Parallel Scalable Solutions AB

Multi-thread & Multi-process

- Problems and Solutions

Håkan Sundell , Ph.D.

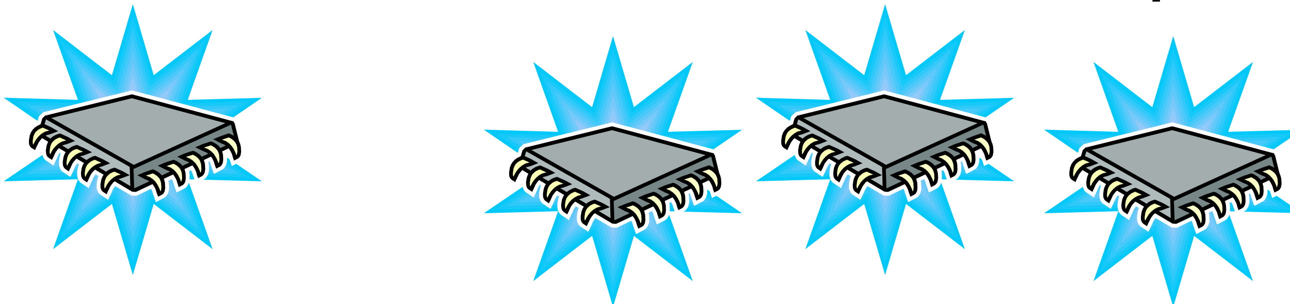
Outline

- Problems in program development
  - Description of core issues
  - Traditional solutions
- Our solutions
  - Technology
  - What we offer

Multi-thread and Multi-process

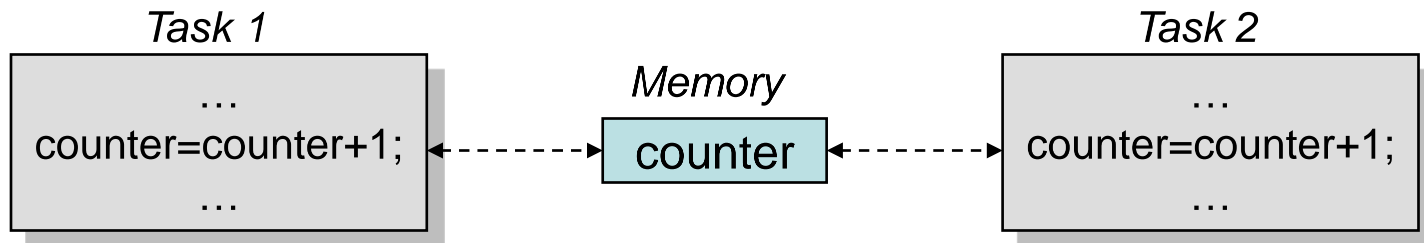
- Programs consist of many tasks

- That execute on one or more processors



Communication

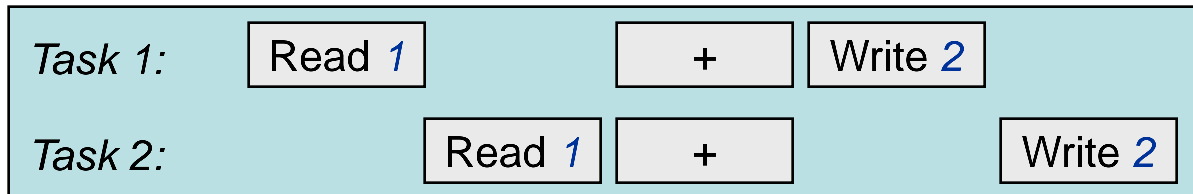
- Threads/Processes need to communicate and work with shared resources.
  - Alternative 1: Message Passing
    - High overhead and Abstract system design.
  - Alternative 2: Shared Memory
    - Fast and Intuitive



Critical Sections

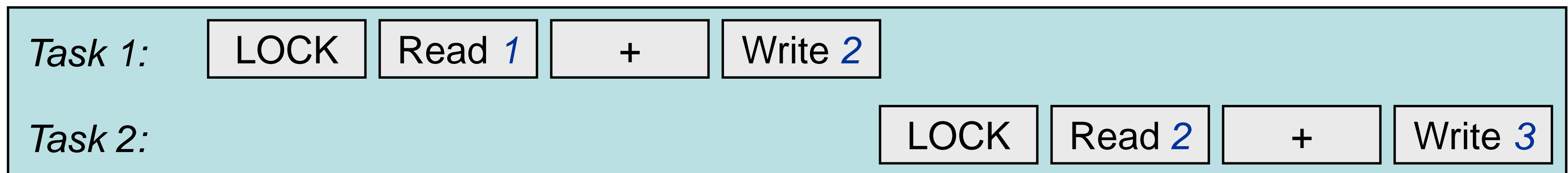
- Problem: operations on shared variables in programming languages are not atomic.

counter=counter+1; = Read + + + Write



! counter=2, but should be 3!

