ECE5610/CSC6220
Introduction to Parallel and Distribution Computing

Lecture 2: Parallel Programs
Creating a Parallel Program

- Assumption: Sequential algorithm is given
  - Sometimes need very different algorithm, but beyond scope of the course

- Key steps:
  - Identify work that can be done in parallel
  - Partition work and perhaps data among processes
  - Manage data access, communication and synchronization
  - Note: work includes computation, data access and I/O

- Main goal: Speedup
  \[
  Speedup\ (p) = \frac{\text{Performance}(p)}{\text{Performance}(1)} = \frac{\text{Time}(1)}{\text{Time}(p)}
  \]

- Secondary goals: low programming effort and low resource demands
Some Important Concepts

- **Task:**
  - The smallest unit of concurrency in parallel computation
  - Executed sequentially; concurrency is only across tasks
  - E.g. an iteration of a loop
  - Fine-grained versus coarse-grained tasks

- **Process (thread):**
  - Abstract entity that performs the tasks
  - Tasks are assigned to processes
  - Processes communicate and synchronize to perform their tasks

- **Processor:**
  - Physical engine on which process executes
  - Processes virtualize machine to programmer
    - first write program in terms of processes, then map to processors

- **Parallel program**
  - Computation ➔ Tasks ➔ Processes ➔ Processors
Steps in Creating a Parallel Program

– 4 steps: Decomposition, Assignment, Orchestration, Mapping
  • Done by programmer or system software (compiler, runtime, ...)
  • Issues are the same, so assume programmer does it all explicitly
Step 1: Decomposition

- Break up computation into tasks to be divided among processes
  - Identify concurrency: the work that can be done in parallel
    - Tasks may become available dynamically
    - Number of available tasks may vary with time
  - Decide appropriate level at which to exploit concurrency
    - Too much: high overheads of management
    - Too few: variations in work performed across tasks
  - In general: precedence relationships between tasks
- Characteristics
  - Task creation: static versus dynamic
  - Task granularity: uniform versus non-uniform
  - Concurrency: number of available tasks
- Goal:
  - Enough tasks to keep processes busy, but not too many
    - Number of tasks available at a time is upper bound on achievable speedup
Trying to Exploit Currency: Example

- 2-phase calculation
  - sweep over \( n \)-by-\( n \) grid and do some independent computation
  - sweep again and add each value to global sum

- Time for first phase = \( \frac{n^2}{p} \)

- Second phase serialized at global variable, so time = \( n^2 \)

- Speedup \( \leq \frac{2n^2}{\frac{n^2}{p} + n^2} \) \( \leq 2 \)

- Trick: divide second phase into \( p \) tasks
  - accumulate into private sums during sweep
  - add per-process private sums into global sum

- Parallel time is \( \frac{n^2}{p} + \frac{n^2}{p} + p \), and speedup at best \( \frac{2p}{2 + (p/n)^2} \)
Example: Pictorial Depiction

(a) \( n^2 \) work done concurrently

(b) \( \frac{n^2}{p} \) work done concurrently

(c) \( \frac{n^2}{p} \) work done concurrently

Time
Step 2: Assignment

• Specifying mechanism to divide work up among processes
  ➢ E.g. which process computes which task?
  ➢ Together with decomposition, also called *partitioning*
  ➢ Goal: Balance workload, reduce communication and management cost

• Structured approaches usually work well
  ➢ Code inspection (parallel loops) or understanding of application
  ➢ Well-known heuristics
  ➢ Static versus dynamic assignment

• Division of responsibility between programmer and architecture
  ➢ Programmers worry about partitioning first
    • Usually independent of architecture or programming model
    • However, cost and complexity of using primitives may affect decisions
  ➢ Architects assume program is reasonably partitioned
    • Cannot do anything if this is not the case!
Step 3: Orchestration

- Issues
  - Structuring communication
    
    Assignment of tasks produces need for inter-process interactions
  - Organizing data structures and scheduling tasks temporally
  - Naming data
  - Whether to communicate implicitly or explicitly and in small or large messages;
  - Synchronization

