Johan Seland

Parallel Programming Patterns

SINTEF Petroleum Development Workshop – Session 3

Trondheim - 9. December 2010

Overview

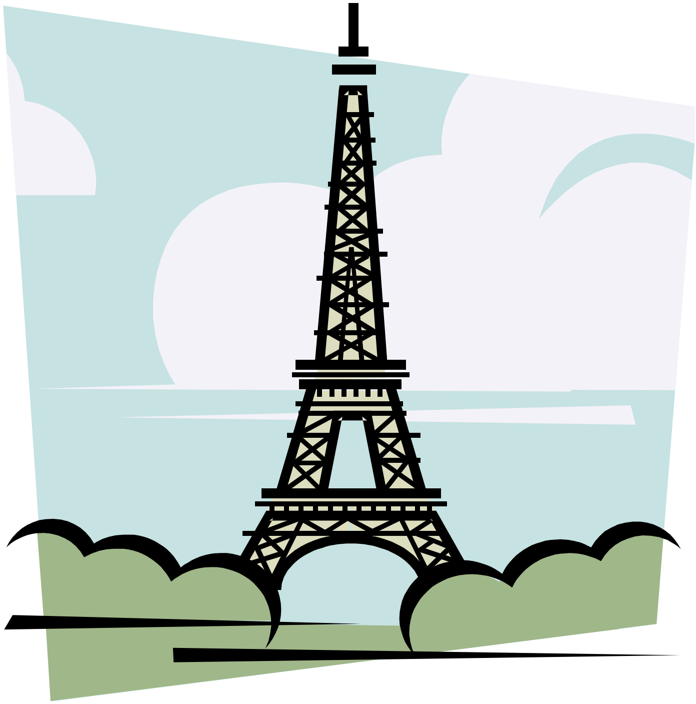
- Introduction and vocabulary
- Limits to performance
  - Amdahls Law vs Gustafson Law
- Concurrency
  - Domain decomposition
  - Task parallelism
- Synchronization
  - Fences
  - Barriers
  - Mutex es
  - Semaphores
- Testing parallel programs

A word about patterns

A **Pattern** is:

- General
- Reusable
- Based on a proven design
- Not directly translatable into code

- Originated in architecture
  - Christopher Alexander 1977
- Now part of the Software Eng. Vocabulary
  - Gang of Four, 1987
- There is also Anti-Patterns



Vocabulary

- Task
  - Sequence of instructions that operate together
- Thread (process)
  - An executing task
- Core
  - The underlying hardware executing a thread
- Load balancing/scheduling
  - The mapping of threads to cores
- Race condition
  - The outcomes varies as the scheduling of threads varies
- Deadlock
  - A cycle of threads that are waiting on each other
- Critical section
  - Part of task that access a common resource

Flynns Taxonomy of Computer Systems

- Single Instruction, Single Data (SISD)
  - A sequential computer
  - Example: Mobile phones, low-end laptops
- Single Instruction, Multiple Data (SIMD)
  - A single instruction applied to multiple data streams
  - Example: Vector unit of CPUs, some GPUs
- Multiple Instruction, Single Data (MISD)
  - Multiple instructions on a single data stream.
  - Example: Fault tolerant systems (space shuttle)
- Multiple Instruction, Multiple Data (MIMD)
  - Multiple processors simultaneously executing different instructions on different data
  - Example: Multi-Core CPUs, clusters, some GPUs

Most TOP500 supercomputers are based on MIMD

Limits to performance

Amdahls Law

- Presented by Gene Amdahl in 1967
  - Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities
- Find maximum expected improvement performance
- Overly pessimistic in practice
- Contradicted by Gustafsons Law

