CS 3006 Parallel and Distributed Computer

Fall 2022

1. Learn about parallel and distributed computer architectures.(1)
2. Implement different parallel and distributed programming paradigms and algorithms using Message-Passing Interface (MPI) and OpenMP.(4)
3. Perform analytical modelling, dependence, and performance analysis of parallel algorithms and programs.(2)
4. Use Hadoop or MapReduce programming model to write bigdata applications.(5)

Week # 9 – Lecture # 22, 23, 24

20th, 22nd, 23rd Rabi ul Awwal, 1444

17th, 19th, 20th October 2022

Dr. Nadeem Kafi Khan

Lecture # 22 – Topics (Lab # 9)

- Review of MPI_Send/Recv
- MPI_Get_Count – to get more info. using MPI_Recv return status
- Parallizing the Trapezoidal Rule
- Collective Operation like MPI_Broadcast, MPI_Reduce, etc.
- MPI_Reduce and MPI_AllReduce
- Collective vs Point-to-Point communication
- Non-Blocking Send and Receive. MPI_Wait and MPI_Test
- Reading and Distributing a vector. MPI_Scatter, MPI_Gather, MPI_AllGather

```
int MPI_Send(
    void*        msg_buf  /* input */,
    int          msg_size /* input */,
    MPI_Datatype msg_type /* input */,
    int          dest     /* input */,
    int          tag      /* input */,
    MPI_Comm     comm     /* input */)

int MPI_Recv(
    void*        msg_buf  /* output */,
    int          msg_size /* input */,
    MPI_Datatype msg_type /* input */,
    int          source   /* input */,
    int          tag      /* input */,
    MPI_Comm     comm     /* input */,
    MPI_Status*  status_p /* output */)
```

- MPI_Send may behave differently with regard to buffer size, cutoffs and blocking.
- MPI_Recv always blocks until a matching message is received.

Data types

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG	signed long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

```
int MPI_Get_count(
        MPI_Status*   status_p   /* in   */,
        MPI_Datatype  type       /* in   */,
        int*          count_p    /* out  */);
```

```
const int MAX_NUMBERS = 100;
int numbers[MAX_NUMBERS];
int number_amount;

if (world_rank == 0) {
  srand(time(NULL));
  number_amount = (rand() / (float) RAND_MAX) * MAX_NUMBERS;
  MPI_Send(numbers, number_amount, MPI_INT, 1, 0, MPI_COMM_WORLD);
  printf("0 sent %d numbers to 1\n", number_amount);
} else if (world_rank == 1) {
  MPI_Status status;
  MPI_Recv(numbers, MAX_NUMBERS, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
  MPI_Get_count( & status, MPI_INT, & number_amount); //check how many actually recv.
  printf("1 received %d numbers from 0. Message source = %d, tag = %d\n",
        number_amount, status.MPI_SOURCE, status.MPI_TAG);
}
```

Parallelizing the Trapezoidal Rule

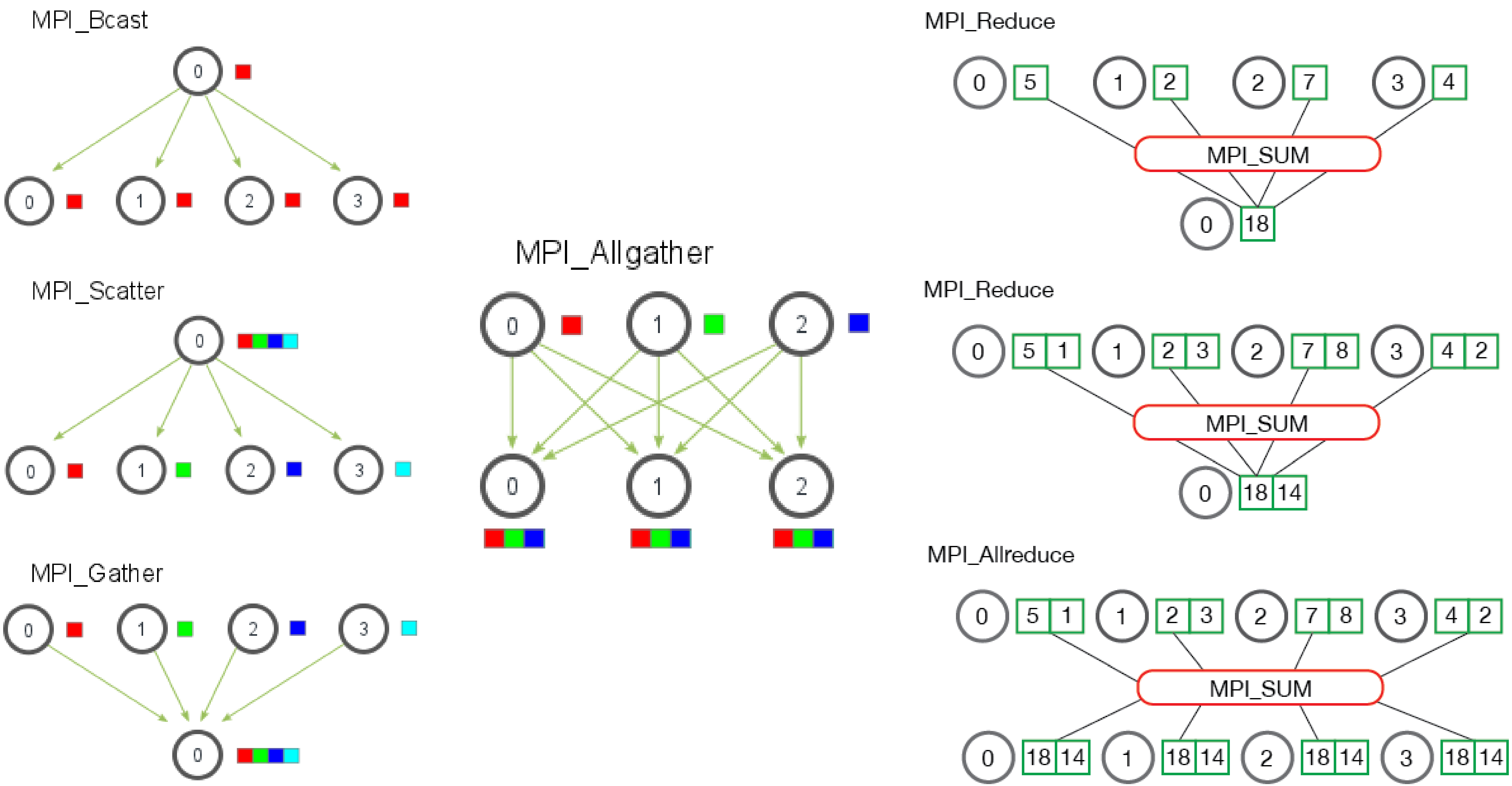
1. Partition problem solution into tasks.
2. Identify communication channels between tasks.
3. Aggregate tasks into composite tasks.
4. Map composite tasks to cores.

```
Get a, b, n;
h = (b−a)/n;
local_n = n/comm_sz;
local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;
local_integral = Trap(local_a, local_b, local_n, h);
if (my_rank != 0)
   Send local_integral to process 0;
else /* my_rank == 0 */
   total_integral = local_integral;
   for (proc = 1; proc < comm_sz; proc++) {
      Receive local_integral from proc;
      total_integral += local_integral;
   }
}
if (my_rank == 0)
   print result;
```

