Creating a Parallel Program

- Assumption: Sequential algorithm is given
  - Sometimes need very different algorithm, but beyond scope

- Pieces of the job:
  - Identify work that can be done in parallel
  - Partition work and perhaps data among processes
  - Manage data access, communication and synchronization
  - Note: work includes computation, data access and I/O

- Main goal: Speedup (plus low prog. effort and resource needs)

  - \[ \text{Speedup} (p) = \frac{\text{Performance}(p)}{\text{Performance}(1)} \]

  - For a fixed problem:

  - \[ \text{Speedup} (p) = \frac{\text{Time}(1)}{\text{Time}(p)} \]
Steps in Creating a Parallel Program

- 4 steps: Decomposition, Assignment, Orchestration, Mapping
  - Done by programmer or system software (compiler, runtime, ...)
  - Issues are the same, so assume programmer does it all explicitly

Some Important Concepts

- Task:
  - Arbitrary piece of undecomposed work in parallel computation
  - Executed sequentially; concurrency is only across tasks
  - E.g. a particle/cell in Barnes-Hut, a ray or ray group in Raytrace
  - Fine-grained versus coarse-grained tasks

- Process (thread):
  - Abstract entity that performs the tasks assigned to processes
  - Processes communicate and synchronize to perform their tasks

- Processor:
  - Physical engine on which process executes
  - Processes virtualize machine to programmer
    - first write program in terms of processes, then map to processors

© C. Xu, 2001
Decomposition

- Break up computation into tasks to be divided among processes
  - Tasks may become available dynamically
  - No. of available tasks may vary with time
- i.e. identify concurrency and decide level at which to exploit it

- Goal: Enough tasks to keep processes busy, but not too many
  - No. of tasks available at a time is upper bound on achievable speedup

Limited Concurrency: Amdahl’s Law

- Most fundamental limitation on parallel speedup
- If fraction $s$ of seq execution is inherently serial, speedup $\leq 1/s$
- Example: 2-phase calculation
  - sweep over $n$-by-$n$ grid and do some independent computation
  - sweep again and add each value to global sum
- Time for first phase $= n^2/p$
- Second phase serialized at global variable, so time $= n^2$
- Speedup $\leq \frac{2n^2}{\frac{n^2}{p} + n^2}$ or at most 2
- Trick: divide second phase into two
  - accumulate into private sum during sweep
  - add per-process private sum into global sum
- Parallel time is $n^2/p + n^2/p + p$, and speedup at best $\frac{2n^2}{2n^2 + p^2}$
Pictorial Depiction

Concurrency Profiles

- Cannot usually divide into serial and parallel part
Concurrency Profiles

- Area under curve is total work done, or time with 1 processor
- Horizontal extent is lower bound on time (infinite processors)
- Speedup = Area under concurrency profile / Horizontal extent

\[
\text{Speedup} = \frac{\sum_{k=1}^{\infty} f_k k}{\sum_{k=1}^{\infty} f_k \left\lfloor \frac{k}{p} \right\rfloor}, \quad \text{base case: } \frac{1}{s + \frac{L_s}{p}}
\]

where \( f_k \) is the number of x-axis points that have concurrency \( k \)

Assignment

- Specifying mechanism to divide work up among processes
  - E.g. which process computes forces on which stars, or which rays
  - Together with decomposition, also called partitioning
  - Balance workload, reduce communication and management cost

- Structured approaches usually work well
  - Code inspection (parallel loops) or understanding of application
  - Well-known heuristics
    - Static versus dynamic assignment

- As programmers, we worry about partitioning first
  - Usually independent of architecture or prog model
  - But cost and complexity of using primitives may affect decisions

- As architects, we assume program does reasonable job of it
Orchestration

- Naming data
- Structuring communication
- Synchronization
  - Organizing data structures and scheduling tasks temporally

- Goals
  - Reduce cost of communication and synch. as seen by processors
  - Reserve locality of data reference (incl. data structure organization)
  - Schedule tasks to satisfy dependences early
  - Reduce overhead of parallelism management

- Closest to architecture (and programming model & language)
  - Choices depend a lot on comm. abstraction, efficiency of primitives
  - Architects should provide appropriate primitives efficiently

© C. Xu, 2001

Mapping

- After orchestration, already have parallel program

- Two aspects of mapping:
  - Which processes will run on same processor, if necessary
  - Which process runs on which particular processor
    - mapping to a network topology

- One extreme: \textit{space-sharing}
  - Machine divided into subsets, only one app at a time in a subset
  - Processes can be pinned to processors, or left to OS

- Another extreme: complete resource management control to OS
  - OS uses the performance techniques we will discuss later

- Real world is between the two
  - User specifies desires in some aspects, system may ignore

- Usually adopt the view: process \textless{}-> processor

© C. Xu, 2001
### Parallelizing Computation vs. Data

- Above view is centered around computation
  - Computation is decomposed and assigned (partitioned)

- Partitioning Data is often a natural view too
  - Computation follows data: *owner computes*
  - Grid example; data mining; High Performance Fortran (HPF)

- But not general enough
  - Distinction between comp. and data stronger in many applications
    - Barnes-Hut, Raytrace (later)
  - Retain computation-centric view
  - Data access and communication is part of orchestration

### High-level Goals

High performance (speedup over sequential program)

