Parallel Programming

SCPD Master Module

Emil Slusanschi
emil.slusanschi@cs.pub.ro
University Politehnica of Bucharest

Acknowledgement

- The material in this course has been adapted from various (cited) authoritative sources by Lawrence Rauchwerger from the Parasol Lab and myself and is used with his approval
- The presentation has been put together with the help of Dr. Mauro Bianco, Antoniu Pop, Tim Smith and Nathan Thomas from the Parasol Lab at Texas A&M University

Grading@PP

- Activity during lectures – 1 point
  - Presence in class for the lectures is **compulsory** but does not insure the point – you have to (try to) participate **actively**
- Project work – 5 points
  - Similar to APP:
    - 3/coding, 1/documentation, 1/presentation, *1/bonus*
  - Topics from subjects related to the PP
  - Teams of 2-3 people – independent grading
  - Subject can also be done in the "research" hours – at the end a paper/presentation should emerge
- Oral exam – 4 points
  - 5-10 minutes / person
  - 2-3 subjects from the lecture
  - Can be replaced by holding a talk during the semester on a topic agreed with me in advance

Table of Contents (subject to change)

- Introduction to Parallelism
- Introduction to Programming Models
- Shared Memory Programming
- Message Passing Programming
- Shared Memory Models
- PGAS (Parallel Global Address Space) Languages
- Other Programming Models

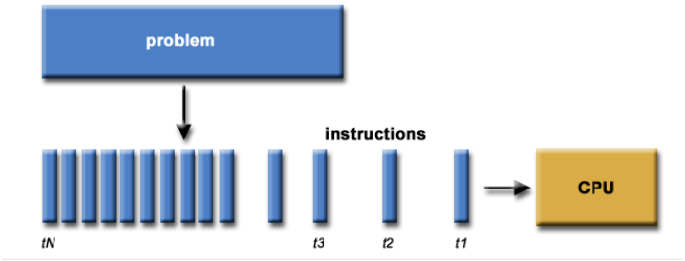
What Will You Get from this Lecture

- (New) Ideas about parallel processing
- Different approaches to parallel programming
- Various programming models used in parallel computing
- Practical experiences with some of the models/technologies presented in this lecture

Table of Contents

Introduction to Parallelism

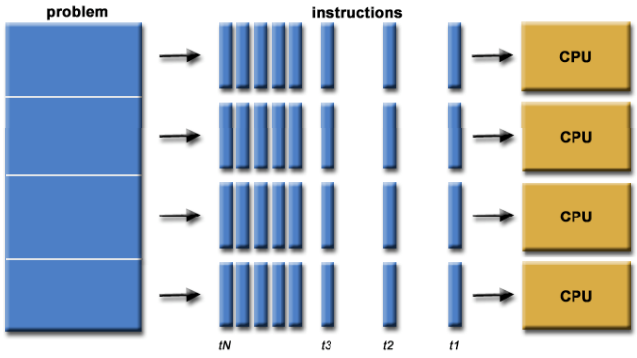
- Sequential Computing
  - Single CPU executes stream of instructions.



Adapted from: http://www.llnl.gov/computing/tutorials/parallel_comp

Introduction to Parallelism

- Parallel computing
  - Partition problem into multiple, concurrent streams of instructions.



Classification

Flynn's Taxonomy (1966-now)		Nowadays
SISD *Single Instruction* *Single Data*	**SIMD** *Single Instruction* *Multiple Data*	**SPMD** *Single Program* *Multiple Data*
MISD *Multiple Instructions* *Single Data*	**MIMD** *Multiple Instructions* *Multiple Data*	**MPMD** *Multiple Program* *Multiple Data*

- Execution models impact the above programming model
- Traditional computer is SISD
- SIMD is *data parallelism* while MISD is pure *task parallelism*
- MIMD is a mixed model (harder to program)
- SPMD and MPMD are less synchronized than SIMD and MIMD
- SPMD is most used model, but MPMD is becoming popular