- Goals
  - Reduce cost of communication and synchronization (as seen by processors)
  - Preserve locality of data reference
  - Schedule tasks to satisfy dependences early
  - Reduce overhead of parallelism management

- Orchestration choices depend on:
  - Available primitives (programming model and language)
  - Efficiency of these primitives (architecture)
Step 4: Mapping

• After orchestration, a parallel program is ready to run

• Two aspects of mapping:
  ➢ Which processes will run on same processor, if necessary (collocation)
  ➢ Which process runs on which particular processor (placement)
    • mapping to a network topology

• One extreme: space-sharing
  ➢ Machine divided into subsets, only one app at a time in a subset
  ➢ Processes can be pinned to processors, or left to OS

• Another extreme: complete resource management control to OS
  ➢ OS tries to achieve better resource sharing and utilization, and speed up execution of a parallel program (e.g., gang scheduling)

• Real world is between the two
  • User specifies desires in some aspects, system takes care of the rest

➡ Mapping in multiprogrammed systems is an active research area
Parallelizing Computation vs. Data

• So far, parallelization is centered around computation
  ➢ Computation is decomposed and assigned (partitioned)
  ➢ Data partition (if present) arises from how tasks access data
• Alternative view: Partitioning Data
  ➢ Very natural perspective in data parallel models
  ➢ Same operation on each element of a data structure
  ➢ Computation follows data: *owner computes*
  ➢ E.g., grid-based computations, data mining
• But not general enough
  ➢ Strong distinction between computation and data in many applications
    • Barnes-Hut, Raytrace
  ➢ Computation-centric view is more general
High-level Goals

High speedup but with low resource usage and development effort

Table 2.1 Steps in the Parallelization Process and Their Goals

<table>
<thead>
<tr>
<th>Step</th>
<th>Architecture-Dependent?</th>
<th>Major Performance Goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decomposition</td>
<td>Mostly no</td>
<td>Expose enough concurrency but not too much</td>
</tr>
<tr>
<td>Assignment</td>
<td>Mostly no</td>
<td>Balance workload</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reduce communication volume</td>
</tr>
<tr>
<td>Orchestration</td>
<td>Yes</td>
<td>Reduce noninherent communication via data locality</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reduce communication and synchronization cost as seen by the processor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reduce serialization at shared resources</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Schedule tasks to satisfy dependences early</td>
</tr>
<tr>
<td>Mapping</td>
<td>Yes</td>
<td>Put related processes on the same processor if necessary</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Exploit locality in network topology</td>
</tr>
</tbody>
</table>

Implications

Algorithm designers: high-performance and low resource needs
Architects: high-performance, low cost, reduced programming effort
An Example of Parallel Program

- Ocean: simulating ocean currents
  - Regular structure
  - Amenable to both data- and computation-oriented partitioning

- Model: set of two-dimensional grids (cross sections)
  - Discretize in space and time
    - Each variable (pressure, velocity, temperature) has a value per grid point
    - Finer spatial and temporal resolution ➞ greater accuracy
  - The equations of motion are solved at all points in one time step, then for another time step.
  - Concurrency across and within grid computations

(a) Cross sections
(b) Spatial discretization of a cross section
What do parallel Programs Look Like?

• Examine a simplified version of a piece of ocean simulation
  ➢ Iterative equation solver (using a finite differencing method)
  ➢ Examine each step in detail

• Illustrate parallel program in low-level parallel language
  ➢ C-like pseudo-code with simple extensions for parallelism
  ➢ Expose basic communication and synchronization primitives in each of the programming models
  ➢ Standard sequential languages augmented with primitives for parallelism is the state of most real parallel programming today
Grid Equation Solver Kernel

Expression for updating each interior point:


- Gauss-Seidel (near-neighbor) sweeps to convergence
  - interior n-by-n points of (n+2)x(n+2) updated in each sweep
  - updates done in-place in grid, and difference from previous value computed
  - accumulate partial differences into global difference at end of every sweep
  - check if error has converged (to within a tolerance parameter)
  - if so, exit solver; if not, do another sweep
Sequential Pseudocode of the Equation Solver Kernel

1. int n; /*size of matrix: (n + 2-by-n + 2) elements*/
2. float **A, diff = 0;

3. main()
4. begin
5. read(n); /*read input parameter: matrix size*/
6. A ← malloc (a 2-d array of size (n+2) by (n+2) doubles);
7. initialize(A); /*initialize matrix A somehow*/
8. Solve (A); /*call the routine to solve equation*/
9. 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 diff = 0, temp;
while (!done) do /*outermost loop over sweeps*/
diff = 0; /*initialize maximum difference to 0*/
for i ← 1 to n do /*sweep over inter. points*/
for j ← 1 to n do
  temp = A[i, j]; /*save old value of element*/
  A[i, j+1] + A[i+1, j]
  diff += abs(A[i, j] - temp);
end for
end for
if (diff/(n*n) < TOL) then done = 1;
end while
end procedure
Decomposition: Looking for Concurrency

• Simple way is to examine loop iterations
  ➢ Dependence analysis: if not enough concurrency, then look further
  ➢ Not much concurrency here at this level (both loops sequential)
• Examine fundamental dependences, ignoring loop structure

Regular dependence:
Task (i,j) at sweep k, T^k(i,j), is dependent on
- task (i-1, j) at sweep k, T^k(i-1, j),
- task (i, j-1) at sweep k, T^k(i,j-1),
- task (i+1,j) at sweep k-1, T^{k-1}(i+1,j),
- task (i, j+1) at sweep k-1, T^{k-1} (i, j+1).
Grid Solver Example: Decomposition

- Concurrency $O(n)$ along anti-diagonals,
- Serialization $O(n)$ along diagonal

But how to exploit this parallelism
- Option 1: Retain existing loop structure
  - No way to parallelize it
- Option 2: Restructure loops to loop over anti-diagonals
  - Global synchronize between iterations (*barrier*)
  - Problem: benefit of parallel is overwhelmed by synchronization cost
- Option 3: Exploit application knowledge
  - Reorder grid traversal using red-black ordering (chess-board pattern)
Exploit Application Knowledge: red-black ordering

- Left-to-right, top-to-bottom ordering not fundamental to Gauss-Seidel
- Red-black ordering
  - Decompose grid into two sets of points (as in a chess-board)
  - Different ordering of updates: may converge quicker or slower
  - Red sweep and black sweep are each fully parallel
  - Global synchronization between them (conservative but convenient)
- Ocean uses red-black; we use simpler, asynchronous one to illustrate
  - No red-black, simply ignore dependences within sweep
  - With multiple processes, the ordering is unpredictable.
Grid Solver Example: Code for Decomposition

15. while (!done) do
   /*a sequential loop*/
16.   diff = 0;
17.   for_all i ← 1 to n do
       /*a parallel loop nest*/
18.     for_all j ← 1 to n do
19.       temp = A[i,j];
   A[i,j+1] + A[i+1,j]);
21.       diff += abs(A[i,j] - temp);
22.   end for_all
23. end for_all
24. end while
25. if (diff/(n*n) < TOL) then done = 1;
26. end while

• We use simpler example of asynchronous code: no dependencies
  ➢ for-all leaves assignment to the system
    • But implicit global synchronization at the end of for-all loop
  ➢ As shown: a task is a single grid point, so O(n^2) tasks;
  ➢ To decompose into rows: make line 18 loop sequential ➔ O(n) tasks (for_all ➔ for)
Grid Solver Example: Assignment

- Static assignments
  - block assignment: Row $i$ is assigned to process $\left\lfloor \frac{i}{p} \right\rfloor$
  - Cyclic assignment: Process $i$ is assigned rows $i$, $i+p$, $i+2p$…

- Block assignment reduces communication requirement
Grid Solver Example: Assignment (cont’d)

