Parallel Programming Overview

- Task Parallelism
- OS support for task parallelism
- Parameter Studies
- Domain Decomposition
- Sequence Matching

Work Assignment

- Static scheduling
  - Divide work into n pieces which will take equal time where n is number of workers

- Dynamic scheduling
  - If tasks are of widely different sizes (times) there is a load balancing problem
  - Assign subset of sub-tasks to slaves
  - When slave finished assign another sub-task
  - Observations:
    - Still load balancing problem at end
    - Minimize by making sub-tasks small
    - If sub-tasks too small communication overhead will impact performance adversely

Master/Slave paradigm

- Divide task into nearly parallel sub-tasks
- Start the master
- Start the slaves
- Master communicates sub-task specifications to slaves
- Slaves perform sub-tasks
- Slaves communicate results to master
- Master ensures that all results have been collected
- Shut down slaves
- Shut down master

Unix OS Concepts for Parallel Programming

- Unix Process (task)
  - Executable code
  - Instruction pointer
  - Stack
  - Logical registers
  - Heap
  - Private address space
  - Task forking to create dependent processes – thousands of clock cycles
- Thread – “lightweight process”
  - Logical registers
  - Stack
  - Shared address space
  - Hundreds of clock cycles to create/destroy/synchronize threads
Local Process execution

- All processes children of init
- Processes spawned using fork-exec combination
- Fork creates a copy of the process
  - Differs from parent only in returned value from fork
    - 0 in child, pid of child in parent
- Exec substitute another program executable for the current program image
- If not familiar with this read 7.2.2 in book for details

Remote Process Execution/File Access

- Rsh
- Ssh
  - Note that ssh slower due to encryption
  - Ssh can do X forwarding – usually syntax to turn this off (-x)
  - Can be problems with NFS mounted file system due to all nodes trying to write .Xauthority file
- NFS
- Rcp
- Scp
- ftp/sftp
- Rdist — maintain identical copies of files across hosts
- Rsync — detect differences between files on different hosts and only transfer diffs

Interprocess Communication with Sockets

- See section 7.2.5 in book
- Also http://www.cs.kent.edu/~farrell/sys2002/

Parameter Studies

- Run same sequential program multiple times with different input data (parameters)
- Trivially parallel
- Common where one wants to see which set of parameters give the best approximation to known (experimental or theoretical results)
- Also where one wants to document the effects of parameters on results
  - See example in book (section 7.3) on testing compiler optimization flags
Parallel Search

- Sequence matching in Computational Biology
  - Databases of nucleotide (RNA or DNA) or amino acid sequences
  - Encoded as strings of characters
  - Information derived by matching given string exactly or approximately against large database
- Example program for matching: BLAST
  - Uses data in FASTA format
  - A program formatdb will build an indexed database from them
  - One can then use BLAST to search for similarities
    - Get list of matches and similarities

BLAST in Parallel

- Database can be large – 1.4 million sequences
- Use parallelism to compare all against all
- Use master/slave paradigm
  - Distribute entire db to each slave
  - Slaves run BLAST with input file of subset of db – chuck
    - Chunk sent to slave over socket
  - Slaves are persistent
  - Output of slaves copies to an output directory using scp
  - Master listens on a socket for added slaves so they can come and go
  - If slaves die they can be replaced with minimal impact
  - Master keeps track of chunk status and checkpoints so restart is possible.

Generic Parallel Programming Models

Single Program Multiple Data Stream (SPMD)
- Each CPU accesses same object code
- Same application run on different data
  - Data exchange may be handled explicitly/implicitly
- "Natural" model for SIMD machines
- Most commonly used generic parallel programming model
  - Message passing
  - Shared memory
  - Usually uses process/task ID to differentiate

Multiple Program Multiple Data Stream (MPMD)
- Each CPU accesses different object code
- Each CPU has only data/instructions needed
- "Natural" model for MIMD machines

Parallel "Architectures" – Mapping Hardware Models to Programming Models
### Methods of Problem Decomposition for Parallel Programming

Want to map Problem + Algorithms + Data to Architecture

Conceptualize mapping via e.g., pseudocode

Realize mapping via programming language

- Data Decomposition - data parallel program
  - Each processor performs the same task on different data
  - Example - grid problems

- Task (Functional) Decomposition - task parallel program
  - Each processor performs a different task
  - Example - signal processing – adding/subtracting frequencies from spectrum

- Other Decomposition methods

### Categories of Parallel Problems

**Generic Parallel Problem “Architectures” (after G Fox)**

- Ideally Parallel (Embarrassingly Parallel, “Job-Level Parallel”)
  - Same application run on different data
  - Example: Parameter Studies

