Module 3: Fundamentals of Parallel Computers: ILP vs TLP

Lecture 6: Preliminaries of Parallel Programming

The Lecture Contains:

Previous Next

Parallel Programming

Prolog: Why Bother?

- As an architect why should you be concerned with parallel programming?
  - Understanding program behavior is very important in developing high-performance computers
  - An architect designs machines that will be used by the software programmers: so need to understand the needs of a program
  - Helps in making design trade-offs and cost/performance analysis i.e. what hardware feature is worth supporting and what is not
  - Normally an architect needs to have a fairly good knowledge in compilers and operating systems

Agenda

- Steps in writing a parallel program
- Example

Writing a Parallel Program

- Start from a sequential description
- Identify work that can be done in parallel
- Partition work and/or data among threads or processes
  - Decomposition and assignment
- Add necessary communication and synchronization
  - Orchestration
- Map threads to processors (Mapping)
- How good is the parallel program?
  - Measure speedup = sequential execution time/parallel execution time = number of processors ideally

Previous Next

Some Definitions

- Task
  - Arbitrary piece of sequential work
  - Concurrency is only across tasks
  - Fine-grained task vs. coarse-grained task: controls granularity of parallelism (spectrum of grain: one instruction to the whole sequential program)
- Process/thread
  - Logical entity that performs a task
  - Communication and synchronization happen between threads
- Processors
  - Physical entity on which one or more processes execute

Decomposition

- Find concurrent tasks and divide the program into tasks
  - Level or grain of concurrency needs to be decided here
  - Too many tasks: may lead to too much of overhead communicating and synchronizing between tasks
  - Too few tasks: may lead to idle processors
  - Goal: Just enough tasks to keep the processors busy
- Number of tasks may vary dynamically
  - New tasks may get created as the computation proceeds: new rays in ray tracing
  - Number of available tasks at any point in time is an upper bound on the achievable speedup

Previous Next

Static Assignment

- Given a decomposition it is possible to assign tasks statically
  - For example, some computation on an array of size N can be decomposed statically by assigning a range of indices to each process: for k processes P 0 operates on indices 0 to (N/k)-1, P 1 operates on N/k to (2N/k)-1,…, P k-1 operates on (k-1)N/k to N-1
  - For regular computations this works great: simple and low-overhead
- What if the nature of computation depends on the index?
  - For certain index ranges you do some heavy-weight computation while for others you do something simple
  - Is there a problem?

Dynamic Assignment

- Static assignment may lead to load imbalance depending on how irregular the application is
- Dynamic decomposition/assignment solves this issue by allowing a process to dynamically choose any available task whenever it is done with its previous task
  - Normally in this case you decompose the program in such a way that the number of available tasks is larger than the number of processes
  - Same example: divide the array into portions each with 10 indices; so you have N/10 tasks
  - An idle process grabs the next available task
  - Provides better load balance since longer tasks can execute concurrently with the smaller ones

Previous Next

Dynamic Assignment

- Dynamic assignment comes with its own overhead
  - Now you need to maintain a shared count of the number of available tasks
  - The update of this variable must be protected by a lock
  - Need to be careful so that this lock contention does not outweigh the benefits of dynamic decomposition
- More complicated applications where a task may not just operate on an index range, but could manipulate a subtree or a complex data structure
  - Normally a dynamic task queue is maintained where each task is probably a pointer to the data
  - The task queue gets populated as new tasks are discovered

Decomposition Types

- Decomposition by data
  - The most commonly found decomposition technique
  - The data set is partitioned into several subsets and each subset is assigned to a process
  - The type of computation may or may not be identical on each subset
  - Very easy to program and manage
- Computational decomposition
  - Not so popular: Tricky to program and manage
  - All processes operate on the same data, but probably carry out different kinds of computation
  - More common in systolic arrays, pipelined graphics processor units (GPUs) etc.

Previous Next

Orchestration

- Involves structuring communication and synchronization among processes, organizing data structures to improve locality, and scheduling tasks
  - This step normally depends on the programming model and the underlying architecture
- Goal is to
  - Reduce communication and synchronization costs
  - Maximize locality of data reference
  - Schedule tasks to maximize concurrency: do not schedule dependent tasks in parallel
  - Reduce overhead of parallelization and concurrency management (e.g., management of the task queue, overhead of initiating a task etc.)