- All collective operations must be called by *all* processes in the communicator
- MPI_Bcast is called by both the sender (called the root process) and the processes that are to receive the broadcast
  - MPI_Bcast is not a "multi-send"
  - "root" argument is the rank of the sender; this tells MPI which process originates the broadcast and which receive

```
if (my_rank == 0) {
   printf("Enter a, b, and n\n");
   scanf("%lf %lf %d", a_p, b_p, n_p);
}
MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Seen by all processes
master and slaves


MPI_Bcast
0
0
1
2
3
MPI_Scatter
0
0
1
2
3
MPI_Gather
0
1
2
3
0
MPI_Allgather
0
1
2
0
1
2
MPI_Reduce
0
5
1
2
2
7
3
4
MPI_SUM
0
18
MPI_Reduce
0
5
1
1
2
3
2
7
8
3
4
2
MPI_SUM
0
18
14
MPI_Allreduce
0
5
1
1
2
3
2
7
8
3
4
2
MPI_SUM
0
18
14
1
18
14
2
18
14
3
18
14

MPI_Reduce

```
int MPI_Reduce(
      void*          input_data_p   /* in   */,
      void*          output_data_p  /* out  */,
      int            count          /* in   */,
      MPI_Datatype   datatype       /* in   */,
      MPI_Op         operator       /* in   */,
      int            dest_process   /* in   */,
      MPI_Comm       comm           /* in   */);
```

Operation Value	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

```
MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0,
     MPI_COMM_WORLD);
```

```
double local_x[N], sum[N];
. . .
MPI_Reduce(local_x, sum, N, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
```

Collective vs. Point-to-Point Communications

- All the processes in the communicator must call the same collective function.

- For example, a program that attempts to match a call to MPI_Reduce on one process with a call to MPI_Recv on another process is erroneous, and, in all likelihood, the program will hang or crash.

- The arguments passed by each process to an MPI collective communication must be "compatible."

- For example, if one process passes in 0 as the dest_process and another passes in 1, then the outcome of a call to MPI_Reduce is erroneous, and, once again, the program is likely to hang or crash.

- Point-to-point communications are matched on the basis of tags and communicators.
- Collective communications don't use tags.
- They're matched solely on the basis of the communicator and the order in which they're called.

MPI_Allreduce

- Useful in a situation in which all of the processes need the result of a global sum in order to complete some larger computation.

```
int MPI_Allreduce(
        void*          input_data_p   /* in  */,
        void*          output_data_p  /* out */,
        int            count          /* in  */,
        MPI_Datatype   datatype       /* in  */,
        MPI_Op         operator       /* in  */,
        MPI_Comm       comm           /* in  */);
```

Non-blocking is not part of the syllabus. You are encourage to include it in your Semester Project code.

Non-blocking Send

```
int MPI_Isend(
    void*          msg_buf  /* input */,
    int            msg_size /* input */,
    MPI_Datatype   msg_type /* input */,
    int            dest     /* input */,
    int            tag      /* input */,
    MPI_Comm       comm     /* input */,
    MPI_Request   &request /* output */)
```

Identical to MPI_Send but added argument for getting handle to MPI_Request struct.

Non-blocking Receive

```
int MPI_Irecv(
    void*          msg_buf  /* output */,
    int            msg_size /* input */,
    MPI_Datatype   msg_type /* input */,
    int            source   /* input */,
    int            tag      /* input */,
    MPI_Comm       comm     /* input */,
    MPI_Request   &request /* output */)
```

Identical to MPI_Read but replaces MPI_Status argument with MPI_Request struct. Status is now retrieved in the MPI_Wait.

Non-blocking Test

```
int MPI_Test(
  MPI_Request   request  /* input */,
  int           &flag    /* output */
  MPI_Status    &status  /* output */)
```

Flag returns 1 if completed, otherwise 0.

Blocking Wait

```
int MPI_Wait(
  MPI_Request  request /* input */,
  MPI_Status   &status /* output */)
```

Same status that would have been returned from MPI_Recv, not as useful in MPI_Send.

```
MPI_Issend(buf,count,datatype,dest,tag,comm,request);
do {
    MPI_Test(request,flag,status);
} while (flag != 1);
```

```
MPI_Isend (buf, count, datatype, dest, tag, comm, request);
MPI_Wait (request, status);
if (status....) {
```

Scatter

- MPI_Scatter can be used in a function that reads in an entire vector on process 0 but only sends the needed components to each of the other processes.

```
int MPI_Scatter(
      void*          send_buf_p   /* in  */,
      int            send_count   /* in  */,
      MPI_Datatype   send_type    /* in  */,
      void*          recv_buf_p   /* out */,
      int            recv_count   /* in  */,
      MPI_Datatype   recv_type    /* in  */,
      int            src_proc     /* in  */,
      MPI_Comm       comm         /* in  */);
```

Gather

- Collect all of the components of the vector onto process 0, and then process 0 can process all of the components.

