Parallel and Concurrent Programming
Introduction and Foundation

Marwan Burelle

marwan.burelle@lse.epita.fr
http://wiki-prog.kh405.net
Outline

1 Introduction

2 Foundations and Issues
   Theories and Models
   Tasks Systems
   Program Determinism
   Maximal Parallelism

3 Definitions
Evolutions

- Next evolutions in processor tends more on growing of cores’ number.
- GPU and similar extensions follows the same path and introduce extra parallelism possibilities.
- Network evolutions and widespread of internet fortify clustering techniques and grid computing.
- Concurrency and parallelism are no recent concern, but are emphasized by actual directions of market.
Nature of Parallel Programming

Synchronization

Parallel Programming

Algorithms

Determinism

Definitions

Introduction

Foundations and Issues
Tools and Techniques

- **Threads APIs:**
  - POSIX Threads
  - Win32 Threads
  - JavaThreads
  - ...

- **Parallelism at Language Level:**
  - Concurrent Pascal
  - Ada
  - Oz/Mozart
  - Erlang
  - F#
  - go (google’s new language)
  - ...

- **Higher Levels Library and related:**
  - OpenMP
  - Boost’s Threads
  - Intel’s TBB
  - Cuda/OpenCL/DirectX Compute Shaders
  - ...
Models and theories

- $\pi$–calculus
- Actor Model
- Ambient calculus (and Boxed Ambient)
- CSP (Communicating Sequential Processes)
- CCS (Calculus of Communicating Systems)
- Message Passing
- Futures
- Critical Section
- Locks, Conditions, Semaphores and Monitors
- Petri Nets
- Bisimulation
- Trace Theory
- ...
A Bit of History

- **late 1950’s**: first discussion about parallel computing.
- **1962**: *D825* by *Burroughs Corporation* (four processors.)
- **1967**: Amdahl and Slotnick published a debate about feasibility of parallel computing and introduce *Amdahl’s law* about limit of speed-up due to parallel computing.
- **1969**: *Honeywell’s Multics* introduced first *Symmetric Multiprocessor* (SMP) system capable of running up to eight processors in parallel.
- **1976**: The first *Cray-1* is installed at *Los Alamos National Laboratory*. The major breakthrough of *Cray-1* is its *vector* instructions set capable of performing an operation on each element of a vector in parallel.
- **1983**: *CM-1 Connection Machine* by *Thinking Machine* offers 65536 1-bit processors working on a *SIMD* (Single Instruction, Multiple Data) model.
A Bit of History (2)

- **1991**: *Thinking Machine* introduced CM-5 using a *MIMD* architecture based on a fat tree network of SPARC RISC processors.

- **1990’s**: Modern micro-processors are often capable of being run in an SMP (Symmetric MultiProcessing) model. It began with processors such as *Intel’s 486DX, Sun’s UltraSPARC, DEC’s Alpha IBM’s POWER* . . . Early SMP architectures were based on motherboard providing two or more sockets for processors.

- **2002**: *Intel* introduced the first processor with *Hyper-Threading* technology (running two threads on one physical processor) derived from DEC previous work.

- **2006**: First multi-core processors appear (several processors in one ship.)

- . . .
Global Lecture Overview

1. **Introduction to parallelism**  
   *(this course)*

2. **Synchronization**  
   How to enforce safe data sharing using various synchronization techniques.

3. **Illustrative Examples, Algorithms and Data Structures**  
   How to adapt or write algorithms and data structures in a parallel world (shared queues, tasks scheduling, lock free structures . . .)

4. **Multi-threading Programming with POSIX Threads**  
   threads, mutexes, conditions, read/write locks, barrier, semaphores . . .

5. **TBB and other higher-level tools**  
   Programming using Intel’s TBB, overview of go programming language . . .
Theories and Models
Amdahl’s law

If $P$ is a part of a computation that can be made parallel, then the maximum speed-up (with respect to the sequential version) of running this program on a $N$ processors machine is:

$$\frac{1}{(1 - P) + \frac{P}{N}}$$
Amdahl’s law

- Suppose, half of a program can be made parallel and we run it on a four processors, then we have a maximal speed-up of: $\frac{1}{(1-0.5)+\frac{0.5}{4}} = 1.6$ which means that the program will run 60% faster.