- Straightforward solution: Apply mutual exclusion



Scheduling

- Problem: We have many threads and few processors.
  - Each thread might have different importance (priorities)
- Solution: Threads must be scheduled to alternately use the processors.
  - Scheduling algorithm (Operating System)
  - Pre-emption of running threads, so that all threads will have the chance to run, a little bit at a time.
    - Pre-emption can not be done too often as the task of changing the running thread is very costly.

Multiprocessors

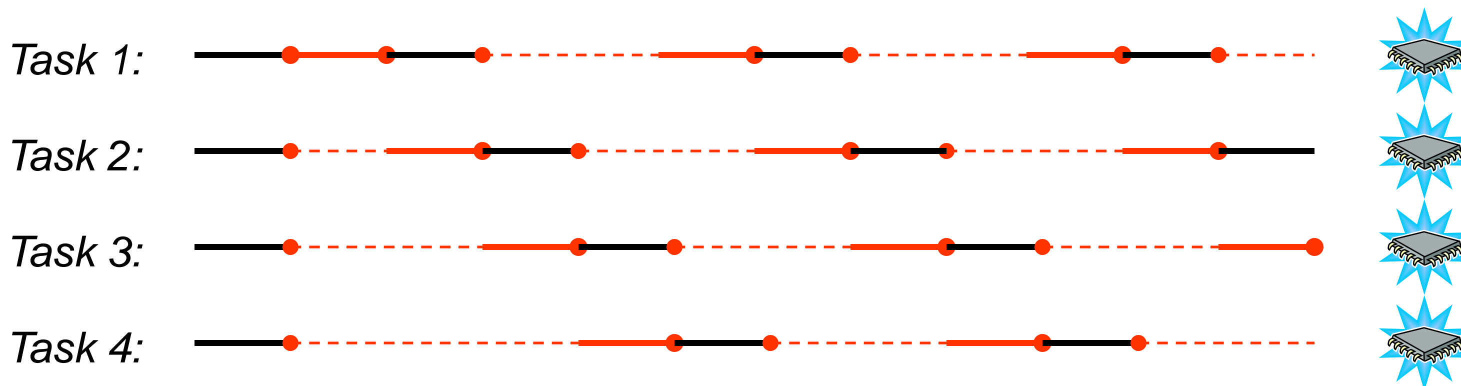
- Needs for performance are increasing
  - Single processor speed has reached its limit (due to power/heat)
- We need multiple processors to increase performance!
  - Multiprocessor systems.
  - Hyper-threading and/or multi-core technology.

Solutions are not compositional!

- Critical Sections + Scheduling:
  - **Blocking**. Threads must wait and get delayed. Significant overhead costs (waiting, pre-emption), i.e. reduced performance.
  - **Deadlocks**. Reduced fault-tolerance, if one thread fails, all other might also fail.
  - **Unpredictable Real-time Behavior**. Threads might not really execute with the set priority.

Critical Sections + Multiprocessors:

- **Reduced Parallelism**. Several threads with overlapping critical sections (in red) will cause waiting processors to go idle (indicated with - - - - -).



Communication+Scheduling+Uni/Multiprocessors:

New Solution

- Avoid Critical Sections!
  - **Avoid Blocking**. Increased performance, better utilization, i.e. less hardware needed ($).
  - **Avoid Deadlocks**. Increased fault-tolerance as failed tasks can not affect others to fail.
  - **Reliable Real-time Behavior**. Access to shared resources will not affect priorities of threads.
  - **Increased Parallelism**. Increased overall performance, better utilization, i.e. less hardware needed ($).

Non-Blocking Synchronization

- Avoiding critical sections: Can it be done?
- Yes, the key lies in how mutual exclusion (i.e. mutex, semaphore) is implemented in actual hardware (i.e. processors).
  - Atomic primitives in hardware can atomically update one memory word.
- Sophisticated solutions can exploit the same atomic primitives to support access to shared resources without locks, i.e. non-blocking.

Non-Blocking Algorithms

- **Lock-Free**. Guarantees that always one operation is making progress.
  - Based on the assumption of a low-conflict scenario, and retries until success.
  - Low overhead, high performance and high scalability.
- **Wait-Free**. Guarantees that any operation will finish in a finite time.
  - Complex solutions that handles high-conflict scenarios with a strict ”ticket” order.
  - High overhead, medium performance and medium scalability.

Other Believers

- **Researchers.** Since over 30 years ago, leading researchers have addressed the problems with critical sections and parellelism and sought for practical non-blocking solutions.
- **Real applications**. Since several years, non-blocking technology are used in a wide range of applications.
  – Applications span from business and trade applications, music and synthesizer applications to even operating system kernels.
  – The Java programming language now contains basic non-blocking constructions.