```
int MPI_Gather(
      void*          send_buf_p   /* in  */,
      int            send_count   /* in  */,
      MPI_Datatype   send_type    /* in  */,
      void*          recv_buf_p   /* out */,
      int            recv_count   /* in  */,
      MPI_Datatype   recv_type    /* in  */,
      int            dest_proc    /* in  */,
      MPI_Comm       comm         /* in  */);
```

Reading and distributing a vector

```
void Read_vector(
      double    local_a[]   /* out */,
      int       local_n     /* in  */,
      int       n           /* in  */,
      char      vec_name[]  /* in  */,
      int       my_rank     /* in  */,
      MPI_Comm  comm        /* in  */) {

   double* a = NULL;
   int i;
```

```
if (my_rank == 0) {
   a = malloc(n*sizeof(double));
   printf("Enter the vector %s\n", vec_name);
   for (i = 0; i < n; i++)
      scanf("%lf", &a[i]);
   MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE,
         0, comm);
   free(a);
} else {
   MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE,
         0, comm);
}
```

Print a distributed vector

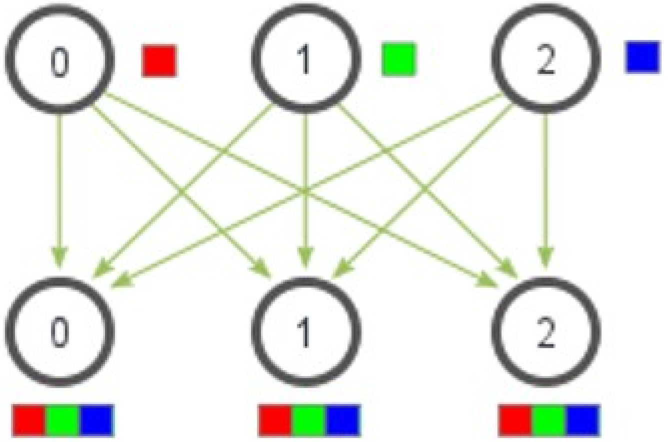
```
void Print_vector(
      double    local_b[]  /* in */,
      int       local_n    /* in */,
      int       n          /* in */,
      char      title[]    /* in */,
      int       my_rank    /* in */,
      MPI_Comm  comm       /* in */) {

   double* b = NULL;
   int i;
```

```
if (my_rank == 0) {
   b = malloc(n*sizeof(double));
   MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE,
         0, comm);
   printf("%s\n", title);
   for (i = 0; i < n; i++)
      printf("%f ", b[i]);
   printf("\n");
   free(b);
} else {
   MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE,
         0, comm);
}
```

Allgather

- Concatenates the contents of each process' send_buf_p and stores this in each process' recv_buf_p.
- As usual, recv_count is the amount of data being received from each process.

MPI_Allgather

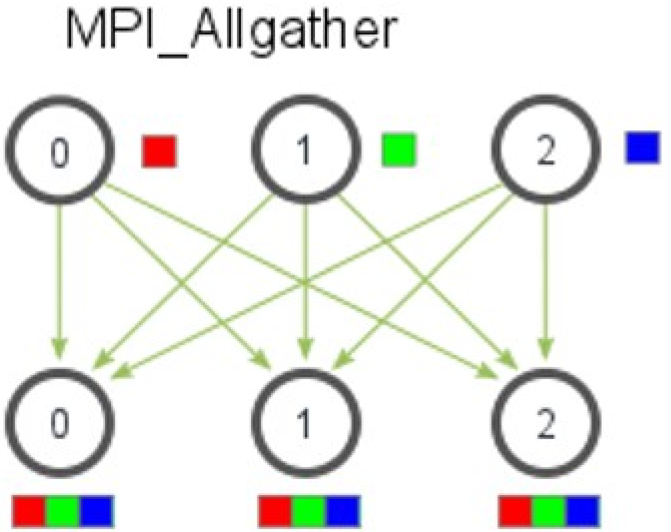



```
int MPI_Allgather(
      void*          send_buf_p   /* in  */,
      int            send_count   /* in  */,
      MPI_Datatype   send_type    /* in  */,
      void*          recv_buf_p   /* out */,
      int            recv_count   /* in  */,
      MPI_Datatype   recv_type    /* in  */,
      MPI_Comm       comm         /* in  */);
```

Matrix-vector multiplication

$$\begin{array}{|c|c|c|c|}\hline a_{00} & a_{01} & \cdots & a_{0,n-1} \\ \hline a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\ \hline \end{array} \quad \begin{array}{|c|}\hline x_0 \\ \hline x_1 \\ \hline \vdots \\ \hline x_{n-1} \\ \hline \end{array} = \begin{array}{|c|}\hline y_0 \\ \hline y_1 \\ \hline \vdots \\ \hline y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\ \hline \vdots \\ \hline y_{m-1} \\ \hline \end{array}$$

```
void Mat_vect_mult(
      double    local_A[]  /* in  */,
      double    local_x[]  /* in  */,
      double    local_y[]  /* out */,
      int       local_m    /* in  */,
      int       n          /* in  */,
      int       local_n    /* in  */,
      MPI_Comm  comm       /* in  */) {
   double* x;
   int local_i, j;
   int local_ok = 1;
   x = malloc(n*sizeof(double));
   MPI_Allgather(local_x, local_n, MPI_DOUBLE,
         x, local_n, MPI_DOUBLE, comm);

