COSC 6374
Parallel Computation

Algorithm structure

Edgar Gabriel
Spring 2008

Parallelization Strategy

Finding Concurrency
Structure the problem to expose exploitable concurrency

Algorithm Structure
Structure the algorithm to take advantage of concurrency

Supporting Structure
Intermediate stage between Algorithm Structure and Implementation
- program structuring
- definition of shared data structures

Implementation Mechanism
Mapping of the higher level patterns onto a programming environment
Finding concurrency

- Result
  - A task decomposition that identifies tasks that can execute concurrently
  - A data decomposition that identifies data local to each task
  - A way of grouping tasks and ordering them according to temporal constraints

Algorithm structure

- Organize by tasks
  - Task Parallelism
  - Divide and Conquer
- Organize by data decomposition
  - Geometric decomposition
  - Recursive data
- Organize by flow of data
  - Pipeline
  - Event-based coordination
Task parallelism (I)

- Problem can be decomposed into a collection of tasks that can execute concurrently
- Tasks can be completely independent (embarrassingly parallel) or can have dependencies among them
- All tasks might be known at the beginning or might be generated dynamically

Task parallelism (II)

- Tasks:
  - There should be at least as many tasks as UEs (typically many, many more)
  - Computation associated with each task should be large enough to offset the overhead associated with managing tasks and handling dependencies
- Dependencies:
  - Ordering constraints: sequential composition of task-parallel computations
  - Shared-data dependencies: several tasks have to access the same data structure
Shared data dependencies

• Shared data dependencies can be categorized as follows:
  - Removable dependencies: an apparent dependency that can be removed by code transformation

```c
int i, ii=0, jj=0;
for (i=0; i<N; i++ ) {
  ii = ii + 1;
  d[ii] = big_time_consuming_work (ii);
  jj = jj + ii;
  a[jj] = other_big_time_consuming_work (jj);
}
```

```c
for (i=0; i<N; i++ ) {
  d[i] = big_time_consuming_work (i);
  a[(i*i)/2] = other_big_time_consuming_work((i*i)/2);
}
```

Shared data dependencies (II)

- Separable dependencies: replicate the shared data structure and combine the copies into a single structure at the end
  - Remember the matrix-vector multiply using column-wise block distribution in the first MPI lecture?
- Other dependencies: non-resolvable, have to be followed
Task scheduling

- Schedule: the way in which tasks are assigned to UEs for execution
- Goal: load balance - minimize the overall execution of all tasks
- Two classes of schedule:
  - Static schedule: distribution of tasks to UEs is determined at the start of the computation and not changed anymore
  - Dynamic schedule: the distribution of tasks to UEs changes as the computation proceeds
Static schedule

- Tasks are associated into blocks
- Blocks are assigned to UEs
- Each UE should take approximately same amount of time to complete task
- Static schedule usually used when
  - Availability of computational resources is predictable (e.g. dedicated usage of nodes)
  - UEs are identical (e.g. homogeneous parallel computer)
  - Size of each task is nearly identical

Dynamic scheduling

- Used when
  - Effort associated with each task varies widely/is unpredictable
  - Capabilities of UEs vary widely (heterogeneous parallel machine)
- Common implementations:
  - usage of task queues: if a UE finishes current task, it removes the next task from the task-queue
  - Work-stealing:
    - each UE has its own work queue
    - once its queue is empty, a UE steals work from the task queue of another UE
Dynamic scheduling

- Trade-offs:
  - Fine grained (=shorter, smaller) tasks allow for better load balance
  - Fine grained task have higher costs for task management and dependency management

Divide and Conquer algorithms
Divide and Conquer

- A problem is split into a number of smaller sub-problems
- Each sub-problem is solved independently
- Sub-solutions of each sub-problem will be merged to the solution of the final problem
- Problems of Divide and Conquer for Parallel Computing:
  - Amount of exploitable concurrency decreases over the lifetime
  - Trivial parallel implementation: each function call to solve is a task on its own. For small problems, no new task should be generated, but the baseSolve should be applied