- Result: Theoretical speedup is limited by serial part of code

Amdahls Law - Equations

- Total running time of serial program is given by:

$$T_{total}(1) = T_{setup} + T_{compute}(1) + T_{final}$$

- Using *P* processors we get:

$$T_{total}(P) = T_{setup} + \frac{T_{compute}(1)}{P} + T_{final}$$

- The relative speedup is:

$$S(P) = \frac{T_{total}(1)}{T_{total}(P)}$$

Amdahls Law - Equations II

$$S(P) = \frac{T_{total}(1)}{T_{total}(P)}$$

- The serial fraction is:

$$\gamma = \frac{T_{setup} + T_{final}}{T_{total}(1)}, 0 \le \gamma \le 1$$

- Inserting this into *S(P)* give **Amdahls Law**:

$$S(P) = \frac{T_{total}(1)}{(\gamma + \frac{1-\gamma}{P})T_{total}(1)}$$

$$= \frac{1}{\gamma + \frac{1-\gamma}{P}}$$

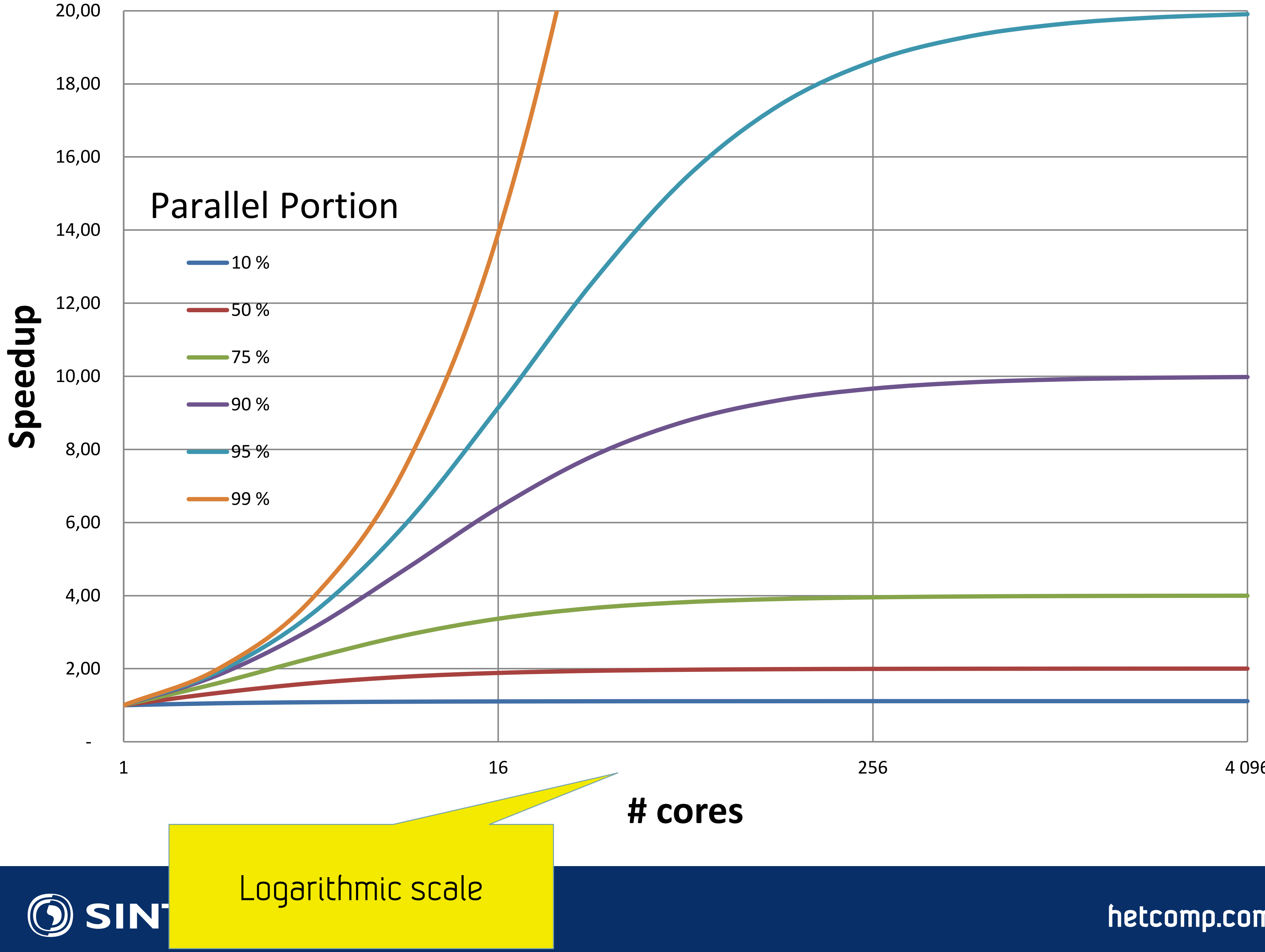
Amdahls Law - Equations III

- Assume an infinite number of processors

$$\lim_{P\to\infty} S(P) = \lim_{P\to\infty} \frac{1}{\gamma + \frac{1-\gamma}{P}} = \gamma^{-1}$$

- The maximum performance increase is bound by the serial fraction

Plot of Amdahls law



Gustafson Law

- Amdahls law does not incorporate increased problem size
- We are interested in solving the largest possible problem in reasonable time

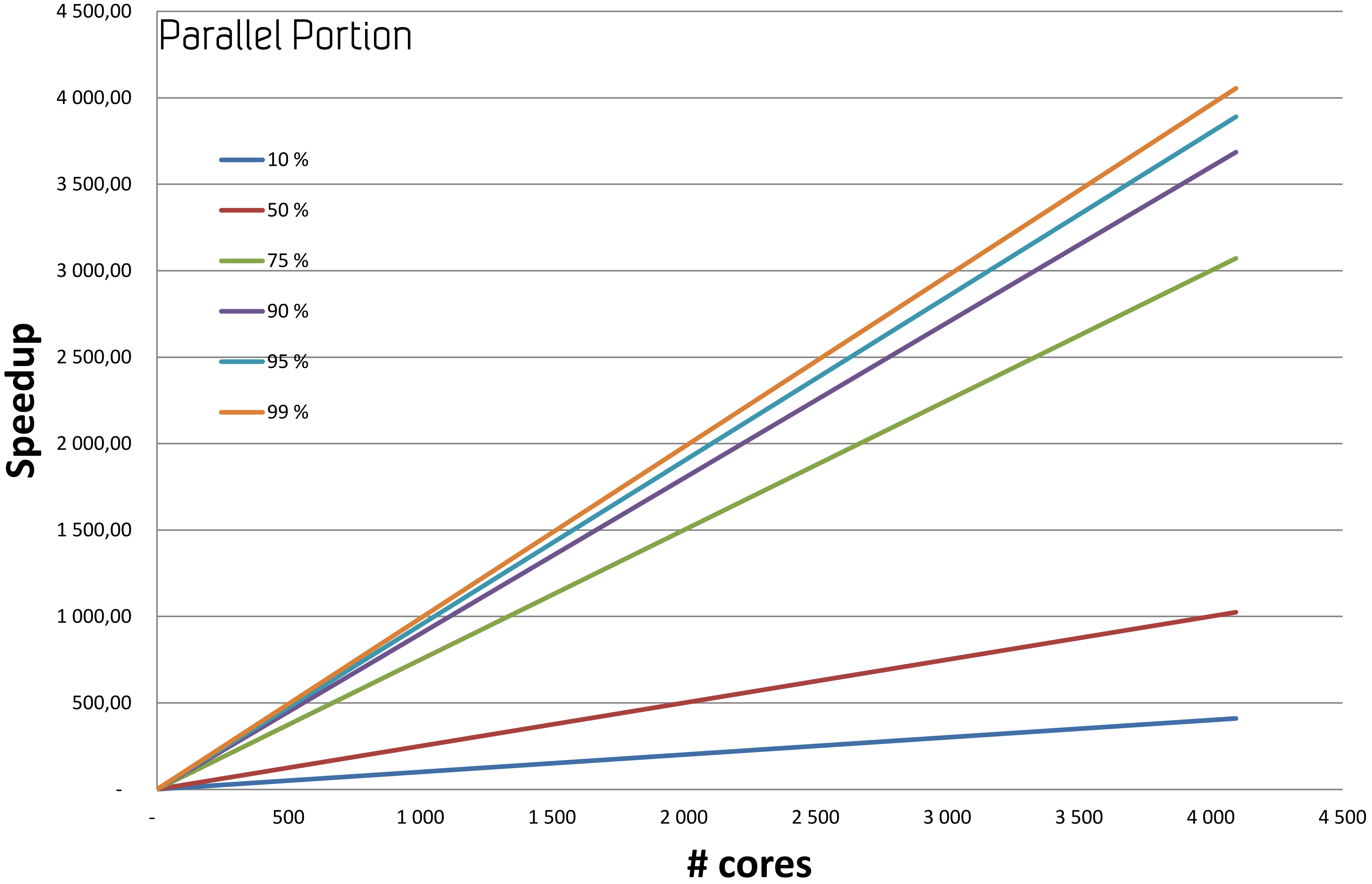
$$T_{total}(1) = T_{setup} + P \cdot T_{compute}(P) + T_{final}$$