   for (local_i = 0; local_i < local_m; local_i++) {
      local_y[local_i] = 0.0;
      for (j = 0; j < n; j++)
         local_y[local_i] += local_A[local_i*n+j]*x[j];
   }
   free(x);
}  /* Mat_vect_mult */
```



$$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1}$$

Lecture # 23 – Topics

- Interaction among Tasks running of different processors
- Task Interaction Graph
    - Capturing data dependencies
- Task Interaction graph- An example
    - Sparse Matrix Vector Multiplication
    - Task interaction graph
- Processes and Mapping
- Criteria for Mapping
- Mapping data query to processes – Example

Task Interaction Graphs

- Subtasks generally exchange data with others in a decomposition.
  - For example, even in the trivial decomposition of the dense matrix-vector product, if the vector is not replicated across all tasks, they will have to communicate elements of the vector.

- The graph of tasks (nodes) and their interactions/data exchange (edges) is referred to as a *task interaction graph*.

  - *Task interaction graphs* represent data dependencies,
  - *Task dependency graphs* represent control dependencies.

The pattern of interaction among tasks is captured by what is known as a ***task-interaction graph***. The nodes in a task-interaction graph represent tasks and the edges connect tasks that interact with each other. The nodes and edges of a task-interaction graph can be assigned weights proportional to the amount of computation a task performs and the amount of interaction that occurs along an edge, if this information is known. The edges in a task-interaction graph are usually undirected, but directed edges can be used to indicate the direction of flow of data, if it is unidirectional. The edge-set of a task-interaction graph is usually a superset of the edge-set of the task-dependency graph. In the database query example discussed earlier, the task-interaction graph is the same as the task-dependency graph. We now give an example of a more interesting task-interaction graph that results from the problem of sparse matrix-vector multiplication.

Example 3.3 Sparse matrix-vector multiplication

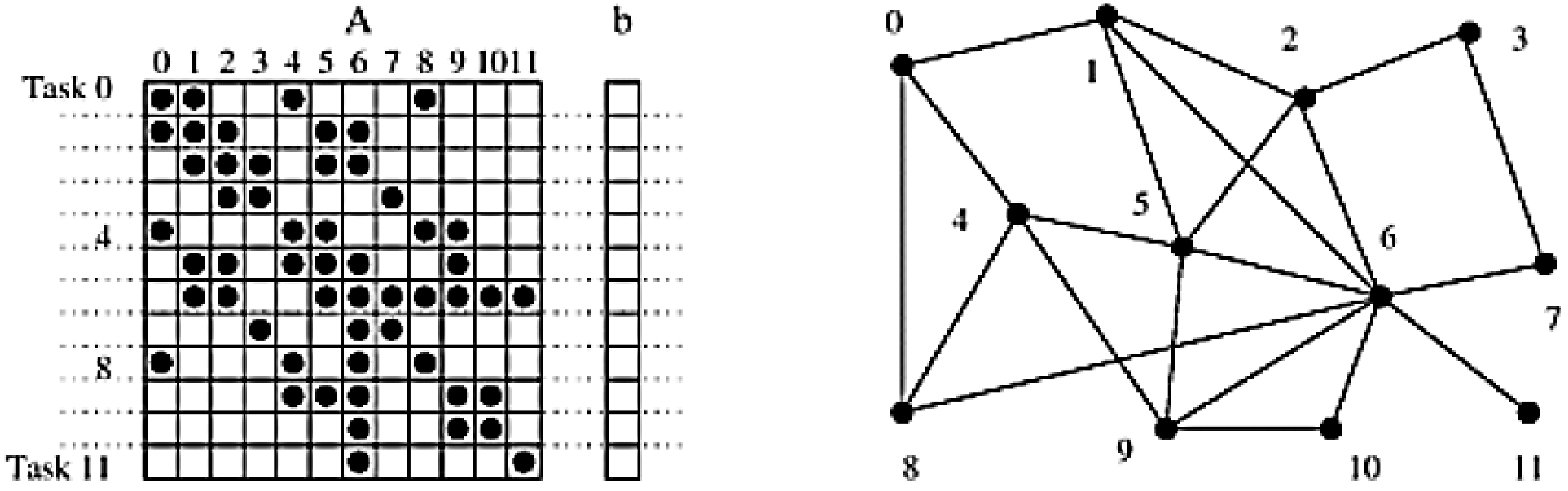
Consider the problem of computing the product $y = Ab$ of a sparse $n \times n$ matrix A with a dense $n \times 1$ vector b. A matrix is considered sparse when a significant number of entries in it are zero and the locations of the non-zero entries do not conform to a predefined structure or pattern. Arithmetic operations involving sparse matrices can often be optimized significantly by avoiding computations involving the zeros.