Divide and Conquer

- Implementation:
  - On shared memory machines, a divide and conquer algorithm can easily be mapped to a fork/join model
    - A new task is forked (=created)
    - After this task is done, it joins the original task (=destroyed)
  - On distributed memory machines: task queues
    - Often implemented using the Master/Worker framework - discussed later in this course
Divide and Conquer

```c
int solve ( Problem P )
{
    int solution;

    /* Check whether we can further partition the problem */
    if (baseCase(P) ) {
        solution = baseSolve(P);   /* No, we can’t */
    } else {                     /* yes, we can */
        Problem subproblems[N];
        int subsolutions[N];

        subproblems = split (P);   /* Partition the problem */
        for ( i=0; i < N; i++ ) {
            subsolutions[i] = solve ( subproblems[i]);
        }
        solution = merge (subsolutions);
    }
    return ( solution );
}
```

Task Parallelism using Master-Worker framework
Task Parallelism using work stealing

Worker Process 1  Worker Process 2

Geometric decomposition

- For all applications relying on data decomposition
  - All processes should apply the same operations on different data items
- Key elements:
  - Data decomposition
  - Exchange and update operation
  - Data distribution and task scheduling
2-D Example Laplace equation

- Parallel domain decomposition
  - Data exchange at process boundaries required
- Halo cells / Ghost cells
  - Copy of the last row/column of data from the neighbor process

Recursive Data

- Typically applied in recursive data structures
  - Lists, trees, graphs
- Data decomposition: recursive data structure is completely decomposed into individual elements
- Example: prefix scan operation
  - Each process has an element of an overall structure such as a linked list, e.g. an integer $x_i$
    - Lets denote the value of the $x$ on process $i$ as $x_i$
  - At the end of the prefix scan operation process $k$ holds the sum of all elements of $x_i$ for $i=0...k$
Recursive data (II)

- Example for eight processes

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_0$</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>$x_3$</td>
<td>$x_4$</td>
<td>$x_5$</td>
<td>$x_6$</td>
<td>$x_7$</td>
</tr>
<tr>
<td>Before prefix scan</td>
<td>After prefix scan</td>
<td>Before prefix scan</td>
<td>After prefix scan</td>
<td>Before prefix scan</td>
<td>After prefix scan</td>
<td>Before prefix scan</td>
<td>After prefix scan</td>
</tr>
<tr>
<td>$x_0$</td>
<td>$x_0 + x_1$</td>
<td>$x_0 + x_1$</td>
<td>$x_2 + x_3$</td>
<td>$x_4 + x_5$</td>
<td>$x_0 + x_1$</td>
<td>$x_2 + x_3$</td>
<td>$x_4 + x_5$</td>
</tr>
</tbody>
</table>

Sequential implementation

Each process forwards its sum to the next process
- n messages/ time steps required for n processes
Recursive data approach

\[ \sum (x_0 : x_0) \sum (x_1 : x_1) \sum (x_2 : x_2) \sum (x_3 : x_3) \sum (x_4 : x_4) \sum (x_5 : x_5) \sum (x_6 : x_6) \sum (x_7 : x_7) \]

Recursive Data (III)

- Very fine grained concurrency
- Restructuring of the original algorithm often required
- Parallel algorithm requires substantially more work, which can however be executed in less time-steps
Algorithm structure - Pipeline pattern

- Calculation can be viewed in terms of **data flowing** through a sequence of stages
- Computation performed on many data sets
  - Compare to pipelining in processors on the instruction level

Pipeline pattern (II)

- Amount of concurrency limited to the number of stages of the pipeline
- Patterns works best, if amount of work performed by various stages is roughly equal
- Filling the pipeline: some stages will be idle
- Draining the pipeline: some stages will be idle
- Non-linear pipeline: pattern allows for different execution for different data items
Pipeline pattern (III)

- Implementation:
  - Each stage typically assigned to a process/thread
  - A stage might be a data-parallel task itself
  - Computation per task has to be large enough to compensate for communication costs between the tasks

Algorithm structure - Event-based coordination

- Pipeline pattern assumes a regular, non-changing data flow
- Event-based coordination assumes irregular interaction between tasks
- Real world example:
  - Data items might flow in both directions
  - Each data item might take a different path
- Major problem: deadlock avoidance
Event-based coordination (II)

• Examples:
  - Any discrete event simulations
    • E.g. traffic control systems
    • Simulation of digital circuits