Parallel and Concurrent Programming
Introduction and Foundation

Marwan Burelle

marwan.burelle@lse.epita.fr
Outline

1. Introduction
   Parallelism in Computer Science
   Nature of Parallelism
   Global Lecture Overview

2. Being Parallel
   Gain?
   Models of Hardware Parallelism
   Decomposition

3. Foundations
   Tasks Systems
   Program Determinism
   Maximal Parallelism
Outline

4 Interacting with CPU Cache
   False Sharing
   Memory Fence

5 Mutual Exclusion
   Classic Problem: Shared Counter
   Critical Section and Mutual Exclusion
   Solutions with no locks

6 Definitions
Evolutions

- Next evolutions in processor tends more on 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

Algorithms

Parallel Programming

Determinism

Synchronization

Introduction

Parallelism in Computer Science
Nature of Parallelism
Global Lecture Overview
Being Parallel
Foundations
Interacting with CPU Cache
Mutual Exclusion
Definitions
Parallelism in Computer Science
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 was 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 appears (several processors in one ship.)
How Can I Code In Parallel?

Processus Based  No special support needed for parallelization but require complex inter-process communication. Not really useful.

System Level Threads  Using threads provides by system’s libraries (like Unix pthreads). Lake of higher level features and portabilities issues.

Portable Threads Libs  Solves portability issues of system threads.

Higher Level Libs  Can really increase efficiency and provide clever approach. May still need lower level interractions.

Language Level  Seriously ? Still very experimental.
Nature of Parallelism
Hardware Parallelism

- **Symetric MultiProcessor**: generic designation of identical processors running in parallel under the same OS supervision.

- **Multi-Core**: embedding multiple processor unit on the same die: decrease cost, permit sharing of common elements, permit shared on die cache.

- **Hyper-Threading**: using the same unit (core or processor) to run two (or more) logical unit: each logical unit *run slower* but parallelism can offer gain and opportunities of more efficient interleaving (using independent elements of the hardware.)
- **Vectorized Instructions**: usual extensions providing parallel execution of a single operation on all elements of a vector.

- **Non-Uniform Memory Access (NUMA)**: architecture where each processor has its own memory with dedicated access (to limit false sharing for example.)
In order to take advantage of hardware parallelism, the *operating system* **must** support SMP.

*Operating Systems* can offer parallelism to application even when hardware is not, by using *multi-programming*.

The system have two ways to provide parallelism: threads and processus.

When only processus are available, no memory conflicts can arise, but processus have to use various communication protocole to exchange data and synchronized themselves.

When memory is shared, programs have to manage memory accesses to avoid conflict, but data exchange between threads is far simpler.
Programming Parallel Applications

- Operating systems supporting parallelism and threads, normally provide API for parallel programming.
- Support for parallelism can be either primitive in the programming language or added by means of API and libraries.
- The most common paradigm is the explicit use of threads backend by the Kernel.
- Some languages try to offer implicit parallelism, or at least parallel blocks, but producing smart parallel code is tedious.
- Some hardware parallelism features (vector instructions or multi-operations) may need special support from the language.
- Modern frameworks for parallelism find a convenient way by abstracting system threads management and offering more simple threads manipulation (OpenMP, Intel’s TBB . . . )
Introduction to parallelism
(this course)

Synchronization and Threads
How to enforce safe data sharing using various synchronization techniques, illustrated using Threads from C11/C++11.

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

TBB and other higher-level tools
Programming using Intel’s TBB . . .
Being Parallel

Introduction

Being Parallel

Gain?

Models of Hardware

Parallelism

Decomposition

Foundations

Interacting with

CPU Cache

Mutual Exclusion

Definitions
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 !

- Amdahl’s law show us that we should first consider the amount of parallelism rather than the number of processors.
Gustafson’s law

Let $P$ be the number of processors and $\alpha$ the sequential fraction of the parallel execution time, then we define the scaled-speedup, noted $S(P)$, as:

$$S(P) = P + \alpha \times (P - 1)$$

- Amdahl’s law consider a fixed amount of work and focus on minimal time execution.
- Gustafson’s law consider a fixed execution time and describe the increased size of problem.
- The main consequences of Gustafson’s law is that, we can always increase size of the solved problem (for a fixed amount of time) by increasing the number of processor.
Some Results

Amdahl’s law speedup depending on parallelism part and number of processors.

