X. Decomposition and Orchestration

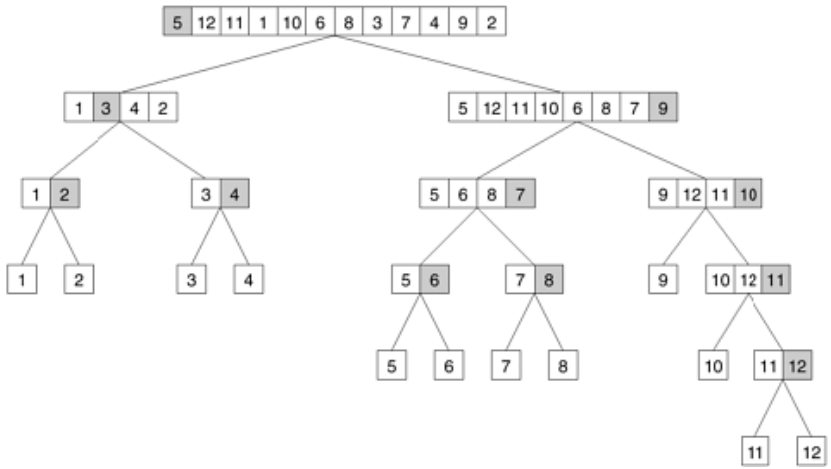
April 27th, 2009

Content

- Decompositions:
  - recursive,
  - data,
  - exploratory,
  - speculative and
  - hybrid
- Orchestration under the
  - data parallel,
  - shared-address space and
  - message passing model

Recursive Decomposition

- Is a method for inducing concurrency in problems that can be solved using the divide-and-conquer strategy.
  - A problem is solved by first dividing it into a set of independent subproblems.
  - Each one of these subproblems is solved by recursively applying a similar division into smaller subproblems followed by a combination of their results.
  - The divide-and-conquer strategy results in natural concurrency, as different subproblems can be solved concurrently.
- Example: Quicksort.
  - Sorting a sequence *A* of *n* elements using the commonly used quicksort algorithm.
  - We define a task as the work of partitioning a given subsequence.
  - Fig also represents the task graph for the problem.



Recursive decomp.: finding the minimum

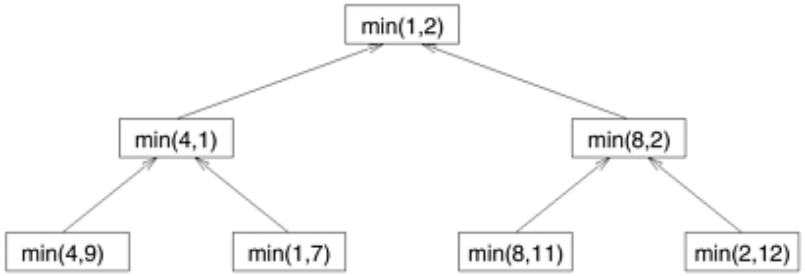
- Sometimes, it is possible to restructure a computation to make it amenable to recursive decomposition even if the commonly used algorithm for the problem is not based on the divide-and-conquer strategy.
- Example: the problem of finding the minimum element in an unordered sequence A of n elements.
  - The serial algorithm for solving this problem scans the entire sequence A, recording at each step the minimum element found so far as illustrated in the following serial algorithm.
  - Once we restructure this computation as a divide-and-conquer algorithm, we can use recursive decomposition to extract concurrency.
  - 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
```

Recursive decomp.: finding the minimum

- In this algorithm,
  - *A* is split into two subsequences, each of size *n*/2,
  - find the minimum for each of these subsequences by performing a recursive call.
  - Now the overall minimum element is found by selecting the minimum of these two subsequences.
  - The recursion terminates when there is only one element left in each subsequence.
- It is easy to construct a task-dependency graph for this problem.
- Fig. illustrates a task-dependency graph for finding the minimum of eight numbers where each task is assigned the work of finding the minimum of two numbers.

- 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

- Is a powerful and commonly used method for deriving concurrency in algorithms that operate on large data structures.
- The decomposition of computations is done in two steps.
  1. The data on which the computations are performed is partitioned,
  2. This data partitioning is used to induce a partitioning of the computations into tasks.
- The operations that these tasks perform on different data partitions are usually similar (e.g., matrix multiplication that follows) or are chosen from a small set of operations (e.g., LU factorization).
- One must explore and evaluate all possible ways of partitioning the data and determine which one yields a natural and efficient computational decomposition.
- ***Partitioning Output Data.***
  - In many computations, each element of the output can be computed independently of others as a function of the input.
  - In such computations, a partitioning of the output data automatically induces a decomposition of the problems into tasks
    - each task is assigned the work of computing a portion of the output.

Example 1 for partitioning the output data

- **Matrix multiplication**
- Consider the problem of multiplying two *n* x *n* matrices *A* and *B* to yield a matrix *C*.
- Fig. shows a decomposition of this problem into four tasks.
- Each matrix is considered to be composed of 4 blocks or sub-matrices defined by splitting each dim. of the matrix into half.
- The 4 submatrices of *C*, roughly of size *n*/2 x *n*/2 each, are then independently computed by 4 tasks as the sums of the appropriate products of submatrices of A and B.

(a) Partitioning of input and output matrices into 2 x 2 submatrices.

