Multicore Programming

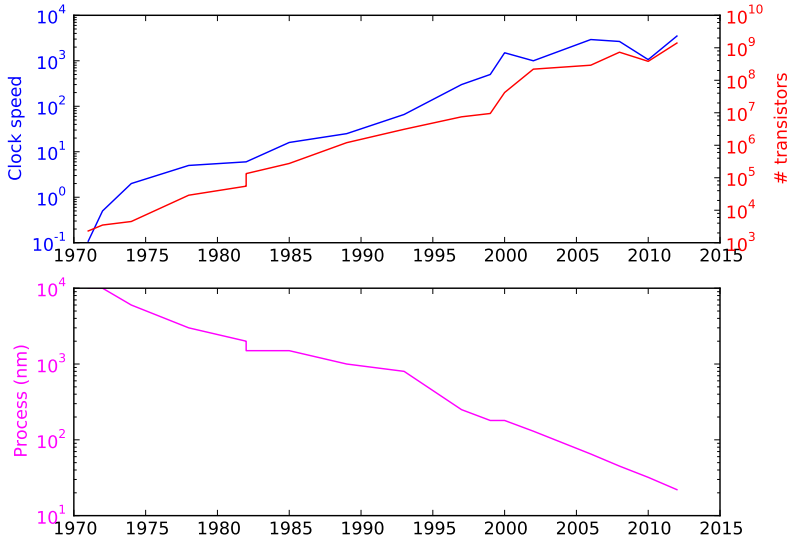
Frédéric Goualard

Laboratoire d'Informatique de Nantes-Atlantique, UMR CNRS 6241
Office #208

Organization

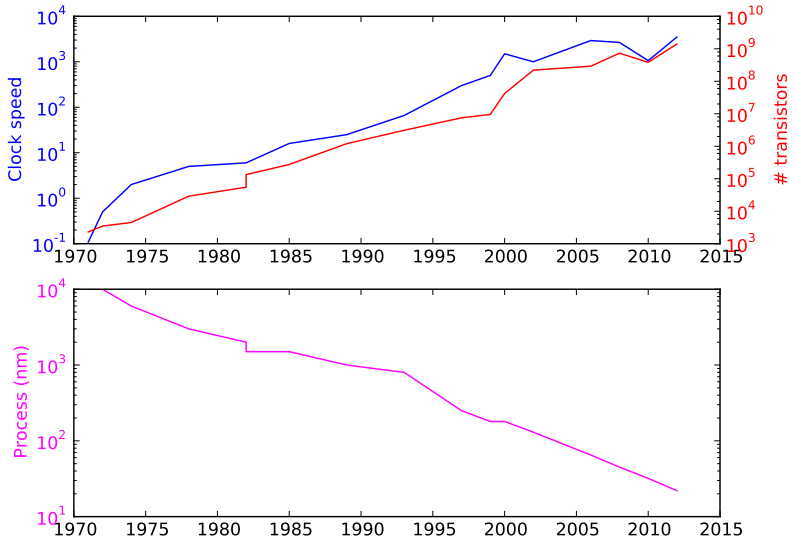
- 11 lectures/tutorials
- 4 lab. work slots
- 2 exams (one on a tutorial slot, one final)
- 1 lab. assignment (2 slots 1:20)
- Interleaving with "*Constraint Programming*" course: every other week

Motivations



Source: Intel Corp.

