# Parallelism (PAR)

## Parallel programming principles: Data decomposition

Eduard Ayguadé, Julita Corbalán,
Daniel Jiménez and Gladys Utrera

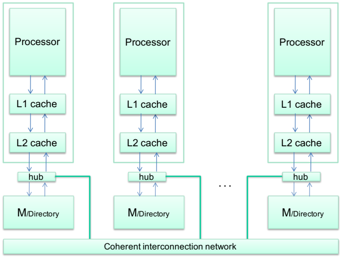
Computer Architecture Department
Universitat Politècnica de Catalunya

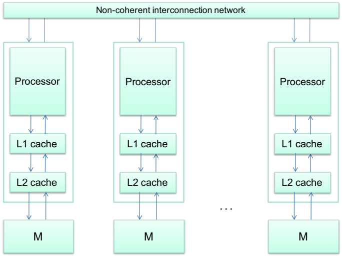
Course 2017/18 (Spring semester)

## Outline

## Why, when and how?

- Used to derive concurrency for problems that operate on large amounts of data focusing on the multiplicity of data
  - E.g. Elements in vectors, rows/columns/slices in matrices, elements in a list and subtrees in a tree
- ... for architectures in which memory plays a performance role





## Motivating example

```
#define n 100
#pragma omp parallel
#pragma omp single
for (iter=0; i<num_iters; iter++) {
    #pragma omp taskloop num_tasks(4)
    for (int i=0; i<n; i++)
        b[i] = foo1(a[i]);

    #pragma omp taskloop num_tasks(4)
    for (int i=0; i<n; i++)
        c[i] = foo2(b[i]);

    #pragma omp taskloop num_tasks(4)
    for (int i=0; i<n; i++)
        a[i] = foo3(c[i]);
}
```

**Vectors a, b and c are distributed across the memories of the NUMA system, as follows**

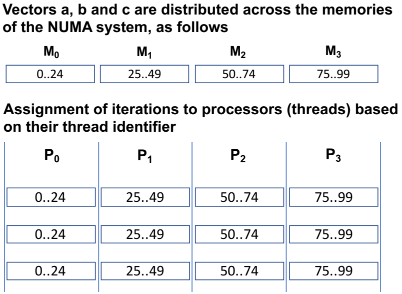
| $M_0$ | $M_1$ | $M_2$ | $M_3$ |
|---|---|---|---|
| 0..24 | 25..49 | 50..74 | 75..99 |

**Possible assignment of iterations to processors (threads) in the different loops**

| $P_0$ | $P_1$ | $P_2$ | $P_3$ |
|---|---|---|---|
| 25..49 | 50..74 | 0..24 | 75..99 |
| 25..49 | 75..99 | 0..24 | 50..74 |
| 50..74 | 25..49 | 75..99 | 0..24 |

## Motivating example (cont.)

```
// Solution based on for work-sharing
#pragma omp parallel
for (iter=0; i<num_iters; iter++) {
    #pragma omp for schedule(static) nowait
    for (int i=0; i<n; i++)
        b[i] = foo1(a[i]);
    #pragma omp for schedule(static) nowait
    for (int i=0; i<n; i++)
        c[i] = foo2(b[i]);
    #pragma omp for schedule(static) nowait
    for (int i=0; i<n; i++)
        a[i] = foo3(c[i]);
}
```


**Vectors a, b and c are distributed across the memories of the NUMA system, as follows**

| $M_0$ | $M_1$ | $M_2$ | $M_3$ |
|---|---|---|---|
| 0..24 | 25..49 | 50..74 | 75..99 |

**Assignment of iterations to processors (threads) based on their thread identifier**

| $P_0$ | $P_1$ | $P_2$ | $P_3$ |
|---|---|---|---|
| 0..24 | 25..49 | 50..74 | 75..99 |
| 0..24 | 25..49 | 50..74 | 75..99 |
| 0..24 | 25..49 | 50..74 | 75..99 |

## Why, when and how? (cont.)

- Step 1: Identify the data used and/or produced in the computations
  - Output data, input data or both
- Step 2: Partition this data across various tasks
  - Linear or geometric decomposition
  - Recursive decomposition
- Step 3: Obtain a computational partitioning that corresponds to the data partitioning: owner-computes rule
- Step 4: In distributed–memory architectures, add the necessary data allocation and movement actions

