Threaded Programming

V. Erkcan Özcan
Boğaziçi University

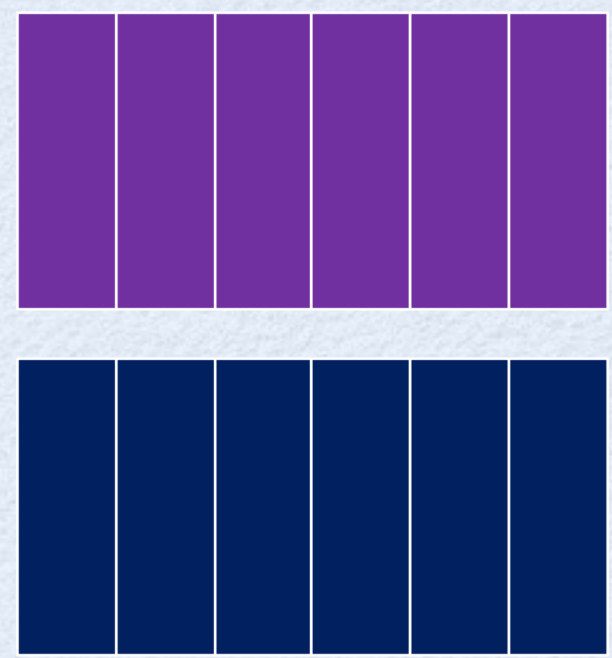
Based on past ISOTDAQ lectures by Giuseppe Avolio, Gökhan Ünel, Giovanna L. Miotto…

ISOTDAQ'17, February 1, 2017, Amsterdam

What is Concurrency?

- Compulsory wikipedia descriptions:
    - In computer science, **concurrency** or **concurrence** is a property of systems in which several computations are executing simultaneously, and potentially interacting with each other. The computations may be executing on multiple cores in the same chip, preemptively time-shared threads on the same processor, or executed on physically separated processors.
    - **Concurrent computing** is a form of computing in which several computations are executing during overlapping time periods—concurrently—instead of sequentially (one completing before the next starts).

Task switching on a single core computer



Parallelism on a dual-core computer

A Bit of History

- late 1950s: time-sharing using interrupts and multiple CPUs discussed by Gill.
- 1960s: Burroughs D825, IBM System/360 - first multiCPU systems.
- 1962: E. Codd, "Multiprogramming" - mutex.
- 1965: E. W. Dijkstra, "Solution of a Problem in Concurrent Programming Control" - semaphore (seinpaal).

Dennis and Van Horn [11] have used the words "locus of control within an instruction sequence," to describe a process; the alternative term "thread" (suggested by V. Vyssotsky) is suggestive of the abstract concept embodied in the term "process."

- 1995: POSIX.1c (IEEE std 1003.1c): pthreads
  - pthread_create(), pthread_join(), etc.

Solution of a Problem in Concurrent Programming Control

E. W. Dijkstra
Technological University, Eindhoven, The Netherlands

A number of mainly independent sequential-cyclic processes with restricted means of communication with each other can be made in such a way that at any moment one and only one of them is engaged in the "critical section" of its cycle.

Introduction

Given in this paper is a solution to a problem for which, to the knowledge of the author, has been an open question since at least 1962, irrespective of the solvability. The paper consists of three parts: the problem, the solution, and the proof. Although the setting of the problem might seem somewhat academic at first, the author trusts that anyone familiar with the logical problems that arise in computer coupling will appreciate the significance of the fact that this problem indeed can be solved.

Concurrency, Why?

Concurrency, Why?

- The old answers:
  - Driving slow devices, such as disks, terminals, printers, networks, etc. Your program can still do useful work in the other threads while it is handling such devices.
  - Enduser is impatient: People want to do multiple things with the computer at the same time.
    - Reduce latency: You can respond to an enduser request fast and then do the dirty work later.
  - Multiple clients: In a system with shared resources (file server, web server, etc.), clients requests can run "simultaneously".

Concurrency, Why?

- The old answers:
  - Driving slow devices, such as disks, terminals, printers, networks, etc. Your program can still do useful work in the other threads while it is handling such devices.
  - Enduser is impatient: People want to do multiple things with the computer at the same time.
    - Reduce latency: You can respond to an enduser request fast and then do the dirty work later.
  - Multiple clients: In a system with shared resources (file server, web server, etc.), clients requests can run "simultaneously".
- Good coding answers:
  - Group related piece of code together
  - Identify and separate areas of functionality