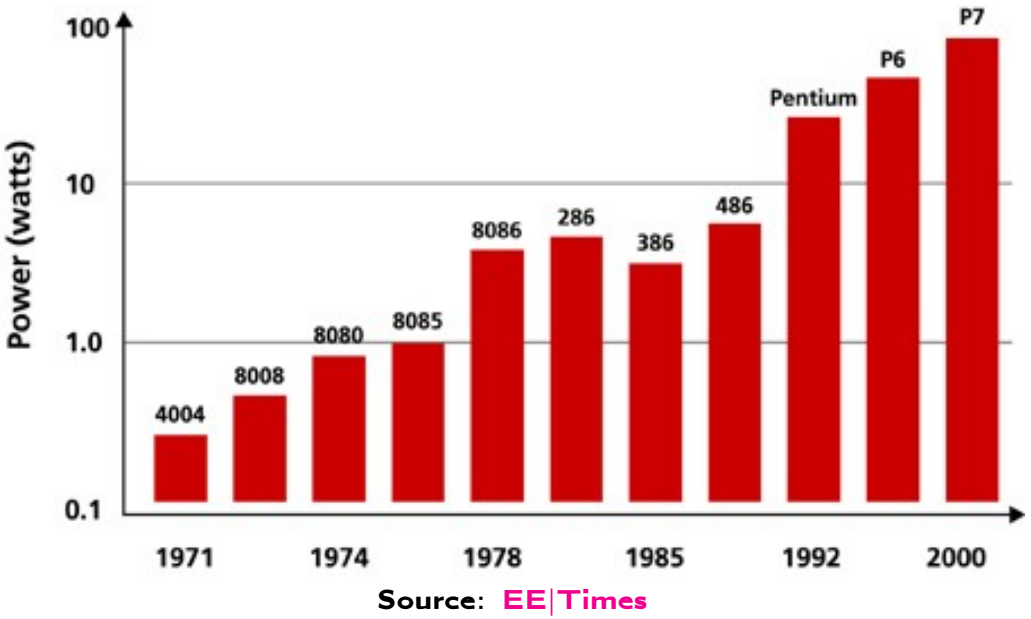
Motivations



Source: Intel Corp.

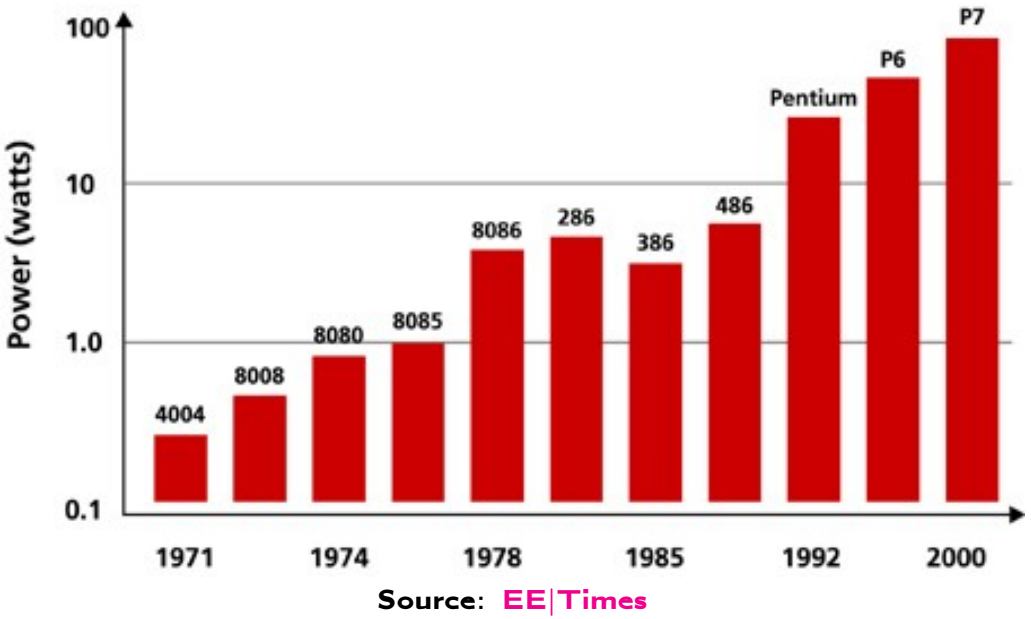
- 1986–2002: performance of single processor computer increases 50% per year ($\times$58 in 10 years)
- 2002–now: performance increases only 20% per year ($\times$6 in 10 years)
- Atom radius: 0.3 nm (reaching limits of Physics)

Motivations



- 1986–2002: performance of single processor computer increases 50% per year ($\times$58 in 10 years)
- 2002–now: performance increases only 20% per year ($\times$6 in 10 years)
- Atom radius: 0.3 nm (reaching limits of Physics)
- Power dissipation : more than 200W in current processors

