Lecture 2: Parallel Programs

- Topics: consistency, parallel applications, parallelization process

Sequential Consistency

- A multiprocessor is sequentially consistent if the result of the execution is achievable by maintaining program order within a processor and interleaving accesses by different processors in an arbitrary fashion

- For example, the code below should ensure mutual exclusion on a sequentially consistent machine

```
Initially A = B = 0
  P1                    P2
A ← 1                 B ← 1
 …                      …
if (B == 0)           if (A == 0)
  Crit.Section          Crit.Section
```

Relaxing Memory Ordering

```
Initially A = B = 0
  P1                  P2
A ← 1              B ← 1
 …                   …
if (B == 0)        if (A == 0)
  Crit.Section       Crit.Section
```

- Executing memory accesses in order is extremely slow; we attempt optimizations → seq consistency is lost
- For example, each processor can be out-of-order; within P1, the write to A and the read of B are independent since they refer to different memory locations
- Ooo execution will allow each process to enter CS

Relaxed Consistency Models

- In order to write correct programs, the programmer must understand that memory accesses do not always happen in order

- The consistency model specifies how memory ordering differs from that of sequential consistency

- If the programmer demands sequential consistency in places, he/she can impose it with special fence instructions – a fence ensures that we make progress only after completing earlier memory accesses

- Fences are slow – a better understanding of the program and the consistency model can eliminate some fences

Parallel Application Examples

- Simulating ocean currents
- Simulating evolution of galaxies
- Visualizing complex scenes using raytracing
- Mining data for associations

Ocean

- Simulates motion of water currents, influenced by wind, friction, etc.

- We examine a horizontal cross-section of the ocean at a time and the cross-section is modeled as a grid of equidistant points

- At each time step, the value of each variable at each grid point is updated based on neighboring values and equations of motion

- Apparently high concurrency

Barnes-Hut

- Problem studies how galaxies evolve by simulating mutual gravitational effects of *n* bodies

- A naïve algorithm computes pairwise interactions in every time step (O(n^2)) – a hierarchical algorithm can achieve good accuracy and run in O(*n log n*) by grouping distant stars

- Apparently high concurrency, but varying star density can lead to load balancing issues

Data Mining

- Data mining attempts to identify patterns in transactions
- For example, association mining computes sets of commonly purchased items and the conditional probability that a customer purchases a set, given they purchased another set of products
- Itemsets are iteratively computed – an itemset of size *k* is constructed by examining itemsets of size *k-1*
- Database queries are simpler and computationally less expensive, but also represent an important benchmark for multiprocessors

Parallelization Process

- Ideally, a parallel application must be constructed by designing a parallel algorithm from scratch

- In most cases, we begin with a sequential version – the quest for efficient automated parallelization continues…

- Converting a sequential program involves:
  - Decomposition of the computation into *tasks*
  - Assignment of tasks to *processes*
  - Orchestration of data access, communication, and synchronization
  - Mapping or binding processes to processors

Partitioning

- Decomposition and Assignment are together called partitioning – partitioning is algorithmic, while orchestration is a function of the programming model and architecture

- The number of tasks at any given time is the level of concurrency in the application – the average level of concurrency places a bound on speedup (Amdahl's Law)

- To reduce inter-process communication or load imbalance, many tasks may be assigned to a single process – this assignment may be either static or dynamic

- We will assume that processes do not migrate (fixed mapping) in order to preserve locality

Parallelization Goals

Step	Architecture-Dependent?	Major Performance Goals
Decomposition	Mostly no	Expose enough concurrency
Assignment	Mostly no	Balance workload Reduce communication volume
Orchestration	Yes	Reduce communication via data locality Reduce communication and synch cost Reduce serialization at shared resources Schedule tasks to satisfy dependences early
Mapping	Yes	Put related processes on same processor Exploit locality in network topology

