PARALLEL PROGRAMMING

Project Seminar “Parallel Programming”, Summer Semester 2011
Introduction
Parallel program design
Patterns for parallel programming
  - A: Algorithm structure
  - B: Supporting structures
INTRODUCTION
Many different models reflecting the various different parallel hardware architectures

2 or rather 3 most common models:
- Shared memory
- Distributed memory
- Hybrid models (combining shared and distributed memory)
PARALLEL PROGRAMMING MODELS

Shared memory

Distributed memory
PROGRAMMING CHALLENGES

Shared memory
- Synchronize memory access
- Locking vs. potential race conditions

Distributed memory
- Communication bandwidth and resulting latency
- Manage message passing
- Synchronous vs. asynchronous communication
2 common standards as examples for the 2 parallel programming models:

- Open Multi-Processing (OpenMP)
- Message passing interface (MPI)
OpenMP

- Collection of libraries and compiler directives for parallel programming on shared memory computers

- Programmers have to explicitly designate blocks that are to run in parallel by adding directives like:

  #pragma omp parallel

- OpenMP then creates a number of threads executing the designated code block
Library with routines to manage message passing for programming on distributed memory computers

Messages are sent from one process to another

Routines for synchronization, broadcasts, blocking and non blocking communication
MPI EXAMPLE

MPI.Scatter

MPI.Gather
PARALLEL PROGRAM DESIGN

General strategies for finding concurrency
General approach: Analyze a problem to identify exploitable concurrency

Main concept is *decomposition*: Divide a computation into smaller parts all or some of which can run concurrently
SOME TERMINOLOGY

- **Tasks**: Programmer-defined units into which the main computation is decomposed
- **Unit of execution (UE)**: Generalization of processes and threads
Decompose a problem into tasks that can run concurrently

- Few large tasks vs. many small tasks
- Minimize dependencies among tasks
Group tasks to simplify managing their dependencies

Tasks within a group run at the same time

Based on decomposition: Group tasks that belong to the same high-level operations

Based on constraints: Group tasks with the same constraints
Order task groups to satisfy constraints among them

Order must be:
- Restrictive enough to satisfy constraints
- Not too restrictive to improve flexibility and hence efficiency

Identify dependencies – e.g.:
- Group A requires data from group B

Important: Also identify the independent groups

Identify potential deadlocks
**DATA DECOMPOSITION**

- Decompose a problem’s data into units that can be operated on relatively independent

- Look at problem’s central data structures

- Decomposition already implied by or basis for task decomposition

- Again: Few large chunks vs. many small chunks
  - Improve flexibility: Configurable granularity
Share decomposed data among tasks

- Identify task-local and shared data
- Classify shared data: read/write or read only?
- Identify potential race conditions
- Note: Sometimes data sharing implies communication
PATTERNS FOR PARALLEL PROGRAMMING
How can the identified concurrency be used to build a program?

3 examples for typical parallel algorithm structures:

- Organize by tasks: Divide & conquer
- Organize by data decomposition: Geometric/domain decomposition
- Organize by data flow: Pipeline
**Principle:** Split a problem recursively into smaller solvable sub problems and merge their results

**Potential concurrency:** Sub problems can be solved simultaneously
**Precondition:** Sub problems can be solved independently

**Efficiency constraint:** Split and merge should be trivial compared to sub problems

**Challenge:** Standard base case can lead to too many too small tasks

- End recursion earlier?
Principle: Organize an algorithm around a linear data structure that was decomposed into concurrently updatable chunks

Potential concurrency: Chunks can be updated simultaneously
Example: Simple blur filter where every pixel is set to the average value of its surrounding pixels

- Image can be split into squares
- Each square is updated by a task
- To update square border information from other squares is required
Again: Granularity of decomposition?

Choose square/cubic chunks to minimize surface and thus nonlocal data

Replicating nonlocal data can reduce communication → “ghost boundaries”

Optimization: Overlap update and exchange of nonlocal data

Number of tasks > number of UEs for better load balance
- Principle based on analogy assembly line: Data flowing through a set of stages
- Potential concurrency: Operations can be performed simultaneously on different data items
Example: Instruction pipeline in CPUs

- Fetch (instruction)
- Decode
- Execute
- ...

PIPELINE
- Precondition: Dependencies among tasks allow an appropriate ordering
- Efficiency constraint: Number of stages $<<$ number of processed items
- Pipeline can also be nonlinear
Intermediate stage between problem oriented algorithm structure patterns and their realized in a programming environment

Structures that “support” the realization of parallel algorithms

4 examples:
- Single program, multiple data (SPMD)
- Task farming/Master & Worker
- Fork & Join
- Shared data
SINGLE PROGRAM, MULTIPLE DATA

- Principle: The same code runs on every UE processing different data

- Most common technique to write parallel programs!
Program stages:

1. Initialize and obtain unique ID for each UE
2. Run the same program on every UE: Differences in the instructions are driven by the ID
3. Distribute data by decomposing or sharing/copying global data

Risk: Complex branching and data decomposition can make the code awful to understand and maintain
Principle: A master task ("farmer") dispatches tasks to many worker UEs and collects ("farms") the results
TASK FARMING/MASTER & WORKER

**master (1)**
- Initiate computation
- Create tasks
- Launch workers
- sleep
- Collect results
- Terminate computation

**workers (1 to N)**
- Initialize
- Compute results
- Done? (Y/N)
- Exit
Precondition: Tasks are relatively independent

Master:
- Initiates computation
- Creates a bag of tasks and stores them e.g. in a shared queue
- Launches the worker tasks and waits
- Collects the results and shuts down the computation

Workers:
- While the bag of tasks is not empty pop a task and solve it

Flexible through indirect scheduling
Optimization: Master can become a worker too
Principle: Tasks create (“fork”) and terminate (“join”) other tasks dynamically.

Example: An algorithm designed after the Divide & Conquer pattern.
Mapping the tasks to UEs can be done directly or indirectly.

**Direct**: Each subtask is mapped to a new UE
- Disadvantage: UE creation and destruction is expensive
- Standard programming model in OpenMP

**Indirect**: Subtasks are stored inside a shared queue and handled by a static number of UEs

Concept behind OpenMP
Problem: Manage access to shared data

Principle: Define an access protocol that assures that the results of a computation are correct for any ordering of the operations on the data
Model shared data as an (abstract) data type with a fixed set of operations.

Operations can be seen as transactions (→ ACID properties).

Start with a simple solution and improve performance step-by-step:
- Only one operation can be executed at any point in time.
- Improve performance by separating operations into noninterfering sets.
- Separate operations in read and write operations.
- Many different lock strategies...
QUESTIONS?
REFERENCES


- Images from Mattson et al. 2004