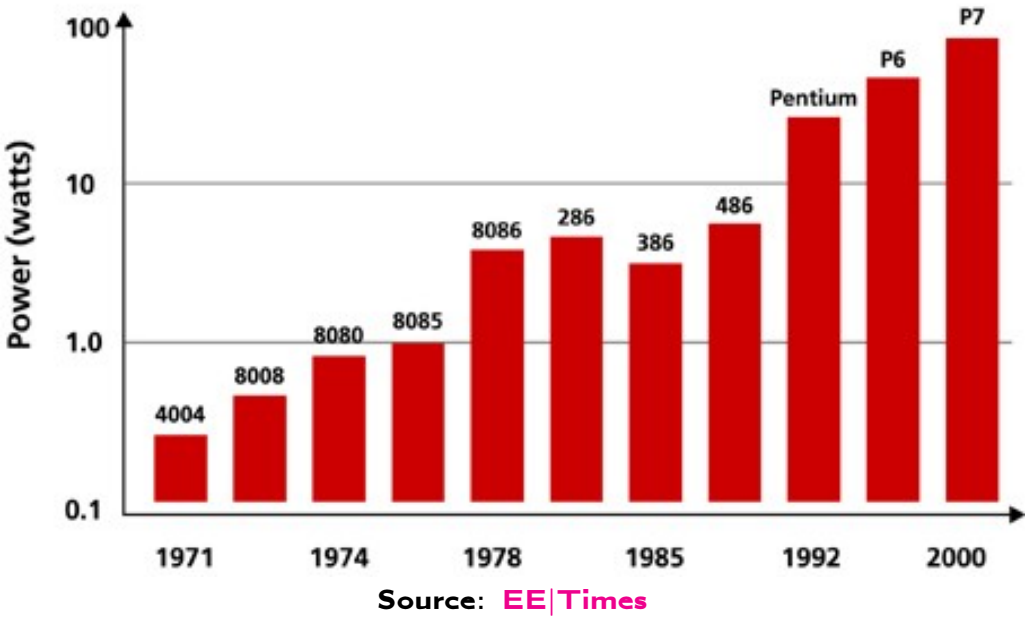
Motivations



- 1986–2002: performance of single processor computer increases 50% per year ($\times$58 in 10 years)
- 2002–now: performance increases only 20% per year ($\times$6 in 10 years)
- Atom radius: 0.3 nm (reaching limits of Physics)
- Power dissipation : more than 200W in current processors

End of the line for mono-processors

Motivations



- 1986–2002: performance of single processor computer increases 50% per year ($\times$58 in 10 years)
- 2002–now: performance increases only 20% per year ($\times$6 in 10 years)
- Atom radius: 0.3 nm (reaching limits of Physics)
- Power dissipation : more than 200W in current processors

End of the line for mono-processors

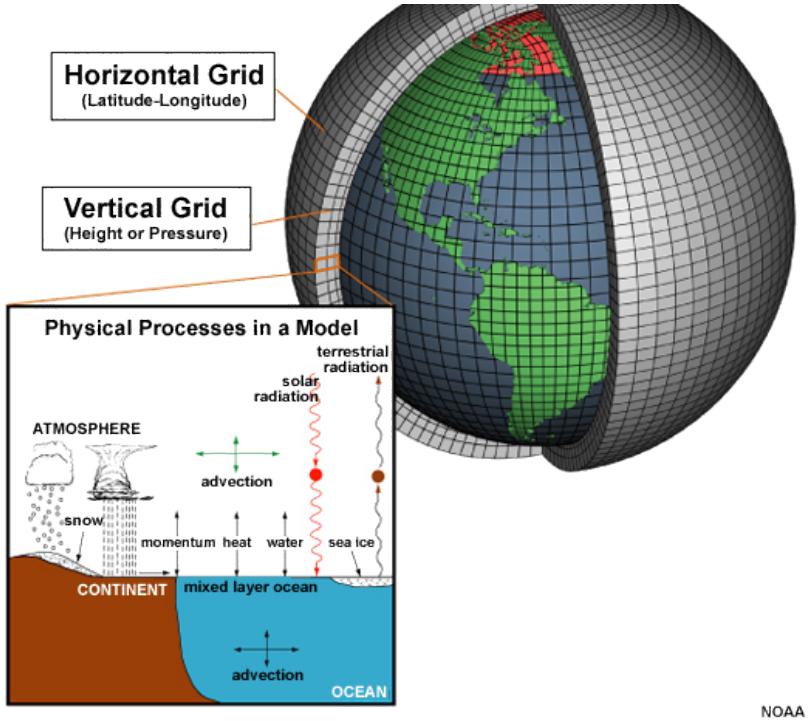
Rise of multicore processors and parallelism

Why parallelism?