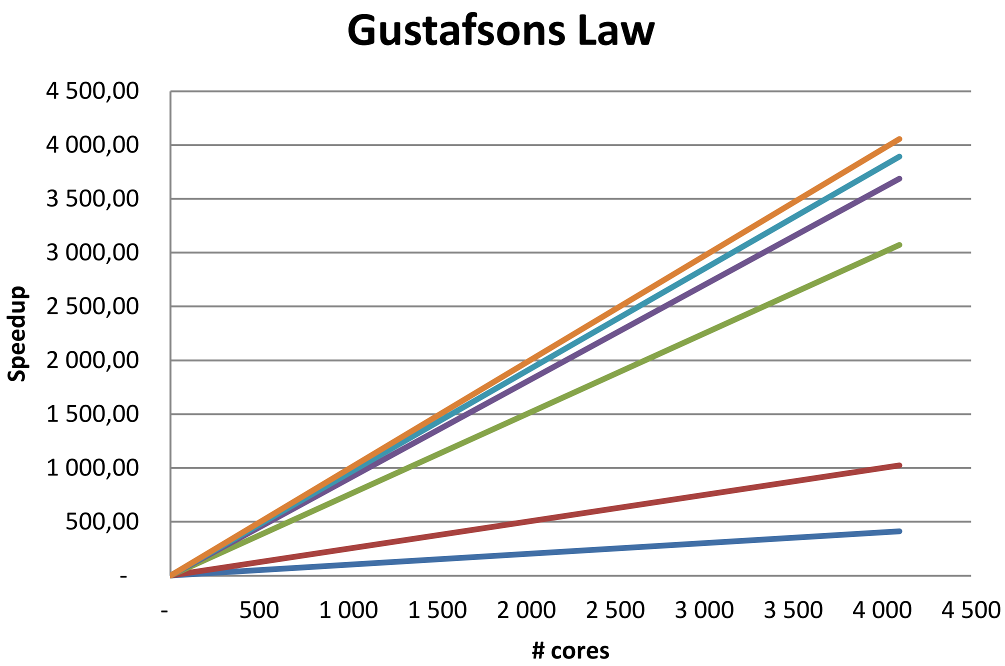
$$\gamma_{scaled} = \frac{T_{setup} + T_{final}}{T_{total}(P)}$$