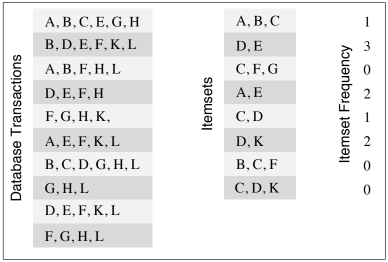
## Guidelines for data decomposition

- Data can be partitioned in various ways – this may critically impact performance
  - Generate comparable amounts of work (for load balancing)
  - Maximize data locality (or minimize the need for task interactions)
    - Minimize volume of data involved in task interactions
    - Minimize frequency of interactions
    - Minimize contention and hot spots
  - Overlap computation with interactions to "hide" their effect
- Parameterizable data partition
  - number of data chunks, size, ...
- Simplicity

## Example

Counting the instances of given itemsets in a database of transactions

**(a) Transactions (input), itemsets (input), and frequencies (output)**

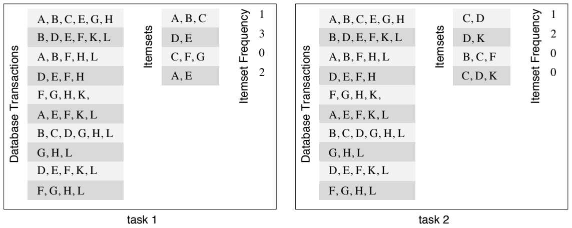


| Database Transactions | Itemsets | Itemset Frequency |
|---|---|---|
| A, B, C, E, G, H | A, B, C | 1 |
| B, D, E, F, K, L | D, E | 3 |
| A, B, F, H, L | C, F, G | 0 |
| D, E, F, H | A, E | 2 |
| F, G, H, K, | C, D | 1 |
| A, E, F, K, L | D, K | 2 |
| B, C, D, G, H, L | B, C, F | 0 |
| G, H, L | C, D, K | 0 |
| D, E, F, K, L | | |
| F, G, H, L | | |

## Output data decomposition

- Partition of the output data structures across tasks. Input data structures may follow the same decomposition or require replication in order to avoid task interactions
- Example: the itemset frequencies are partitioned across tasks
  - The database of transactions needs to be replicated
  - The itemsets can be partitioned across tasks as well (reduce memory utilization)

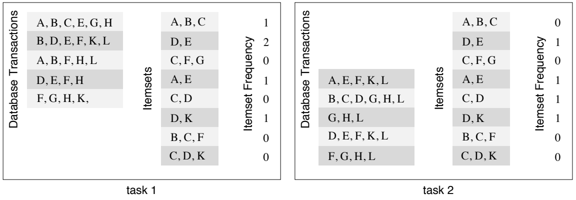
**(b) Partitioning the frequencies (and itemsets) among the tasks**



## Input data decomposition

- Partition the input data structures across tasks. It may require combining partial results in order to generate the output data structures
- Example: the database transactions can be partitioned, but it requires the itemsets to be replicated. Final aggregation of partial counts for all itemsets

**Partitioning the transactions among the tasks**



task 1

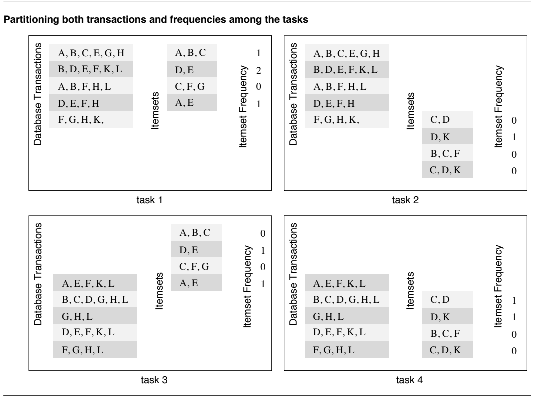
| Database Transactions | Itemsets | Itemset Frequency |
|---|---|---|
| A, B, C, E, G, H | A, B, C | 1 |
| B, D, E, F, K, L | D, E | 2 |
| A, B, F, H, L | C, F, G | 0 |
| D, E, F, H | A, E | 1 |
| F, G, H, K, | C, D | 0 |
| | D, K | 1 |
| | B, C, F | 0 |
| | C, D, K | 0 |

task 2

| Database Transactions | Itemsets | Itemset Frequency |
|---|---|---|
| | A, B, C | 0 |
| | D, E | 1 |
| | C, F, G | 0 |
| A, E, F, K, L | A, E | 1 |
| B, C, D, G, H, L | C, D | 1 |
| G, H, L | D, K | 1 |
| D, E, F, K, L | B, C, F | 0 |
| F, G, H, L | C, D, K | 0 |