<table>
<thead>
<tr>
<th></th>
<th>30%</th>
<th>50%</th>
<th>60%</th>
<th>75%</th>
<th>90%</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 CPU</td>
<td>5.3%</td>
<td>33.3%</td>
<td>53.8%</td>
<td>100.0%</td>
<td>185.7%</td>
</tr>
<tr>
<td>8 CPU</td>
<td>31.1%</td>
<td>77.8%</td>
<td>116.2%</td>
<td>220.0%</td>
<td>515.4%</td>
</tr>
<tr>
<td>16 CPU</td>
<td>36.8%</td>
<td>88.2%</td>
<td>131.9%</td>
<td>255.6%</td>
<td>661.9%</td>
</tr>
<tr>
<td>32 CPU</td>
<td>39.7%</td>
<td>93.9%</td>
<td>140.6%</td>
<td>276.5%</td>
<td>764.9%</td>
</tr>
</tbody>
</table>
Two Faces Of A Same Law

- It has been proved that both laws express the same result but in different form.
- Variation in results are often due to a misleading definition of Amdahl’s $P$ representing the parallelizable part of a sequential program and Gustafson’s $\alpha$ representing the amount of time spent in the linear part during a parallel execution.
Models of Hardware Parallelism
### Flynn’s Taxonomy

<table>
<thead>
<tr>
<th>Single Data</th>
<th>Single Instruction</th>
<th>Multiple Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Data</td>
<td>SISD</td>
<td>MISD</td>
</tr>
<tr>
<td>Multiple Data</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.
• Actual processors provide MIMD parallelism in the form of Symmetric Multi-Processor (SMP)
• Modern processors also provides vector based instruction set extension that provides SIMD parallelism (like MMX or SSE instructions for x86)
• GPGPU are more or less used in a SIMD like fashion which is much more accurate for a very large number of core.
Decomposition
• To move from a linear program to a parallel program, we have to break our activities in order to think in parallel
• When decomposing you have to take into account various aspects:
  • Activities independance (how much synchronization we need)
  • Load Balancing (using all available threads)
  • Decomposition overhead

Choosing the right decomposition is the most critical choice you have to make when designing a parallel program
Decomposition Strategies

**Task Driven**
the problem is split in (almost) independent tasks ran in parallel;

**Data Driven**
all running tasks perform the same operations on a partition of the original data set;

**Data Flow Driven**
the whole activities is decomposed in a chain of dependent tasks, a pipeline, where each task depends on the output of the previous one.
Task Driven Decomposition

- In order to perform task driven decomposition, you need to:
  - list activities in your program,
  - establish groups of dependent activities that will form standalone tasks,
  - identify interaction and possible data conflict between tasks
- Task driven decomposition is technically simple to implement and can be particularly efficient when activities are well segmented
- On the other hand, this strategy is highly constraint by the nature of your activities, their relationship and dependencies.
- Load balancing (maintaining thread activities at its maximum) is often hard to achieve due to the fixed nature of the decomposition.
Data Driven Decomposition

• In order to perform data driven decomposition, you need to:
  • Defines the common task performed on each subset of data
  • Find a coherent data division strategy that respect load balancing, data conflict and other memory issue (such as cache false sharing,)
  • You often have to prepare a recollection phase to compute final result (probably a fully linear computation,)

• Data driven decomposition scales pretty well, as data set size grows partitionning becomes more efficient against sequential or task based approach;

• Care must be taken when choosing data partitionning in order to obtain maximal performances

• Too much partitionning or too small data set will probably induce higher overhead.
• In order to perform data flow driven decomposition, you need to:
  • Split activities along the flow of execution in order to identify tasks
  • Model data exchange between tasks
  • Choose a data partitionning (the flow’s grain) strategy

• With respect to Ford’s concept of production line, execution time for a chunk of data correspond to the execution time of the longest task, the global time is thus this execution time multiply by the number of chunks plus two times the cost of the whole line;

• Needs careful design and conception, data channels and efficient data partitionning, probably the more complex (but the more realistic) approach
• Data driven approach yield pretty good result when performing single set of operations on huge set of data;
• Task driven approach are more suited for concurrency issues (performing various activities in parallel to mask waiting time.)
• Data flow driven decomposition can be used, together with another decomposition, as a skeleton for the global flow.
• Data flow driven decomposition requires complex infrastructure but provides a more adaptable solution for a complete chain of activities;
• Of course, in realistic situation, we’ll often choose a mid-term decomposition mixing various approaches.
There are various design patterns (we’ll see some later) dedicated to parallel programming, here are some examples:

- **Divide and Conquer**: data decomposition (divide) and recollection (conquer)
- **Pipeline**: data flow decomposition
- **Wave Front**: parallel topological traversal of a graph of tasks
- **Geometric Decomposition**: divide data-set in rectangles
Tasks Systems
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 tasks 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:

\[
\begin{align*}
T1 & : x = 1 & & T1 : \text{IN}(T1)=\emptyset \\
& & & \text{OUT}(T1) = \{x\} \\
T2 & : y = 5 & & T2 : \text{IN}(T2)=\emptyset \\
& & & \text{OUT}(T2) = \{y\} \\
T3 & : z = x + y & & T3 : \text{IN}(T3)=\{x, y\} \\
& & & \text{OUT}(T3) = \{z\} \\
T4 & : w = |x - y| & & T4 : \text{IN}(T4)=\{x, y\} \\
& & & \text{OUT}(T4) = \{w\} \\
T5 & : r = (z + w)/2 & & T5 : \text{IN}(T5)=\{z, w\} \\
& & & \text{OUT}(T5) = \{r\}
\end{align*}
\]
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 $\text{IN}(T)$ always appears before $\text{OUT}(T)$.
- Finally, an execution can be modelized by an ordered sequence of input and output sets of memory locations.
Execution *(example)*

**Example:**

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

- \( T_{11} : x = 1 \)
- \( T_{12} : y = x + 1 \)
- \( T_{21} : y = 1 \)
- \( T_{22} : x = y - 1 \)

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

- \( E_1 = \text{IN}(T_{11}); \text{OUT}(T_{11}); \text{IN}(T_{12}); \text{OUT}(T_{12}); \text{IN}(T_{21}); \text{OUT}(T_{21}); \text{IN}(T_{22}); \text{OUT}(T_{22}) \)
- \( E_2 = \text{IN}(T_{21}); \text{OUT}(T_{21}); \text{IN}(T_{22}); \text{OUT}(T_{22}); \text{IN}(T_{11}); \text{OUT}(T_{11}); \text{IN}(T_{12}); \text{OUT}(T_{12}) \)
- \( E_3 = \text{IN}(T_{11}); \text{IN}(T_{21}); \text{OUT}(T_{11}); \text{OUT}(T_{21}); \text{IN}(T_{12}); \text{IN}(T_{22}); \text{OUT}(T_{22}); \text{OUT}(T_{12}) \)

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

- \( E_1 \) $x = 0$ and $y = 1$
- \( E_2 \) $x = 1$ and $y = 2$
- \( E_3 \) $x = 0$ and $y = 2$
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!
Introduction

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 $<\rangle$ 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 $(E, <\rangle$ is the definition of a set, $E$, of tasks and a dependency order $<\rangle$ on $E$. It describes a combination of several sequential programs into a parallel program (or a fully sequential program if $<\rangle$ 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 $L(S_1 \times S_2) = L(S_1) . L(S_2)$ (\_\_\_ 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 $E_1 \cap E_2 = \emptyset$.)
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 $S_1 = \{T_1\}$, $S_2 = \{T_2 \ T3 \ T4\}$, $S_3 = \{T5 \ T6 \ T7\}$ and $S_4 = \{T8\}$. Then the resulting system (described by the graph above) is:

$$S = S_1 \times (S_2/\!\!/S_3) \times S_4$$
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.)
- $\text{IN}(T_1) \cap \text{OUT}(T_2) = \text{IN}(T_2) \cap \text{OUT}(T_1) = \text{OUT}(T_1) \cap \text{OUT}(T_2) = \emptyset$
The $NI$ definitions is 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.)

**Theorem**

Let $S = (E, <)$ be a tasks system, $S$ is a deterministic system if every pair of tasks in $E$ are non-interfering.
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 $X$. 