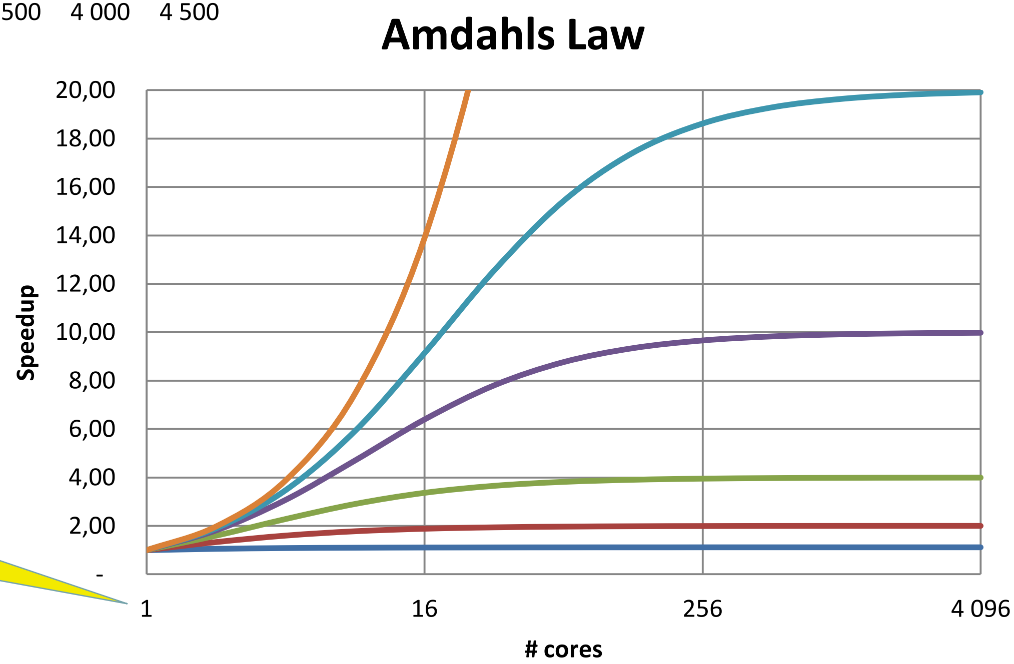
$$S(P) = \gamma_{scaled} + P(1 - \gamma_{scaled})$$

- Assume γ is **independent of** P
  - S is then **linear in P**

Plot of Gustafson law







Logarithmic scale

Dicussion

- Experience shows that Amdahls law is overly pessimistic
  - But you will always have some serial parts
- Many real world scenarios demonstrate almost linear speedup
- Some cases see superlinear speedup!
- Both models are simplified
  - Parallelism also introduces overhead

- Don't forget Donald Knuth: *Premature Optimization is the root of all evil*
- Is the potential speedup worth the extra effort?
  - Up front and maintenance wide?

Concurrency

Concurrency

Definition of CONCURRENCE

a : the simultaneous occurrence of events or circumstances

b : the meeting of concurrent lines in a point

Definition of CONCURRENT

a : operating or occurring at the same time

b : running parallel

from *Merriam Webster*

Concurrency

- Concurrency can be found at many levels
- Concurrency exists in two forms:
  1. Data parallel
  2. Task parallel
- Not mutually exclusive
  - A complex program will have both
  - The line between them is blurred

Data parallelism

- The same task is executed as many threads on different pieces of the data
- Examples:
  - Rendering
  - (Dense) linear algebra
  - FFTs
  - Max/Min computations