One possible way of decomposing this computation is to partition the output vector y and have each task compute an entry in it. Figure 3.6(a) illustrates this decomposition. In addition to assigning the computation of the element $y[i]$ of the output vector to Task i, we also make it the "owner" of row $A[i, *]$ of the matrix and the element $b[i]$ of the input vector. Note that the computation of $y[i]$ requires access to many elements of b that are owned by other tasks. So Task i must get these elements from the appropriate locations. In the message-passing paradigm, with the ownership of $b[i]$,Task i also inherits the responsibility of sending $b[i]$ to all the other tasks that need it for their computation. For example, Task 4 must send $b[4]$ to Tasks 0, 5, 8, and 9 and must get $b[0]$, $b[5]$, $b[8]$, and $b[9]$ to perform its own computation. The resulting task-interaction graph is shown in Figure 3.6(b). ■

Task Interaction graph- An example

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} ax + by + cz + dw \\ ex + fy + gz + hw \\ ix + jy + kz + lw \\ mx + ny + oz + pw \end{bmatrix}$$

Figure 3.6. A decomposition for sparse matrix-vector multiplication and the corresponding task-interaction graph. In the decomposition Task *i* computes $\sum_{0 \le j \le 11, A[i,j] \ne 0} A[i,j].b[j]$**.**



(a) (b)

Processes and Mapping

- **Mapping**: the mechanism by which tasks are assigned to processes for execution.
- **Process**: a logic computing agent that performs tasks, which is an abstract entity that uses the code and data corresponding to a task to produce the output of that task.
- Why use processes rather than processors?
  - We rely on OS to map processes to physical processors.
  - We can aggregate tasks into a process

Criteria of Mapping

1. Maximize the use of concurrency by mapping independent tasks onto different processes
2. Minimize the total completion time by making sure that processes are available to execute the tasks on critical path as soon as such tasks become executable
3. Minimize interaction among processes by mapping tasks with a high degree of mutual interaction onto the same process.

Basis for Choosing Mapping

Task-dependency graph

Makes sure the max. concurrency

Task-interaction graph

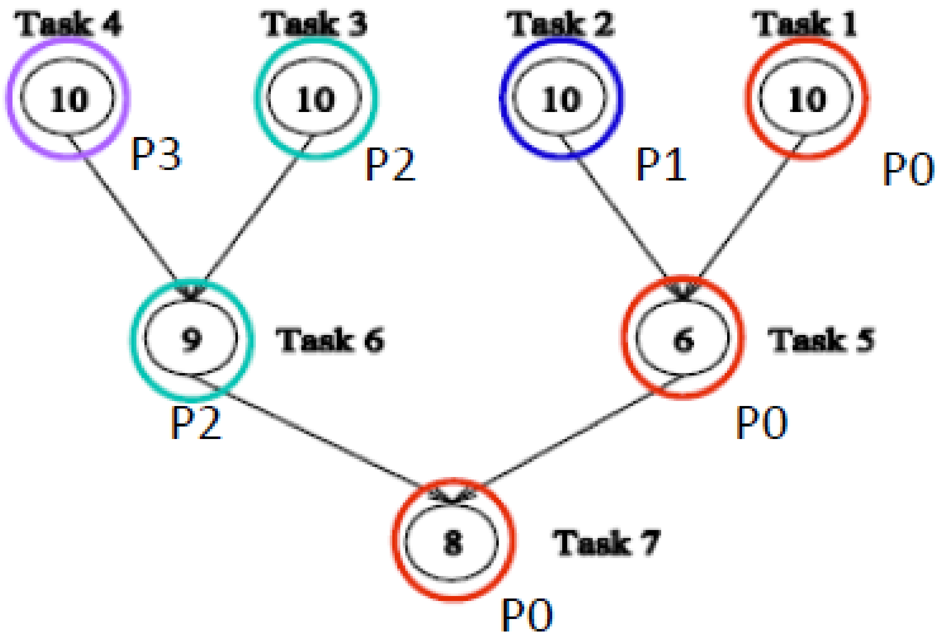
Minimum communication.

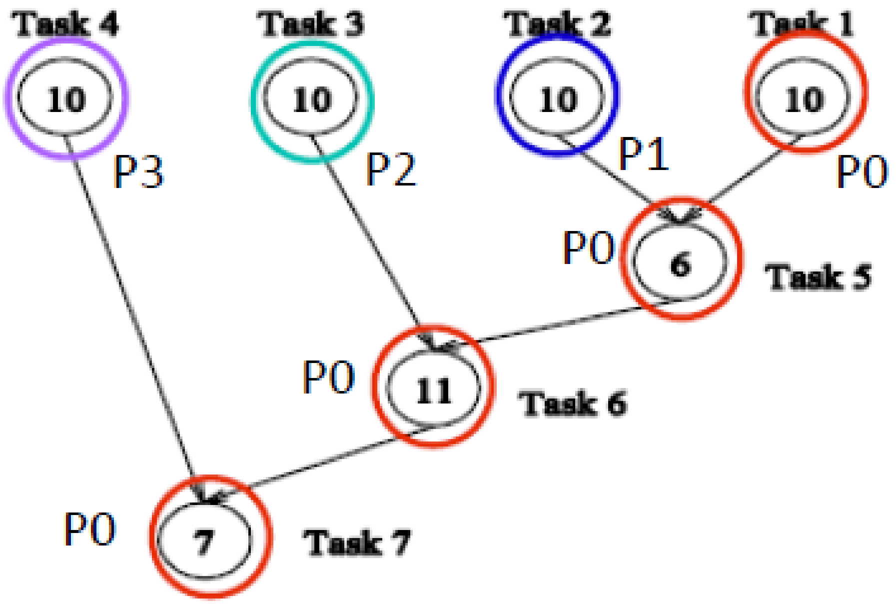
These criteria often conflict with each other.