(b) A decomposition of matrix multiplication into four tasks based on the partitioning of the matrices

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

(a)

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

(b)

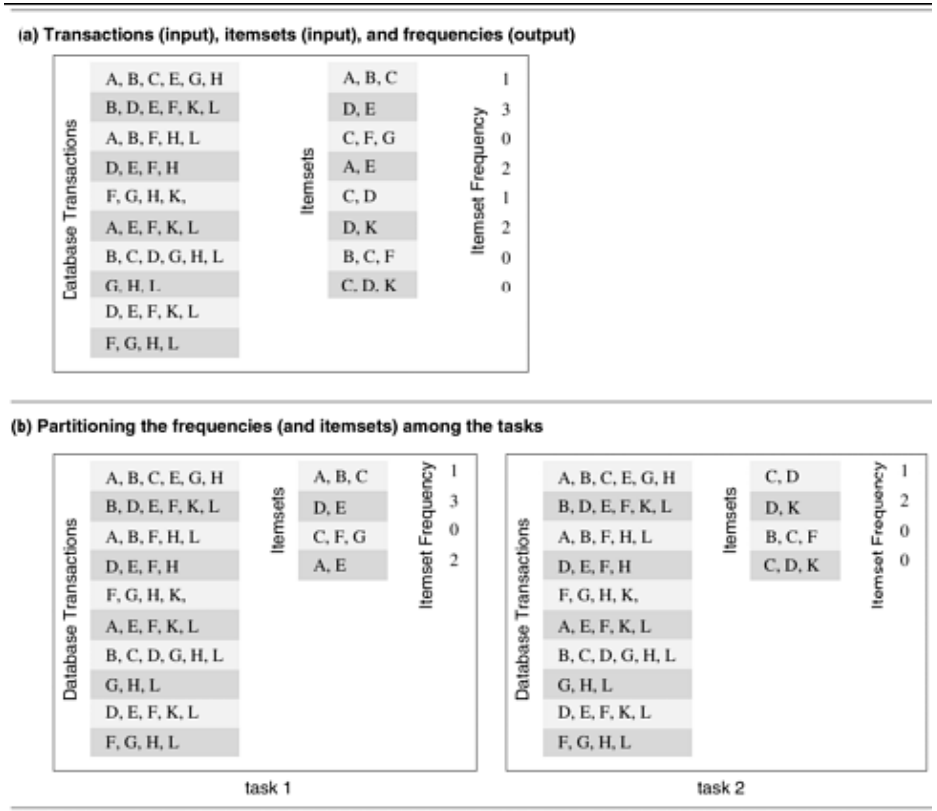
Example 1 for partitioning the output data

- Most matrix algs, including matrix-vector and matrix-matrix multiplication, can be formulated in terms of block matrix operations.
  - The matrix is viewed as composed of blocks or submatrices and the scalar arithmetic ops on its elements are replaced by the equivalent matrix ops on the blocks.
  - The results of the element and the block versions of the algorithm are mathematically equivalent.
  - Block versions of matrix algorithms are often used to aid decomposition.
- Data-decomposition is distinct from the decomposition of the computation into tasks.
  - Although the two are often related, a given data-decomposition does not result in a unique decomposition into tasks.
  - Example, Fig. shows two other decompositions of matrix multiplication, each into eight tasks, corresponding to the same data-decomposition as used in previous Fig.

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

Example 2 for partitioning the output data

- **Computing frequencies of itemsets in a transaction database.**
- Given a set *T* containing *n* transactions and a set *I* containing *m* itemsets.
- Each transaction and itemset contains a small no. items, out of a possible set of items.
- Example: *T* is a grocery stores database of customer sales with each transaction being an individual grocery list of a shopper & itemset could be a group of items in the store.
  - If the store desires to find out how many customers bought each of the designated groups of items, then it would need to find the no. times that each itemset in *I* appears in all the transactions (the no. transactions of which each itemset is a subset of)
- Fig. (a) shows an example
  - The database shown consists of 10 transactions, and
  - We are interested in computing the frequency of the 8 itemsets shown in the second column.
  - The actual frequencies of these itemsets in the database (output) are shown in the third column.
  - For instance, itemset {D, K} appears twice, once in the second and once in the ninth transaction.
- Fig. (b) shows how the computation of frequencies of the itemsets can be decomposed into 2 tasks
  - by partitioning the output into 2 parts and having each task compute its half of the frequencies.

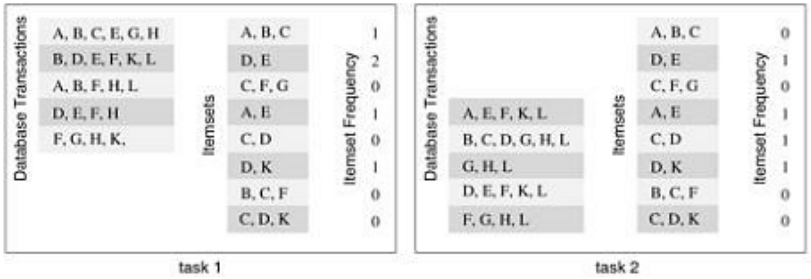


Partitioning input data

- Remark: Partitioning of output data can be performed only if each output can be naturally computed as a function of the input.
  - In many algorithms, it is not possible or desirable to partition the output data.
  - For example:
    - While finding the minimum, maximum, or the sum of a set of numbers, the output is a single unknown value.
    - In a sorting algorithm, the individual elements of the output cannot be efficiently determined in isolation.
- It is sometimes possible to partition the input data, and then use this partitioning to induce concurrency.
  - A task is created for each partition of the input data and this task performs as much computation as possible using these local data.
  - Solutions to tasks induced by input partitions may not directly solve original probl.
    - In such cases, a follow-up computation is needed to combine the results.
- Example: finding the sum of N numbers using p processes ($N > p$):
  - we can partition the input into p subsets of nearly equal sizes.
  - Each task then computes the sum of the numbers in one of the subsets.
  - Finally, the p partial results can be added up to yield the final result.