## Input *and* output data decompositon

- Input and output data decomposition could be combined
- Example: the database and itemsets (input) and counts (output) can be decomposed



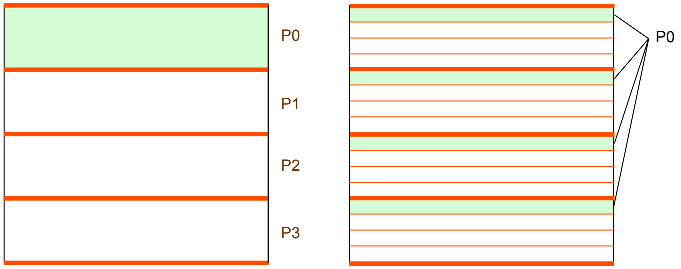
## The Owner Computes rule

It defines who is responsible for doing the computations:

- In the case of output data decomposition, the owner computes rule implies that the output is computed by the task to which the output data is assigned.
- In the case of input data decomposition, the owner computes rule implies that all computations that use the input data are performed by the task to which the input is assigned.

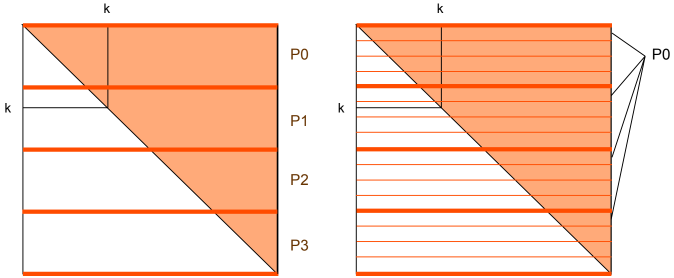
## Data distributions for geometric decomposition

Block (left) and cyclic (right) data decompositions



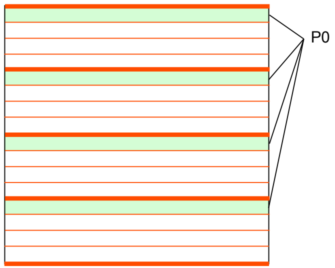
## Data distributions for geometric decomposition

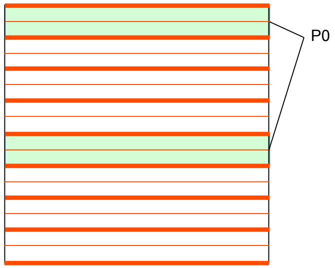
Block (left) and cyclic (right) data decompositions in a triangular iteration space



## Data distributions for geometric decomposition

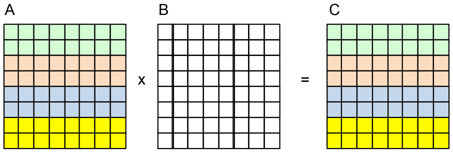
Cyclic (left) and block-cyclic (right) data decompositions





## Example: matrix multiply

```
void matmul (double C[MATSIZE][MATSIZE],
             double A[MATSIZE][MATSIZE],
             double B[MATSIZE][MATSIZE])
{
   for (int i=0; i<MATSIZE; i++)
      for (int j=0; j<MATSIZE; j++)
         for (int k=0; k<MATSIZE; k++)
            C[i][j] += A[i][k]*B[k][j];
}
```



$A$ and $C$ partitioned by rows on 4 processors (logically in shared memory architectures, physically in distributed memory architectures). $B$ is replicated.