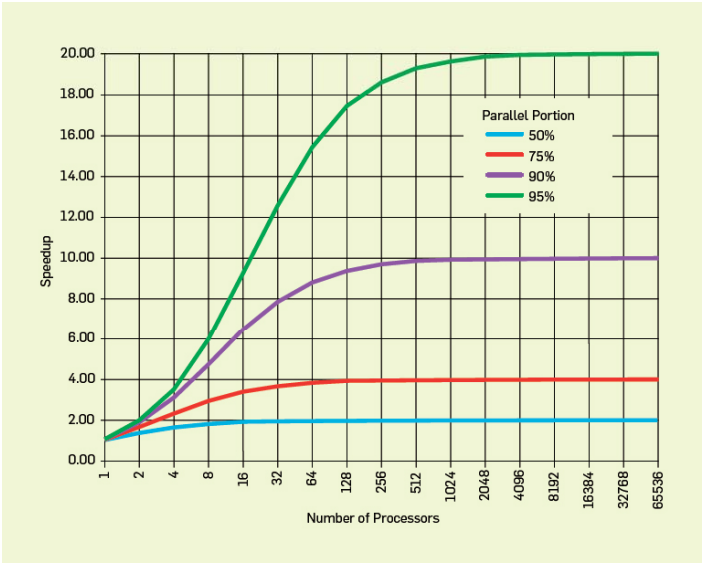
Introduction to Parallelism

- Goal of parallel computing
  - Save time - reduce wall clock time.
    - Speedup - $\frac{\text{wall-clock time of serial execution}}{\text{wall-clock time of parallel execution}}$
  - Solve larger problems - problems that take more memory than available to 1 CPU.

Reduce wall clock time

- Methods
  - Parallelizing serial algorithms (parallel loops)
    - Total number of operations performed changes only slightly
    - Scalability may be poor (Amdahl's law)
  - Develop parallel algorithms
    - Total number of operations may increase, but the running time decreases
- Work Complexity
  - Serialization: parallel algorithm executed sequentially Serializing parallel algorithm may lead to sub-optimal sequential complexity

Amdahl's law revisited



Performance Models

- Abstract Machine Models (PRAM, BSP, and many, many others)
  - Allow asymptotical analysis and runtime estimations
  - Often inaccurate for selecting the right implementation/algorithm on a given architecture
- Programming Primitives Behavior
  - Allow the selection of the right implementation
  - Increases programming effort

Abstract Machine

- PRAM (Parallel RAM, shared memory)
  - Processors access a shared flat memory
  - Performing an operation or accessing a memory location has cost = 1
- BSP (Bulk Synchronous Parallel, distributed memory)
  - Computation proceeds through supersteps
  - Cost of a superstep is $w+hg+l$
  - w is the time for computing on local data
  - h is the size of the largest message sent
  - g and l are architectural parameters describing network bandwidth and latency, respectively

Table of Contents

Parallel Programming Models

Many languages and libraries exist for creating parallel applications.

Each presents a programming model to its users.

During this course, we'll discuss criteria for evaluating a parallel model and use them to explore various approaches.

OpenMP	Charm++	Linda
Pthreads	UPC	MapReduce
Cilk	STAPL	Matlab DCE
TBB	X10	OpenCL
HPF	Fortress	
MPI	Chapel	

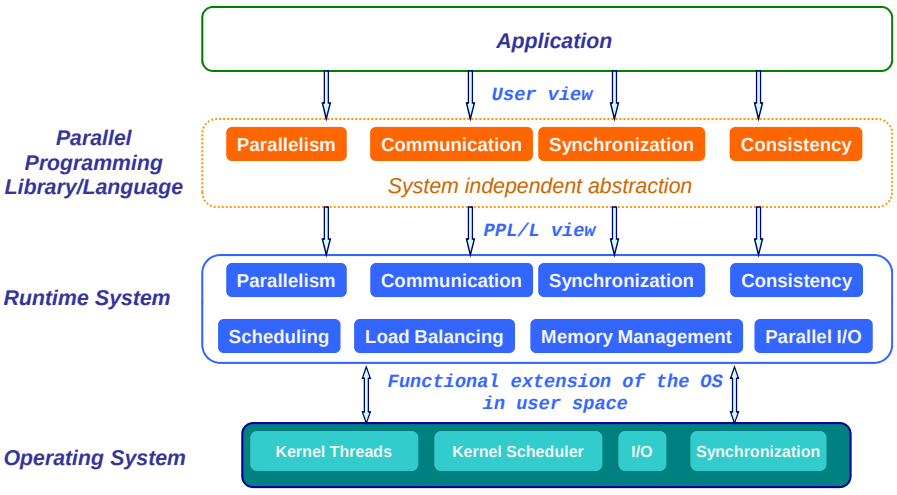
Programming Models Evaluation

What should we consider when evaluating a parallel programming model?