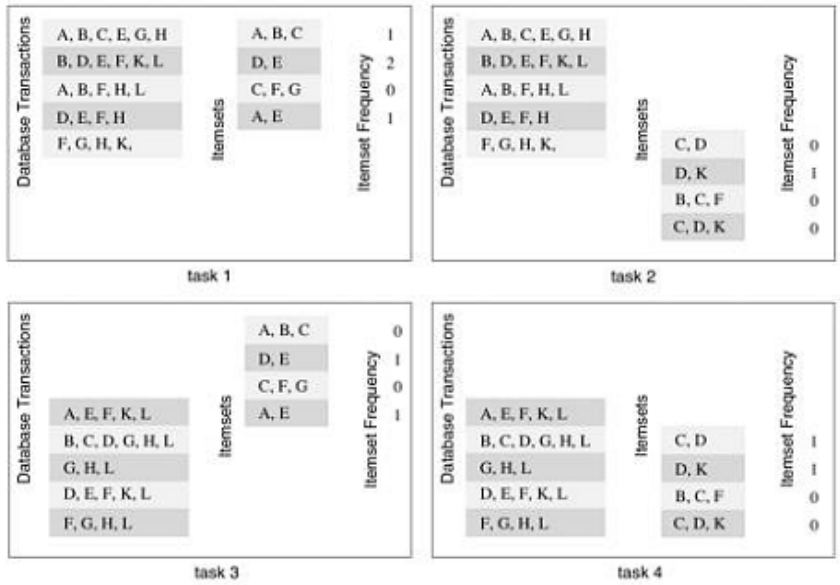
Example for input data partitioning

- The problem of computing the frequency of a set of itemsets
- Can also be decomposed based on a partitioning of input data.
- Fig. shows a decomposition based on a partitioning of the input set of transactions.
  - Each of the two tasks computes the frequencies of all the itemsets in its respective subset of transactions.
  - The two sets of frequencies, which are the independent outputs of the two tasks, represent intermediate results.
  - Combining the intermediate results by pairwise addition yields the final result.



Partitioning both Input and Output Data

- In some cases, in which it is possible to partition the output data, partitioning of input data can offer additional concurrency.
- Example: consider the 4-way decomposition shown in Fig. for computing itemset frequencies.
  - both the transaction set and the frequencies are divided into two parts and a different one of the four possible combinations is assigned to each of the four tasks.
  - Each task then computes a local set of frequencies.
  - Finally, the outputs of Tasks 1 and 3 are added together, as are the outputs of Tasks 2 and 4.

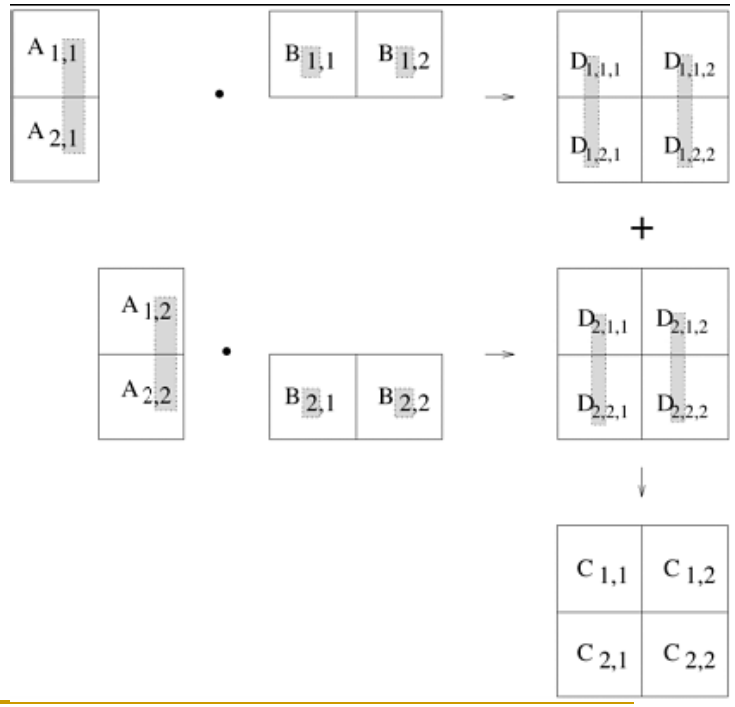


Partitioning Intermediate Data

- Partitioning intermediate data can sometimes lead to higher concurrency than partitioning input or output data.
- Often, the intermediate data are not generated explicitly in the serial alg. for solving the problem

⇒ Some restructuring of the original alg. may be required to use intermediate data partitioning to induce a decomposition.

- Example: matrix multiplication
  - Recall that the decompositions induced by a 2 x 2 partitioning of the output matrix *C* have a maximum degree of concurrency of four.
  - Increase the degree of concurrency by introducing an intermediate stage in which 8 tasks compute their respective product submatrices and store the results in a temporary 3-d matrix *D*, as shown in Fig.
  - Submatrix *Dk,i,j* is the product of *Ai,k* and *Bk,j*.



Partitioning Intermediate Data - example

- A partitioning of the intermediate matrix *D* induces a decomposition into eight tasks. Decomposition (see Fig.)
- After the multiplication phase, a relatively inexpensive matrix addition step can compute the result matrix C.
- All submatrices *D*,i,j* with the same second and third dimensions *i* and *j* are added to yield *Ci,j*.
- The eight tasks numbered 1 through 8 in Fig. perform $O(n^3/8)$ work each in multiplying *n*/2 x *n*/2 submatrices of *A* and *B*.
- Then, four tasks numbered 9 through 12 spend O(*n*2/4) time each in adding the appropriate *n*/2 x *n*/2 submatrices of the intermediate matrix *D* to yield the final result matrix *C*.