For example, a decomposition into one task (or no decomposition at all) minimizes interaction but does not result in a speedup at all!

- Finding a balance that optimizes the overall parallel performance is the key to a successful parallel algorithm.
- Therefore, mapping of tasks onto processes plays an important role in determining how efficient the resulting parallel algorithm is.

Example: Mapping Database Query to Processes





No two nodes in a level have dependencies, therefore, single level tasks are assigned to different processes.

- 4 processes can be used in total since the max. concurrency is 4.
- Assign all tasks within a level to different processes.

Lecture # 24 – Topics

- Decomposition Techniques
  - Recursive Decomposition
  - Data Decomposition
  - Exploratory Decomposition
  - Speculative Decomposition

Decomposition Techniques

So how does one decompose a task into various subtasks?

While there is no single recipe that works for all problems, we present a set of commonly used techniques that apply to broad classes of problems. These include:

- **recursive decomposition**
- **data decomposition**
- **exploratory decomposition**
- **speculative decomposition**

Recursive Decomposition

- Generally suited to problems that are solved using the divide-and-conquer strategy.
- A given problem is first decomposed into a set of sub-problems.
- These sub-problems are recursively decomposed further until a desired granularity is reached.

Recursive Decomposition: Example

A classic example of a divide-and-conquer algorithm on which we can apply recursive decomposition is Quicksort.

Task generation is dynamic and the task size is non-uniform.

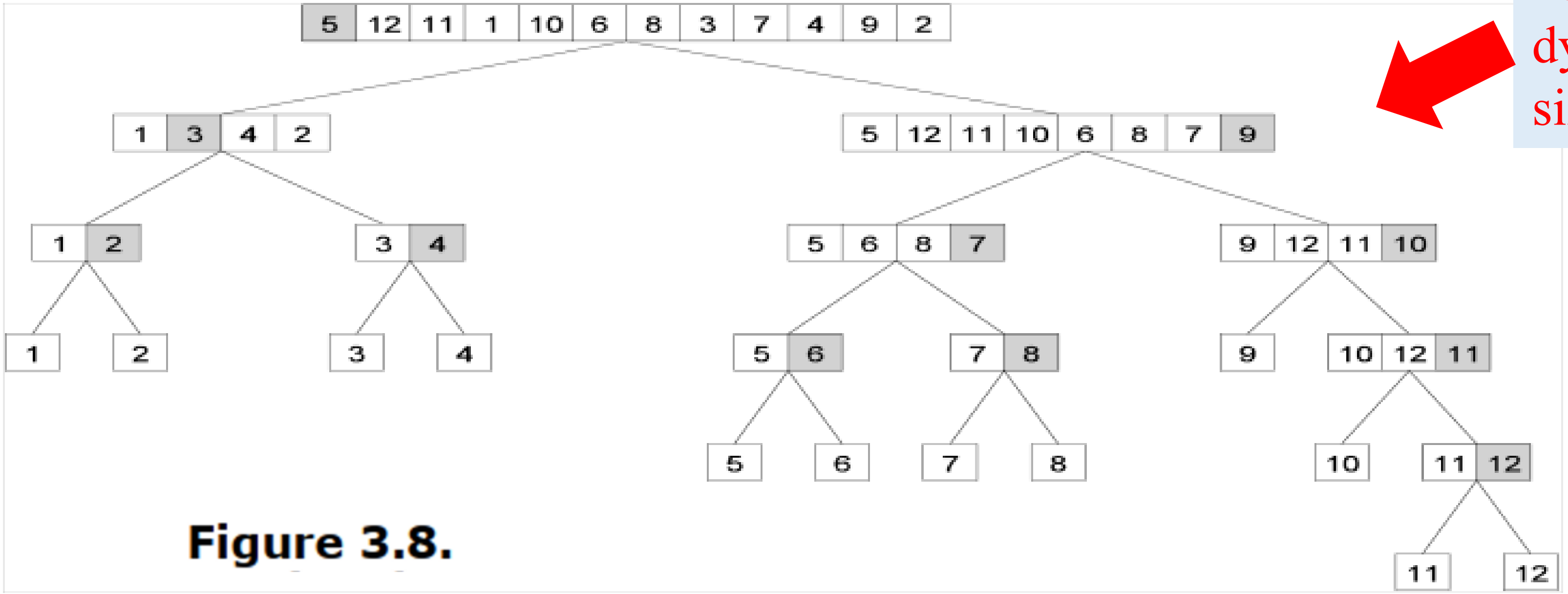


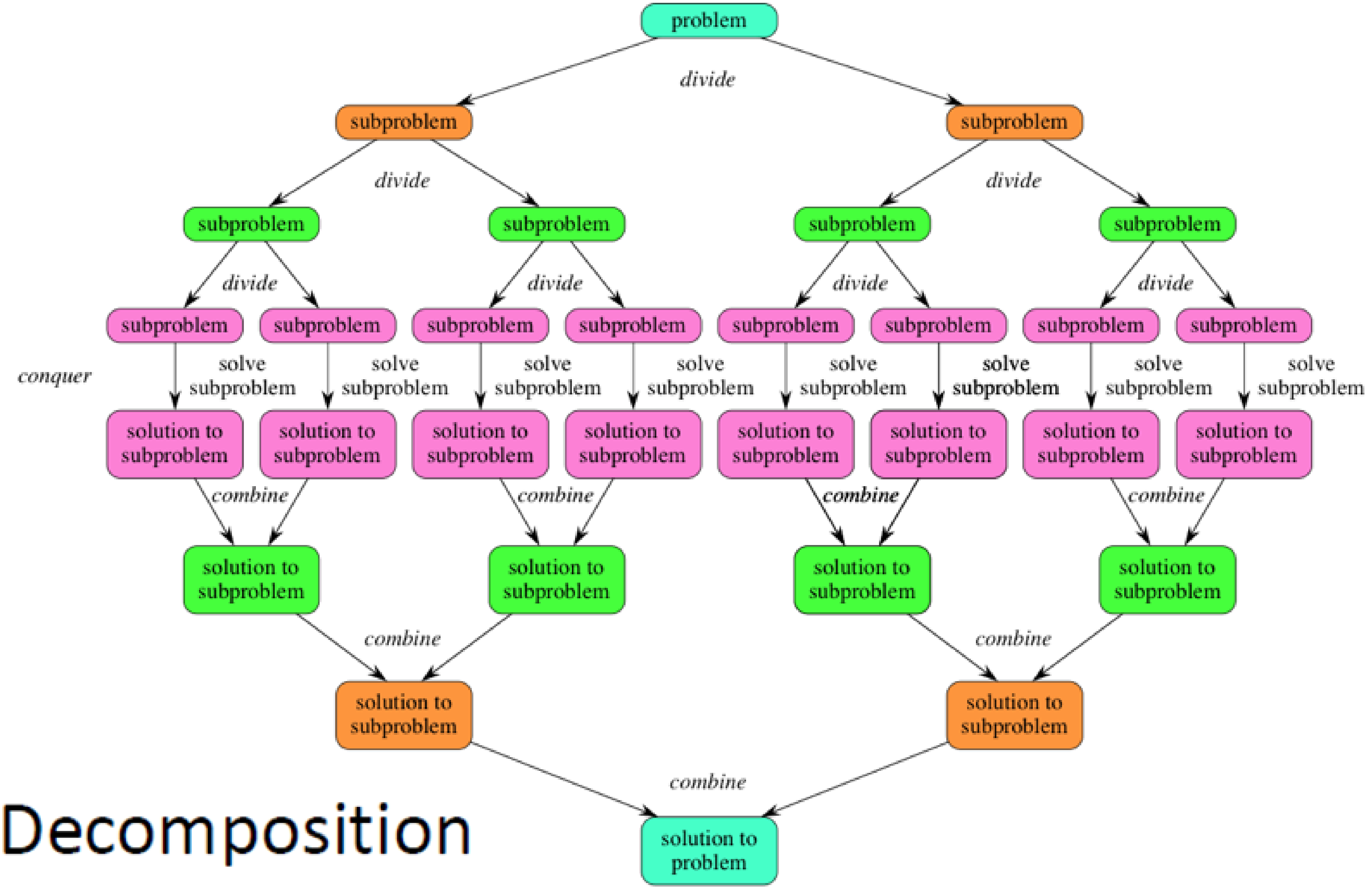
Figure 3.8.

In this example, a task represents the work of partitioning a (sub)array. Note that each subarray represents an independent subtask. This can be repeated recursively.

https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/divide-and-conquer-algorithms

You should think of a divide-and-conquer algorithm as having three parts:

1. **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
2. **Conquer** the subproblems by solving them recursively. If they are small enough, solve as base cases.
3. **Combine** the solutions to the subproblems into the solution for the original problem.



Recursive Decomposition

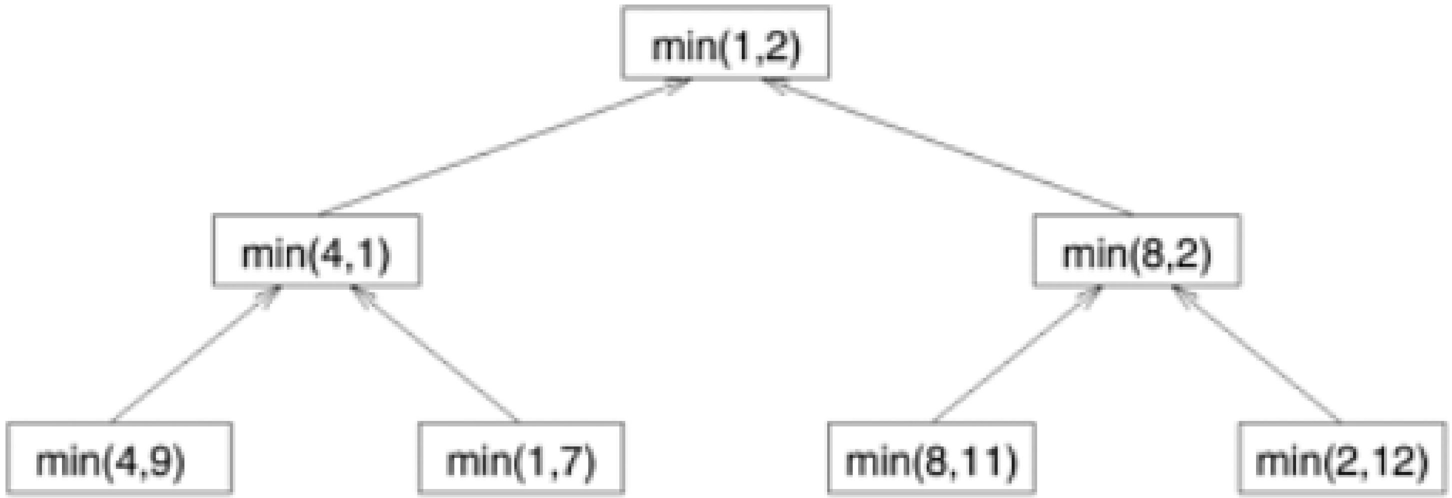
Algorithm 3.1 A serial program for finding the minimum in an array of numbers *A* of length *n*.