- Almost Ideally Parallel
  - Similar to Ideal case, but with “minimal” coordination required
  - Example: Linear Monte Carlo calculations, integrals

- Pipeline Parallelism
  - Problem divided into tasks that have to be completed sequentially
  - Can be transformed into partially sequential tasks
  - Example: DSP filtering

- Synchronous Parallelism
  - Each operation performed on all/most of data
  - Operations depend on results of prior operations
  - All processes must be synchronized at regular points
  - Example: Modeling Atmospheric Dynamics

- Loosely Synchronous Parallelism
  - Similar to Synchronous case, but with “minimal” intermittent data sharing
  - Example: Modeling Diffusion of contaminants through groundwater

### Designing and Building Parallel Applications

**Attributes of Parallel Algorithms**

- Concurrency - Many actions performed “simultaneously”
- Modularity - Decomposition of complex entities into simpler components
- Locality - Want high ratio of local memory access to remote memory access
- Usually want to minimize communication/computation ratio
- Performance
  - Measures of algorithmic “efficiency”
    - Execution time
    - Complexity usually ~ Execution Time
    - Scalability

**Partitioning**

- Break down main task into smaller ones – either identical or “disjoint”.

**Communication phase**

- Determine communication patterns for task coordination, communication algorithms.

**Agglomeration**

- Evaluate task and/or communication structures wrt performance and implementation costs. Tasks may be combined to improve performance or reduce communication costs.

**Mapping**

- Tasks assigned to processors; maximize processor utilization, minimize communication costs. Mapping may be either static or dynamic.

May have to **iterate whole process** until satisfied with expected performance

- Consider writing application in parallel, using either SPMD message-passing or shared-memory
- Implementation (software & hardware) may require rework, additional refinement or re-design
Designing and Building Parallel Applications

Partitioning
- Geometric or Physical decomposition (Domain Decomposition) – partition data associated with problem
- Functional tasks decomposition – partition into disjoint tasks, associated with problem
- Divide and Conquer – partition problem into two simpler problems of approximately equivalent “size” – iterate to produce set of indivisible subproblems

Programming Methodologies - Practical Aspects

Bulk of parallel programs written in Fortran, C, or C++
- Generally, best compiler, tool support for parallel program development

Bulk of parallel programs use MessagePassing with MPI
- Performance, portability, mature compilers, libraries for parallel program development

Data and/or tasks are split up onto different processors by:
- Distributing the data/tasks onto different CPUs, each with local memory (MPPs, MPI)
- Distribute work of each loop to different CPUs (SMPs, OpenMP, Pthreads)
- Hybrid - distribute data onto SMPs and then within each SMP distribute work of each loop (or task) to different CPUs within the box (SMP-Cluster, MPI/OMPI, LAM)

Generic Parallel Programming Software Systems

Message-Passing
- Local tasks, each encapsulating local data
- Explicit data exchange
- Supports both SPMD and MPMD
- Supports both task and data decomposition
- Most commonly used
- Example API: MPI, PVM Message-Passing libraries

Data Parallel
- Ideally SPMD
- Supports data decomposition
- Data mapping locale may be either explicit/implicit
- Example: MPI compiler

Shared-Memory
- Tasks share common address space
- No explicit transfer of data - supports both task and data decomposition
- Cache SPMD, MPMD
- Thread-based for performance, threads should be running on separate CPUs
- Example API: OpenMP, Pthreads

Hybrid: Combination of Message-Passing and Shared-Memory - supports both task and data decomposition
- Example: OpenMP / MPI

Typical Data Decomposition for Parallelism

Example: Solve 2-D Wave Equation:

Original partial differential equation:

\[ \frac{\partial^2 \Psi}{\partial t^2} = D \frac{\partial^4 \Psi}{\partial x^4} + B \frac{\partial^4 \Psi}{\partial y^4} \]

Finite Difference Approximation:

\[ f^{n+1} - f^n \approx \Delta t \left( D \frac{f^{n+1} - 2f^n + f^{n-1}}{\Delta x^2} + B \frac{f^{n+1} - 2f^n + f^{n-1}}{\Delta y^2} \right) \]
**Sending Data Between CPUs**

Finite Difference Approximation: \[
\frac{f^{n+1} - f^n}{\Delta t} = D \frac{f_{i+1}^{n+1} - 2f_i^{n+1} + f_{i-1}^{n+1}}{\Delta x^2} + B \frac{f_{j+1}^{n+1} - 2f_j^{n+1} + f_{j-1}^{n+1}}{\Delta y^2}
\]

Sample Pseudo Code:

```plaintext
if (taskid=0) then
  li = 1
  ui = 25
  lj = 1
  uj = 25
  send(1:25) = f(25,1:25)
elseif (taskid=1) then
  ... 
elseif (taskid=2) then
  ... 
elseif (taskid=3) then
  ... 
end if
```

```plaintext
do j = lj,uj
  do i = li,ui
    work on f(i,j)
  end do
end do
```

**Basics of Task Parallel Decomposition - SPMD**

Same program will run on 2 different CPUs.