Case Study: Ocean Kernel

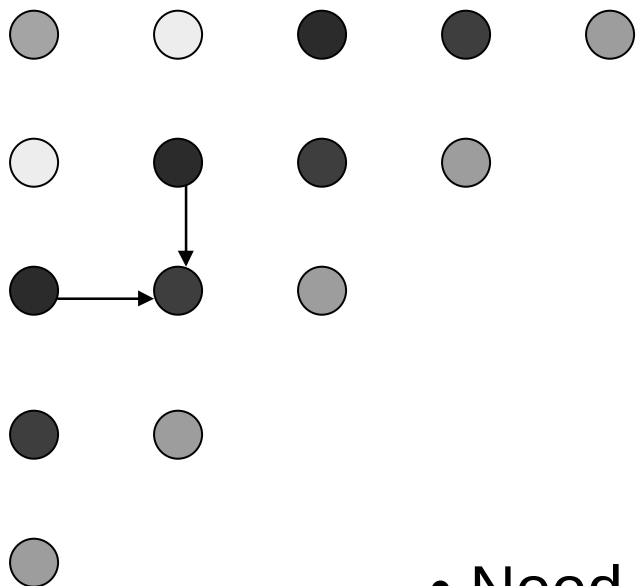
- Gauss-Seidel method: sweep through the entire 2D array and update each point with the average of its value and its neighboring values; repeat until the values converge

- Since we sweep from top to bottom and left to right, the averaging step uses new values for the top and left neighbors, and old values for the bottom and right neighbors

Ocean Kernel

```
Procedure Solve(A)
begin
  diff = done = 0;
  while (!done) do
      diff = 0;
      for i ← 1 to n do
         for j ← 1 to n do
            temp = A[i,j];
            A[i,j] ← 0.2 * (A[i,j] + neighbors);
            diff += abs(A[i,j] – temp);
         end for
      end for
      if (diff < TOL) then done = 1;
  end while
end procedure
```

Concurrency



- Need synch after every anti-diagonal
- Potential load imbalance

Algorithmic Modifications

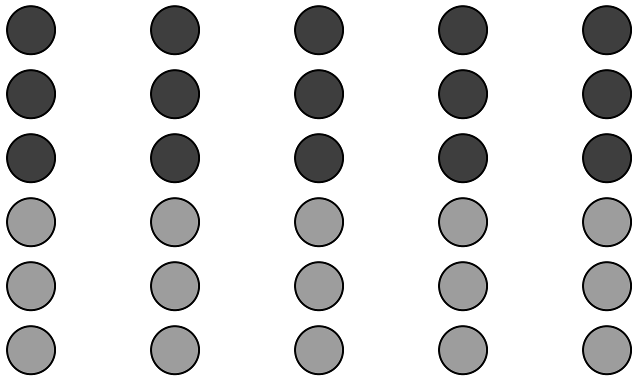
- Red-Black ordering: the grid is colored red and black similar to a checkerboard; sweep through all red points, then sweep through all black points; there are no dependences within a sweep

- Asynchronous updates: ignore dependences within a sweep → you may or may not get the most recent value

- Either of these algorithms expose sufficient concurrency, but you may or may not converge quickly

Assignment

- With the asynchronous method, each process can be assigned a subset of all rows



- What is the degree of concurrency?

- What is the communication to computation ratio

Orchestration

- Orchestration is a function of the programming model and architecture

- Consider the shared address space model – by using the following primitives, the program appears very similar to the sequential version:
  - CREATE: creates *p* processes that start executing at procedure *proc*
  - LOCK and UNLOCK: acquire and release mutually exclusive access
  - BARRIER: global synchronization: no process gets past the barrier until *n* processes have arrived
  - WAIT_FOR_END: wait for *n* processes to terminate

Shared Address Space Model

```
int  n, nprocs;
float  **A, diff;
LOCKDEC(diff_lock);
BARDEC(bar1);


main()
begin
   read(n); read(nprocs);
   A ← G_MALLOC();
   initialize (A);
   CREATE (nprocs,Solve,A);
   WAIT_FOR_END (nprocs);
end main
```

```
procedure Solve(A)
    int i, j, pid, done=0;
    float temp, mydiff=0;
    int mymin = 1 + (pid * n/procs);
    int mymax = mymin + n/nprocs -1;
    while (!done) do
       mydiff = diff = 0;
       BARRIER(bar1,nprocs);
       for i ← mymin to mymax
          for j ← 1 to n do
             …
          endfor
       endfor
       LOCK(diff_lock);
       diff += mydiff;
       UNLOCK(diff_lock);
       BARRIER (bar1, nprocs);
       if (diff < TOL) then done = 1;
       BARRIER (bar1, nprocs);
    endwhile
```

Title

- Bullet