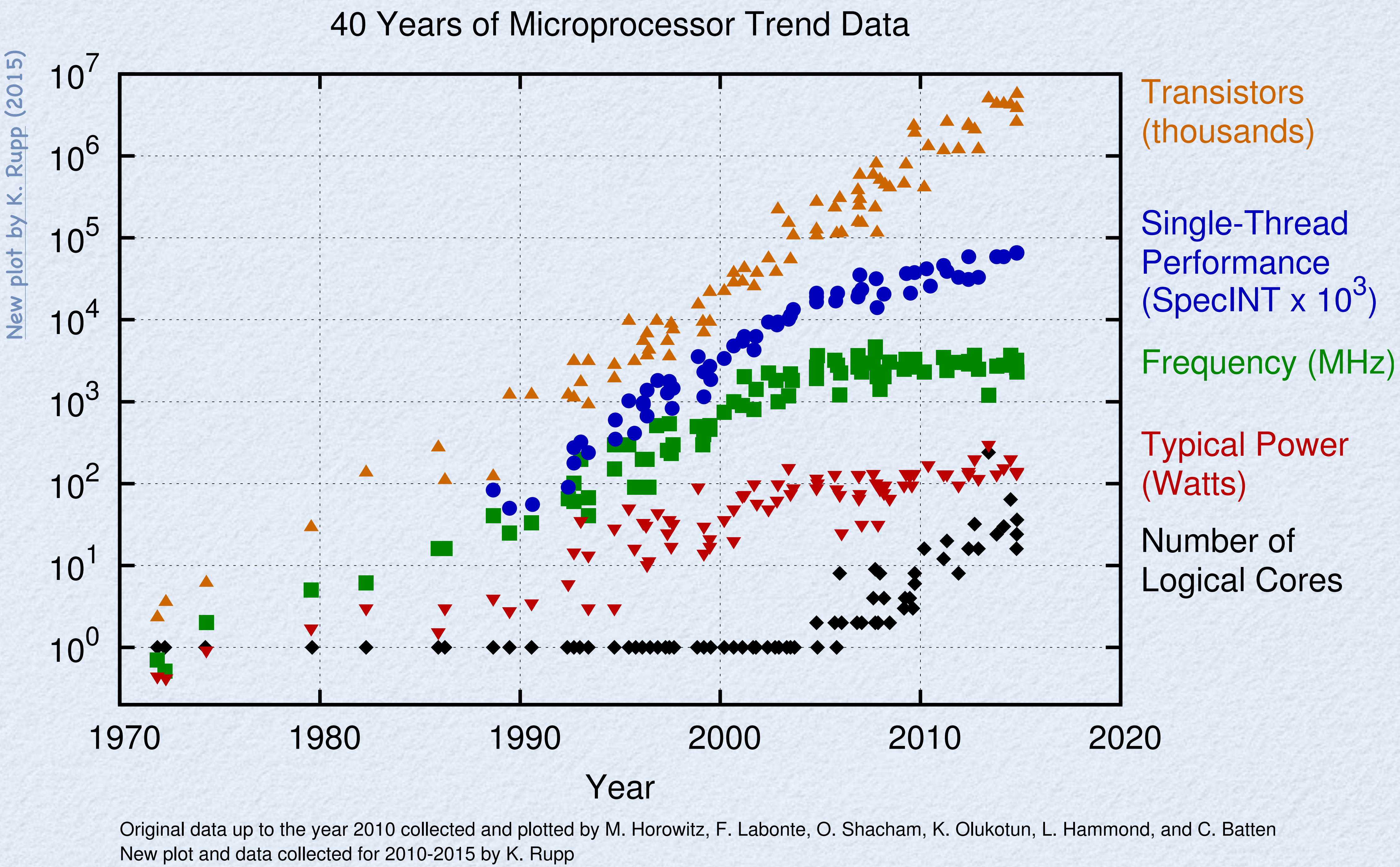
Concurrency, Why?

- Compulsory stackexchange answer (from 2011):

> 25
>
> Here's a quick and easy motivation: If you want to code for anything but the smallest, weakest systems, you *will* be writing concurrent code.
>
> Want to write for the cloud? Compute instances in the cloud are small. You don't get big ones, you get lots of small ones. Suddenly your little web app is a concurrent app. If you designed it well, you can just toss in more servers as you gain customers. Else you have to learn how while your instance has its load average pegged.
>
> OK, you want to write desktop apps? Everything has a dual-or-more-core-CPU. Except the least expensive machines. And people with the least expensive machines probably aren't going to fork over for your expensive software, are they?
>
> Maybe you want to do mobile development? Hey, the iPhone 4S has a dual-core CPU. The rest won't be far behind.
>
> Video games? Xbox 360 is a multi-CPU system, and Sony's PS3 is essentially a multi-core system.
>
> You just can't get away from concurrent programming unless you are working on tiny, simple problems.
>
> share improve this answer
>
> answered Oct 21 '11 at 3:27
>
> ObscureRobot
>
> 534 • 3 • 6

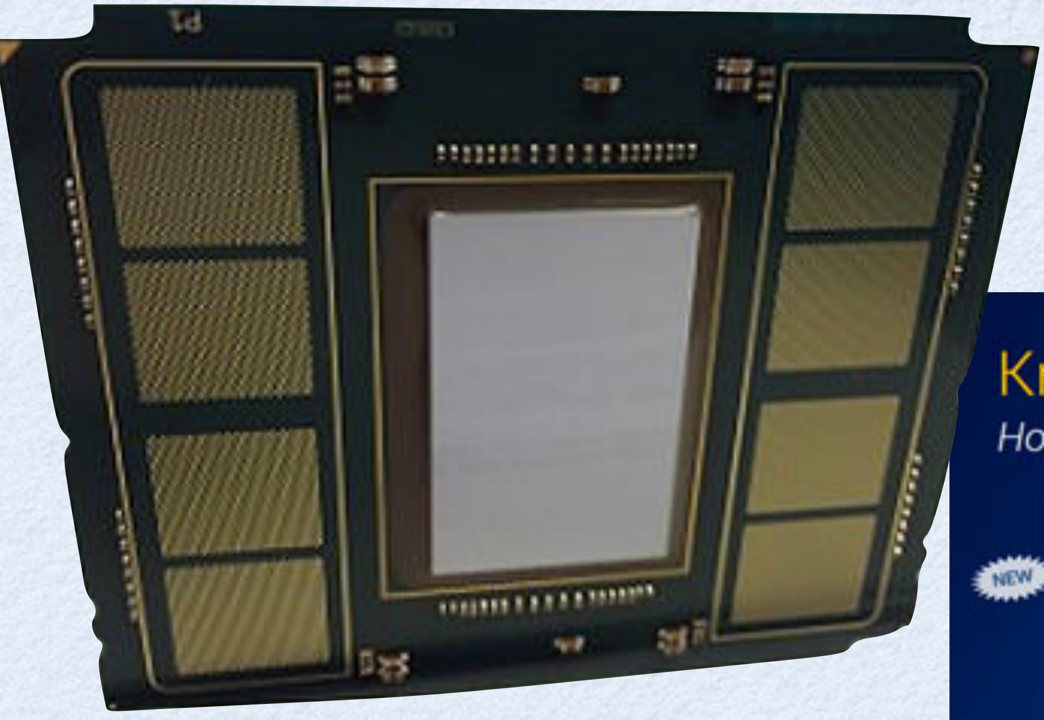
Yes, it is a bit cheesy, but gets the idea right.