## Example: matrix multiply (OpenMP)

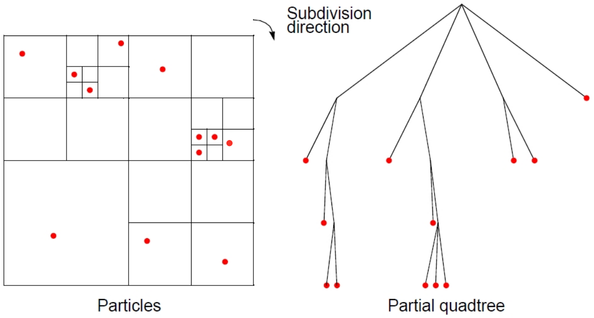
```
void matmul (double C[MATSIZE][MATSIZE],
             double A[MATSIZE][MATSIZE],
             double B[MATSIZE][MATSIZE])
{
   int i, j, k;

#pragma omp parallel
   {
   int myid = omp_get_thread_num();
   int numprocs = omp_get_num_threads();
   int i_start =  myid * (MATSIZE/numprocs);
   int i_end = i_start + (MATSIZE/numprocs);
   if (myid == numprocs-1) i_end = MATSIZE;

   for (int i=i_start; i<i_end; i++)
      for (int j=0; j<MATSIZE; j++)
         for (int k=0; k<MATSIZE; k++)
            C[i][j] += A[i][k]*B[k][j];
   }
}
```

## Example: matrix multiply (OpenMP)

```
void matmul (double C[MATSIZE][MATSIZE],
             double A[MATSIZE][MATSIZE],
             double B[MATSIZE][MATSIZE])
{
   int i, j, k;

#pragma omp parallel
   {
   int myid = omp_get_thread_num();
   int numprocs = omp_get_num_threads();
   int i_start =  myid * (MATSIZE/numprocs);
   int i_end = i_start + (MATSIZE/numprocs);
   int rem = MATSIZE % numprocs;
   if (rem != 0) {
       if (myid < rem) {
           i_start += myid;
           i_end += (myid+1);
           }
       else {
           i_start += rem;
           i_end += rem;
           }
       }
   ...
   }
}
```

## Data distributions for recursive decomposition

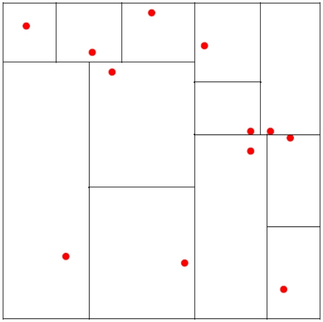
Quadtree to represent particles in an N-body problem



- Each leaf node stores position and mass for a body
- Other nodes store center of mass and total mass for all bodies below

## Data distributions for recursive decomposition

Orthogonal distribution of the particles of an N-body, so that in each bi-partition the number of particles in each side is halved (load balancing)



# Example: N-body computation (sequential)

## Sequential code

```
void main() {
 // Initialize tree
 for (t=0; t<tmax; t++) {
   for (i=0;i<N;i++) doTimeStep(tree, node[i]);  // node[i] points to body i in the tree
   // Update the positions and velocities
   // Migrate bodies if required in the tree
 }
}
```

## TreeNode structure

```
typedef struct {
   ...
   char     isLeaf
   TreeNode *quadrant[2][2];
   double   F; // force on node
   double   center_of_mass[3];
   double   mass_of_center;
   ...
} TreeNode;
```

## Calculate forces implementation

```
void doTimeStep(TreeNode* subTree, TreeNode* body) {
  if(subTree) {
    if(!subTree->isLeaf && !distant(subTree, body)) {
      for(int i=0; i<2; i++)
       for(int j=0; j<2; j++)
           doTimeStep(subTree->quadrant[i][j], body);
    }
    else // subtree is a leaf
      calcForces(subTree, body); // update F field for body
  }
}
```

A distant subtree is approximated as a single body with mass/center

## Example: N-body computation (data decomposition)

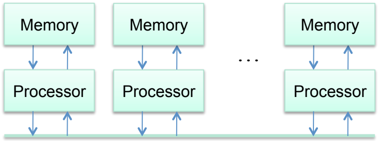
Each thread computes the forces in each node caused by the sub-tree assigned to it

```
void main() {
   // initialize tree
   ...
   #pragma omp parallel private(subtree) num_threads(4)
   {
      // Each thread will get a subtree
      subtree = partition(tree, omp_get_thread_num(), omp_get_num_threads());
      for (int t=0; t<tmax; t++) {
          for (int i=0;i<N;i++) doTimeStep(subtree, node[i]);
          // Update the positions and velocities
          ...
          if (...) {   // Migrate bodies if required in the quad-tree
              ...
              subtree = partition(tree, omp_get_thread_num(), omp_get_num_threads());
          }
      }
   }
}
```

## Outline

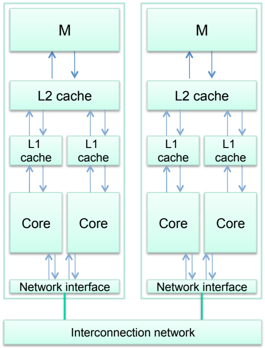
## Why hardware needs to provide data sharing?