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.)
Maximal Parallelism
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 $S = (E, \prec)$ there exists an equivalent system with maximal parallelism $S_{max} = (E, \prec_{max})$ with $(\prec_{max})$ defined as:

$$
T_1 \prec_{max} T_2 \text{ if } \begin{cases} 
T_1 \prec T_2 \\
\land \ \text{OUT}(T_1) \neq \emptyset \land \text{OUT}(T_2) \neq \emptyset \\
\land \left( \land \ \text{IN}(T_1) \cap \text{OUT}(T_2) \neq \emptyset \\
\lor \ \text{IN}(T_2) \cap \text{OUT}(T_1) \neq \emptyset \\
\lor \ \text{OUT}(T_1) \cap \text{OUT}(T_2) \neq \emptyset \end{cases}
$$
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 use 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.
Interracting with CPU Cache
• Modern CPUs rely on memory cache to prevent memory access bottleneck;
• In SMP architecture, cache are mandatory: there’s only one memory bus! (NUMA architectures try to solve this)
• Access to shared data induce memory locking and cache updates (thus waiting time for your core.)
• Even when data are not explicitly shared, cache management can become your worst enemy!
This part is Intel/x86 oriented, technical details may not correspond to other processors.

- Each cache line have a special state: **Invalid(I)**, **Shared(S)**, **Exclusive(E)**, **Modified(M)**
- Cache line in state S are shared among core and only used for reading.
- Cache line in state E or M are only owned by one core.
- When a core tries to write to some memory location in a cache line, it will forced any other core to *loose* the cache line (putting it to state I for example.)
- Cache mechanism worked as a read/write lock that track modifications and consistency between values in cache line and value in memory.
- The pipeline (and somehow the compiler) try to *anticipate* write needs so cache line are directly acquired in E state (*fetch for write*)
False Sharing
False Sharing

CPU0

CPU1

Array

Memory

Cache Update

Invalidate Cache Line

Write

Memory Update
Be nice with cache

There’s no standard ways to control cache interaction, even using ASM code. Issues described previously may or may not happen in various contexts. Rather than providing a seminal solutions, we must rely on guidelines to prevent cache false sharing.

- Avoid as much as possible shared data;
- Prefer *threads’ local storages* (local variable, localy allocated buffers . . .);
- When returning set of values, allocate a container per working thread;
- Copy shared data before using it;
- When possible use a thread oriented allocator (modern allocator will work with separated pools of memory per unit rather than one big pool);
Be nice with cache

--- Bad Style ---

```c
int main()
{
    int             res[2];
    pthread_t      t[2];
    pthread_create(t, NULL, run, res);
    pthread_create(t+1, NULL, run, res+1);
    pthread_join(t[0], NULL);
    pthread_join(t[1], NULL);
    // use the result
    return 0;
}
```

--- Better Style ---

```c
int main()
{
    int         *r0, *r1;
    pthread_t    t[2];
    r0 = malloc(sizeof (int));
    r1 = malloc(sizeof (int));
    pthread_create(t, NULL, run, r0);
    pthread_create(t+1, NULL, run, r1);
    pthread_join(t[0], NULL);
    pthread_join(t[1], NULL);
    // use the result
    return 0;
}
```
**Parallel and Concurrent Programming**

**Introduction and Foundation**

Marwan Burelle

**Introduction**

Being Parallel

Foundations

Interacting with CPU Cache

False Sharing

Memory Fence

Mutual Exclusion

Definitions

---

**Even Better**

```c
int main()
{
    int *res[2];
    pthread_t t[2];

    // Provide no containers
    pthread_create(t, NULL, run, NULL);
    pthread_create(t+1, NULL, run, NULL);

    // let threads allocate the result
    // and collect it with join.
    pthread_join(t[0], res);
    pthread_join(t[1], res+1);
}
```
Some stats ...
### Some stats ...

<table>
<thead>
<tr>
<th>Input ((n))</th>
<th>Bad Practice</th>
<th>Good Practice</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0.0446</td>
<td>0.0028</td>
<td>0.0418</td>
</tr>
<tr>
<td>2048</td>
<td>0.1878</td>
<td>0.0107</td>
<td>0.177078</td>
</tr>
<tr>
<td>4096</td>
<td>0.7140</td>
<td>0.0423</td>
<td>0.671683</td>
</tr>
<tr>
<td>8192</td>
<td>2.7296</td>
<td>0.1688</td>
<td>2.560812</td>
</tr>
<tr>
<td>16384</td>
<td>7.4191</td>
<td>0.6817</td>
<td>6.73742</td>
</tr>
<tr>
<td>65536</td>
<td>122.0350</td>
<td>10.7911</td>
<td>111.2439</td>
</tr>
<tr>
<td>131072</td>
<td>682.3750</td>
<td>43.1707</td>
<td>639.2043</td>
</tr>
</tbody>
</table>