- Parallel Execution Model
- Productivity
- Performance
- Portability

Table of Contents

Parallel Execution Model



Parallel Execution Model

- Parallel Programming Model (user view)
  - Parallelism
  - Communication
  - Synchronization
  - Memory consistency
- Runtime System (RTS)
  - Introduction, definition and objectives
  - Usual services provided by the RTS
  - Portability / Abstraction

Parallel Programming Model (user view)

- Parallelism
- Communication
- Synchronization
- Memory consistency

PPM – Implicit Parallelism

Implicit parallelism (single-threaded view)

- User not required to be aware of the parallelism
  - User writes programs unaware of concurrency
    - Possible re-use previously implemented sequential algorithms
    - Often minor modifications to parallelize
  - User not required to handle synchronization or communication
    - Dramatic reduction in potential bugs
    - Straightforward debugging (with appropriate tools)
- Productivity closer to sequential programming
- Performance may suffer depending on application
- E.g. Matlab DCE, HPF, OpenMP*, Charm++*

* at various levels of implicitness

PPM – Explicit Parallelism

Explicit parallelism (multi-threaded view)

- User required to be aware of parallelism
  - User required to write parallel algorithms
    - Complexity designing parallel algorithms
    - Usually impossible to re-use sequential algorithms (except for embarrassingly parallel ones)
  - User responsible for synchronization and/or communication
    - Major source of bugs and faulty behaviors (e.g. deadlocks)
    - Hard to debug
    - Hard to even reproduce bugs
- Considered low-level
  - Productivity usually secondary
  - Best performance when properly used, but huge development cost
  - E.g. MPI, Pthreads

PPM – Mixed Parallelism

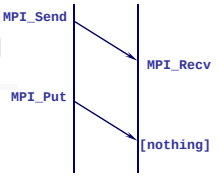
Mixed view

- Basic usage does not require parallelism awareness
- Optimization possible for advanced users

- Benefits from the two perspectives
  - High productivity for the general case
  - High performance possible by fine-tuning specific areas of the code
- E.g. STAPL, Chapel, Fortress

Table of Contents

PPM – Explicit Communication



Explicit Communication

- Message Passing (two-sided communication, P2P)
  - User explicitly sends/receives messages (e.g., MPI)
  - User required to match every Send operation with a Receive
  - Implicitly synchronizes the two threads
    - Often excessive synchronization (reduces concurrency)
    - Non-blocking operations to alleviate the problem (e.g., MPI_Isend/Recv)
- One-sided communication
  - User uses get/put operations to access memory (e.g., MPI-2, GASNet, Cray T3D)
  - No implicit synchronization (i.e., asynchronous communication)

PPM – Explicit Communication

Explicit Communication – Active Message, RPC, RMI

- Based on Message Passing
- Messages activate a handler function or method on the remote side
- Asynchronous
  - No return value (no **`get`** functions)
  - Split-phase programming model (e.g. Charm++, GASNet)
    - Caller provides a callback handler to asynchronously process “return” value
- Synchronous
  - Blocking semantic (caller stalls until acknowledgement/return is received)
  - Possibility to use **`get`** functions
- Mixed (can use both)
  - E.g., ARMI (STAPL)

PPM – Implicit Communication

Implicit Communication

- Communication through shared variables
- Synchronization is primary concern
  - Condition variables, blocking semaphores or monitors
  - Full/Empty bit
- Producer/consumer between threads are expressed with synchronizations
- Increases productivity
  - User does not manage communication
  - Reduced risk of introducing bugs