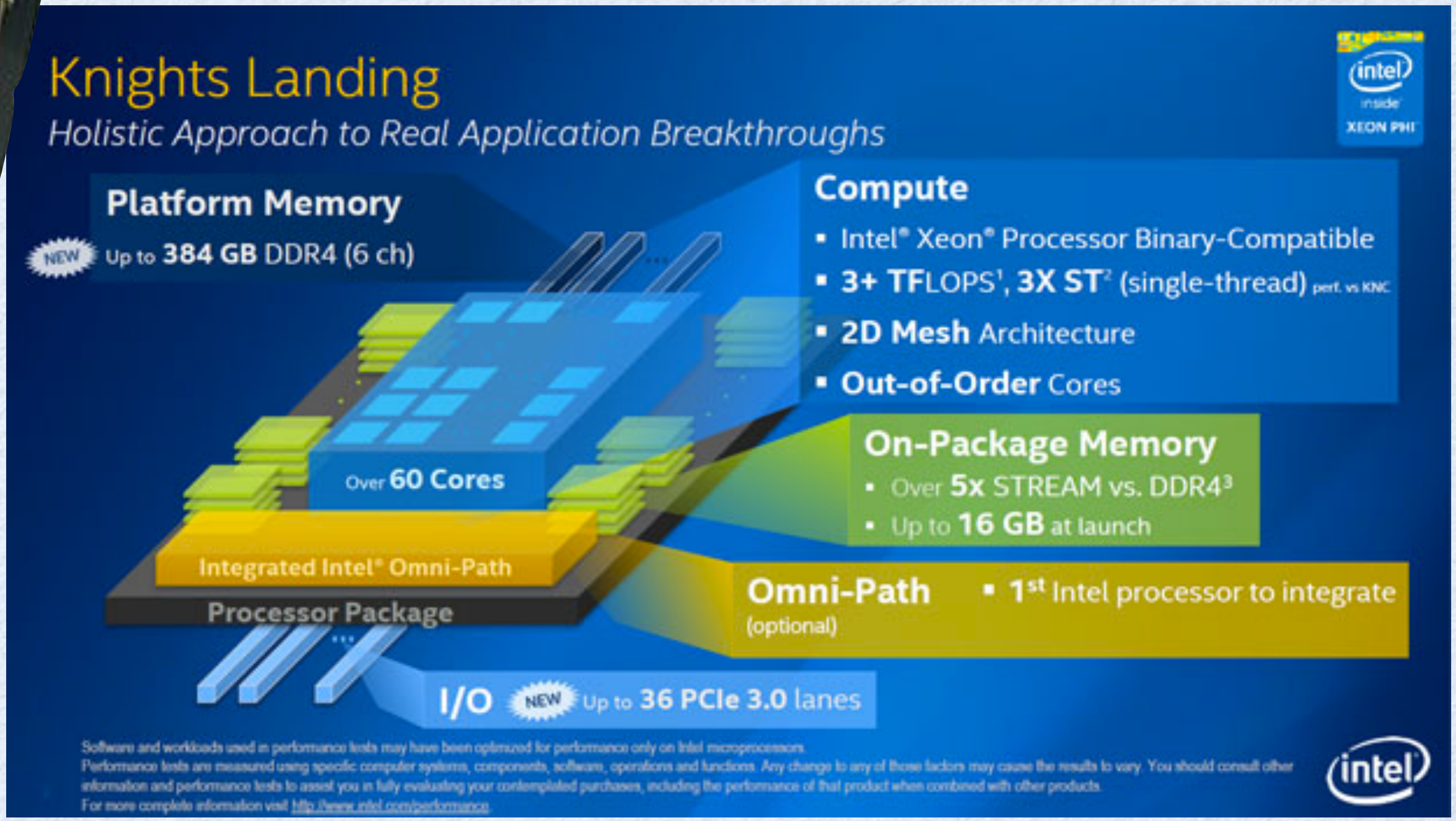
Concurrency is the Future



- Single-thread performance increasing very slowly (thanks to dynamic clock frequency adjustments), but don't expect more for-free faster code.
- Transistor count still going logarithmic; concurrent computing is the way of the future!

Already a Quarter Thousand!