- Second Fig. shows the task-dependency graph

Stage I

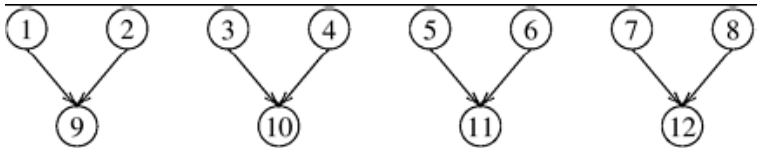
$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} \begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \end{pmatrix} \\ \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,2}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

Stage II

$$\begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \end{pmatrix} + \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,2} & D_{2,2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

A decomposition induced by a partitioning of D

$$\begin{aligned}
\text{Task 01:} &\quad D_{1,1,1} = A_{1,1}B_{1,1} \\
\text{Task 02:} &\quad D_{2,1,1} = A_{1,2}B_{2,1} \\
\text{Task 03:} &\quad D_{1,1,2} = A_{1,1}B_{1,2} \\
\text{Task 04:} &\quad D_{2,1,2} = A_{1,2}B_{2,2} \\
\text{Task 05:} &\quad D_{1,2,1} = A_{2,1}B_{1,1} \\
\text{Task 06:} &\quad D_{2,2,1} = A_{2,2}B_{2,1} \\
\text{Task 07:} &\quad D_{1,2,2} = A_{2,1}B_{1,2} \\
\text{Task 08:} &\quad D_{2,2,2} = A_{2,2}B_{2,2} \\
\text{Task 09:} &\quad C_{1,1} = D_{1,1,1} + D_{2,1,1} \\
\text{Task 10:} &\quad C_{1,2} = D_{1,1,2} + D_{2,1,2} \\
\text{Task 11:} &\quad C_{2,1} = D_{1,2,1} + D_{2,2,1} \\
\text{Task 12:} &\quad C_{2,2} = D_{1,2,2} + D_{2,2,2}
\end{aligned}$$

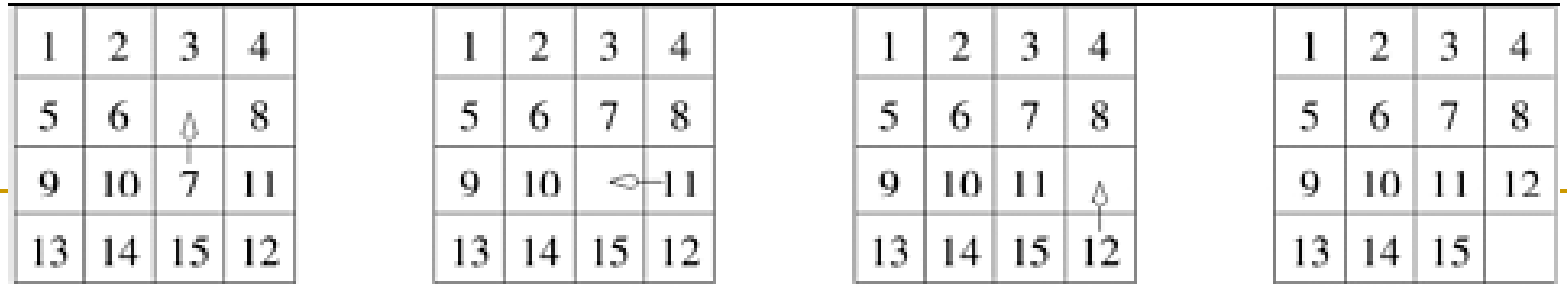


Data decomp.: the Owner-Computes Rule

- A decomposition based on partitioning output or input data is also widely referred to as the owner-computes rule.
- The idea behind this rule is that each partition performs all the computations involving data that it owns.
- Depending on the nature of the data or the type of data-partitioning, the owner-computes rule may mean different things:
  - When we assign partitions of the input data to tasks, then the owner-computes rule means that a task performs all the computations that can be done using these data.
  - If we partition the output data, then the owner-computes rule means that a task computes all the data in the partition assigned to it.

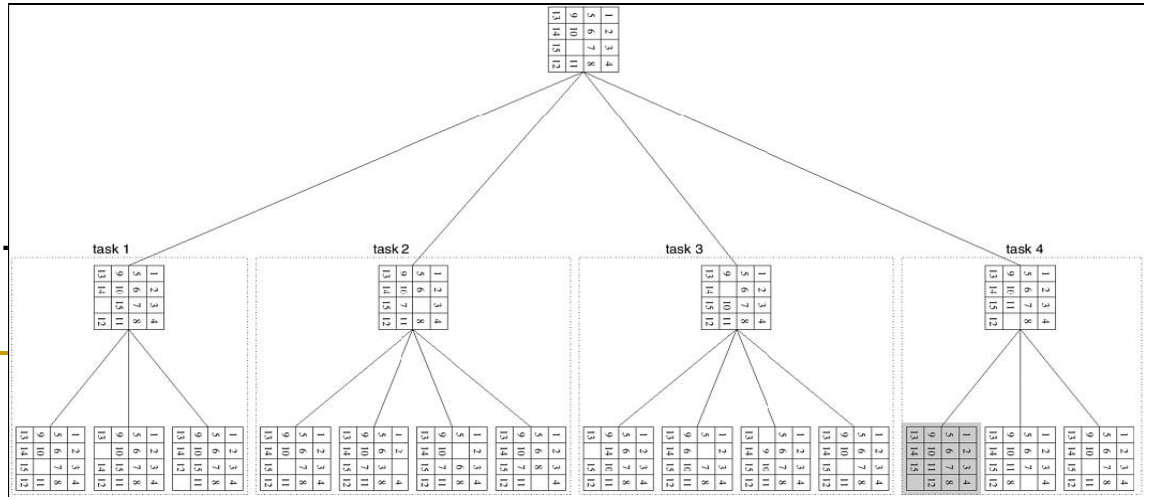
Exploratory Decomposition

- Is used to decompose problems whose underlying computations correspond to a search of a space for solutions.
- Partition the search space into smaller parts, and search each one of these parts concurrently, until the desired solutions are found.
- Example: consider the 15-puzzle problem.
  - Consists of 15 tiles numbered 1 through 15 and one blank tile placed in a 4 x 4 grid.
  - A tile can be moved into the blank position from a position adjacent to it, thus creating a blank in the tile's original position.
  - Four moves are possible: up, down, left, and right.
  - The initial and final configurations of the tiles are specified.
  - The objective is to determine any sequence or a shortest sequence of moves that transforms the initial configuration to the final configuration.