Task decomposition analysis has defined 2 tasks (a and b) to be done by 2 CPUs.

```plaintext
program.f
...
initialize
... if TaskID=A then do task a
elseif TaskID=B then do task b
end if
... end program
```

```plaintext
Task A Execution Stream
... program.f
 ...
  initialize
 ...
  if TaskID=A then do task a
  elseif TaskID=B then do task b
  end if
... end program
```

```plaintext
Task B Execution Stream
... program.f
 ...
  initialize
 ...
  if TaskID=A then do task a
  elseif TaskID=B then do task b
  end if
... end program
```

**Typical Task Parallel Decomposition**

- **Signal processing**
  - Use one processor for each independent task
  - Can use more processors if one is overloaded

**Multi-Level Task Parallelism**

Implementation: MPI and OpenMP
**Parallel Application Performance Concepts**
- Parallel Speedup
- Parallel Efficiency
- Parallel Overhead
- Limits on Parallel Performance

<table>
<thead>
<tr>
<th>Parallel Application Performance Concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Parallel Speedup: $S(n) = \frac{t_s}{t_p}$</td>
</tr>
<tr>
<td>- Parallel Efficiency: $E(n) = \frac{t_s}{t_p + t_c}$</td>
</tr>
<tr>
<td>- Parallel Overhead: $t_c$</td>
</tr>
<tr>
<td>- Limits on Parallel Performance</td>
</tr>
</tbody>
</table>

**Limits of Parallel Computing**
- Theoretical upper limits
  - Amdahl’s Law
  - Gustafson’s Law
- Practical limits
  - Communication overhead
  - Synchronization overhead
  - Extra operations necessary for parallel version
- Other Considerations
  - Time used to rewrite (existing) code

<table>
<thead>
<tr>
<th>Limits of Parallel Computing</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Theoretical upper limits</td>
</tr>
<tr>
<td>- Practical limits</td>
</tr>
<tr>
<td>- Other Considerations</td>
</tr>
</tbody>
</table>

**Parallel Computing - Theoretical Performance Upper Limits**
- All parallel programs contain:
  - Parallel sections
  - Serial sections
- Serial sections limit the parallel performance
- Amdahl’s Law provides a theoretical upper limit on parallel performance for size-constant problems
Amdahl’s Law

- Amdahl’s Law places a strict limit on the speedup that can be realized by using multiple processors
  - Effect of multiple processors on run time for size-constant problems
  - Effect of multiple processors on parallel speedup, S:

\[ t_s = \left( \frac{f_s}{N} + \frac{f_p}{N} \right) \]

Where:

- \( f_s \) = serial fraction of code
- \( f_p \) = parallel fraction of code
- \( N \) = number of processors
- \( t_1 \) = sequential execution time

Gustafson’s Law

Consider scaling problem size as processor count increased

- \( T_s \) = serial execution time
- \( T_p(N,W) \) = parallel execution time for same problem, size \( W \), on \( N \) CPUs
- \( S(N,W) \) = Speedup on problem size \( W \), \( N \) CPUs

\[
S(N,W) = \frac{T_s + T_p(1,W)}{T_s + T_p(N,W)}
\]

Consider case where \( T_p(N,W) \sim W^2/N \)

\[
S(N,W) \to \frac{N T_s + N W^2}{N T_s + W^2} \to N
\]

Gustafson’s Law provides some hope for parallel applications to deliver on the promise.
Parallel Programming Analysis -
Example
Consider solving 2-D Poisson’s equation by iterative method
on a regular grid with M points –

\[ u_{ij}^{n+1} = c_1 \cdot u_{ij}^n - c_2 \cdot \left( u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n \right) \]

- \( t_c \) = time to perform a floating-point computation - addition, multiply
- \( T_{serial} \) = time to do serial iteration update = 8 \( M \cdot t_c \)
- \( T_{parallel} = 6 \cdot M \cdot t_c + \left( t_c + \frac{\sqrt{M}}{N} t_\ell \right) + 2 \cdot \frac{M}{N} t_c + \log N (t_\ell + t_c + t_\ell) \)