- Dynamic assignment:
  - Get a row index, work on the row, get a new row, and so on
  - Adapt at runtime to balance load
  - Can increase communication and reduce locality
  - Can increase task management overheads
Grid Solver Example: Orchestration

- Requirements
  - Communication
    - Values of neighboring grid points must be available
  - Synchronization
    - Next iteration (or alternating red-black sweeps) cannot proceed until all grid points have been evaluated in the current sweep

- Language support for orchestration is programming-model-specific
  - Data parallel
    - Construct loops, computation and data structure decomposition, collective operations
  - Share memory
    - Process creation, mutual exclusion, global synchronization, post-wait
  - Message-passing
    - Process creation, synchronous and asynchronous send/receive, global synchronization
Data Parallel Model: Orchestration Support

• Global declarations ➔ Shared data; all other data ➔ private data

• Dynamic allocation of shared data
  ➢ \textit{G\_MALLOC}(global malloc)

• Concurrent loops
  ➢ \textit{for\_all}
  ➢ Parallel processes are implicitly active: only within \textit{for\_all} body

• Decomposition of data and computation
  ➢ \textit{DECOMP \textit{arr}[BLOCK, *, nprocs]},
    • Alternatively, \textit{DECOMP \textit{arr}[CYCLIC, *, nprocs]}, \textit{DECOMP \textit{arr}[BLOCK, BLOCK, nprocs]},
  ➢ Specifies assignment of data elements to processes
  ➢ Owner-computes: specifies assignment of iterations to processes

• Collective operations of associativity
  ➢ \textit{REDUCE} (other such as broadcast, etc)
  ➢ should be implemented efficiently by underlying system
Orchestration: Data Parallel Solver

1. int n, nprocs;
   /*grid size (n + 2-by-n + 2) and number of processes*/
2. float **A, diff = 0;

3. main()
4. begin
5.   read(n);
6.   read(nprocs);
   /*read input grid size and number of processes*/
7.   A ← G_MALLOC (a 2-d array of size n+2 by n+2 doubles);
8.   initialize(A);  /*initialize the matrix A somehow*/
9.   Solve (A);     /*call the routine to solve equation*/
10. end
Data Parallel Solver

10. procedure Solve(A) /*solve the equation system*/
11.   float **A;       /*A is an (n + 2-by-n + 2) array*/
12.   begin
13.   int i, j, done = 0;
14.   float mydiff = 0, temp;
14a.  DECOMP A[BLOCK,*, nprocs];
15.   while (!done) do /*outermost loop over sweeps*/
16.     mydiff = 0;    /*initialize maximum difference to 0*/
17.     for_all i ← 1 to n do /*sweep over non-border points of grid*/
18.       for_all j ← 1 to n do
19.         temp = A[i,j];  /*save old value of element*/
22.         mydiff += abs(A[i,j] - temp);
23.       end for_all
24.     end for_all
24a.    REDUCE (mydiff, diff, ADD);
25.     if (diff/(n*n) < TOL) then done = 1;
Shared Memory Model: Orchestration Support

• Process creation and termination
  - CREATE(num_procs, proc, args)
  - WAIT_FOR_END(number)
• Dynamic allocation of shared data
  - G_MALLOC()
• Mutual exclusion
  - LOCK(name): acquire mutually exclusive accesses
  - UNLOCK(name): release access
• Global synchronization
  - BARRIER(name, number)
  - no process gets pass barrier until number have arrived
• Point-to-point synchronization
  - WAIT(flag): wait for flag to be set (spin or block)
  - SIGNAL(flag): set flag, wait up waiting processes
  - Producer-consumer sharing, semaphores
Shared Memory Model: Grid Solver Example

- Grid declared as a shared array
  - All processes can access it just as in sequential program