<table>
<thead>
<tr>
<th>Step</th>
<th>Architecture-Dependent?</th>
<th>Major Performance Goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decomposition</td>
<td>Mostly no</td>
<td>Expose enough concurrency but not too much</td>
</tr>
<tr>
<td>Assignment</td>
<td>Mostly no</td>
<td>Balance workload</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reduce communication volume</td>
</tr>
<tr>
<td>Orchestration</td>
<td>Yes</td>
<td>Reduce noninherent communication via data locality</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reduce communication and synchronization cost as seen by the processor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reduce serialization at shared resources</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Schedule tasks to satisfy dependences early</td>
</tr>
<tr>
<td>Mapping</td>
<td>Yes</td>
<td>Put related processes on the same processor if necessary</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Exploit locality in network topology</td>
</tr>
</tbody>
</table>

© C. Xu, 2001
High-level Goals

- Implications for algorithm designers and architects

  - **Algorithm designers:**
  - high-performance and low resource needs

  - **Architects:**
  - high-performance, low cost, reduced programming effort
  - e.g. gradually improving perf. with programming effort
    may be preferable to sudden threshold after large
    programming effort

What Parallel Programs Look Like
Parallelization of An Example Program

- Motivating problems all lead to large, complex programs
- Examine a simplified version of a piece of Ocean simulation
  - Iterative equation solver
- Illustrate parallel program in low-level parallel language
  - C-like pseudocode with simple extensions for parallelism
  - Expose basic comm. and synch. primitives that must be supported
  - State of most real parallel programming today

Grid Solver Example

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Expression for updating each interior point:


- Simplified version of solver in Ocean simulation

© C. Xu, 2001
Grid Solver Example

- Gauss-Seidel (near-neighbor) sweeps to convergence
  - interior n-by-n points of (n+2)x(n+2) updated in each sweep
  - updates done in-place in grid, and difference from previous value computed
  - accumulate partial differences into global difference at end of every sweep
  - check if error has converged (to within a tolerance parameter)
  - if so, exit solver; if not, do another sweep

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 (an (n+2)x(n+2)array);
7. initialize(A); /* initialize matrix A somehow*/
8. Solve (A); /* call the routine to solve equation*/
9. end main
procedure Solve (A) /* solve the equation system */
float **A; /* A is an (n + 2)-by-(n + 2) array */
begin
int i, j, done = 0;
float diff = 0, temp;
while (!done) do
/* outermost loop over sweeps */
diff = 0; /* initialize maximum difference to 0 */
for i ← 1 to n do
/* sweep over inter. points */
for j ← 1 to n do
    temp = A[i, j]; /* save old value of element */
    diff += abs(A[i, j] - temp);
end for
end for
if (diff/(n*n) < TOL) then done = 1;
end while
end procedure

**Decomposition**

- Simple way to identify concurrency is to look at loop iterations — *dependence analysis*; if not enough concurrency, then look further
- Not much concurrency here at this level (all loops *sequential*)
- Examine fundamental dependences, ignoring loop structure
**Decomposition**

- Concurrency $O(n)$ along anti-diagonals, serialization $O(n)$ along diagonal
- Retain loop structure, use pt-to-pt synch;
  - Problem: too many synch ops.
- Restructure loops, use global synch;
  - Problem: imbalance and too much synch

© C. Xu, 2001

**Exploit Application Knowledge**

- Reorder grid traversal: red-black ordering

- Red sweep and black sweep are each fully parallel:
- Global synch between them (conservative but convenient)
- Ocean uses red-black; we use simpler, asynchronous one to illustrate
  - no red-black, simply ignore dependences within sweep
  - sequential order same as original, parallel program *deterministic*

© C. Xu, 2001
Assignment

• Static assignments (given decomposition into rows)
  - Block assignment: Row \( i \) is assigned to process \( \left\lfloor \frac{i}{p} \right\rfloor \)

\[
\begin{array}{cccccccc}
 & & & & & & \circ & \circ \\
& & & & & & \circ & \circ \\
& & & & & & \circ & \circ \\
\end{array}
\]

\[
\begin{array}{cccccccc}
 & & & & & & \circ & \circ \\
& & & & & & \circ \circ & \circ \\
& & & & & & \circ \circ & \circ \\
\end{array}
\]

\[
\begin{array}{cccccccc}
 & & & & & & \circ \circ \circ \circ & \circ \\
& & & & & & \circ \circ \circ \circ & \circ \\
& & & & & & \circ \circ \circ \circ & \circ \\
\end{array}
\]

Cyclic assignment: Process \( i \) is assigned rows \( i, i+p, i+2p \)

Block cyclic:

Deciding How to Manage Concurrency

– *Static* versus *Dynamic* techniques

– Static:
  • Algorithmic assignment based on input; won’t change
  • Low runtime overhead
  • Computation must be predictable
  • Preferable when applicable (except in multiprogrammed/heterogeneous environment)

– Dynamic:
  • Adapt at runtime to balance load
  • Can increase communication and reduce locality
  • Can increase task management overheads
**Dynamic Assignment**

- Dynamic Tasking:
  
  - E.g. “Self-scheduling” of loop iterations
    
    - Get a row index, work on the row, then get a new row, and so on.
  
  - Deal with unpredictability in program or environment (e.g. Branch and bound optimization)
    
    - computation, communication, and memory system interactions
    
    - multiprogramming and heterogeneity
    
    - used by runtime systems and OS too
  
  - Pool of tasks; take and add tasks until done

---

**Dynamic Tasking with Task Queues**

- Centralized versus distributed queues
  
  - Task stealing with distributed queues
    
    - Can compromise comm and locality, and increase synchronization
    
    - Whom to steal from, how many tasks to steal, ...
    
    - Termination detection
    
    - Maximum imbalance related to size of task

![Diagram](attachment:image.png)

© C. Xu, 2001