- For the same program, running on a 32 processors will have a speed up of 1.94.

- We can observe that when $N$ tends toward infinity, the speed-up tends to 2! We can not do better than two time faster even with a relatively high number of processors!
## Flynn’s Taxonomy

<table>
<thead>
<tr>
<th>Single Data</th>
<th>Single Instruction</th>
<th>Multiple Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>SISD</td>
<td>SISD</td>
<td>MISD</td>
</tr>
<tr>
<td>SIMD</td>
<td>SIMD</td>
<td>MIMD</td>
</tr>
</tbody>
</table>

- **SISD**: usual non-parallel systems
- **SIMD**: performing the same operations on various data (like vector computing.)
- **MISD**: uncommon model where several operations are performed on the same data, usually implies that all operations must agreed on the result (fault tolerant code such as in space-shuttle controller.)
- **MIMD**: most common actual model.
Tasks Systems
Tasks

- We will describe parallel programs by a notion of task.
- A task $T$ is an instruction in our program. For the sake of clarity, we will limit our study to task of the form:

$$T : \text{VAR} = \text{EXPR}$$

where \( \text{VAR} \) is a memory location (can be seen as a variable) and \( \text{EXPR} \) are usual expressions with variables, constants and basic operators, but no function calls.

- A task \( T \) can be represented by two sets of memory locations (or variables): \( \text{IN}(T) \) the set of memory locations used as input and \( \text{OUT}(T) \) the set of memory locations affected by \( T \).

- \( \text{IN}(T) \) and \( \text{OUT}(T) \) can, by them self, be seen as elementary task (as reading or writing values.) And thus our finest grain description of a program execution will be a sequence of \( \text{IN}() \) and \( \text{OUT}() \) tasks.
Example:

Let $P_1$ be a simple sequential program we present it here using task and memory locations sets:

- **T1**: $x = 1$
  - $\text{IN}(T1) = \emptyset$
  - $\text{OUT}(T1) = \{x\}$

- **T2**: $y = 5$
  - $\text{IN}(T2) = \emptyset$
  - $\text{OUT}(T2) = \{y\}$

- **T3**: $z = x + y$
  - $\text{IN}(T3) = \{x, y\}$
  - $\text{OUT}(T3) = \{z\}$

- **T4**: $w = |x - y|$
  - $\text{IN}(T4) = \{x, y\}$
  - $\text{OUT}(T4) = \{w\}$

- **T5**: $r = (z + w)/2$
  - $\text{IN}(T5) = \{z, w\}$
  - $\text{OUT}(T5) = \{r\}$
Execution and Scheduling

- Given two sequential programs (a list of tasks) a parallel execution is a list of tasks resulting of the composition of the two programs.
- Since, we do not control the scheduler, the only constraint on an execution is the preservation of the order between tasks of the same program.
- Scheduling does not understand our notion of task, it rather works at assembly instructions level, and thus, we can assume that a task $T$ can be interleaved with another task between the realisation of the subtask $\text{IN}(T)$ and the realisation of the subtask $\text{OUT}(T)$.
- As for tasks, the only preserved order is that $\text{IN}(T)$ allways appears before $\text{OUT}(T)$.
- Finally, an execution can be modelized by an ordered sequence of input and output sets of memory locations.
Execution \textit{(example)}

\begin{itemize}
\item \textbf{Example:}
\end{itemize}

Given the two programs $P_1$ and $P_2$:

\begin{center}
\begin{tabular}{ll}
$T_{11}$ : $x = 1$ & $T_{21}$ : $y = 1$\\
$T_{12}$ : $y = x + 1$ & $T_{22}$ : $x = y - 1$
\end{tabular}
\end{center}

The following sequences are valid parallel execution of $P_1//P_2$:

\begin{center}
\begin{tabular}{l}
$E_1$ = \texttt{IN(T11); OUT(T11); IN(T12); OUT(T12); IN(T21); OUT(T21); IN(T22); OUT(T22)}\\
$E_2$ = \texttt{IN(T21); OUT(T21); IN(T22); OUT(T22); IN(T11); OUT(T11); IN(T12); OUT(T12)}\\
$E_3$ = \texttt{IN(T11); IN(T21); OUT(T11); OUT(T21); IN(T12); IN(T22); OUT(T22); OUT(T12)}
\end{tabular}
\end{center}