```
procedure SERIAL_MIN (A, n)
begin
min = A[0];
for i := 1 to n - 1 do
    if (A[i] < min) min := A[i];
endfor;
return min;
end SERIAL_MIN
```

Algorithm 3.2 A recursive program for finding the minimum in an array of numbers *A* of length *n*.

```
procedure RECURSIVE_MIN (A, n)
begin
if (n = 1) then
    min := A[0];
else
    lmin := RECURSIVE_MIN (A, n/2);
    rmin := RECURSIVE_MIN (&(A[n/2]), n - n/2);
    if (lmin < rmin) then
        min := lmin;
    else
        min := rmin;
    endelse;
endelse;
return min;
end RECURSIVE_MIN
```

Figure 3.9. The task-dependency graph for finding the minimum number in the sequence {4, 9, 1, 7, 8, 11, 2, 12}. Each node in the tree represents the task of finding the minimum of a pair of numbers.



Data Decomposition

- *Ideal for problems that operate on large data structures*
- **Steps**
  1. The data on which the computations are performed are partitioned
  2. Data partition is used to induce a partitioning of the computations into tasks.
- Data Partitioning
  – Partition output data
  – Partition input data
  – Partition input + output data
  – Partition intermediate data

Data Decomposition: Output Data Decomposition

- Often, each element of the output can be computed independently of others (but simply as a function of the input).
- A partition of the output across tasks decomposes the problem naturally.

- Input: if each output is described as a function of the input directly. Some combination of the individual results may be necessary.
- Output data decomposition: if it applies, it can result in less communication.
- Intermediate data decomposition more rare.
- Owner computes rules: the process that owns a part of the data performs all the computations related to it.

Output Data Decomposition: Example

Consider the problem of multiplying two $n \times n$ matrices A and B to yield matrix C. The output matrix C can be partitioned into four tasks as follows:

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

Task 1: $C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$

Task 2: $C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$

Task 3: $C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$

Task 4: $C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

Output Data Decomposition: Example

A given data decomposition does not result in a unique decomposition into tasks.

Figure 3.11. Two examples of decomposition of matrix multiplication into eight tasks.

Decomposition I	Decomposition II
Task 1: $C_{1,1} = A_{1,1}B_{1,1}$	Task 1: $C_{1,1} = A_{1,1}B_{1,1}$
Task 2: $C_{1,1} = C_{1,1} + A_{1,2}B_{2,1}$	Task 2: $C_{1,1} = C_{1,1} + A_{1,2}B_{2,1}$
Task 3: $C_{1,2} = A_{1,1}B_{1,2}$	Task 3: $C_{1,2} = A_{1,2}B_{2,2}$
Task 4: $C_{1,2} = C_{1,2} + A_{1,2}B_{2,2}$	Task 4: $C_{1,2} = C_{1,2} + A_{1,1}B_{1,2}$
Task 5: $C_{2,1} = A_{2,1}B_{1,1}$	Task 5: $C_{2,1} = A_{2,2}B_{2,1}$
Task 6: $C_{2,1} = C_{2,1} + A_{2,2}B_{2,1}$	Task 6: $C_{2,1} = C_{2,1} + A_{2,1}B_{1,1}$
Task 7: $C_{2,2} = A_{2,1}B_{1,2}$	Task 7: $C_{2,2} = A_{2,1}B_{1,2}$
Task 8: $C_{2,2} = C_{2,2} + A_{2,2}B_{2,2}$	Task 8: $C_{2,2} = C_{2,2} + A_{2,2}B_{2,2}$