- Single Program Multiple Data (SPMD) style
  - Assignment controlled by values of variables used as loop bounds
1. int n, nprocs; /* matrix dimension and number of processors to be used */
2a. float **A, diff; /*A is global 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 synchronization
    BARDEC(bar2) between sweeps*/
    BARDEC(bar3)
3. main()
4. begin
5. read(n); read(nprocs);
6. A ← G_MALLOC (a n+2 x n+2 double array);
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 (nprocs–1);
    /*wait for all child processes created to terminate*/
9. end main
procedure Solve(A)

float **A; /* A is entire n+2-by-n+2 shared array*/

begin
int i,j, pid, done = 0;

float temp, mydiff = 0; /* private variables*/

int mymin = 1 + (pid * n/nprocs); /* assume that n is exactly */

int mymax = mymin + n/nprocs - 1 /* divisible by nprocs for simplicity*/

while (!done) do /* outer loop over all diagonal elements */

mydiff = diff = 0; /* set global diff to 0 (OK for all to it) */

BARRIER(bar1,nprocs); /* ensure all reach here before anyone modifies diff*/

for i ← mymin to mymax do /* for each of my rows */

for j ← 1 to n do /* for all non-border elements in the row */

temp = A[i,j];


mydiff += abs(A[i,j] - temp);

endfor

endfor

BARRIER(bar1, nprocs);

lock(diff_lock); /* update global diff if necessary*/

diff += mydiff; /* critical section */

UNLOCK(diff_lock);

BARRIER(bar2, nprocs); /* ensure all reach here before checking if done */

if (diff/(n*n) < TOL) then done = 1; /* check convergence; */

BARRIER(bar3, nprocs); /* all get same answer */

endwhile

end procedure
Shared Memory Model: Grid Solver (cont’d)

- Single Program Multiple Data (SPMD):
  - not lockstep or even necessarily same instructions
  - Assignment of iterations controlled by values of variables used as loop bounds
    - Unique pid per process, used to control assignment
  - done condition evaluated redundantly by all
  - Code that does the update identical to sequential program
    - Since grid array is in shared space
    - each process has private mydiff variable
- Most interesting special operations are for synchronization
  - accumulations into shared diff have to be mutually exclusive
  - why the need for all the barriers?
**Grid Solver: Need for Mutual Exclusion**

- Code each process executes:
  - load the value of \( \text{diff} \) into register \( r1 \)
  - add the register \( r2 \) to register \( r1 \)
  - store the value of register \( r1 \) into \( \text{diff} \)

- A possible interleaving:

  \[
  \begin{align*}
  &\text{P1} &\text{P2} \\
  r1 &\leftarrow \text{diff} & r1 &\leftarrow \text{diff} \\
  & r1 \leftarrow r1+r2 & r1 \leftarrow r1+r2 \\
  & \text{diff} \leftarrow r1 & \text{diff} \leftarrow r1 \\
  \end{align*}
  \]

  \{P1 gets 0 in its \( r1 \}\}
  \{P2 also gets 0\}
  \{P1 sets its \( r1 \) to 1\}
  \{P2 sets its \( r1 \) to 1\}
  \{P1 sets \( \text{diff} \) to 1\}
  \{P2 also sets \( \text{diff} \) to 1\}

- Need the set of operations to be atomic (mutually exclusive)
- Use of \( \text{mydiff} \) to reduce contention for the lock
Grid Solver: Need for Barrier

- **BARRIER**(name, nprocs): wait here till nprocs processes get here
  - Built using lower level primitives
  - Global sum example: wait for all to accumulate before using *sum*
  - Often used to separate phases of computation
  - Conservative form of preserving dependences, but easy to use

- **Application of Barriers in the Grid Solver**
  - **Line 25d:**
    - Ensures that all processes have updated *diff*
    - Needed for ensuring that the tolerance check is correct.
  - **Line 25f**
    - Ensures that each process waits for all others to get done before exiting or get into next iteration.
  - **Line 16a**
    - Ensures that there is no race condition between Lines 16 and 25b
    - Else, an arbitrarily slow process can reset *diff* after a faster process has updated it for the next iteration
Message Passing Model: Orchestration Support