At the end of each executions we can observe each value in both memory locations $x$ and $y$:

\begin{center}
\begin{tabular}{ll}
$E_1$ & $x = 0$ and $y = 1$\\
$E_2$ & $x = 1$ and $y = 2$\\
$E_3$ & $x = 0$ and $y = 2$
\end{tabular}
\end{center}
The issue!

- In the previous example, it is obvious that two different executions of the same parallel program may give different results.
- In a linear programming, given fixed inputs, programs’ executions always give the same result.
- Normally, programs and algorithms are supposed to be deterministic, using parallelism it is obviously not always the case!
Program Determinism
Tasks’ Dependencies

- In order to completely describe parallel programs and parallel executions of programs, we introduce a notion of dependencies between tasks.
- Let $E$ be a set of tasks and $\langle \rangle$ a well founded dependency order on $E$.
- A pair of tasks $T_1$ and $T_2$ verify $T_1 < T_2$ if the sub-task $\text{OUT}(T_1)$ must occurs before the sub-task $\text{IN}(T_2)$.
- A Task System $(E, \langle \rangle)$ is the definition of a set, $E$, of tasks and a dependency order $\langle \rangle$ on $E$. It describes a combination of several sequential programs into a parallel program (or a fully sequential program if $\langle \rangle$ is total.) Tasks of a same sequential program have a natural ordering, but we can also define ordering between tasks of different programs, or between programs.
Task Language

Let \( E = \{T_1, \ldots, T_n\} \) be a set of task, \( A = \{\text{IN}(T_1), \ldots, \text{OUT}(T_n)\} \) a vocabulary based on sub-task of \( E \) and \((<)\) an ordering relation on \( E \).

The language associated with a task system \( S = (E, <) \), noted \( L(S) \), is the set of words \( \omega \) on the vocabulary \( A \) such that for every \( T_i \) in \( E \) there is exactly one occurrence of \( \text{IN}(T_i) \) and one occurrence of \( \text{OUT}(T_i) \) and the former appearing before the latter. If \( T_i < T_j \) then \( \text{OUT}(T_i) \) must appear before \( \text{IN}(T_j) \).

- We can define the product of system \( S_1 \) and \( S_2 \) by \( S_1 \times S_2 \) such that \( L(S_1 \times S_2) = L(S_1).L(S_2) \) (\( \cdot \cdot \cdot \) is the concatenation of language.)
- We can also define parallel combination of task system: \( S_1//S_2 = (E_1 \cup E_2, <_1 \cup <_2) \) (where \( E_1 \cap E_2 = \emptyset \).)
Tasks’ Dependencies and Graph

- The relation ($<$) can sometimes be represented using directed graph (and thus graph visualization methods.)
- In order to avoid overall complexity, we use the smallest relation ($<_{\text{min}}$) with the same transitive closure as ($<$) rather than ($<$) directly.
- Such a graph is of course directed and without cycle. Vertexes are task and an edge between from $T_1$ to $T_2$ implies that $T_1 < T_2$.
- This graph is often call Precedence Graph.
If we define \( S_1 = \{T_1\} \), \( S_2 = \{T_2, T_3, T_4\} \), \( S_3 = \{T_5, T_6, T_7\} \) and \( S_4 = \{T_8\} \). Then the resulting system (described by the graph above) is:

\[
S = S_1 \times (S_2 \slash S_3) \times S_4
\]
Notion of Determinism

Deterministic System

A deterministic task system $S = (E, <)$ is such that for every pair of words $\omega$ and $\omega'$ of $L(S)$ and for every memory locations $X$, sequences values affected to $X$ are the same for $\omega$ and $\omega'$.

A deterministic system, is a tasks system where every possible executions are not distinguishable by only observing the evolution of values in memory locations (observational equivalence, a kind of bisimulation.)
Determinism

• The previous definition may seem too restrictive to be useful.

