Parallelization of an Example Program

[§2.3] In this lecture, we will consider a parallelization of the kernel of the Ocean application.

**Goals**: Illustrate parallel programming in a low-level parallel language. (C-like pseudocode with simple extensions for parallelism.)

Expose basic communication and synchronization primitives that must be supported.

This is the state of most real parallel programming today.

**The serial program**

The equation solver solves a PDE on a grid, using a finite-differencing method.

It operates on a regular 2D grid of \((n+2)\) by \((n+2)\) elements.

- The border rows & columns contain boundary elements that do not change.
- The interior \(n\)-by-\(n\) points are updated, starting from their initial values.

Expression for updating each interior point:

\[
\]
• The old value at each point is replaced by the weighted average of itself and its 4 nearest-neighbor points.
• Updates are done from left to right, top to bottom.
  ° The update computation for a point sees the new values of points above and to the left, and
  ° the old values of points below and to the right.
This form of update is called the Gauss-Seidel method.

During each sweep, the solver computes how much each element changed since the last sweep.

• If this difference is less than a “tolerance” parameter, the solution has converged.
• If so, exit solver; if not, do another sweep.

Here is the code for the solver.

```c
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 the matrix A somehow*/
8. Solve (A); /*call the routine to solve equation*/
9. end main
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 diff = 0, temp;
15. while (!done) do /*outermost loop over sweeps*/
16.    diff = 0; /*initialize maximum difference to 0*/
17.    for i ← 1 to n do /*sweep over nonborder points of grid*/
18.       for j ← 1 to n do /*save old value of element*/
19.          temp = A[i, j];
22.          diff += abs(A[i, j] - temp);
23.       end for
24.    end for
25.    if (diff/(n*n) < TOL) then done = 1;
26. end while
27. end procedure
```
Decomposition
A simple way to identify concurrency is to look at loop iterations.

We do dependence analysis of sweeps; if there’s not enough concurrency, then we look further.

Is there much concurrency in this example? Can we perform more than one sweep concurrently?

Next, we examine fundamental dependencies, ignoring loop structure.

Note that—

- Computation proceeds from left to right and top to bottom.
- Thus, to compute a point, we use
  - the updated values from the point above and the point to the left, but
  - the “old” values of the point itself and its neighbors below and to the right.

Here is a diagram that illustrates the dependencies.

![Diagram](image)

The horizontal and vertical lines with arrows indicate dependencies.

The dashed lines along the antidiagonal connect points with no dependencies that can be computed in parallel.

Of the $O(n^2)$ work in each sweep, $\exists$ concurrency proportional to $n$ along antidiagonals.

How could we exploit this parallelism?
• We can leave loop structure alone and let loops run in parallel, inserting synchronization ops to make sure a value is computed before it is used.
   Is this a good idea?

• We can change the loop structure, making
  - the outer \texttt{for} loop (line 17) iterate over anti-diagonals, and
  - the inner \texttt{for} loop (line 18) iterate over elements within an antidiagonal.
   Is this a good idea?

Note that according to the Gauss-Seidel algorithm, we don’t have to update the points from left to right and top to bottom.

It is just a convenient way to program on a uniprocessor.

We can compute the points in another order, as long as we use updated values frequently enough (if we don’t, the solution will only converge more slowly).

\textit{Red-black ordering}

Let’s divide the points into alternating “red” and “black” points:
To compute a red point, we don’t need the updated value of any other red point. But we need the updated values of 2 black points.

And similarly for computing black points.

Thus, we can divide each sweep into two phases.

- First we compute all red points.
- Then we compute all black points.

True, we don’t use any updated black values in computing red points.

But we use all updated red values in computing black points.

Whether this converges more slowly or faster than the original ordering depends on the problem.

But it does have important advantages for parallelism.

- How many red points can be computed in parallel?

- How many black points can be computed in parallel?

Red-black ordering is effective, but it doesn’t produce code that can fit on a TV screen.

A simpler decomposition

Another ordering that is simpler but still works reasonably well is just to ignore dependencies between grid points within a sweep.

An sweep just updates points based on their nearest neighbors, regardless of whether the neighbors have been updated yet.

Global synchronization is still used between sweeps, however.

Now execution is no longer deterministic; the number of sweeps needed, and the results, may depend on the number of processors used.
But for most reasonable assignments of processors, the number of sweeps will not vary much.

Let’s look at the code for this.

```
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];
22.       diff += abs(A[i, j] - temp);
23.     end for_all
24.   end for_all
25.   if (diff/(n*n) < TOL) then done = 1;
26.   end while
```

The only difference is that `for` has been replaced by `for_all`.

A `for_all` just tells the system that all iterations can be executed in parallel.

With `for_all` in both loops, all \( n^2 \) iterations of the nested loop can be executed in parallel.

Whether or not they are assigned and orchestrated in this way is up to the system; the program doesn’t ordain it.

We could write the program so that the computation of one row of grid points must be assigned to a single processor. How would we do this?

With each row assigned to a different processor, each task has to access about \( 2n \) grid points that were computed by other processors; meanwhile, it computes \( n \) grid points itself.

So the communication-to-computation ratio is \( O(\_\_\_\).\)

**Assignment**

How can we statically assign rows to processes?
• One option is “block assignment”—Row $i$ is assigned to process $\lfloor i/p \rfloor$.

<table>
<thead>
<tr>
<th>$p_0$</th>
<th>$p_1$</th>
<th>$p_2$</th>
<th>$p_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Another option is “cyclic assignment”—Process $i$ is assigned rows $i, i+p, i+2p$, etc.

(We could instead use dynamic assignment, where a process gets an index, works on the row, then gets a new index, etc.