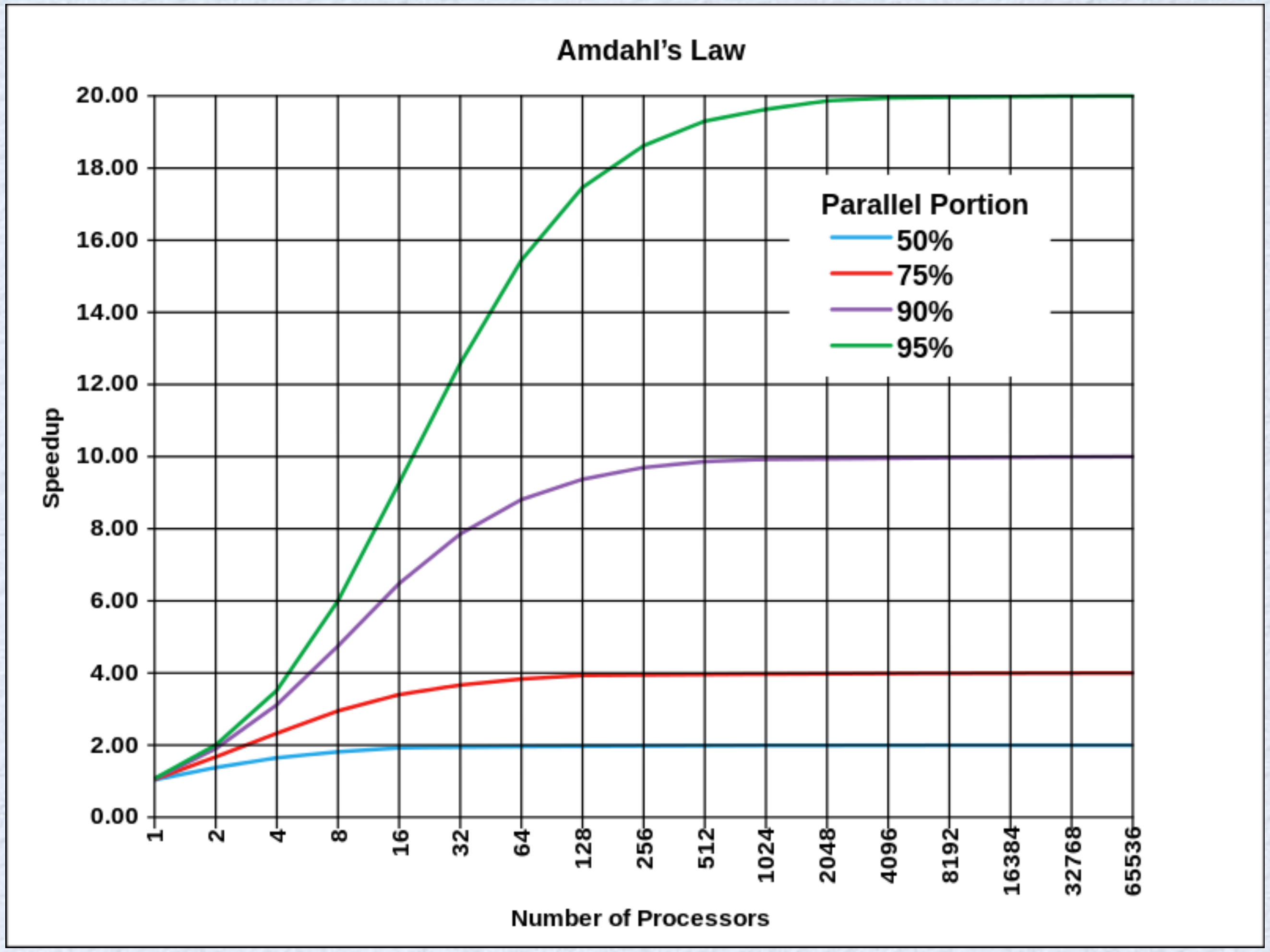
- Intel Xeon Phi 7290 (previously called "Knights Landing") - 72 cores with 4 threads each (4-way SMT), ie. 288 logical cores!
    - Launched in Q2 of last year (2016).

SMT = Simultaneous multithreading

Amdahl's Law

$$S_{\text{latency}}(s) = \frac{1}{1 - p + \frac{p}{s}},$$

- Rather straightforward argument.
  - $S_{latency}$ = theoretical speedup in latency of the execution of the whole task.
  - s is the speedup factor (say number of parallel processors) for the execution of the part of the task that benefits from paralellisation;
  - p is the percentage of the execution time of the whole task concerning the part that benefits from the improvement of the resources of the system before the improvement.

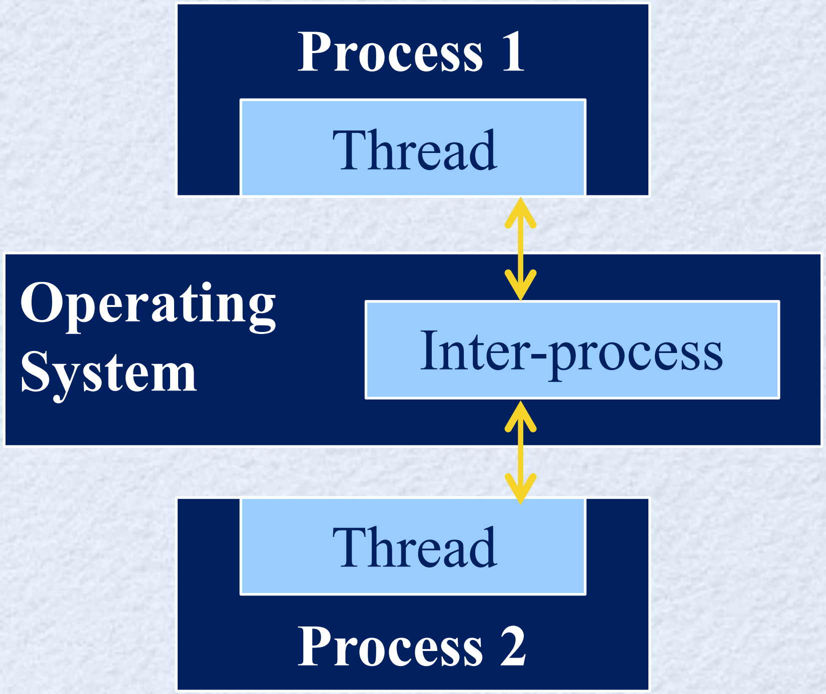


Daniels220 at English Wikipedia

Example: if 95% of the program can be parallelised, the theoretical maximum speedup using parallel computing would be 20 times, no matter how many processors are used.