Output Data Decomposition: Example

A given data decomposition does not result in a unique decomposition into tasks.

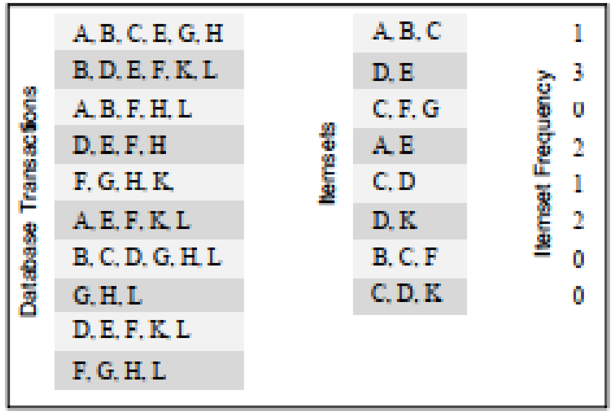
Figure 3.11. Two examples of decomposition of matrix multiplication into eight tasks.

Decomposition I	Decomposition II
Task 1: $C_{1,1} = A_{1,1}B_{1,1}$	Task 1: $C_{1,1} = A_{1,1}B_{1,1}$
Task 2: $C_{1,1} = C_{1,1} + A_{1,2}B_{2,1}$	Task 2: $C_{1,1} = C_{1,1} + A_{1,2}B_{2,1}$
Task 3: $C_{1,2} = A_{1,1}B_{1,2}$	Task 3: $C_{1,2} = A_{1,2}B_{2,2}$
Task 4: $C_{1,2} = C_{1,2} + A_{1,2}B_{2,2}$	Task 4: $C_{1,2} = C_{1,2} + A_{1,1}B_{1,2}$
Task 5: $C_{2,1} = A_{2,1}B_{1,1}$	Task 5: $C_{2,1} = A_{2,2}B_{2,1}$
Task 6: $C_{2,1} = C_{2,1} + A_{2,2}B_{2,1}$	Task 6: $C_{2,1} = C_{2,1} + A_{2,1}B_{1,1}$
Task 7: $C_{2,2} = A_{2,1}B_{1,2}$	Task 7: $C_{2,2} = A_{2,1}B_{1,2}$
Task 8: $C_{2,2} = C_{2,2} + A_{2,2}B_{2,2}$	Task 8: $C_{2,2} = C_{2,2} + A_{2,2}B_{2,2}$

Output Data Decomposition: Example

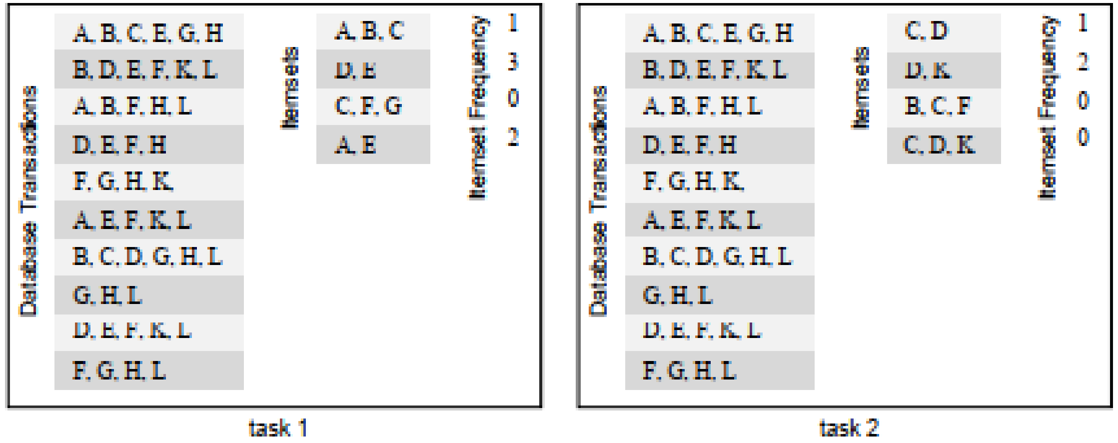
Consider the problem of counting the instances of given itemsets in a database of transactions. In this case, the output (itemset frequencies) can be partitioned across tasks.

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



Problem: Find the number of times that each itemset in I appears in all the transactions; i.e., the number of transactions of which each itemset is a subset of.

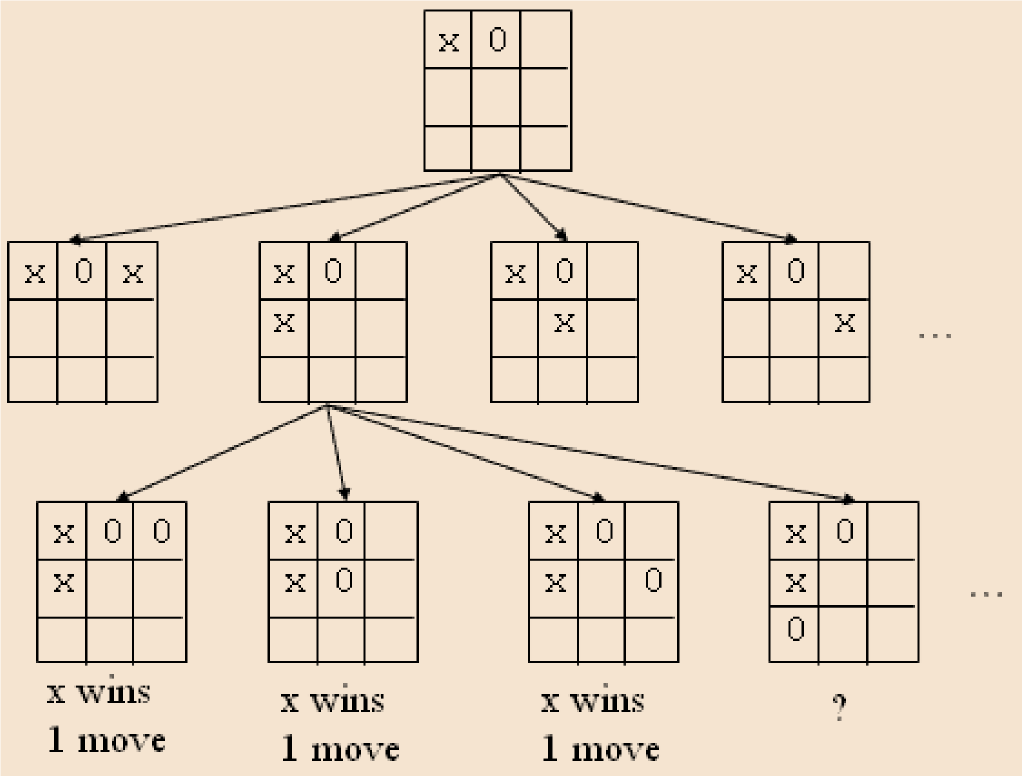
(b) Partitioning the frequencies (and itemsets) among the tasks



Part (b) shows how two tasks can achieve results by partitioning the output into two parts and having each task compute its half of the frequencies.

Exploratory Decomposition

- Example: **looking for the best move in a game**.
- Simple case: generate all possible configurations from the starting position.
- Send each of the configurations to a child process.
- *Each process will look for possible best moves for the opponent recursively eventually using more processes*.
- When it finds the result, it sends it back to the parent.
- *The parent selects the best move from all of the results received from the child* (eventually the worst move for the opponent).



Speculative Decomposition

- Switch statement in a program: We wait to know the value of the expression and execute only the corresponding case.
- In speculative decomposition *we execute some or all of the cases in advance.*
- When the value of the expression is know, *we keep only the results from the computation to be executed in that case*.
- The *gain in performance comes from anticipating the possible computations*.

Sequential version	Parallel version

<pre>compute expr;
switch (expr) {
  case 1:
    compute a1;
    break;
  case 2:
    compute a2;
    break;
  case 3:
    compute a3;
    break;...
}</pre>
	<pre>Slave(i)
	{
	  compute ai;
	  Wait(request);
	  if (request)
	    Send(ai, 0);
	}
	Master()
	{
	  compute expr;
	  swicth (expr) {
	    case 1:
	      Send(request, 1);
	      Receive(a1, i);
	  ...
	}</pre>

The difference with the exploratory decomposition is that we can compute the possible states before the next move is performed.