Speedup of 20 % may look good enough...

...but not for all applications:

- Data mining
- Earthquake simulation
- Global climate modeling
- Computational chemistry
- Bioinformatics
- Cryptography
- ...



NOAA

Transition to parallelism

- Automatic parallelization of sequential programs

Transition to parallelism

- Automatic parallelization of sequential programs
  - Many tried... all failed
  - Applicable only to simple cases (e.g., replacing FPU instructions by SSE ones)

Transition to parallelism

- Automatic parallelization of sequential programs
  - Many tried... all failed
  - Applicable only to simple cases (e.g., replacing FPU instructions by SSE ones)
- Many different kinds of "parallelism"
  - *task parallelism* or *data parallelism*

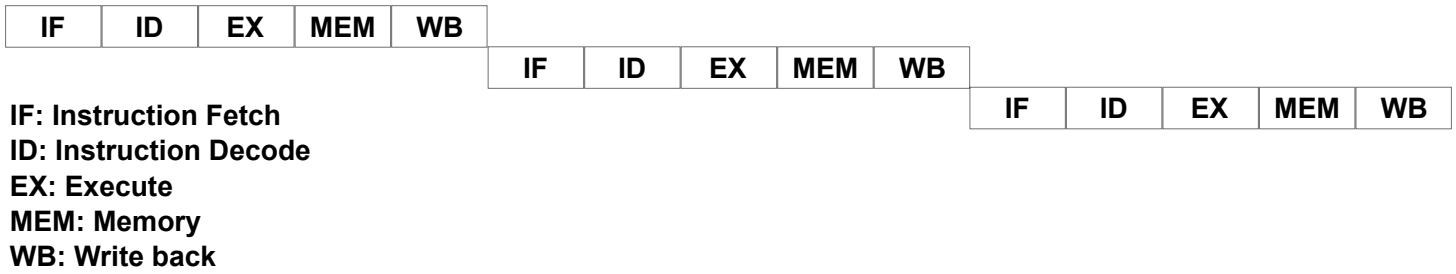
Transition to parallelism

- Automatic parallelization of sequential programs
  - Many tried... all failed
  - Applicable only to simple cases (e.g., replacing FPU instructions by SSE ones)
- Many different kinds of "parallelism"
  - *task parallelism* or *data parallelism*
- New problems arise

Transition to parallelism

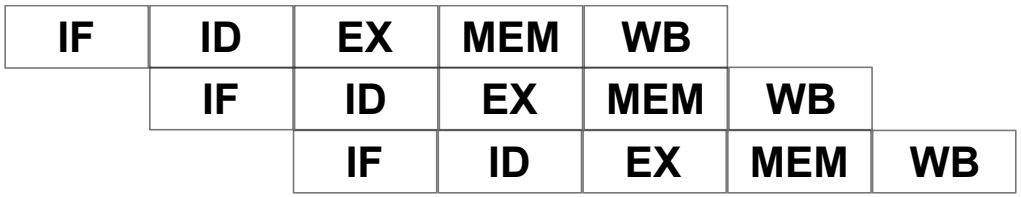
- Automatic parallelization of sequential programs
  - Many tried... all failed
  - Applicable only to simple cases (e.g., replacing FPU instructions by SSE ones)
- Many different kinds of "parallelism"
  - *task parallelism* or *data parallelism*
- New problems arise
  - Communication/coordination between *threads*/processors
  - Synchronization
  - Barriers to successful parallelization (see *Amdahl's law*)
  - Ever changing trends in devising parallel machines: the hot parallel algorithm of today may not be usable on tomorrow's best parallel computer
  - ...

