
  - in iteration 1, messages are sent to  $k$ , with  $k \leq K$ , processes to sort cuts of the original sequence
  - in successive iterations, as cuts of the original sequence are already sorted, the number of processes at work are progressively reduced
  - in the last iteration, the original sequence has been transformed in such a manner that only two parts already sorted remain, so only one process is required
- all other processes perform the sorting process on cuts of the original sequence; when they are no longer required, they terminate

## *Binary sorting a list of numbers*

```
[ruib@ruib-laptop basic3]$ mpicc -Wall -o binSortNum binSortNum.c
```

```
[ruib@ruib-laptop basic3]$ mpiexec -n 10 ./binSortNum
Too few processes!
```

```
[ruib@ruib-laptop basic3]$ mpiexec -n 12 ./binSortNum
```

### **Unsorted sequence**

```
327 264 179 583 652 409 469 454 362 946 921 62 761 716 633 258 266
292 976 850 59 235 376 370 765 664 926
```

### **Sorted sequence**

```
59 62 179 235 258 264 266 292 327 362 370 376 409 454 469 583 633
652 664 716 761 765 850 921 926 946 976
```

```
[ruib@ruib-laptop basic3]$ mpiexec -n 16 ./binSortNum
```

### **Unsorted sequence**

```
962 244 417 227 721 394 241 796 249 348 913 983 629 408 700 159 710
537 526 766 826 61 423 674 712 590 340
```

### **Sorted sequence**

```
61 159 227 241 244 249 340 348 394 408 417 423 526 537 590 629 674
700 710 712 721 766 796 826 913 962 983
```

```
[ruib@ruib-laptop basic3]$
```



## *Binary sorting a list of names - 1*

```
[ruib@ruib-laptop basic3]$ mpicc -Wall -o binSortListNames binSortListNames.c -lm
```

```
[ruib@ruib-laptop basic3]$ mpiexec -n 10 ./binSortListNames listNames.txt  
Too few processes!
```

```
[ruib@ruib-laptop basic3]$ mpiexec -n 14 ./binSortListNames listNames.txt
```

### **Unsorted list**

susana  
claudino  
margarida  
alfredo  
luis  
joaquina  
francisca  
teresa  
ana  
duarte  
aurora  
silvino  
celestino  
jaime  
josefina  
hermengarda  
afonso  
agostinha  
maria  
augusto  
leonilde  
fernando  
rui  
renata  
jorge  
georgina  
rosalina  
vitorino  
adelino

DETI

## *Binary sorting a list of names - 2*

### **Sorted list**

```
adelino  
afonso  
agostinha  
alfredo  
ana  
augusto  
aurora  
celestino  
claudino  
duarte  
fernando  
francisca  
georgina  
hermengarda  
jaime  
joaquina  
jorge  
josefina  
leonilde  
luis  
margarida  
maria  
renata  
rosalina  
rui  
silvino  
susana  
teresa  
vitorino  
[ruib@ruib-laptop basic3]$
```

DETI

## *Suggested reading*

- *Introduction to HPC with MPI for Data Science*, Nielsson F., Springer International, 2016
  - Chapter 2: *Introduction to MPI: The Message Passing Interface*
- *MPI: A Message-Passing Interface Standard (Version 3.1)*, Message Passing Interface Forum, 2015