COSC 6374
Parallel Computation

Parallel Design Patterns (I)

Edgar Gabriel
Spring 2009

Performance Metrics (I)

- **Speedup**: how much faster does a problem run on \( p \) processors compared to 1 processor?
  
  \[
  S(p) = \frac{T_{\text{total}}(1)}{T_{\text{total}}(p)}
  \]
  
  - Optimal: \( S(p) = p \) (linear speedup)

- **Parallel Efficiency**: Speedup normalized by the number of processors
  
  \[
  E(p) = \frac{S(p)}{p}
  \]
  
  - Optimal: \( E(p) = 1.0 \)
Performance Metrics (II)

- Example: Application A takes 35 min. on a single processor, 27 on two processors and 18 on 4 processors.

\[
S(2) = \frac{35}{27} = 1.29 \\
S(4) = \frac{35}{18} = 1.94 \\
E(2) = \frac{1.29}{2} = 0.645 \\
E(4) = \frac{1.94}{4} = 0.485
\]

Amdahl’s Law (I)

- Basic idea: most applications have a (small) sequential fraction, which limits the speedup

\[
T_{\text{total}} = T_{\text{sequential}} + T_{\text{parallel}} = \frac{fT_{\text{Total}}}{p} + (1-f)T_{\text{Total}}
\]

\[
f: \text{fraction of the code which can only be executed sequentially}
\]

\[
S(p) = \frac{T_{\text{total}}(1)}{(f + \frac{1-f}{p})T_{\text{total}}(1)} = \frac{1}{f + \frac{1-f}{p}}
\]
Amdahl’s Law (II)

- Amdahl’s Law assumes, that the problem size is constant
- In most applications, the sequential part is independent of the problem size, while the part which can be executed in parallel is not.
Performance Metrics (III)

- **Scaleup**: ratio of the execution time of a problem of size \( n \) on 1 processor to the execution time of the same problem of size \( n^p \) on \( p \) processors.

\[
S_p(n) = \frac{T_{\text{total}}(1,n)}{T_{\text{total}}(p,n^p)}
\]

- Optimally, execution time remains constant, e.g.

\[
T_{\text{total}}(p,n) = T_{\text{total}}(2p,2n)
\]

Timing functions in MPI (I)

- Can be done e.g. by `gettimeofday()`
- **MPI functions provided:**

```c
double MPI_Wtime (void);
double MPI_Wtick (void);
```

- **MPI_Wtime** returns a floating-point number of seconds, representing elapsed wall-clock time since some time in the past.
  - The times returned are local to the node that called them. There is no requirement that different nodes return `the same time.``
- **MPI_Wtick**: returns the resolution of MPI_WTIME in seconds.
Timing functions in MPI (II)

double starttime, endtime, elapsedtime;
...
starttime = MPI_Wtime();
/** do some incredibly complex calculations */
endtime = MPI_Wtime();
elapsedtime = endtime - starttime;

• Timing rules:
  - Make sure you time longer than the clock resolution (e.g. on a regular LINUX box clock resolution is ~10ms)
  - Rule of thumb: >100 times the clock resolution

Communication Metrics

• Latency:
  - minimal time to send a very short message from one processor to another
  - Unit: ms, µs

• Bandwidth:
  - amount of data which can be transferred from one processor to another in a certain time frame
  - Units: Bytes/sec, KB/s, MB/s, GB/s
            Bits/sec, Kb/s, Mb/s, Gb/s,
            baud
Common benchmarks to determine communication characteristics (I)

- **Ping-pong:**
  - Process A sends a message to Process B
  - Process B sends the message back
  - Time measured for both operations and divided by two
Design patterns

- A design pattern is a way of reusing abstract knowledge about a problem and its solution
  - Patterns are devices that allow programs to share knowledge about their design
- A pattern is a description of the problem and the essence of its solution
  - Documenting patterns is one way to reuse and share the information about how it is best to solve a specific design problem
- A pattern should be sufficiently abstract to be reused in different settings
- Patterns often rely on object characteristics such as inheritance and polymorphism

History of Design Patterns

- Architect Christopher Alexander
  - *A Pattern Language* (1977)
  - *A Timeless Way of Building* (1979)
- “Gang of four”
  - Erich Gamma
  - Richard Helm
  - Ralph Johnson
  - John Vlissides
- *Design Patterns: Elements of Reusable Object-Oriented Software* (1995)
  - Many since
  - Conferences, symposia, books
Pattern elements

- Name
  - A meaningful pattern identifier
- Problem description
- Solution description
  - Not a concrete design but a template for a design solution that can be instantiated in different ways
- Consequences
  - The results and trade-offs of applying the pattern

Abstraction Levels

Patterns exist on different abstraction levels:
- **basic building blocks** (algorithms and components), provided by language or by libraries
  - e.g. hash tables, linked lists, sort algorithms, math
  - e.g. inheritance and polymorphism, encapsulation
- **design patterns**: general design problems in particular context concerning **several classes**
  - e.g. GOF-design patterns
- **architecture patterns**: architecture decisions concerning the **whole system** (or subsystem)
  - e.g. client-server, data centric etc.
Terminology (I)

- **Task:**
  - Sequence of instructions operating as a group
  - A logical part of an algorithm/program
  - Example: multiplication of two matrices. A task could be
    - Inner products between rows and columns of the matrices
    - Individual iterations of the loops involved in the matrix multiplication
  - Usually, a problem can be decomposed in several (different) ways leading to very different tasks

Terminology (II)

- **Unit of execution (UE)**
  - Generic abstraction for threads and processes
  - Process: collection of resources that enable the execution of program instructions
  - Thread: shares the process’s environment
    - Considered ‘lightweight’ compared to a process, since switching between several threads is significantly cheaper than switching between several processes.

