Parallel Programming

Kai Shen

Parallel Programming Steps

Converting a sequential application to a parallel one

- Decomposition into tasks
- Assign tasks to processors
- Orchestrate data access and synchronization

Task Decomposition

- Decomposition:
  - over different functions
  - over different data segments (over loop iterations)
- Two tasks are dependent if they must follow their order in the sequential program so that the program execution results aren't changed.
- Three types of data dependences:
  - Read-after-write
  - Write-after-read
  - Write-after-write
- Only "read-after-write" is called true dependence

Task Decomposition

- How good is the decomposition?
  - How much parallelism in the resulted parallel program?
  - Best-case running time under parallel execution – Assuming dependent tasks must run serially, we can build a DAG of task dependencies and critical path length indicates lower bound of parallel execution time.
- Tradeoff on task granularity:
  - smaller tasks may offer more parallelism/concurrency
  - smaller tasks require more management/programming overhead

Task Assignment

Assign tasks to processors
- You often have more tasks than the number of processors

Goals:
- Load balance
- Minimize inter-processor data movement ⇨ maximize locality

Ways:
- Static assignment (possibly poor load balancing)
- Dynamic assignment

Orchestration

- Access shared data
  - Shared-memory parallel platform
  - Distributed-memory parallel platform
- Performance implication: remote data access is expensive.

Orchestration

- Synchronization
  - Mechanism to enforce execution ordering between parallel tasks
  - Maintain dependences
  - Avoid races: e.g., "`counter++`" may be compiled into the following instruction sequence:

```
register1 = counter;
register1 = register1 + 1;
counter   = register1;
```

    Image two tasks running "`counter++`" in parallel.
- Synchronization primitives: mutex lock, condition, barrier, …
- Performance implication: synchronization incurs costs
  - Time to execute synchronization primitives; cost of execution blocking
  - Granularity of synchronization

Dynamic Task Assignment

- How does it work?
  - Maintain a centralized queue of ready tasks, protected by synchronization primitives like mutex lock
  - Each thread grabs a task at the beginning; grabs another task after completing the current one
  - New tasks may be generated on the fly and added to queue
- Advantage: good load balancing
- Disadvantages with dynamic task assignment
  - Data locality may be lost in the interests of load balancing
  - Synchronization contention on manipulating the task queue

Dynamic Task Assignment

- Disadvantages with dynamic task assignment
  - Locality may be lost in the interests of load balancing
  - Synchronization contention on manipulating the task queue
- Fixable through distributed queues with work stealing
  - Each thread has an exclusive task queue with good locality and no contention
  - When out of work (load imbalance), steal some task from another queue with ready tasks

Parallel Programming Example: Successive Over Relaxation

- SOR implements a mathematical model for many natural phenomena, e.g., heat dissipation, ocean currents
- Given a 2D grid of data, for some number of iterations:
  - For each internal grid point, compute average of its four neighbors

```
for (i=1; i<n; i++)
   for (j=1; j<n; j++)
      temp[i][j] = 0.25 * (grid[i-1][j]+grid[i+1][j]+grid[i][j-1]+grid[i][j+1]);
for( i=1; i<n; i++)
   for( j=1; j<n; j++)
      grid[i][j] = temp[i][j];
```


Parallel Programming Example: Successive Over Relaxation

```
for (i=1; i<n; i++)
   for (j=1; j<n; j++)
      temp[i][j] = 0.25 * (grid[i-1][j]+grid[i+1][j]+grid[i][j-1]+grid[i][j+1]);
for( i=1; i<n; i++)
   for( j=1; j<n; j++)
      grid[i][j] = temp[i][j];
```

- Dependences:
  - First (i,j) loop nest?
  - Second (i,j) loop nest?
  - Between the two loop nests?
  - Between two iterations?

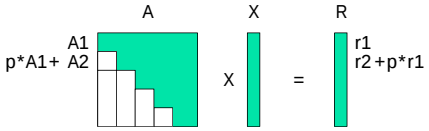
Parallel Programming Example: Successive Over Relaxation

```
for (i=1; i<n; i++)
   for (j=1; j<n; j++)
      temp[i][j] = 0.25 * (grid[i-1][j]+grid[i+1][j]+grid[i][j-1]+grid[i][j+1]);
for( i=1; i<n; i++)
   for( j=1; j<n; j++)
      grid[i][j] = temp[i][j];
```

- Task decomposition:
  - 1D partitioning: each task manages some columns or rows
  - 2D partitioning: each task manages a 2D block of the grid
- Impact on data movement/communication?

Parallel Programming Example: Gaussian Elimination

- Solving a system of linear equations
- Reduce an equation matrix into an equivalent upper-diagonal matrix



- Partial pivoting to maintain numerical stability