• In fact, one can exclude local memory locations (i.e. memory locations not shared with other programs) of the observational property.

• In short, the deterministic behavior can be limited to a restricted set of meaningful memory locations, excluding temporary locations used for inner computations.

• The real issue here is the provability of the deterministic behavior: one cannot possibly test every execution path of a given system.

• We need a finite property independent of the scheduling (i.e. a property relying only on the system.)
Non-Interference

- Non-Interference (NI) is a general property used in many context (especially language level security.)
- Two tasks are non-interfering, if and only if the values taken by memory locations does not depend on the order of execution of the two tasks.

Non Interference

Let $S = (E, <)$ be a tasks system, $T_1$ and $T_2$ be two task of $E$, then $T_1$ and $T_2$ are non-interfering if and only if, they verify one of the two following properties:

- $T_1 < T_2$ or $T_2 < T_1$ (the system force a particular order.)
- $\text{IN}(T_1) \cap \text{OUT}(T_2) = \text{IN}(T_2) \cap \text{OUT}(T_1) = \emptyset$
- $\text{OUT}(T_1) \cap \text{OUT}(T_2) = \emptyset$
\textbf{NI} and determinism

- The \textit{NI} definitions is a based on the contraposition of the Bernstein’s conditions (defining when two tasks are dependent.)

- Obviously, two non-interfering tasks do not introduce non-deterministic behavior in a system (they are already ordered or the order of their execution is not relevant.)

\textbf{Theorem}

\textit{Let }$S = (E, <)\text{ be a tasks system, } S \text{ is a deterministic system if every pair of tasks in } E \text{ are non-interfering.}$
Equivalent Systems

- We now extend our use of observational equivalence to compare systems.
- The idea is that we cannot distinguish two systems that have the same behavior (affect the same sequence of values in a particular set of memory locations.)

Equivalent Systems

Let $S_1 = (E_1, <_1)$ and $S_2 = (E_2, <_2)$ be two tasks systems. $S_1$ and $S_2$ are equivalent if and only if:

- $E_1 = E_2$
- $S_1$ and $S_2$ are deterministic
- For every words $\omega_1 \in L(S_1)$ and $\omega_2 \in L(S_2)$, for every (meaningful) memory location $X$, $\omega_1$ and $\omega_2$ affect the same sequence of values to $X$. 
Maximal Parallelism
Now that we can define and verify determinism of tasks systems, we need to be able to assure a kind of maximal parallelism.

Maximal parallelism describes the minimal sequentiality and ordering needed to stay deterministic.

A system with maximal parallelism can’t be more parallel without introduction of non-deterministic behavior (and thus inconsistency.)

Being able to build (or transform systems into) maximally parallel systems, guarantees usage of a parallel-friendly computer at its maximum capacity for our given solution.
Maximal Parallelism

A tasks system with maximal parallelism, is a tasks where one can not remove dependency between two tasks $T_1$ and $T_2$ without introducing interference between $T_1$ and $T_2$.

Theorem

For every deterministic system $S = (E, <)$ there exists an equivalent system with maximal parallelism $S_{max} = (E, <_{max})$ with $(<_{max})$ defined as:

$T_1 <_{max} T_2$ if

$$
\begin{cases}
T_1 < < T_2 \\
\land \text{OUT}(T_1) \neq \emptyset \land \text{OUT}(T_2) \neq \emptyset \\
\land \left( \text{IN}(T_1) \cap \text{OUT}(T_2) \neq \emptyset \right) \\
\land \left( \bigvee \text{IN}(T_2) \cap \text{OUT}(T_1) \neq \emptyset \right) \\
\land \left( \bigvee \text{OUT}(T_1) \cap \text{OUT}(T_2) \neq \emptyset \right)
\end{cases}
$$
Usage of Maximal Parallelism

• Given a graph representing a system, one can reason about parallelism and performances.

• Given an (hypothetical) unbound material parallelism, the complexity of a parallel system is the length of the longest path in the graph from initial tasks (tasks with no predecessors) to final tasks (tasks with no successor.)

• Classical analysis of dependency graph can be used to spot critical tasks (tasks that can’t be late without slowing the whole process) or find good planned executions for non-parallel hardware.