  - Fig. illustrates sample initial and final configurations and a sequence of moves leading from the initial configuration to the final configuration.



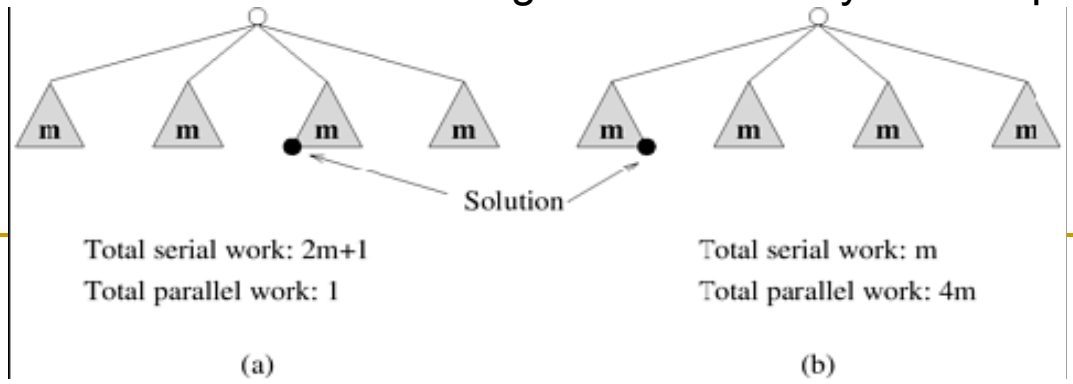
Ex. Exploratory Decomposition: puzzle

- The puzzle is typically solved using tree-search techniques.
  - Starting from the initial configuration, all possible successor configurations are generated.
  - A configuration may have 2, 3, or 4 possible successor configurations, each corresponding to the occupation of the empty slot by one of its neighbors.
  - The task of finding a path from initial to final configuration now translates to finding a path from one of these newly generated configurations to the final configuration.
  - Since one of these newly generated configurations must be closer to the solution by one move (if a solution exists), we have made some progress towards finding the solution.
- The configuration space generated by the tree search is the *state space graph*.
  - Each node of the graph is a configuration and each edge of the graph connects configurations that can be reached from one another by a single move of a tile.
- One method for solving this problem in parallel:
  - First, a few levels of configurations starting from the initial configuration are generated serially until the search tree has a sufficient number of leaf nodes
  - Now each node is assigned to a task to explore further until at least one of them finds a sol.
  - As soon as one of the concurrent tasks finds a solution it can inform the others to terminate their searches.
- Figure illustrates
  one such decomposition
  into four tasks
  in which task 4 finds the solution.



Exploratory vs. data decomposition

- The tasks induced by data-decomposition are performed in their entirety & each task performs useful computations towards the solution of the prob.
- In exploratory decomposition, unfinished tasks can be terminated as soon as an overall solution is found.
  - Portion of the search space searched (& the aggregate amount of work performed) by a parallel formulation can be different from that searched by a serial alg.
  - The work performed by the parallel formulation can be either smaller or greater than that performed by the serial algorithm.
- Example: consider a search space that has been partitioned into four concurrent tasks as shown in Fig.
  - If the solution lies right at the beginning of the search space corresponding to task 3 (Fig. (a)), then it will be found almost immediately by the parallel formulation.
  - The serial algorithm would have found the solution only after performing work equivalent to searching the entire space corresponding to tasks 1 and 2.
  - On the other hand, if the solution lies towards the end of the search space corresponding to task 1 (Fig (b)), then the parallel formulation will perform almost four times the work of the serial algorithm and will yield no speedup.

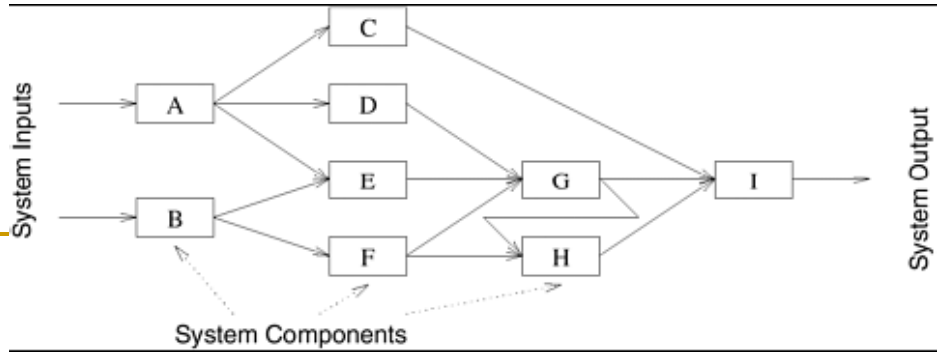


Speculative Decomposition

- Is used when a program may take one of many possible computationally significant branches depending on the output of other computations that precede it.
  - While one task is performing the computation whose output is used in deciding the next computation, other tasks can concurrently start the computations of the next stage.
- This scenario is similar to evaluating one or more of the branches of a switch statement in C in parallel before the input for the switch is available.
  - While one task is performing the computation that will eventually resolve the switch, other tasks could pick up the multiple branches of the switch in parallel.
  - When the input for the switch has finally been computed, the computation corresponding to the correct branch would be used while that corresponding to the other branches would be discarded.
  - The parallel run time is smaller than the serial run time by the amount of time required to evaluate the condition on which the next task depends because this time is utilized to perform a useful computation for the next stage in parallel.
- This parallel formulation of a switch guarantees at least some wasteful computation.
- In order to minimize the wasted computation, a slightly different formulation of speculative decomposition could be used, especially in situations where one of the outcomes of the switch is more likely than the others.
  - In this case, only the most promising branch is taken up a task in parallel with the preceding computation.
  - In case the outcome of the switch is different from what was anticipated, the computation is rolled back and the correct branch of the switch is taken.