**The bad code take more then 90% longer, due to false sharing.**

Both code do heavy computation based on \(n\), bad version share an array between threads for reading and writing, while the other copy the input value and allocate its own container. Time results are in seconds, measured using `tbb::tick_count`. The main program runs 4 threads on 2 dual core processor (no hyperthreading) under FreeBSD.
Discussion

- False sharing can induce an important penalty (as shown with previous example).
- In the presence of more than 2 CPU (on the same dice or not) multiple level of cache can amplified the penalty (this is our case.).
- Most dual-core shares L2 cache but on quad core only pairs of cores share it.
- With multi-processors (as in the experience), there’s probably no shared cache at all: no hope to avoid a full memory access!
- The same issue can even be worse when sharing arrays of locks!
Memory Fence
Modern processors are able to somehow modify execution order.

On multi-processor platforms, this means that apparent ordering may not be respected at execution level (in fact, your compiler is doing the same).

Another mistake is to assume that conditionally executed code cannot happen before the condition is tested. However, the compiler or hardware may speculatively hoist the conditional code above the condition.

Similarly, it is a mistake to assume that a processor cannot read the target of a pointer before reading the pointer. A modern processor does not read individual values from main memory. It reads cache lines . . .
bool Ready;
std::string Message;

// Thread 1 action
void Send( const std::string& src ) {
    Message = src; // C++ hidden memcpy
    Ready = true;
}

// Thread 2 action
bool Receive( std::string& dst ) {
    bool result = Ready;
    if( result ) dst = Message; // C++ hidden memcpy
    return result;
}
• In the previous example, there is no guarantee that the string is effectively copied when the second thread sees the flag becoming true.

• Marking the flag volatile won’t change any things (it doesn’t induce memory fence.)

• Compilers and processor try to optimized memory operations and often use prefetch or similar activities.

• Prefer locks, atomic types or condition variables.
Mutual Exclusion
Classic Problem: Shared Counter
Sharing a counter

• Sharing a counter between two threads (or processus) is a good seminal example to understand the complexity of synchronisation.

• The problem is quite simple: we have two threads monitoring external events, when an event occurs they increase a global counter.

• Increasing a counter $X$ is a simple task of the form:

$$T1 : X = X + 1$$

With associated sets:

$$\text{IN}(T1) = \{X\}$$
$$\text{OUT}(T1) = \{X\}$$

• The two threads execute the same task $T1$ (along with their monitoring activity.) And thus, they are interfering.
Concrete Results

A shared counter without locking.

<table>
<thead>
<tr>
<th>Threads</th>
<th>Average Errors</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3878947.2</td>
<td>46.24%</td>
</tr>
<tr>
<td>4</td>
<td>11575434.6</td>
<td>68.99%</td>
</tr>
<tr>
<td>8</td>
<td>28256366</td>
<td>84.21%</td>
</tr>
<tr>
<td>16</td>
<td>57525557.4</td>
<td>85.72%</td>
</tr>
<tr>
<td>32</td>
<td>119176589.8</td>
<td>88.79%</td>
</tr>
<tr>
<td>64</td>
<td>244048804</td>
<td>90.92%</td>
</tr>
<tr>
<td>128</td>
<td>478563124.4</td>
<td>89.14%</td>
</tr>
<tr>
<td>256</td>
<td>965023132.4</td>
<td>89.87%</td>
</tr>
</tbody>
</table>

Programs run on a 4-cores (8 hyper-threads) CPU. Each threads (system threads using C++11 API) wait for everyone and then perform a $2^{22}$ steps loop where it adds one to the counter at each step.
Pseudo-Code for Counter Sharing

Example:

```plaintext
global int X=0;

guardian(int id):
    for (;;)
        wait_event(id); // wait for an event
        X = X + 1; // T1

main:
    {// // parallel execution
        guardian(0);
        guardian(1);
    }//
```
Critical Section and Mutual Exclusion

Critical Section and Mutual Exclusion
In the previous example, while we can easily see the interference issue, no task ordering can solve it.

- We need an other refinement to enforce consistency of our program.
- The critical task (T1) is called a **critical section**.

**Critical Section**

A section of code is said to be a critical section if execution of this section cannot be interrupted by other process manipulating the same shared data without loss of consistency or determinism.