  - Web servers
  - Databases

Task parallelism

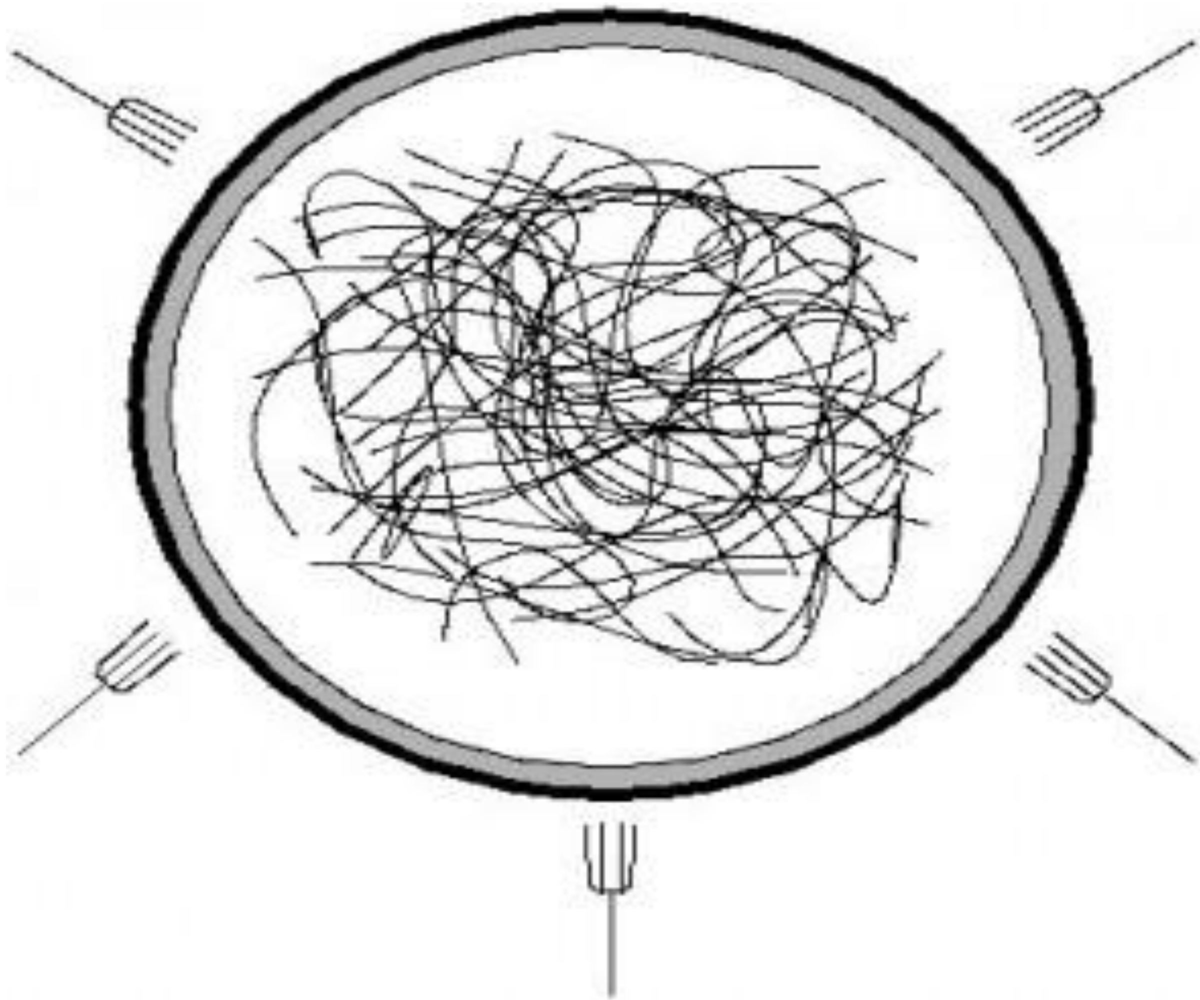
- Different, independent tasks
- Linked by sharing data
- Examples:
  - GUI code
  - Logging
  - Loading data
  - Writing data
  - Networking
  - Monitoring data?

- Hard to find enough tasks to scale to 10++ cores

Discussion

- Concurrency should be identified early
- #Cores on target hardware should be known before choosing algorithm
- Good serial algorithms seldom make good parallel ones

Synchronization



Synchronization

- Most parallel programs require tasks to communicate
- Synchronization must be explicitly handled
- Difficult to enforce automatically
  - Threads are assumed to follow an agreed upon protocol
- Synchronization is expensive
  - Slows down the program
- Common source of bugs
  - Hard to find
  - Hard to reproduce