  - The speedup due to speculative decomposition can add up if there are multiple speculative stages

Example for Speculative Decomposition

- **Parallel discrete event simulation.**
- Consider the simulation of a system that is represented as a network or a directed graph.
  - The nodes of this network represent components.
  - Each component has an input buffer of jobs.
  - The initial state of each component or node is idle.
- An idle component
  - picks up a job from its input queue, if there is one,
  - processes that job in some finite amount of time, and
  - puts it in the input buffer of the components which are connected to it by outgoing edges.
- A component has to wait if the input buffer of one of its outgoing neighbors if full, until that neighbor picks up a job to create space in the buffer.
- The output of a component (and hence the input to the components connected to it) and the time it takes to process a job is a function of the input job.
- The problem: simulate the functioning of the network for a given sequence or a set of sequences of input jobs and compute the total completion time and possibly other aspects of system behavior.
- Fig. shows a simple network for a discrete event solution problem.
- Define speculative tasks that start simulating a subpart of the network, each assuming one of several possible inputs to that stage.
- When an actual input to a certain stage becomes available (as a result of the completion of another selector task from a previous stage), then all or part of the work required to simulate this input would have already been finished
  if the speculation was correct,
  or the simulation of this stage is restarted
  with the most recent correct input
  if the speculation was incorrect.

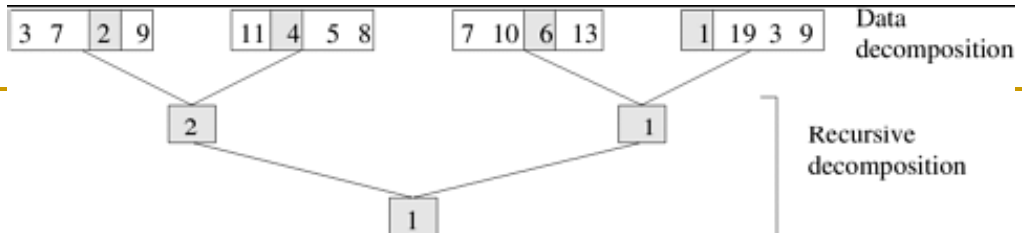


Speculative vs. exploratory decomposition

- In exploratory decomposition
  - the output of the multiple tasks originating at a branch is unknown.
  - the serial algorithm may explore different alternatives one after the other, because the branch that may lead to the solution is not known beforehand
  - ⇒ the parallel program may perform more, less, or the same amount of aggregate work compared to the serial algorithm depending on the location of the solution in the search space.
- In speculative decomposition
  - the input at a branch leading to multiple parallel tasks is unknown
  - the serial algorithm would strictly perform only one of the tasks at a speculative stage because when it reaches the beginning of that stage, it knows exactly which branch to take.
  - ⇒ a parallel program employing speculative decomposition performs more aggregate work than its serial counterpart.

Hybrid Decompositions

- Decomposition techs are not exclusive, and can often be combined together.
  - Often, a computation is structured into multiple stages and it is sometimes necessary to apply different types of decomposition in different stages.
- Example 1: while finding the minimum of a large set of n numbers,
  - a purely recursive decomposition may result in far more tasks than the number of processes, *P*, available.
  - An efficient decomposition would partition the input into *P* roughly equal parts and have each task compute the minimum of the sequence assigned to it.
  - The final result can be obtained by finding the minimum of the *P* intermediate results by using the recursive decomposition shown in Fig
- Example 2: quicksort in parallel.
  - Used a recursive decomposition to derive a concurrent formulation of quicksort.
    - This formulation results in O(*n*) tasks for the problem of sorting a sequence of size n.
    - But due to the dependencies among these tasks and due to uneven sizes of the tasks, the effective concurrency is quite limited.
    - For example, the first task for splitting the input list into two parts takes O(*n*) time, which puts an upper limit on the performance gain possible via parallelization.
  - The step of splitting lists performed by tasks in parallel quicksort can also be decomposed using the input decomposition technique.
    - The resulting hybrid decomposition that combines recursive decomposition and the input data-decomposition leads to a highly concurrent formulation of quicksort.



Orchestration by an example

- Simplified version of a piece or kernel of Ocean problem: its equation solver.
- It uses the equation solver to dig deeper and illustrate how to implement a parallel program using the three programming models.
- The equation solver kernel solves a simple partial differential equation on a grid, using what is referred to as a finite differencing method.
- It operates on a regular, 2-d grid or array of *(n+2)*-by-*(n+2)* elements, such as a single horizontal cross-section of the ocean basin in Ocean.
- The border rows and columns of the grid contain boundary values that do not change, while the interior *n*-by-*n* points are updated by the solver starting from their initial values.
- The computation proceeds over a number of sweeps.
- In each sweep, it operates on all the elements of the grid, for each element replacing its value with a weighted average of itself and its four nearest neighbor elements (above, below, left and right).
- The updates are done in-place in the grid, so a point sees the new values of the points above and to the left of it, and the old values of the points below it and to its right.
- This form of update is called the Gauss-Seidel method.
- During each sweep the kernel also computes the average difference of an updated element from its previous value.
- If this average difference over all elements is smaller than a predefined “tolerance” parameter, the solution is said to have converged & solver exits at the end of the sweep.
- Otherwise, it performs another sweep and tests for convergence again.

Example - decomposition

Sequential:

```
float diff = 0, temp;
while (!done) do
    /*outermost loop over sweeps */
diff = 0; /* initialize max.diff. to 0 */
for i >=1 to n do
/* sweep over non-border points of grid */
for j >=1 to n do
temp = A[i,j]; /* save old value of elem*/
A[i,j] <- 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
     A[i,j+1] + A[i+1,j]);
/*compute average */
diff += abs(A[i,j] - temp);
end for
end for
```

Decomposition

- For programs that are structured in successive loops or loop nests, a simple way to identify concurrency is to start from the loop structure itself.