Concurrency's Flavors 1

- **Multiple processes**
  - Separate applications running at the same time
  - Messages can be exchanged using the inter-process mechanism provided by the Operating System (signals, sockets, files…)
  - Cons
    - Inter-process communication is usually complicated or slow
    - Overhead: duplicating resources needed by the OS and the application itself
  - Pros
    - Easier to write correct concurrent code
    - Processes can be spawn on different nodes connected over a network



Quick tip: use task spooler (ts)

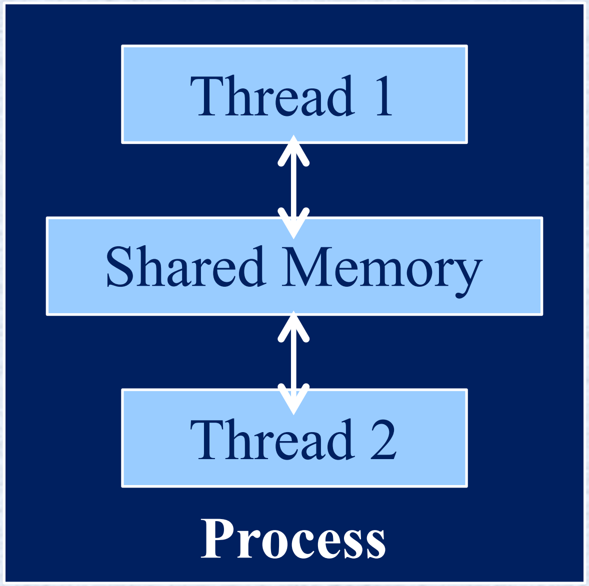
Forking

```
#include <stdio.h>     // printf, stderr, fprintf
#include <sys/types.h> // pid_t
#include <unistd.h>    // _exit, fork, sleep
#include <stdlib.h>    // exit
#include <errno.h>     // errno

int main(void)
{
   pid_t  pid;
   pid = fork();

   if (pid == -1) {
      fprintf(stderr, "can't fork, error %d\n",
errno);
      exit(EXIT_FAILURE);
   }

   if (pid == 0) {
      /* Child process:
       * If fork() returns 0, it is the child
process.
       */
      int j;
      for (j = 0; j < 15; j++) {
         printf("child: %d\n", j);
         sleep(1);
      }
      _exit(0);  /* Note that we do not use
exit() */
   } // end of child
   else
   {

      /* If fork() returns a positive number, we
are in the parent process
       * (the fork return value is the PID of the
newly created child process)
       */
      int i;
      for (i = 0; i < 10; i++)
      {
         printf("parent: %d\n", i);
         sleep(1);
      }
      exit(0);
   }// end of parent

   return 0;
}
```

- A kind of multi-process concurrency is forking.
- Very easy to use, but slow and heavy:
  - a separate address space for the child process with an exact copy of all the memory segments of the parent.

Concurrency's Flavors 2

- **Multiple threads**
  - Threads are often called lightweight processes
    - A process may run one or several threads
    - A thread is the smallest unit of processing that can be scheduled by the OS
  - Resources
    - Shared global memory address space (i.e., access to the same variables)
    - But each thread has its own stack and local variables
  - Threads can be executed simultaneously & asynchronously with respect to the other ones
  - Lower overhead with respect to running multiple processes



Risks of Threads

Safety

- The execution's order of threads is unpredictable
- A thread may access or modify variables another thread is using
- Access to shared data must be coordinated: synchronization

Liveness

- Deadlock: all the threads wait for the same resource
- Starvation: a thread is perpetually denied access to a resource it needs in order to make progress
- Livelock: a thread keeps retrying an operation that will always fail

Performance

One needs to take into account the time spent for synchronization and scheduling.

- Save/restore register state, cache state, etc.
- Too many threads: OS reverts to round-robin; time slice for a thread that locked a resource might expire, locking all others.

Difficult to maintain.

Warning

- "Although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism. Threads, as a model of computation, are wildly nondeterministic, and the job of the programmer becomes one of pruning that non-determinism."

  The Problem with Threads, Edward A. Lee, 2006

- Do it correctly the first time! Debugging this thing is an order of magnitude worse than your sequential code.

A Very First Threaded Program

```
#include <iostream>
#include <thread>
#include <vector>

void testfunction(int tid) {
  std::cout << "Hey I am here " << tid << std::endl;
}

int main() {

  // this gives 8 on my hyperthreaded 4-core machine
  const int nhardthread = std::thread::hardware_concurrency();
  std::cout << "# of hardware units: " << nhardthread << std::endl;

  std::vector<std::thread> t;
  for (int i=0; i<nhardthread; ++i) {
    t.push_back(std::thread(testfunction,i+1)); }

  for (auto& th : t) th.join();

  return 0;
}
```

```
# of hardware units: 8
HHeHHHHeHHyeeeeyee yyyy yyI    I   IIII IIa    a  maaaamaa mmmm mmh    h
ehhhhehhreeeereeerrrrerr eeee ee7    3
6184
25
```

- In C++11 (and onwards), it is very easy to play with threads.
  - Functors can also be made into threads easily, but they are beyond the scope of this lecture.
- We generate one thread per virtual core.
  - Oversubscription is also ok if you know what you are doing.
- Use std::vector to get an array of threads and then join() them.
  - join() makes sure each thread is completed before the main is done.
- The program ends when all the joined threads have finished execution.
- The output is scrambled, as we would expect from 8 threads running concurrently.