- Simple design: distributed-memory architectures



- Each node can only access its own (local) memory hierarchy, through load/store instructions
  - No access to memory locations in other nodes
  - No cache coherency among nodes
- Interconnection network to exchange data between nodes through messages

## Why hardware needs to provide data sharing?

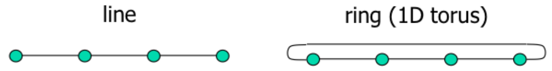


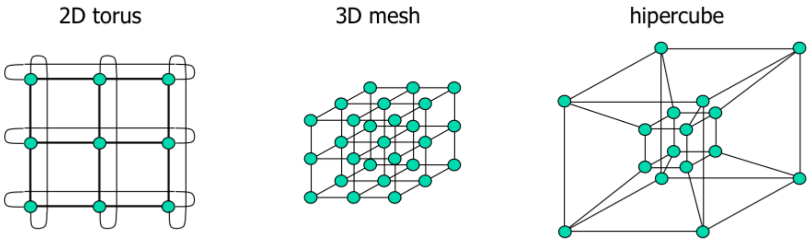
- Each node usually based on a shared–memory multiprocessor architecture (i.e. multi-socket and/or multicore)
- Network interface in each node to inject/retrieve messages to/from the interconnection network

## Interconnection networks

- Interconnection networks are build up of switching elements
  - Switches: devices that contain multiple input and output ports with a crossbar interconnection between them (i.e. any input to any output path available)
- Topology is the pattern in which the individual switches are connected to other switches and to processors and memories (nodes).
  - Direct topologies connect each switch directly to the network interface of a node
  - In indirect topologies at least some of the switches connect to other switches

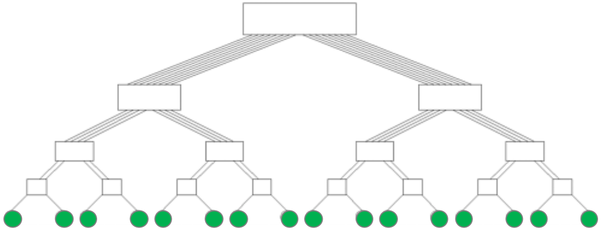
## Interconnection networks: direct topologies

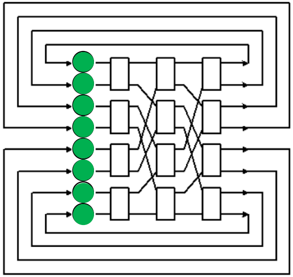




## Interconnection networks: indirect topologies

E.g. fat tree (left) and Omega multistage (right) networks





## Communication metrics

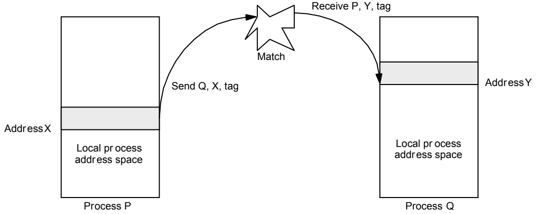
Network topology determines communication metrics (latency and bandwidth) and possibility of contention/congestion

- Latency: How long (e.g. microseconds) does a single data exchange take?
- Bandwidth: What data rate (e.g. Mbytes/sec.) can be sustained?

| Interconnect | Typical latency | Typical bandwidth |
|---|---|---|
| 100 Mbps Ethernet | 75 | 8 |
| 1Gbit/s Ethernet | 60-90 | 90 |
| 10 Gb/s Ethernet | 12-20 | 800 |
| Myricom Myrinet | 2.2-3 | 250-1200 |
| InfiniBand | 2-4 | 900-1400 |

## Communication model

Data exchange using send and receive primitives



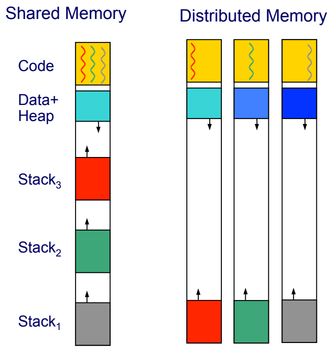
- Send specifies buffer to be sent and receiving process
- Receive specifies sending process and application storage to receive into
- Optional tag on send and matching rule on receive
- Optional implicit synchronization (e.g. blocking receive)