Static assignment of rows to processes reduces concurrency

But block assignment reduces communication, by assigning adjacent rows to the same processor.

How many rows now need to be accessed from other processors?

So the communication-to-computation ratio is now only $O(\ldots)$.

**Orchestration**

Once we move on to the orchestration phase, the computation model affects our decisions.

*Data-parallel model*

In the code below, we assume that global declarations are used for shared data, and that any data declared within a procedure is private.

Global data is allocated with $g\_malloc$. 
Differences from sequential program:

- **for_all** loops
- **decomp** statement
- **mydiff** variable, private to each process
- **reduce** statement

The **decomp** statement has a twofold purpose.

- It specifies the assignment of iterations to processes.
The first dimension (rows) is partitioned into \( nprocs \) contiguous blocks. The second dimension is not partitioned at all.

Specifying \([\text{CYCLIC}, *, nprocs]\) would have caused a cyclic partitioning of rows among \( nprocs \) processes.

Specifying \([*, \text{CYCLIC}, nprocs]\) would have caused a cyclic partitioning of columns among \( nprocs \) processes.

Specifying \([\text{BLOCK}, \text{BLOCK}, nprocs]\) would have implied a 2D contiguous block partitioning.

- It specifies the assignment of grid data to memories on a distributed-memory machine. (Follows owner-computes rule.)

The \textit{mydiff} variable allows local sums to be computed.

The \texttt{reduce} statement tells the system to add together all the \textit{mydiff} variables into the shared \textit{diff} variable.

\textit{SAS model}

In this model, we need mechanisms to create processes and manage them.

After we create the processes, they interact as shown on the right.
What are the main differences between the serial program and this program?

- The first process creates `nprocs - 1` worker processes. All `n` processes execute `Solve`.

  All processes execute the same code (the SPMD model).

  But all do not execute the same instructions at the same time.

- Private variables like `mymin` and `mymax` are used to control loop bounds.
• All processors need to—
  ° complete an iteration before any process tests for convergence.
  ° test for convergence before any process starts the next iteration.

Notice the use of barrier synchronization to achieve this.

• Locks must be placed around updates to \textit{diff}, so that no two processors update it at once. Otherwise, inconsistent results could ensue.

\begin{align*}
  p_1 & \quad p_2 \\
  r_1 & \leftarrow \text{diff} \quad \{ p_1 \text{ gets 0 in its } r_1 \} \\
  r_1 & \leftarrow \text{diff} \quad \{ p_2 \text{ also gets 0} \} \\
  r_1 & \leftarrow r_1 + r_2 \quad \{ p_1 \text{ sets its } r_1 \text{ to 1} \} \\
  r_1 & \leftarrow r_1 + r_2 \quad \{ p_2 \text{ sets its } r_1 \text{ to 1} \} \\
  \text{diff} & \leftarrow r_1 \quad \{ p_1 \text{ sets } \text{diff} \text{ to 1} \} \\
  \text{diff} & \leftarrow r_1 \quad \{ p_2 \text{ also sets } \text{diff} \text{ to 1} \}
\end{align*}

If we allow only one processor at a time to access \textit{diff}, we can avoid this race condition.

What is one performance problem with using locks?

Note that at least some processors need to access \textit{diff} as a non-local variable.

What is one technique that our SAS program uses to diminish this problem of serialization?

\textit{Message-passing model}

The program for the message-passing model is also similar, but again there are several differences.
• There’s no shared address space, so we can’t declare array $A$ to be shared. Instead, each processor holds the rows of $A$ that it is working on.

• The subarrays are of size $(nprocs/2 + 2) \times (n + 2)$. This allows each subarray to have a copy of the boundary rows from neighboring processors. Why is this done?

These *ghost* rows must be copied explicitly, via *send* and *receive* operations.

Note that *send* is not synchronous; that is, it doesn’t make the process wait until a corresponding *receive* has been executed.

What problem would occur if it did?

• Since the rows are copied and then not updated by the processors they have been copied from, the boundary values are more out-of-date than they are in the sequential version of the program. This may or may not cause more sweeps to be needed for convergence.

• The indexes used to reference variables are *local* indexes, not the “real” indexes that would be used if array $A$ were a single shared array.
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); /*read input matrix size and number of processes*/
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; /*private variables*/
15. myA ← malloc(a 2-d array of size [n/nprocs + 2] by n+2); /*my assigned rows of A*/
16. initialize(myA); /*initialize my rows of A, in an unspecified way*/
17. while (!done) do /*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,*]*/
18. for i ← 1 to n' do /*for each of my (nonghost) rows*/
19. for j ← 1 to n do /*for all nonborder elements in that row*/
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 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
Alternatives for send and receive

**send** and **receive** operations may be either—

- **synchronous**, meaning that the program can continue past the **send** (**receive**) only when the corresponding **receive** (**send**) has completed and returned an ack to the sender (receiver).

  In this case, once the operation completes, both processes “know” that the message has been transferred successfully.

- **blocking asynchronous**, meaning that the sender can continue when the message has been taken out of the source processor’s buffer, and is therefore in the care of the system.

  This allows the sender to continue, and to modify the data in its buffer before the receiver has finished receiving it.

  After the **receive** is complete, the sender receives an ack.

  A **receive** returns control after the information has been removed from the buffer in put in the receiver’s address space.

- **nonblocking asynchronous**, meaning that a **send** completes immediately, and a **receive** returns control after specifying the intent to receive.

  This allows the greatest concurrency, but also means that the program isn’t guaranteed that the operations will complete successfully. So the program needs to check for successful completion later.
The various forms of send and receive trade off ________________ for