Parallelism and Parallelism again



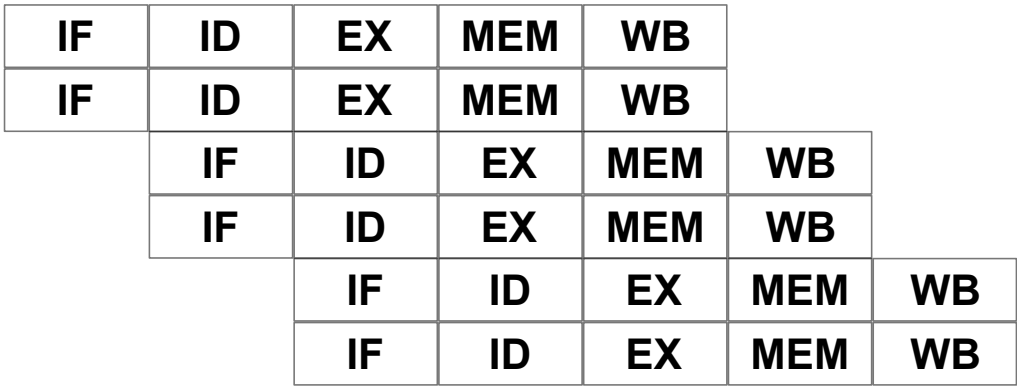
Strictly sequential uniprocessor

Parallelism and Parallelism again



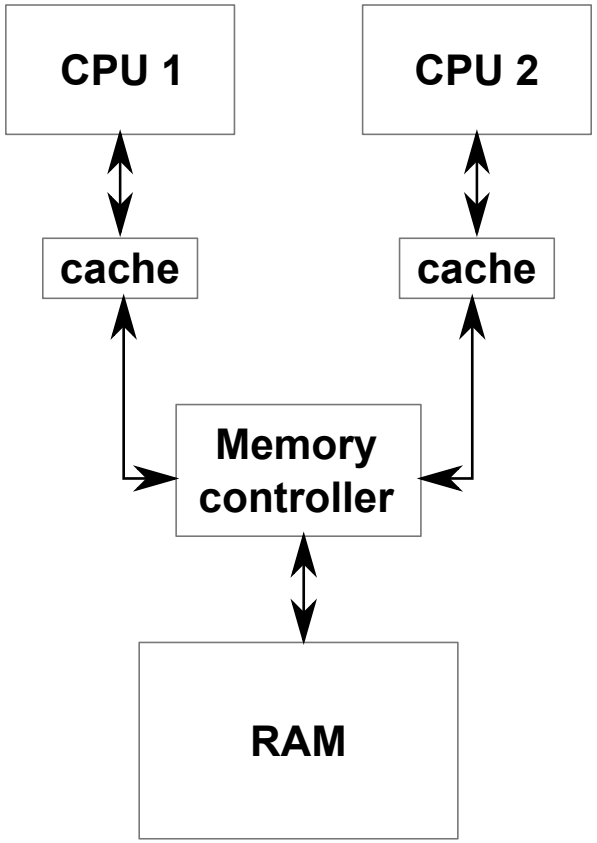
Pipelined uniprocessor (e.g. Intel i486)