## Who does communication?

- Software DSM (distributed-shared memory)
  - Software layer that implements data sharing (and coherence)
  - Transparently to programmer
  - Usually based on page faults (OS involved, high overhead), which uses the communication model to move pages between nodes
- Compiler inserts communication based on programmer annotations (e.g. in Unified Parallel C – UPC)
- Message-passing paradigm (e.g. MPI standard)
  - User-level library exporting the communication model to the programmer, who moves data when necessary, assuming a data distribution

## Outline

## Distributed memory: address space



Programmer needs to

- Distribute work among tasks
- Distribute data among nodes
- Insert task interaction whenever necessary: communication to share data explicitly and synchronization to avoid data races

## Data allocation

- Tasks will access to data that is resident in the memory of the processor that executes the task
  - Specified through extensions in the language.

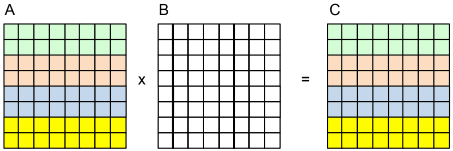
    For example in UPC Unified Parallel C:

    ```
    shared [2] int vector[16];  // block cyclic distribution with block size equals 2
    ```

  - Explicit memory allocation (In MPI standard)

## Example: matrix multiply in MPI

```
void matmul (double C[MATSIZE][MATSIZE],
             double A[MATSIZE][MATSIZE],
             double B[MATSIZE][MATSIZE])
{
   for (int i=0; i<MATSIZE; i++)
      for (int j=0; j<MATSIZE; j++)
         for (int k=0; k<MATSIZE; k++)
            C[i][j] += A[i][k]*B[k][j];
}
```



- Assume that process 0 initially stores $A$, $B$ and $C$ complete
- $A$ and $C$ are distributed by rows ($MATSIZE/nproc$ rows per process)
- $B$ is replicated

## Example: matrix multiply in MPI (cont.)

```
    ...
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &mpiRank);
    MPI_Comm_size(MPI_COMM_WORLD, &mpiSize);
    ...
    n = MATSIZE;
    n_local  = getRowCount(n, mpiRank, mpiSize);
    n_sq     = n * n;
    n_sq2    = n * n_local;
    ...
    A = (double *) malloc(sizeof(double) * (mpiRank ? n_sq2 : n_sq));
    B = (double *) malloc(sizeof(double) *                n_sq );
    C = (double *) malloc(sizeof(double) * (mpiRank ? n_sq2 : n_sq));
    ...
```

where

```
int getRowCount(int rowsTotal, int mpiRank, int mpiSize) {
    /* Adjust slack of rows in case rowsTotal is not exactly divisible */
    return (rowsTotal / mpiSize) + (rowsTotal % mpiSize > mpiRank);
}
```

## Task interaction

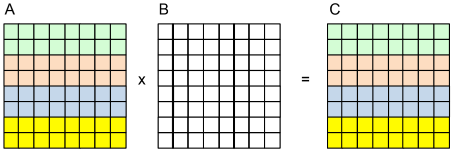
- Task interaction is necessary whenever a task needs an input (or part of it) that is assigned to another task or generates an output (or part of it) that is assigned to another task
- All task interactions (read-only, write-only or read/write) require cooperation (orchestration) of two processes: the task that has the data and the task that wants to access the data
- The message passing model provides the mechanisms to support task interaction

## Task interaction

- Interaction patterns
  - Point to Point (one to one)
  - Scatter and broadcast (one to all)
  - Gather and Reduce (all to one)
  - All to All (each processor sends its data to all others)
- Interactions may imply synchronization, i.e., process waits for interaction to happen (synchronous vs. asynchronous)
- Example: message-passing interface (MPI)

## Example: matrix multiply in MPI (cont.)

```
void matmul (double C[MATSIZE][MATSIZE],
             double A[MATSIZE][MATSIZE],
             double B[MATSIZE][MATSIZE])
{
   for (int i=0; i<MATSIZE; i++)
      for (int j=0; j<MATSIZE; j++)
         for (int k=0; k<MATSIZE; k++)
            C[i][j] += A[i][k]*B[k][j];
}
```



$A$ and $B$ initialised by process 0. Then $A$ distributed by rows ($MATSIZE/nproc$ rows per process) and $B$ replicated.