Illustrating the problem

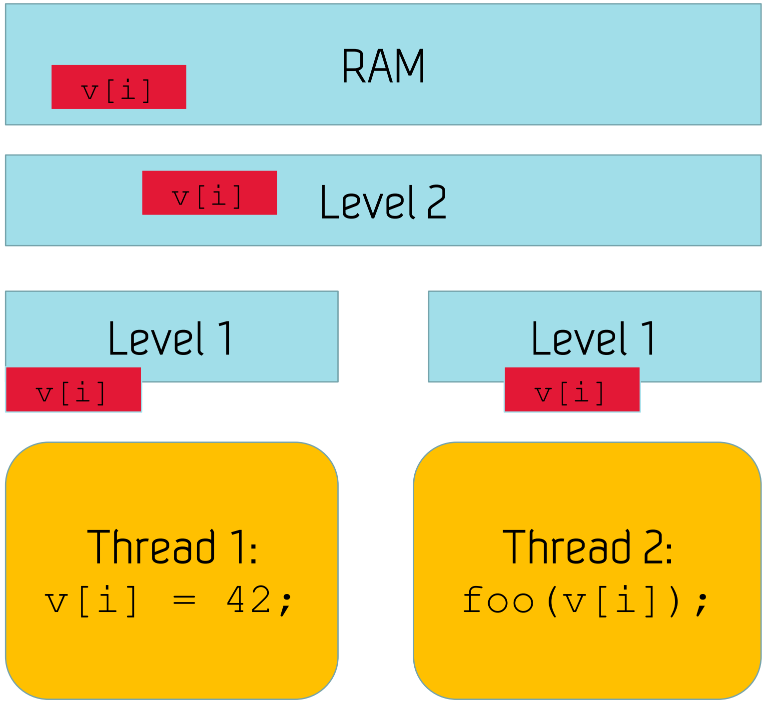
```
int getNextId() {
   static int lastIdUsed = 0;
   return ++lastIdUsed;
}
```

- Assume two threads call getNextId()

Thread one	Thread two	lastIdUsed
43	44	44
44	43	44
43	43	43

Memory fences

- Modern CPUs have complex cache hierarchies
  - Typically three levels deep
- A **fence** ensures that all threads have a consistent view of memory
- Typically invoked by higher level primitives
- Only meaningful in a shared memory setting



Barriers

- Synchronization point:
  - Every thread must arrive before continuing
- Typically used at the end of an iteration
- Explicitly written by the programmer
- Useful in cluster and shared memory processing

Mutex

- Mutex = Mutual Exclusion
- Protects against the simultaneous use of a common resource
  - Example: global variable, network card, write to file
- The mutex is a lock that protects the resource

- Threads must acquire the mutex before entering a critical section
- If the mutex is busy the thread must wait (spin on the lock)
- Remember to release the mutex!

- Coding a mutex is not trivial – use libraries (which generally use HW)

OpenMP Lock Example

```
int main() {
   omp_lock_t lock;
   omp_init_lock( &lock );

    #pragma omp parallel shared (lock)
   {   // non-critical section

       omp_set_lock( &lock );
       // critical section…
       omp_unset_lock( &lock );

       // non-critical section
   }
}
```

Declare and init lock

Wait or Aquire lock

Release lock

Semaphores

- Controls access to common resources (note the plural form)
- Records how many units of a resource is available (counting semaphore)
  - Hands out a permit to the resource

- Example:
  - A library with ten study rooms and ten keys
  - A librarian (semaphore) hands out keys to the rooms
  - Students (threads) must wait if there are no free keys

- A mutex can be seen as a binary-semaphore
  - A mutex has the concept of a "owner"

Monitors

- An object designed to be safely used by several threads
- Often implemented using mutex/semaphores
  - Java, C# has language support
- Monitors often have mechanism for signaling callers when they are "ready"

- Mutex/Semaphore: caller is responsible
- Monitor: callee is responsible

Monitor example

Release lock before returning

```
class A {
private:
   Lock l;
   int lastIdUsed;
public:
   int getNextId() {
        l.aquire();
        int id = ++lastIdUsed;f
        l.release();
        return id;
   }
}
```

Client code does need not worry about locking

```
int main() {
   A a;
   int id = a.getNextId();
}
```

Debugging and testing parallel programs

- Notoriously hard
- Ensure unicore algorithm is correct
- Parameterize the "test space" – make it as big as possible
  - Vary number of cores
  - Vary hardware platform
  - Vary compiler settings
  - Vary input data
  - Run tests in different order
  - Run automatically

Conclusion

- Synchronization protocol must be agreed upon
- Common source of bugs

Reading list