Mapping

- At this point you have a parallel program
  - Just need to decide which and how many processes go to each processor of the parallel machine
- Could be specified by the program
  - Pin particular processes to a particular processor for the whole life of the program; the processes cannot migrate to other processors
- Could be controlled entirely by the OS
  - Schedule processes on idle processors
  - Various scheduling algorithms are possible e.g., round robin: process#k goes to processor#k
  - NUMA-aware OS normally takes into account multiprocessor-specific metrics in scheduling
- How many processes per processor? Most common is one-to-one

Previous Next

An Example

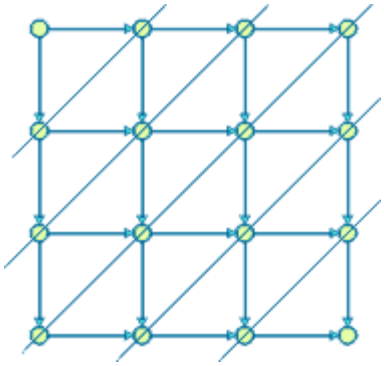
- Iterative equation solver
  - Main kernel in Ocean simulation
  - Update each 2-D grid point via Gauss-Seidel iterations
  - "A[i,,j] = 0.2(A[i,,j]+A[i,,j+1]+A[i,,j-1]+A[i+1,,j]+A[i-1,,j]")
  - Pad the n by n grid to (n+2) by (n+2) to avoid corner problems
  - Update only interior n by n grid
  - One iteration consists of updating all n 2 points in-place and accumulating the difference from the previous value at each point
  - If the difference is less than a threshold, the solver is said to have converged to a stable grid equilibrium

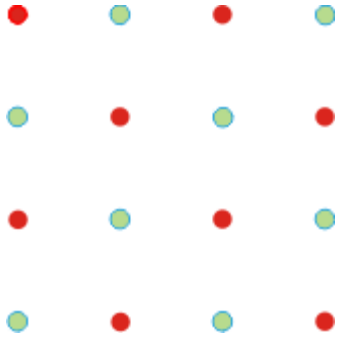
Sequential Program

```
int n;                          begin Solve (A)
float **A, diff;                int i , j, done = 0;
begin main()                    float temp;
read (n); /* size of grid */    while (!done)
Allocate (A);                   diff = 0.0;|
Initialize (A);                 for i = 0 to n-1
Solve (A);                      for j = 0 to n-1
end main                        temp = A[ i,j ];
                                A[ i,,j ] = 0.2(A[ i,,j ]+A[i,,j+1]+A[i,,j-1]+ A[i-1,,j]+A[i+1,,j];)
                                diff += fabs (A[ i,j ] - temp);
                                endfor
                                endfor
                                if (diff/(n*n) < TOL) then done = 1; endwhile
                                end Solve
```

Decomposition

- Look for concurrency in loop iterations
  - In this case iterations are really dependent
  - Iteration (i , j) depends on iterations (i , j-1) and (i-1, j)





  - Each anti-diagonal can be computed in parallel
  - Must synchronize after each anti-diagonal (or pt-to-pt)
  - Alternative: red-black ordering (different update pattern)

- Can update all red points first, synchronize globally with a barrier and then update all black points
  - May converge faster or slower compared to sequential program
  - Converged equilibrium may also be different if there are multiple solutions
  - Ocean simulation uses this decomposition
- We will ignore the loop-carried dependence and go ahead with a straight-forward loop decomposition
  - Allow updates to all points in parallel
  - This is yet another different update order and may affect convergence
  - Update to a point may or may not see the new updates to the nearest neighbors (this parallel algorithm is non-deterministic)

Previous Next

Decomposition

```
while (!done)
diff = 0.0;
for_all i = 0 to n-1
for_all j = 0 to n-1
temp = A[ i , j];
A[ i , j] = 0.2(A[ i , j]+A[ i , j+1]+A[ i , j-1]+A[i-1, j]+A[i+1, j]; )
diff += fabs (A[ i , j] – temp);
end for_all
end for_all
if (diff/(n*n) < TOL) then done = 1;
end while
```

- Offers concurrency across elements: degree of concurrency is n 2
- Make the j loop sequential to have row-wise decomposition: degree n concurrency

Assignment

- Possible static assignment: block row decomposition
  - Process 0 gets rows 0 to (n/p)-1, process 1 gets rows n/p to (2n/p)-1 etc.
- Another static assignment: cyclic row decomposition
  - Process 0 gets rows 0, p, 2p,…; process 1 gets rows 1, p+1, 2p+1,….
- Dynamic assignment
  - Grab next available row, work on that, grab a new row,…
- Static block row assignment minimizes nearest neighbor communication by assigning contiguous rows to the same process

Previous Next