- **Processing element (PE)**
  - Generic term for a hardware element that executes a stream of instructions.
Terminology (III)

- **Load balance**
  - How well work is distributed among different PEs
  - Goal: all PEs should spend the same time with a certain task
  - If all PEs are equal (homogeneous platform):
    - load balance can often be achieved by assigning the same number of data elements to each PE
  - If PEs are different (heterogeneous platform):
    - E.g. some processors are faster than others. For a balanced execution, the faster processes need more data elements than the slower ones

Terminology (IV)

- **Synchronization**
  - Enforces ordering constraints
    - E.g. for non-deterministic task sequences
    - E.g. for concurrent access to the same variable on a shared memory machine
  - If two events must happen at the same time, they are synchronous
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 - Overview

Decomposition
- Task Decomposition
- Data Decomposition

Dependency Analysis
- Group Tasks
- Order Tasks
- Data Sharing

Design Evaluation
Overview (II)

- Is the problem large enough to justify the efforts for parallelizing it?
- Are the key features of the problem and the data elements within the problem well understood?
- Which parts of the problem are most computationally intensive?

Task decomposition

- How can a problem be decomposed into tasks that can execute concurrently?
- Goal: a collection of (nearly) independent tasks
  - Initially, try to find as many as tasks as possible (can be merged later to form larger tasks)

- A task can correspond to
  - A function call
  - Distinct iterations of loops within the algorithm (loop splitting)
  - Any independent sections in the source code
Task decomposition - goals and constraints

- **Flexibility:** the design should be flexible enough to be able to handle any numbers of processes
  - E.g. the number and the size of each task should be a parameter for the task decomposition
- **Efficiency:** each task should include enough work to compensate for the overhead of “generating” tasks and managing their dependencies
- **Simplicity:** tasks should be defined such that it keeps debugging and maintenance easy

Task decomposition

- Two tasks A and B are considered independent if
  \[
  I_A \cap O_B = \emptyset \\
  I_B \cap O_A = \emptyset \\
  O_A \cap O_B = \emptyset 
  \]
Task decomposition - example

Solving a system of linear equations using the Conjugate Gradient Method

Given \( x_0, A, b \)

\[ p_0 = r_0 := b - Ax_0 \]

For \( i=0,1,2,... \)

\[ \alpha = r_i^T p_i \]
\[ \beta = (Ap_i)^T p_i \]
\[ \lambda = \alpha / \beta \]
\[ x_{i+1} = x_i + \lambda p_i \]
\[ r_{i+1} = r_i - \lambda Ap_i \]
\[ p_{i+1} = r_{i+1} - (Ap_{i+1})^T p_i / \beta \]

Data decomposition

- How can a problem’s data be decomposed into units that can be operated (relatively) independently?
  - Most computationally intensive parts of a problem are dealing with large data structures
- Common mechanisms
  - Array-based decomposition (e.g. see example on the next slides)
  - Recursive data structures (e.g. decomposing the parallel update of a large tree data structure)
Data decomposition - goals and constraints

- Flexibility: size and number of data chunks should be flexible (granularity)
- Efficiency:
  - Data chunks have to be large enough that the amount of work with the data chunk compensates for managing dependencies
  - Load balancing
- Simplicity:
  - Complex data types difficult to debug
  - Mapping of local indexes to global indexed often required

Data decomposition - example

- Parallel domain decomposition
Grouping tasks

- How can (small) tasks be grouped to simplify the management of dependencies?
- Tasks derived from task decomposition or data decomposition do not necessarily constitute a flat tree
  - Might be derived from the same high level operation
  - Might have similar constraints

Ordering tasks

- Tasks might be truly independent (trivial parallelism)
  - E.g. search operations
  - E.g. Monte-Carlo Simulations
- Tasks must run simultaneously
  - E.g. often occurring in data decomposition
- Temporal dependency: constraint on the order in which a collection of tasks execute
  - E.g. Task B needs the result of Task A
Temporal dependency - example
Solving a system of linear equations using the Conjugant Gradient Method

Given the vectors \( \mathbf{x}_0, \mathbf{A}, \mathbf{b} \)

\[
\mathbf{p}_0 = \mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0
\]

For \( i=0,1,2,... \)

\[
\begin{align*}
\alpha &= \mathbf{r}_i^T \mathbf{p}_i \\
\beta &= (\mathbf{A}\mathbf{p}_i)^T \mathbf{p}_i \\
\lambda &= \alpha / \beta \\
\mathbf{x}_{i+1} &= \mathbf{x}_i + \lambda \mathbf{p}_i \\
\mathbf{r}_{i+1} &= \mathbf{r}_i - \lambda \mathbf{A}\mathbf{p}_i \\
\mathbf{p}_{i+1} &= \mathbf{r}_{i+1} - (\mathbf{A}\mathbf{r}_{i+1})^T \mathbf{p}_i / \beta
\end{align*}
\]

Data sharing

- Up to this point we have
  - A collection of tasks that can execute independently
  - A data decomposition corresponding to the collection of concurrent tasks
  - Dependencies among the tasks
- Question: how is data shared amongst the tasks?
  - Shared variables (e.g. in OpenMP)
  - Message passing, e.g. ghost cell update