- Examine the individual loops or loop nests in the program one at a time, see if their iterations can be performed in parallel, & determine whether this exposes enough concurrency.
  - Each iteration of the outermost loop, sweeps through the entire grid.
  - These iterations clearly are not independent, since data that are modified in one iteration are accessed in the next.
- Look at the inner loop first (the *j* loop).
  - Each iteration of this loop reads the grid point (A[*i,j*-1]) that was written in the previous iteration.
  - The iterations are therefore sequentially dependent, and we call this a *sequential loop*.
  - The outer loop of this nest is also sequential, since the elements in row *i-1* were written in the previous (*i-1th*) iteration of this loop.
- So this simple analysis of existing loops and their dependences uncovers no concurrency in this case.

Example - decomposition approaches

- An alternative to relying on program structure to find concurrency is to go back to the fundamental dependences in the underlying algorithms used, regardless of program or loop structure.
- Look at the fundamental data dependences at the granularity of individual grid points.
  - Computing a particular grid point in the sequential program uses the updated values of the grid points directly above and to the left.
  - Elements along a given anti-diagonal (south-west to north-east) have no dependences among them and can be computed in parallel, while the points in the next anti-diagonal depend on some points in the previous one.
- From this diagram, we can observe that of the work involved in each sweep, there a sequential dependence proportional to *n* along the diagonal and inherent concurrency proportional to *n*.
- Decompose the work into individual grid points, so updating a single grid point is a task.
- Approach 1:
  - Leave the loop structure of the program as it is
  - Insert point-to-point synchronization to ensure that a grid point has been produced in the current sweep before it is used by the points to the right of or below it.
  - Different loop nests and even different sweeps might be in progress simultaneously on different elements, as long as the element-level dependences are not violated.
  - The overhead of this synchronization at grid-point level may be too high.
- Approach 2:
  - Change the loops: outer loop be over antidiagonals & inner loop be over elements within an anti-diagonal.
  - The inner loop can now be executed completely in parallel, with global synchronization between iterations of the outer for loop to preserve dependences conservatively across antidiagonals.
  - Global synchronization is still very frequent: once per antidiagonal.
  - Also, the number of iterations in the parallel (inner) loop changes with successive outer loop iterations, causing load imbalances among processors especially in the shorter antidiagonals.
- Because of the frequency of synchronization, the load imbalances, and the programming complexity, neither of these approaches is used much on modern architectures.

Example – red-black ordering

- Approach 3: exploiting knowledge of the problem beyond the sequential program itself.
- Gauss-Seidel solution: iterates until convergence, we can update the grid points in a different order as long as we use updated values for grid points frequently enough.
- One such ordering that is used often for parallel versions is called *red-black* ordering.
  - The idea here is to separate the grid points into alternating red points and black points as on a checkerboard, so that no red point is adjacent to a black point or vice versa.
  - To compute a red point we do not need the updated value of any other red point, but only the updated values of the above and left black points (in a standard sweep), and vice versa.
  - We can therefore divide a grid sweep into two phases: first computing all red points and then computing all black points.
  - Within each phase there are no dependences among grid points, so we can compute all red points in parallel, then synchronize globally, and then compute all black points in parallel.
  - Global synchronization is conservative and can be replaced by point-to-point synchronization at the level of grid points—since not all black points need to wait for all red points to be computed—but it is convenient.
- The red-black ordering is different from our original sequential ordering, and can therefore both converge in fewer or more sweeps as well as produce different final values for the grid points (though still within the convergence tolerance).
- Even if we don't use updated values from the current while loop iteration for any grid points, and we always use the values as they were at the end of the previous while loop iteration, the system will still converge, only much slower.
  - This is called Jacobi rather than Gauss-Seidel iteration.

Example – assignment

- Static assignment:
  - The simplest option is a static (predetermined) assignment in which each processor is responsible for a contiguous block of rows: *block assignment*
  - Alternative: *cyclic* assignment in which rows are interleaved among processes.
- Dynamic assignment:
  - each process repeatedly grabs the next available (not yet computed) row after it finishes with a row task
  - it is not predetermined which process computes which rows.
- Static block assignment.
  - Exhibits good load balance across processes as long as the number of rows is divisible by the number of processes, since the work per row is uniform.

Orchestration under the Data Parallel Model

- Diff from sequential code:
  - Dynamically allocated shared data, are allocated with a G_MALLOC (global malloc) call rather than a regular malloc.
  - Use DECOMP statement,
  - Use of for_all loops instead of for loops,
  - Use of a private mydiff variable per process, and
  - Use of a REDUCE statement.
- for_all specify: iterations performed in parallel.
- DECOMP statement has a two-fold purpose.
  - assignment of the iterations to processes:
    - [BLOCK, *, nprocs] assignment: the 1st dim (rows) is partitioned into contiguous pieces among the nprocs processes, & 2nd dimension is not partitioned at all.
    - [CYCLIC, *, nprocs] would have implied a cyclic or interleaved partitioning of rows among nprocs processes,
    - [BLOCK, BLOCK, nprocs] a subblock decomposition,
    - [*, CYCLIC, nprocs] interleaved partitioning of columns.
  - specifies how the grid data should be distributed among memories on a distributed memory machine
- mydiff variable is used to allow each process to first independently compute the sum of the difference values for its assigned grid points.
- REDUCE: directs the system to add all their partial mydiff values together into the shared diff variable.
  - The reduction op may be implemented in a library in a manner best suited to the underlying architecture.