Parallelism and Parallelism again



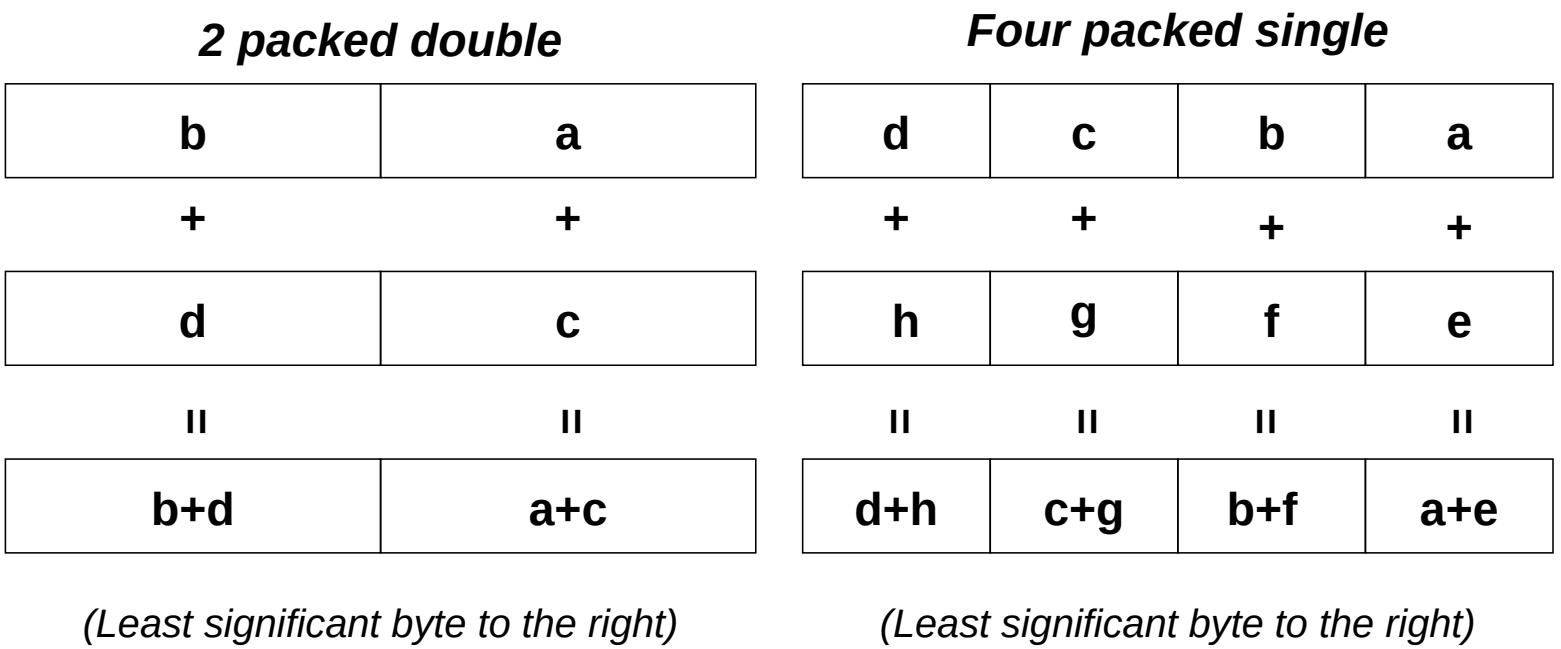
Superscalar processor (e.g. Intel Pentium).
Instruction level parallelism with redundant functional units.

Parallelism and Parallelism again



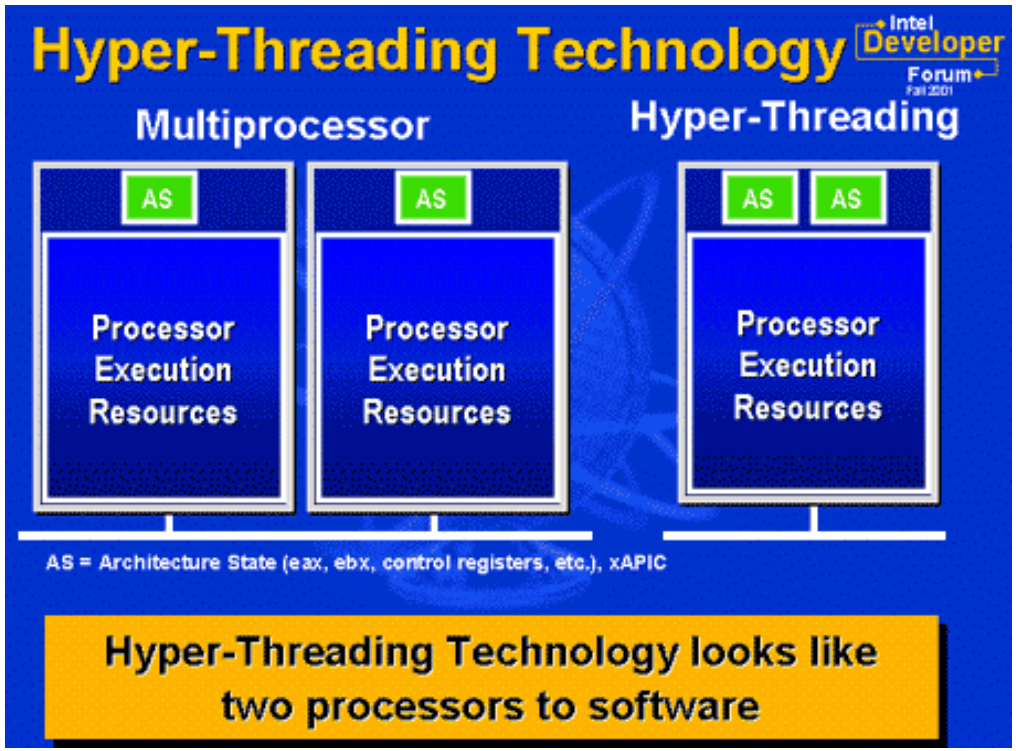
Symmetric Multiprocessor (SMP) (e.g. Pentium Pro)