Mutual Exclusion

```
#include <iostream>
#include <thread>
#include <vector>
#include <mutex>

std::mutex mtx; // lockable object to encapsulate critical sections

void testfunction(int tid) {
  std::lock_guard<std::mutex> guard(mtx); // RAII
  //mtx.lock();
  std::cout << "Hey I am here " << tid << std::endl;
  //mtx.unlock();
}

int main() {

  // this gives 8 on my hyperthreaded 4-core machine
  const int nhardthread = std::thread::hardware_concurrency();

  std::vector<std::thread> t;
  for (int i=0; i<nhardthread; ++i) {
    t.push_back(std::thread(testfunction,i+1)); }

  for (auto& th : t) th.join();

  return 0;
}
```

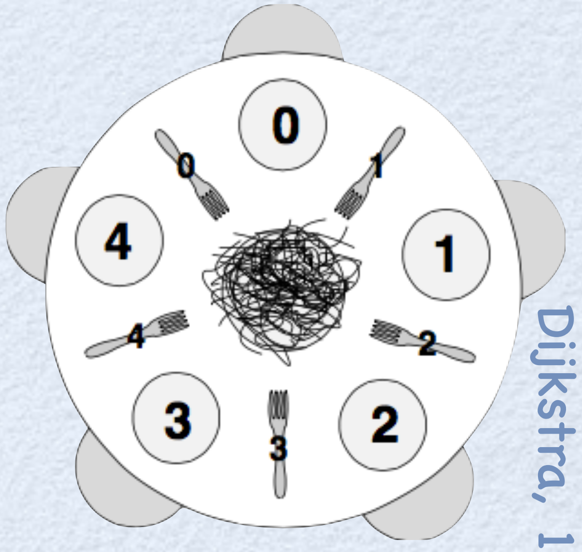
```
Hey I am here 1
Hey I am here 2
Hey I am here 5
Hey I am here 4
Hey I am here 3
Hey I am here 8
Hey I am here 7
Hey I am here 6
```

But wait, we lost all concurrency.

- If we want to unscramble the output, we need that the resource std::cout is accessed by the threads one at a time.
  - We can use a mutex. When the mutex is locked, the execution waits until it gets unlocked.
- RAII: Resource Acquisition Is Initialization
  - Resource (mutex) is allocated during the creation of the lock_guard object, while deallocation happens during object destruction, by the destructor.
- Extra hint: You can do even better than this. You can put a mutex and any other resource (stream, network, etc.) into a class and hide the resource completely from direct access.

Not Just "Resource" Sharing

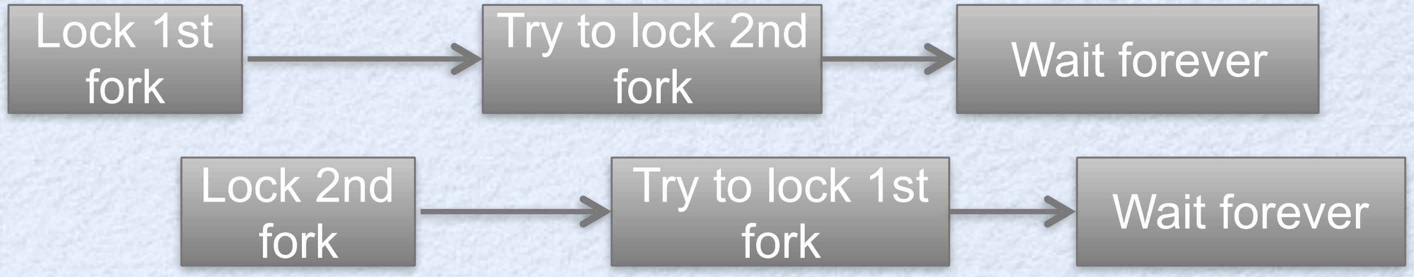
- Don't see the use of the mutex just as a means for different threads reaching out to a limited physical resource (like cout) one at a time.
- You don't have to have threads doing identical jobs; you can have multiple threads of different kinds interacting with each other.
- For a list of classical (and not-so-classical) synchronisation puzzles, see for example, The Little Book of Semaphores, by A. B. Downey.
  - Producer-consumer, readers-writers, dining philosophers, cigarette smokers, barbershop, river crossing, unisex bathroom, sushi bar, child care, etc.
  - The syntax of the language is useful, but these allow you to think of new algorithms in real-life problems.



Dijkstra, 1965.

A Word of Warning

- Having mentioned the dining philosophers, do you see the real problem with it?
  - Here is your pseudocode for each philosopher:
    1. Ponder the nature of reality until the left fork is available; when it is, pick it up.
    2. Ponder the nature of reality until the right fork is available; when it is, pick it up.
    3. When both forks are held, eat for a fixed amount of time.
    4. Put the right fork down.
    5. Put the left fork down.
    6. Repeat from the beginning.



  - If you are not careful, you will end up with **deadlocks**, even with two philosophers. Don't use your mutexes arbitrarily.
- See one solution at: http://rosettacode.org/wiki/Dining_philosophers

Producer-Consumer

- A lot of the tasks encountered in physics applications can be handled using producer-consumer patterns.
- Example:
  - Input threads read data from the detector and places them on to an event filter (EF) input stack.
  - EF threads read records from the EF input stack, process them, and put (some of) them on the output stack.
  - The output threads read records from the output stack and store them.
- There are libraries for implementing this sort of thing, with “low-level” stuff like mutexes hidden from the enduser by the thread-aware queues. Let’s see how they work.

Primitive Producer-Consumer

```cpp
#include <iostream>
#include <thread>
#include <stack>
#include <mutex>
#include <chrono>

std::mutex mtx;
std::stack<int> stk;
using namespace std;

void feed() {
  for (int i=0; i<10; ++i) { // growing food size
    unique_lock<mutex> ulocker(mtx); // like lock_guard, but ...
    stk.push(i);
    ulocker.unlock(); // ... but allows unlock before the destructor
    this_thread::sleep_for(chrono::seconds(1)); }
}

void eat() {
  int hunger = 40; // starting hunger level
  while (hunger>0) {
    unique_lock<mutex> ulocker(mtx);
    if (!stk.empty()) {
      hunger -= stk.top();
      stk.pop();
      ulocker.unlock();
      cout << "I ate something. Hunger level=" << hunger << endl; }
    else ulocker.unlock();
  }
}

int main() {
  thread tf(feed), te(eat);
  tf.join();
  te.join();
  return 0;
}
```

```
I ate something. Hunger level=40
I ate something. Hunger level=39
I ate something. Hunger level=37
I ate something. Hunger level=34
I ate something. Hunger level=30
I ate something. Hunger level=25
I ate something. Hunger level=19
I ate something. Hunger level=12
I ate something. Hunger level=4
I ate something. Hunger level=-5
```

- Food stack is on our shared memory.
  - Hence the mutex.
- Each second feed() puts in larger and larger pieces.
- What is eat() really eating?