## Example: matrix multiply in MPI (cont.)

```
...
/* Initialize A and B using some functions */
if (!mpiRank) {
    ReadfromDisk(A, n_sq, 0); /* 0: from beginning; otherwise: from last element read */
    ReadfromDisk(B, n_sq, 0); /* 0: from beginning; otherwise: from last element read */
}

/* Send A by splitting it in row-wise parts */
if (!mpiRank) {
    currentRow = n_sq2;
    for (i=1; i<mpiSize; i++) {
        sizeToBeSent = n * getRowCount(n, i, mpiSize);
        MPI_Send(A + currentRow, sizeToBeSent, MPI_DOUBLE, i, TAG_INIT,
                 MPI_COMM_WORLD);
        currentRow += sizeToBeSent;
    }
}
else { /* Receive parts of A */
    MPI_Recv(A, n_sq2, MPI_DOUBLE, 0, TAG_INIT, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
}
...
```

## Example: matrix multiply in MPI (cont.)

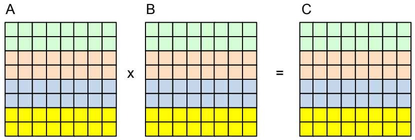
```
...
/* Replicate complete B in each process */
if (!mpiRank) {
    for (i=1; i<mpiSize; i++) {
        MPI_Send(B, n_sq, MPI_DOUBLE, i, TAG_INIT, MPI_COMM_WORLD);
    }
}
else { /* Receive B in each other process  */
    MPI_Recv(B, n_sq, MPI_DOUBLE, 0, TAG_INIT, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
}

/* Let each process initialize C to zero */
for (i=0; i<n_sq2; i++)
    C[i] = 0.0;

/* And finally ... let each process perform its own multiplications */
for (i=0; i<nlocal; i++)
  for (j=0; j<MATSIZE; j++)
     for (k=0; k<MATSIZE; k++)
        C[i][j] += A[i][k]*B[k][j];
...
```

## Example: matrix multiply in MPI (cont.)

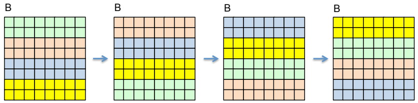
```
...
/* Receive partial results from each slave */
if (!mpiRank) {
    currentRow = n_sq2;
    for (i=1; i<mpiSize; i++) {
        sizeToBeSent = n * getRowCount(n, i, mpiSize);
        MPI_Recv(C + currentRow, sizeToBeSent, MPI_DOUBLE, i, TAG_RESULT,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        currentRow += sizeToBeSent;
    }
}
else /* Send partial results to master */
    MPI_Send(C, n_sq2, MPI_DOUBLE, 0, TAG_RESULT, MPI_COMM_WORLD);

MPI_Finalize();
...
```

## Matrix multiply: stronger memory size constraints

- What if $B$ does not fit completely in worker processors?

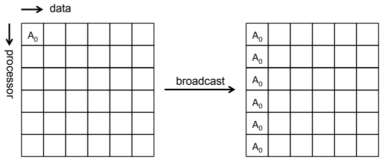


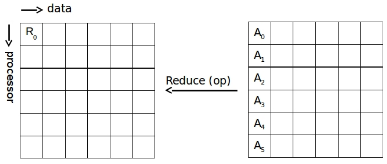
We need to circulate $B$ among processors in order to do the complete product.

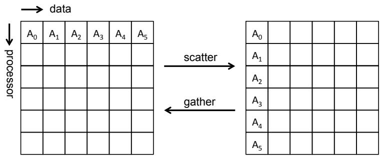


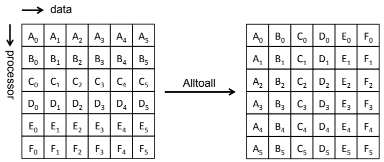
- What if master has the same memory size constraints as workers?