• Tasks systems and maximal parallelism can be used to prove modelization of parallel implementations of sequential programs.

• Maximal parallelism can be also used to effectively measure real gain of a parallel implementation.
QUESTIONS ?
Tasks’ Dependencies

Dependency Ordering Relation

A dependency ordering relation is a partial order which verifies:

- anti-symmetry ($T_1 < T_2$ and $T_2 < T_1$ can not be both true)
- anti-reflexive (we can’t have $T < T$)
- transitive (if $T_1 < T_2$ and $T_2 < T_3$ then $T_1 < T_3$).
Task Language

Let $E = \{T_1, \ldots, T_n\}$ be a set of tasks, $A = \{\text{IN}(T_1), \ldots, \text{OUT}(T_n)\}$ a vocabulary based on sub-tasks of $E$ and $(<)$ an ordering relation on $E$. The language associated with a task system $S = (E, <)$, noted $L(S)$, is the set of words $\omega$ on the vocabulary $A$ such that for every $T_i$ in $E$ there is exactly one occurrence of $\text{IN}(T_i)$ and one occurrence of $\text{OUT}(T_i)$ and the former appearing before the latter. If $T_i < T_j$ then $\text{OUT}(T_i)$ must appear before $\text{IN}(T_j)$. 
Transitive Closure

The transitive closure of a relation (<) is the relation <\(\mathcal{C}\) defined by:

\[
x <\mathcal{C} y \text{ if and only if } \begin{cases} x < y \\ \exists z \text{ such that } x <\mathcal{C} z \text{ and } z <\mathcal{C} y \end{cases}
\]

This relation is the biggest relation that can be obtained from (<) by only adding sub-relation by transitivity.
Notion of Determinism

Deterministic System

A deterministic task system \( S = (E, <) \) is such that for every pair of words \( \omega \) and \( \omega' \) of \( L(S) \) and for every memory locations \( X \), sequences values affected to \( X \) are the same for \( \omega \) and \( \omega' \).

A deterministic system, is a tasks system where every possible executions are not distinguishable by only observing the evolution of values in memory locations (observational equivalence, a kind of bisimulation.)
Non-Interference

Let $S = (E, <)$ be a tasks system, $T_1$ and $T_2$ be two tasks of $E$, then $T_1$ and $T_2$ are non-interfering if and only if, they verify one of the two following properties:

- $T_1 < T_2$ or $T_2 < T_1$ (the system force a particular order.)
- $\text{IN}(T_1) \cap \text{OUT}(T_2) = \text{IN}(T_2) \cap \text{OUT}(T_1) = \emptyset$
- $\text{OUT}(T_1) \cap \text{OUT}(T_2) = \emptyset$
Equivalent Systems

Let $S_1 = (E_1, <_1)$ and $S_2 = (E_2, <_2)$ be two tasks systems. $S_1$ and $S_2$ are equivalent if and only if:

- $E_1 = E_2$
- $S_1$ and $S_2$ are deterministic
- For every words $\omega_1 \in L(S_1)$ and $\omega_2 \in L(S_2)$, for every (meaningful) memory location $X$, $\omega_1$ and $\omega_2$ affect the same sequence of values to $X$. 
Maximal Parallelism

A tasks system with maximal parallelism, is a tasks where one can not remove dependency between two tasks $T_1$ and $T_2$ without introducing interference between $T_1$ and $T_2$.

Theorem

For every deterministic system $S = (E, <)$ there exists an equivalent system with maximal parallelism $S_{max} = (E, <_{max})$ with ($<_{max}$) defined as:

$$T_1 <_{max} T_2 \text{ if } \begin{cases} T_1 \not< T_2 \\ \land \text{OUT}(T_1) \neq \emptyset \land \text{OUT}(T_2) \neq \emptyset \\ \land \left( \lor \text{IN}(T_1) \cap \text{OUT}(T_2) \neq \emptyset \\ \lor \text{IN}(T_2) \cap \text{OUT}(T_1) \neq \emptyset \\ \lor \text{OUT}(T_1) \cap \text{OUT}(T_2) \neq \emptyset \right) \end{cases}$$