Primitive Producer-Consumer

```cpp
#include <iostream>
#include <thread>
#include <stack>
#include <mutex>
#include <chrono>

std::mutex mtx;
std::stack<int> stk;
using namespace std;

void feed() {
  for (int i=0; i<10; ++i) { // growing food size
    unique_lock<mutex> ulocker(mtx); // like lock_guard, but ...
    stk.push(i);
    ulocker.unlock(); // ... but allows unlock before the destructor
    this_thread::sleep_for(chrono::seconds(1)); }
}

void eat() {
  int hunger = 40; // starting hunger level
  while (hunger>0) {
    unique_lock<mutex> ulocker(mtx);
    if (!stk.empty()) {
      hunger -= stk.top();
      stk.pop();
      ulocker.unlock();
      cout << "I ate something. Hunger level=" << hunger << endl; }
    else ulocker.unlock();
  }
}

int main() {
  thread tf(feed), te(eat);
  tf.join();
  te.join();
  return 0;
}
```

```
I ate something. Hunger level=40
I ate something. Hunger level=39
I ate something. Hunger level=37
I ate something. Hunger level=34
I ate something. Hunger level=30
I ate something. Hunger level=25
I ate something. Hunger level=19
I ate something. Hunger level=12
I ate something. Hunger level=4
I ate something. Hunger level=-5
```

- Food stack is on our shared memory.
  - Hence the mutex.
- Each second feed() puts in larger and larger pieces.
- What is eat() really eating?

Lots of CPU cycles.

Condition Variables

```
#include <iostream>
#include <thread>
#include <stack>
#include <mutex>
#include <chrono>

std::mutex mtx;
std::stack<int> stk;
std::condition_variable cvar;
using namespace std;

void feed() {
  for (int i=0; i<10; ++i) { // growing food size
    unique_lock<mutex> ulocker(mtx); // like lock_guard, but ...
    stk.push(i);
    ulocker.unlock(); // ... but allows unlock before the destructor
    cvar.notify_one(); // if there are waiting threads, notify one
    this_thread::sleep_for(chrono::seconds(1)); }
}

void eat() { // non-polling version
  int hunger = 40; // starting hunger level
  while (hunger>0) {
    unique_lock<mutex> ulocker(mtx);
    cvar.wait(ulocker, [](){ return !stk.empty(); });
    hunger -= stk.top();
    stk.pop();
    ulocker.unlock();
    cout << "I ate something. Hunger level=" << hunger << endl; }
}

int main() {
  thread tf(feed), te(eat);
  tf.join();
  te.join();
  return 0;
}
```

- We could also add some sleep (say 50ms) to eat(): when no new food, why not just say, `this_thread::sleep_for(chrono::milliseconds(50));` ?
- But isn't there a method without any polling?
- Once the food is ready we notify (any) one thread that is waiting.
- Hungry eat() is sleeping. If it wakes up on its own, it goes back to sleep if `!stk.empty()`. Otherwise it wakes up thanks to the notification and eats. Yummy…

Are Mutexes Always Necessary?

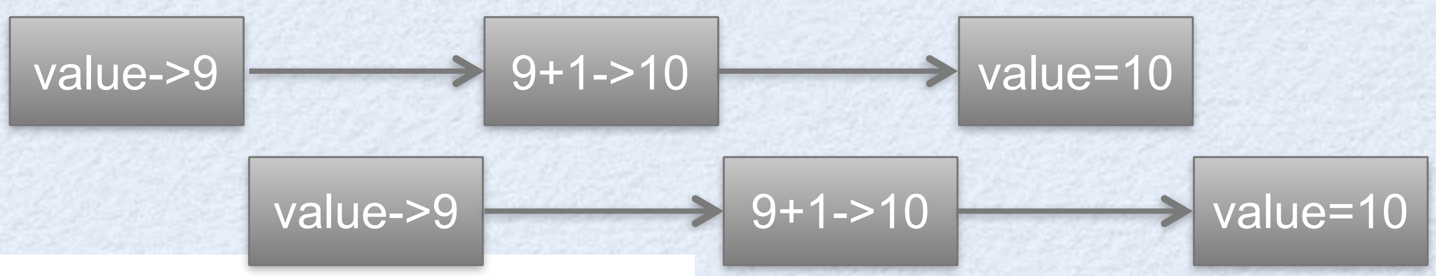
```
#include <iostream>
#include <thread>
#include <vector>
#include <chrono>

int counter(0); // a global counter
int complete(0); // counting the completion of each thread

void testfunction(int tid) {
  std::cout << "Hey I am here " << tid << std::endl;
  for (int i=0; i<10000; ++i)  ++counter;  // an important computation
  complete++;
}

int main() {

  // this gives 8 on my hyperthreaded 4-core machine
  const int nhardthread = std::thread::hardware_concurrency();

  std::vector<std::thread> t;
  for (int i=0; i<nhardthread; ++i) {
    t.push_back(std::thread(testfunction,i+1)); }
  for (auto& th : t) th.join();

  // Give 1.5 seconds to the threads so they finish (no guarantee)
  std::this_thread::sleep_for (std::chrono::milliseconds(1500));

  std::cout << "All the threads counting together = " << counter << std::endl;
  std::cout << "# of completed threads = " << complete << std::endl;

  return 0;
}
```

```
All the threads counting together = 18134
# of completed threads = 8
```

- The same code but this time it also performs a very important computation: “counting up to 10k 8 times”.
  - Scrambling is not an issue now. Whichever thread gets to the resource, let it use it.
  - Removed the mutex, because “we want the counting part to be concurrent”. (You think you are smart, don’t you?)
- We give 1.5 seconds for the completion of threads.
- But the output is wrong and it is changing each time we run the code.
  - Why? What is wrong? Are printing before the threads finish their computations? But #completed is 8.