```
int n, nprocs;
/* grid size (n+2-by-n+2) and number of processes*/
float **A, diff = 0;
main()
begin
read(n); read(nprocs);; /* read input grid size and no.processes*/
A <-G_MALLOC (a 2-d array of size n+2 by n+2 doubles);
initialize(A); /* initialize the matrix A somehow */
Solve (A); /* call the routine to solve equation*/
end main
procedure Solve(A) /* solve the equation system */
float **A; /* A is an n+2 by n+2 array*/
begin
int i, j, done = 0;
float mydiff = 0, temp;
DECOMP A[BLOCK,*];
while (!done) do /* outermost loop over sweeps */
mydiff = 0; /* initialize maximum difference to 0 */
for_all i >=1 to n do /* sweep over non-border points of grid */
for_all j >=1 to n do
temp = A[i,j]; /* save old value of element */
A[i,j] <- 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
A[i,j+1] + A[i+1,j]); /*compute average */
mydiff += abs(A[i,j] - temp);
end for_all
end for_all
REDUCE (mydiff, diff, ADD);
if (diff/(n*n) < TOL) then done = 1;
end while
end procedure
```

Orchestration under the Shared Address Space Model

- Declare the matrix *A* as a single shared arra
- Processes can reference the parts of it they need using loads and stores with exactly the same array indices as in a sequential program.
- Communication will be generated implicitly as necessary.
- With explicit parallel processes we now need mechanisms to:
  - create the processes,
  - coordinate them through synchronization, and
  - control the assignment.
- Differences from the sequential code are shown in italicized bold font
- Comments: in the textbook

```
1. int n, nprocs; /* matrix dimension and number of processors to be used */
2a. float **A, diff; /*A is global (shared) array representing the grid */
/* diff is global (shared) maximum difference in current sweep */
2b. LOCKDEC(diff_lock); /* declaration of lock to enforce mutual exclusion */
2c. BARDEC (bar1); /* barrier declaration for global sync between sweeps*/
3. main()
4. begin
5. read(n); read(nprocs); /* read input matrix size and number of processes*/
6. A <-G_MALLOC (a two-dimensional array of size n+2 by n+2 doubles);
7. initialize(A); /* initialize A in an unspecified way*/
8a. CREATE (nprocs-1, Solve, A);
8 Solve(A); /* main process becomes a worker too*/
8b. WAIT_FOR_END; /* wait for all child processes created to terminate */
9. end main
10. procedure Solve(A)
11. float **A; /*A is a n+2-by-n+2 shared array,as in the sequential program */
12. begin
13. int i,j, pid, done = 0;
14. float temp, mydiff = 0; /* private variables/
14a. int mymin <-1 + (pid * n/nprocs); /*assume that n is divisible by */
14b. int mymax <-mymin + n/nprocs - 1; /* nprocs for simplicity here*/
15. while (!done) do /* outer loop over all diagonal elements */
16. mydiff = diff = 0; /* set global diff to 0 (okay for all to do it) */
17. for i >=mymin to mymax do /* for each of my rows */
18. for j >=1 to n do /* for all elements in that row */
19. temp = A[i,j];
20. A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21. A[i,j+1] + A[i+1,j]);
22. mydiff += abs(A[i,j] - temp);
23. endfor
24. endfor
25a. LOCK(diff_lock); /* update global diff if necessary */
25b. diff += mydiff;
25c. UNLOCK(diff_lock);
25d. BARRIER(bar1,nprocs);/* ensure all have got here before checking if done*/
25e. if (diff/(n*n)<TOL) then done=1; /*check convergence;all get same answer*/
25f. BARRIER(bar1, nprocs); /* see Exercise c */
26. endwhile
27.end procedure
```

Orchestration under the Message Passing Model

```
1. int pid, n, nprocs; /* process id, matrix dimension & no.
       processors to be used */
2. float **myA;
3. main()
4. begin
5. read(n); read(nprocs); /* read input matrix size and number of
       processes*/
8a. CREATE (nprocs-1 processes that start at procedure Solve);
8b. Solve(); /* main process becomes a worker too*/
8c. WAIT_FOR_END; /* wait for all child processes created to
       terminate */
9. end main
10. procedure Solve()
11. begin
13. int i,j, pid, n' = n/nprocs, done = 0;
14. float temp, tempdiff, mydiff = 0; /* private variables/
6. myA <-malloc(2d array of size[n/nprocs+2] by n+2);/*my
       assigned rows of A */
7. initialize(myA); /* initialize my rows of A, in an unspecified way*/
15.while (!done) do
16. mydiff = 0; /* set local diff to 0 */
16a.if (pid != 0) then SEND(&myA[1,0],n*sizeof(float),pid-1,ROW);
16b. if (pid = nprocs-1) then
       SEND(&myA[n',0],n*sizeof(float),pid+1,ROW);
16c. if (pid != 0) then RECEIVE(&myA[0,0],n*sizeof(float),pid-
       1,ROW);
16d. if (pid != nprocs-1) then
       RECEIVE(&myA[n'+1,0],n*sizeof(float),pid+1,ROW);
/*border rows of neighbors have now been copied into myA[0,*]
       and myA[n'+1,*]*/
17. for i >=1 to n' do /* for each of my rows */
18. for j >=1 to n do /* for all elements in that row */
19. temp = myA[i,j];
20. myA[i,j] <- 0.2 * (myA[i,j] + myA[i,j-1] + myA[i-1,j] +
21. myA[i,j+1] + myA[i+1,j]);
22. mydiff += abs(myA[i,j] - temp);
23. endfor
24. endfor
/* communicate local diff values and obtain determine if done;
       can be replaced
by reduction and broadcast */
25a. if (pid != 0) then /* process 0 holds global total diff*/
25b. SEND(mydiff,sizeof(float),0,DIFF);
25c. RECEIVE(mydiff,sizeof(float),0,DONE);
25d. else
25e. for i >=1 to nprocs-1 do /* for each of my rows */
25f. RECEIVE(tempdiff,sizeof(float),*,DONE);
25g. mydiff += tempdiff; /* accumulate into total */
25h. endfor
25i. for i >=1 to nprocs-1 do /* for each of my rows */
25j. SEND(done,sizeof(int),i,DONE);
25k. endfor
25l. endif
26. if (mydiff/(n*n) < TOL) then done = 1;
27. endwhile
28. end procedure
```

Comments: in the textbook