## Collective communications









| | | | | | |
|---|---|---|---|---|---|
| $A_0$ | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ |
| $B_0$ | $B_1$ | $B_2$ | $B_3$ | $B_4$ | $B_5$ |
| $C_0$ | $C_1$ | $C_2$ | $C_3$ | $C_4$ | $C_5$ |
| $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $D_5$ |
| $E_0$ | $E_1$ | $E_2$ | $E_3$ | $E_4$ | $E_5$ |
| $F_0$ | $F_1$ | $F_2$ | $F_3$ | $F_4$ | $F_5$ |

| | | | | | |
|---|---|---|---|---|---|
| $A_0$ | $B_0$ | $C_0$ | $D_0$ | $E_0$ | $F_0$ |
| $A_1$ | $B_1$ | $C_1$ | $D_1$ | $E_1$ | $F_1$ |
| $A_2$ | $B_2$ | $C_2$ | $D_2$ | $E_2$ | $F_2$ |
| $A_3$ | $B_3$ | $C_3$ | $D_3$ | $E_3$ | $F_3$ |
| $A_4$ | $B_4$ | $C_4$ | $D_4$ | $E_4$ | $F_4$ |
| $A_5$ | $B_5$ | $C_5$ | $D_5$ | $E_5$ | $F_5$ |

## Using collectives in matrix multiply

```
...
/* Replicate complete B in each process */
if (!mpiRank) {
    for (i=1; i<mpiSize; i++) {
        MPI_Send(B, n_sq, MPI_DOUBLE, i, TAG_INIT, MPI_COMM_WORLD);
    }
}
else { /* Receive B in each other process  */
    MPI_Recv(B, n_sq, MPI_DOUBLE, 0, TAG_INIT, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
}
...
```

Using a single collective, assuming same number of rows per processor:

```
...
/* Replicate complete B in each process */
MPI_Bcast(B, n_sq, MPI_DOUBLE, 0, MPI_COMM_WORLD);
...
```

## Using collectives in matrix multiply (cont.)

```
...
/* Send A by splitting it in row-wise parts */
if (!mpiRank) {
    currentRow = n_sq2;
    for (i=1; i<mpiSize; i++) {
        sizeToBeSent = n * getRowCount(n, i, mpiSize);
        MPI_Send(A + currentRow, sizeToBeSent, MPI_DOUBLE, i, TAG_INIT,
                 MPI_COMM_WORLD);
        currentRow += sizeToBeSent;
    }
}
else { /* Receive parts of A */
    MPI_Recv(A, n_sq2, MPI_DOUBLE, 0, TAG_INIT, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
}
...
```

Using a single collective, assuming same number of rows per processor:

```
...
sizeToBeSent = n * n/mpiSize;
MPI_Scatter(A,sizeToBeSent,MPI_DOUBLE,A,sizeToBeSent,MPI_DOUBLE,0,MPI_COMM_WORLD)
...
```

## Using collectives in matrix multiply (cont.)

```
...
/* Receive partial results from each slave */
if (!mpiRank) {
    currentRow = n_sq2;
    for (i=1; i<mpiSize; i++) {
        sizeToBeSent = n * getRowCount(n, i, mpiSize);
        MPI_Recv(C + currentRow, sizeToBeSent, MPI_DOUBLE, i, TAG_RESULT,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        currentRow += sizeToBeSent;
    }
}
else /* Send partial results to master */
    MPI_Send(C, n_sq2, MPI_DOUBLE, 0, TAG_RESULT, MPI_COMM_WORLD);
...
```

Using a single collective, assuming same number of rows per processor:

```
...
sizeToBeSent = n * n/mpiSize;
MPI_Gather(C,sizeToBeSent,MPI_DOUBLE,C,sizeToBeSent,MPI_DOUBLE,0,MPI_COMM_WORLD)
...
```

## Minimizing interaction overheads

- Minimize volume of data exchange because of the cost associated with each word that is communicated
- Minimize frequency of interactions because of the startup cost associated with each interaction (try to merge multiple interactions into one, where possible)
- Overlap computations with interactions by using non-blocking communications
  - Non-blocking operations (`MPI_Isend` and `MPI_Irecv`) return (immediately) a "request handler" that can be tested and waited on

## Minimizing interaction overheads (cont.)

- Use collective communications instead of point-to-point primitives (also programming simplicity)
- Minimize contention and hot-spots: Use decentralized techniques, replicate data where necessary
- Replicate computations

# Parallelism (PAR)

## Parallel programming principles: Data decomposition

Eduard Ayguadé, Julita Corbalán,
Daniel Jiménez and Gladys Utrera

Computer Architecture Department
Universitat Politècnica de Catalunya

Course 2017/18 (Spring semester)