- Process creation and termination
  - `CREATE(num_procs, procedure)`
  - `WAIT_FOR_END(nproc)`

- Communication: data-transfer + synchronization
  - `SEND(src_addr, size, dest, tag)`
    - Send size bytes from `src_addr` to `dest` process with tag identifier
  - `RECEIVE(buffer_addr, size, src, tag)`
    - Receive a message of `size` from `src` process with `tag` identifier and store it in `buffer_addr`
  - `SEND_ASYNC, SEND_PROBE`
  - `RECEIVE_ASYNC, RECEIVE_PROBE`

- Global synchronization
  - `BARRIER(name, number)`
    - None gets past BARRIER until number of processes have arrived
Message Passing Model: Grid Solver Example

• Structurally similar to shared memory program (still SPMD), but differs significantly in orchestration
  ➢ Data structures and data access/naming
    • Cannot declare grid A to be a shared array any more
    • Need to compose it logically from per-process private arrays
      ✓ usually allocated in accordance with the assignment of work
      ✓ process assigned a set of rows allocates them locally
  ➢ Communication
    • Transfers of entire rows between traversals
  ➢ synchronization
1. int pid, n, b; /*process id, matrix dimension and number of processors to be used*/
2. float **myA;
3. main()
4. begin
5.   read(n); read(nprocs);
8a. CREATE (nprocs-1, Solve);
8b. Solve(); /*main process becomes a worker too*/
8c. WAIT_FOR_END (nprocs–1);
     /*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;
15.   myA ← malloc(a [n/nprocs + 2] by n+2);
     /*my assigned rows of A*/
16.   initialize(myA);
17.    while (!done) do
        … …
26.   endwhile
27. end procedure
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 (nonghost) rows*/
18.     for j ←← ←← 1 to n do /*for all nonborder elements */
19.       temp = myA[i,j];
22.       mydiff += abs(myA[i,j] - temp);      
23.     endfor
24.   endfor
/*communicate local diff values and determine if
   done; can be replaced by reduction and broadcast*/
25    … …
26. endwhile
27. end procedure
15. while (!done) do
16.     mydiff = 0;

............
/*communicate local diff values and 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(done,sizeof(int),0,DONE);
25d.     else /*pid 0 does this*/
25e.         for i ← 1 to nprocs-1 do /*for each other process*/
25f.             RECEIVE(tempdiff,sizeof(float),*,DIFF);
25g.             mydiff += tempdiff; /*accumulate into total*/
25h.         endfor
25i.     if (mydiff/(n*n) < TOL) then done = 1;
25j.     for i ← 1 to nprocs-1 do /*for each other process*/
25k.         SEND(done,sizeof(int),i,DONE);
25l.     endfor
25m.     endif
26. endwhile
27. end procedure
for i ← 1 to nprocs-1 do {
    RECEIVE(tempdiff, size(float), DIFF);
    Mydiff += tempdiff;
}

for i ← 1 to nprocs-1 do {
    SEND (done, sizeof(int), i, DONE)
    RECEIVE(done, size(float), DONE)
    SEND (mydiff, sizeof(float), 0, DIFF)
}
Message Passing Model: Grid Solver (Cont’d)

- Private portions of grid array
  - Use of ghost rows to store neighbor values
- Core similar, but indices/bounds in local rather than global space
- Communication
  - *Receive* does not transfer data, *send* does
    - unlike SAS which is usually receiver-initiated (load fetches data)
  - Communication done at beginning of iteration, implicit synchronization
  - Communication in whole rows, not one element at a time
- Synchronization
  - Using synchronous sends and receives
    - Could implement locks and barriers with messages
  - Can use REDUCE and BROADCAST library calls to simplify code

```c
/*communicate local diff values and determine if done, using reduction and broadcast*/
25b. REDUCE(0,mydiff,sizeof(float),ADD);
25c. if (pid == 0) then
25i.   if (mydiff/(n*n) < TOL) then done = 1;
25k.   endif
25m. BROADCAST(0,done,sizeof(int),DONE);
```
Send and Receive Alternatives

• Semantic flavors: based on when control is returned

Send/Receive

Synchronous

Asynchronous

Blocking asynch.