Need for Atomicity

- CPU is much faster than the memory. In order to overcome the memory latency, memory caches are used.
  - Each core/CPU has its own cache. However if a value in one cache is updated, the value in the other caches become invalid.
  - There are cache-coherence protocols to overcome these issues. But even with those, unless you apply a memory model in your programming language, you will end up with issues.
  - If we simplify this: Consider one thread reads the value of the counter, but before it writes back the incremented value, another thread reads the old value of the counter.




```
#include <chrono>

std::atomic<int> counter(0); // a global counter
int complete(0); // counting the completion of eac

void testfunction(int tid) {
  std::cout << "Hey I am here " << tid << std::enc
  for (int i=0; i<10000; ++i)  ++counter;  // an i
  complete++;
}
```

- It is important that read-increment-write is done atomically; nothing should be able to break that. Use std::atomic<int>.

```
All the threads counting together = 80000
# of completed threads = 8
```

Messages from the Future

```
#include <iostream>
#include <thread>
#include <vector>
#include <future>

int testfunction(int tid) {
  int mycounter(0);
  std::cout << "Hey I am here " << tid << std::endl;
  for (int i=0; i<10000; ++i)  ++mycounter;  // the difficult computation
  return mycounter;
}

int main() {

  // this gives 8 on my hyperthreaded 4-core machine
  const int nhardthread = std::thread::hardware_concurrency();

  std::vector<std::future<int>> futureresults(8);
  for (int i=0; i<nhardthread; ++i) {
    futureresults[i] = std::async(std::launch::async,testfunction,i+1); }

  int counter(0);
  for (auto& fr : futureresults) {
    counter += fr.get(); }

  std::cout << "All the threads counting together = " << counter << std::endl;

  return 0;
}
```

```
HHeeyy  IHI HHe aeeHyaHmyyeH me   yeI yhII y h e  I aeIraa Imr emma
ea   mah mh1h me2 e
eh r8 …
All the threads counting together = 80000
```

- As an alternative, we can let each thread to perform the difficult computation on local variables and return the result.
- But when will they complete their computations?
  - Who cares? Sometime in the future we will know the result.

Closing Advice

1989 © DEC

An Introduction to Programming with Threads

Andrew D. Birrell

This paper provides an introduction to writing concurrent programs with "threads". A threads facility allows you to write programs with multiple simultaneous points of execution, synchronizing through shared memory. The paper describes the basic thread and synchronization primitives, then for each primitive provides a tutorial on how to use it. The tutorial sections provide advice on the best ways to use the primitives, give warnings about what can go wrong and offer hints about how to avoid these pitfalls. The paper is aimed at experienced programmers who want to acquire practical expertise in writing concurrent programs.

There is a revised version with C#, 2005 © Microsoft.

Memory Barriers: a Hardware View for Software Hackers

Paul E. McKenney
Linux Technology Center
IBM Beaverton
paulmck@us.ibm.com

April 5, 2009

So what possessed CPU designers to cause them to inflict memory barriers on poor unsuspecting SMP software designers?

In short, because reordering memory references allows much better performance, and so memory barriers are needed to force ordering in things like synchronization primitives whose correct operation depends on ordered memory references.

Getting a more detailed answer to this question requires a good understanding of how CPU caches work, and especially what is required to make caches really work well. The following sections:

ing ten instructions per nanosecond, but will require many tens of nanoseconds to fetch a data item from main memory. This disparity in speed — more than two orders of magnitude — has resulted in the multi-megabyte caches found on modern CPUs. These caches are associated with the CPUs as shown in Figure 1, and can typically be accessed in a few cycles.[1]

The Little Book of Semaphores

Second Edition

Version 2.1.5

- Learn it like a pro.
- Old ISOTDAQ lectures are available online.
  - ISOTDAQ'12, Giovanna's lecture with Java.
  - ISOTDAQ'13, Gökhan's lecture with C+11 & pthreads + signal handling.
  - ISOTDAQ'15, Giuseppe's lecture with extra information on the HW side

Conclusion

- Parallel programming is fast becoming a must…
  - It is not easy, but:
    - At least it is easier than it used to be. Concurrent code is also a lot more portable than before.
    - It can be quite a lot of fun.
- This lecture is only the tip of the iceberg.
  - Two "concurrent" learning steps needed: (1) play with the code freely on your own, making your own mistakes; (2) study at least the classical examples from a decent source.
    - More than any other programming experience, concurrent programming requires some reformulating your ideas in foreign ways.
    - Its syntax is easy to learn, but it requires a new point of view.
- Homework:
  - Implement a solution to the dining savages problem (or pick another problem of your liking from the Little Book); OR:
  - Integrate some function, say $\sin^2\theta$, using Monte Carlo integration distributed over a number of threads. Measure the speedup. Test how well hyperthreaded cores behave.

Backups

Debugging Tips

- TIP#1: Try to avoid falling into finding yourself in a situation where you have to do debugging!
  - Use the established patterns of parallel programming.
    - You might even completely forget about mutexes and semaphores as certain patterns are already coded for you in libraries like Boost.
  - In some cases, it might be useful to ask "would my threads work ok when they were running sequentially?"
- Trace buffers: Add a simple message buffer to log which thread is using a given resource at what time. Write a function to dump that buffer when needed. Call that function with gdb.
- Some quick gdb commands:
  - info threads: see list of threads with id numbers
  - thread <idno>: switch to the thread with given idno
  - thread apply <thread ids/all> <command>: send a given gdb command to a list of (or all of) threads.