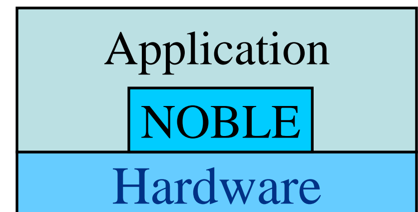
Real Applications: Example

- Splash-2
  - A benchmark suite of several applications for Scientific Computing for multiprocessors.
  - In 2002, researchers performed experiments with replacing simple lock-based constructions with corresponding non-blocking alternatives.
  - Highly considerable increase in scalability for the communication-intensive applications!
    - No increase in applications with very little communication (as it is there that non-blocking synchronization can improve).

Splash-2 Benchmark Suite

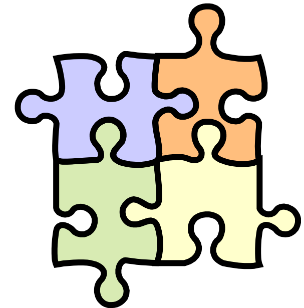
What we offer

- **Custom-designed components**.
  - Wait-free/lock-free solutions for specific purposes.
- **Software library (NOBLE).**
  - State-of-the-art implementations of the very latest wait-free/lock-free versions of common program components.
    - e.g. stack, queue, deque, priority queue, dictionary, list, snapshot, transactions etc.



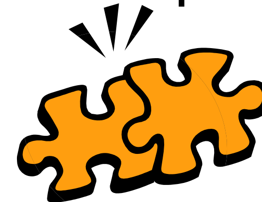
NOBLE Professional Edition: Contents

- Memory Management
  - Memory allocation
  - Memory reclamation (garbage collection)
- Atomic primitives
  - Single-word and Multi-word transactions.
- Common shared data structures
  - Stack
  - Queue
  - Deque
  - Priority Queue
  - Dictionary
  - Linked Lists
  - Snapshots

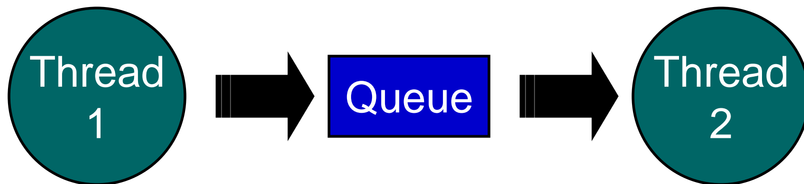


NOBLE Professional Edition: Design

- **Easy to use**. Hides the underlying complexity and shows a common and simple interface.
- **Versatile**. Contains lock-based as well as lock-free/wait-free implementations of each component.
- **Efficient**. Designed for best possible performance.
- **Object-oriented**. Designed in C, and also provided with a generic interface in C++.
- **Configurable**. A lot of optional parameters and functionalities that can be set/tuned to meet specific needs.



NOBLE Professional Edition: Example (C)



Globals

```
#include <Noble.h>
NBLQueueRoot* queue;
```

Main

```
queue=NBLQueueCreateLF();
/* Create and run the threads */
NBLQueueFree(queue);
```

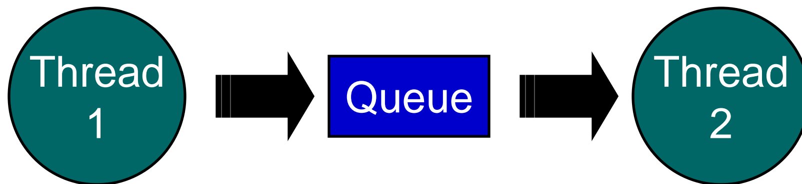
Thread 1

```
NBLQueue *handle=
NBLQueueGetHandle(queue);
...
NBLQueueEnqueue(handle, item);
...
NBLQueueFreeHandle(handle);
```

Thread 2

```
NBLQueue *handle=
NBLQueueGetHandle(queue);
...
item=NBLQueueDequeue(handle);
...
NBLQueueFreeHandle(handle);
```

NOBLE Professional Edition: Example (C++)



Thread 1

```
T* item = new T;
...
queue->Enqueue(item);
```

Globals

```
#include <Noble.h>
NBL::Queue<T>* queue;
```

Main

```
queue=NBL::Queue<T>::CreateLF();
/* Create and run the threads */
delete queue;
```

Thread 2

```
T* item=queue->Dequeue();
...
delete item;
```

How we work

Algorithmic Design

Correctness

Implementation

- *Practical*
- *Efficient*

How we work

- Algorithm design
  - We are experts in designing efficient non-blocking algorithms suited for practice.
- Correctness
  - We create proofs of relevant correctness criteria using analytical and mathematical-style readable text, that cover *all* possible cases of concurrency and interleavings of program statements.
- Implementation
  - Implementation of algorithms in real program code involves many details that need deep understanding of the corresponding algorithm in order to properly inherit the correctness and performance from it.

Areas of possible non-blocking applications

- **Embedded systems**
  - Enhanced predictability and simpler development.
- **Scientific computing**
  - Increased scalability gives results faster.
- **Business and support systems**
  - Handle more transactions per time unit.
- **IT and telecommunication**
  - Make more with same hardware.
- **Games and entertainment**
  - Better performance with same (the users) hardware.

Questions?

Please feel free to contact us for more information!