Nonblocking asynch.

➢ Affect when data structures or buffers can be reused at either end
➢ Affect event sync
  • Synchronous messages provide built-in sync through match
  • Separate event synchronization needed with asynch. messages
➢ Affect ease of programming and performance
➢ With sync messages, our code can be deadlocked. Fix?
Synchronous versus Asynchronous

• Synchronous SEND: returns control to the calling process only when it’s clear that the corresponding RECEIVE has been performed

• Synchronous RECEIVE: returns control when the data has been received into the destination process’s address space

• Asynchronous: Blocking vs Nonblocking

  ➢ Blocking asynchronous SEND: return control to the calling process when the msg has been copied into system buffer.

  ➢ Blocking asynchronous RECEIVE: Similar to synchronous RECEIVE, but a blocking RECEIVE does not send an acknowledgement to the sender

  ➢ Non-blocking SEND: return control immediately

  ➢ Non-blocking RECEIVE: return control after simply posting the intent to RECEIVE
/* exchange border rows between adjacent blocks */
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)

nprocs = 3

pid = 0

16b

deadlock

pid = 1

16a

pid = 2

16a
/* exchange boundary rows between adjacent blocks */

if (pid % 2 = 0 & & pid != nprocs-1) then SEND((&myA[n',0],n*sizeof(float),pid+1,ROW);
if (pid % 2 = 1) then RECEIVE(&myA[0,0],n*sizeof(float),pid-1,ROW);

if (pid % 2 = 1) then SEND((&myA[0,0],n*sizeof(float),pid-1,ROW);
if (pid % 2 = 0 & & pid != nprocs-1) then RECEIVE(& myA[n',0], n*sizeof(float),pid+1,ROW);
if (pid % 2 = 1 && pid != nprocs-1) then SEND((&myA[n',0],n*sizeof(float),pid+1,ROW);
if (pid % 2 = 0 & & pid != 0) then RECEIVE(&myA[0,0],n*sizeof(float),pid-1,ROW);
if (pid % 2 = 0 & & pid != 0) then SEND((&myA[0,0],n*sizeof(float),pid-1,ROW);
if (pid % 2 = 1 && pid != nprocs-1) then RECEIVE(& myA[n',0], n*sizeof(float),pid+1,ROW);
if (pid % 2 = 1 && pid != nprocs-1) then RECEIVE(& myA[n',0], n*sizeof(float),pid+1,ROW);

P0
P1
P2
P3
P4

P0
P1
P2
P3
P4
Orchestration: Summary

• Data Parallel
  ➢ Decomposition of data structures (implicit assignment of tasks)

• Shared address space
  ➢ Shared and private data explicitly separate
    • No correctness need for data distribution
  ➢ Communication implicit in access patterns
  ➢ Synchronization via atomic operations on shared data
    • Synchronization explicit and distinct from data communication

• Message passing
  ➢ Data distribution among local address spaces needed
    • No explicit shared structures (implicit in comm. patterns)
  ➢ Communication is explicit
  ➢ Synchronization implicit in communication
    • With synchronous SEND/RECEIVE primitives;
    • Mutual exclusion for free: only one process updating each address space
Correctness in Grid Solver Program

- Decomposition and assignment (partitioning) similar in all three models
- Orchestration is different
  - Data structures, data access/naming, communication, synchronization

<table>
<thead>
<tr>
<th></th>
<th>SAS</th>
<th>Msg-Passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit global data structure?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Assignment indept of data layout?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Communication</td>
<td>Implicit</td>
<td>Explicit</td>
</tr>
<tr>
<td>Synchronization</td>
<td>Explicit</td>
<td>Implicit</td>
</tr>
<tr>
<td>Explicit replication of border rows?</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Requirements for performance are another story ...