*The overlapping portion of each process, where the shared variables are being accessed.*
Critical Section

Example:

```c
guardian(int id):
    // Restant Section
    for (;;)
        wait_event(id);
    // Entering Section (empty here)
    X = X + 1; // Critical Section
    // Leaving Section (empty here)
```

**Restant Section** : section outside of the critical part

**Critical Section (CS)** : section manipulating shared data

**Entering Section** : code used to enter **CS**

**Leaving Section** : code used to leave **CS**
Expected Properties of CS

- **Mutual Exclusion**: two processes cannot be in a critical section (concerning the same shared data) at the same time.

- **Progress**: a process operating outside of its critical section cannot prevent other processes from entering theirs.

- **Bounded Waiting**: a process attempting to enter its critical region will be able to do so after a finite time.
• **Deadlock**: two process try to enter in CS at the same time and block each other (errors in entering section.)

• **Race condition**: two processes make an assumption that will be invalidated when the execution of the other process will finished (no mutual exclusion.)

• **Starvation**: a process is waiting for CS indefinitely.

• **Priority Inversion**: a complex double blocking situation between process of different priority. In short, a high priority process is consuming execution time waiting for CS, blocking a low priority process already in CS (spin waiting.)
Solutions with no locks
Bad Solution 1

Example:

```
global int X=0;
global int turn=0;

guardian(int id):
    int other = (id+1)%2;
    for (;;)
        wait_event(id);
        while(turn!=id);
    X = X + 1;
    turn=other;
```
Bad Solution 1

- This solution enforces mutual exclusion: turn cannot have two different values at the same time.
- This solution enforces bounded waiting: you can see the other thread passing only one time while waiting.
- This solution does not respect progression:
  - You will wait for entering CS that the other thread is passed (even if it arrived after you.)
  - If the other thread sees no event, it will not go through the CS and won’t let you take your turn!
Example:

```c
global int X=0;
global int ASK[2] = {0;0};

guardian(int id):
    int other = (id+1)%2;
    for (;;)
        wait_event(id);
    ASK[id] = 1;
    while(ASK[other]);
    X = X + 1;
    ASK[id] = 0;
```
Bad Solution 2

- This solution enforce mutual exclusion: turn cannot have two different values at the same time.

- This solution respects progression

- This solution **present a dead lock:**
  - When asking for CS, each thread will set their flag and then waits if necessary
  - Both thread can set their flag simultaneously
  - Thus, both thread will wait each other with escape possibility
Bad Solution 3

Example:

global int X=0;
global int ASK[2] = {0;0};

guardian(int id):
    int other = (id+1)%2;
    for (;;)
        wait_event(id);
        while(ASK[other]);
        ASK[id] = 1;
        X = X + 1;
        ASK[id] = 0;
• This tiny modification removed the dead lock of previous solution

• But, this solution present a race condition, mutual exclusion is violated:
  • When entering CS, a thread will first wait and then set its flag
  • Both thread can enter the waiting loop before the other one has set its flag and then just pass
  • Both thread can thus enter the CS: lost game!
The Petterson’s Algorithm

Example:

```c
global int X=0;
global int turn=0;
global int ASK[2] = {0;0};

guardian(int id):
    int other = (id+1)%2;
    for (;;)
        wait_event(id);
        ASK[id] = 1;
        turn=other;
        while(turn!=id && ASK[other]);
        X = X + 1;
        ASK[id] = 0;
```
The Petterson’s Algorithm

- The previous algorithm satisfies *mutual exclusion*, *progress* and *bounded waiting*.

- The solution is limited to two process but can be generalized to any number of processes.

- This solution is *hardware/system independent*.

- The main issue is *spin wait*: a process waiting for CS is consuming time resources, opening risks of *priority inversion*.
Definitions
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-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 $\text{IN}(T_j)$.
Transitive Closure

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

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

This relation is the biggest relation that can be obtained from (<) by only adding sub-relation by transitivity.
Tasks’ Dependencies and Graph

Equivalent Relation

Let $< \text{be a well founded partial order, any relation } <_{eq} \text{ is said to be equivalent to } < \text{ if and only if, } <_{eq} \text{ has the same transitive closure as } <$.

Kernel ($<_{min}$)

The kernel $<_{min}$ of a relation $<$ is the smallest relation equivalent to $<$, that is if we suppress any pair $x <_{min} y$ to the relation it is no longer equivalent.
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 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.)
- $\text{IN}(T_1) \cap \text{OUT}(T_2) = \text{IN}(T_2) \cap \text{OUT}(T_1) = \emptyset$
- $\text{OUT}(T_1) \cap \text{OUT}(T_2) = \emptyset$
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 $X$. 

\[ \text{back} \]
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:

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