Parallelism and Parallelism again



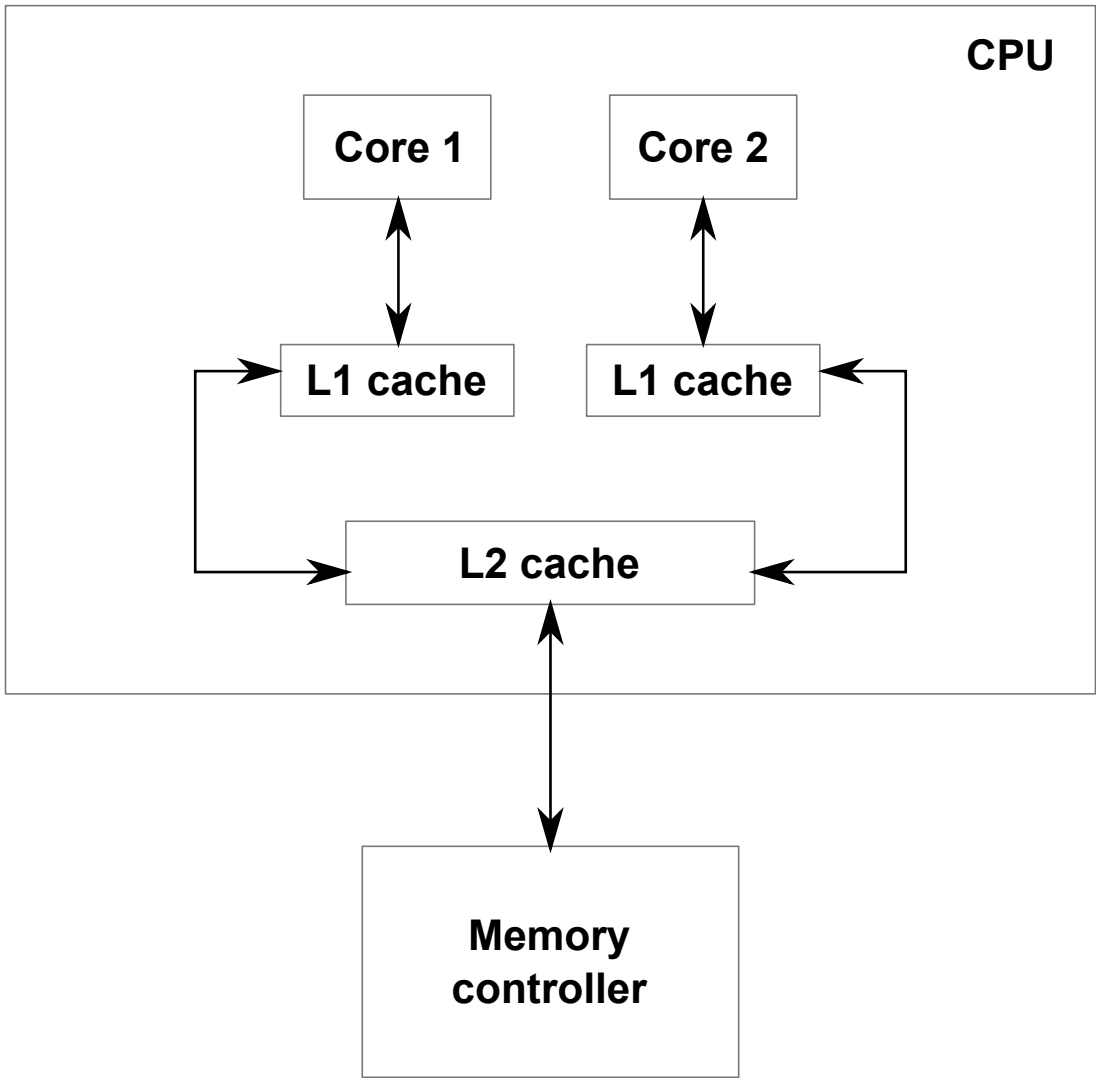
SSE Instructions (SIMD) (e.g., Pentium III)

Parallelism and Parallelism again



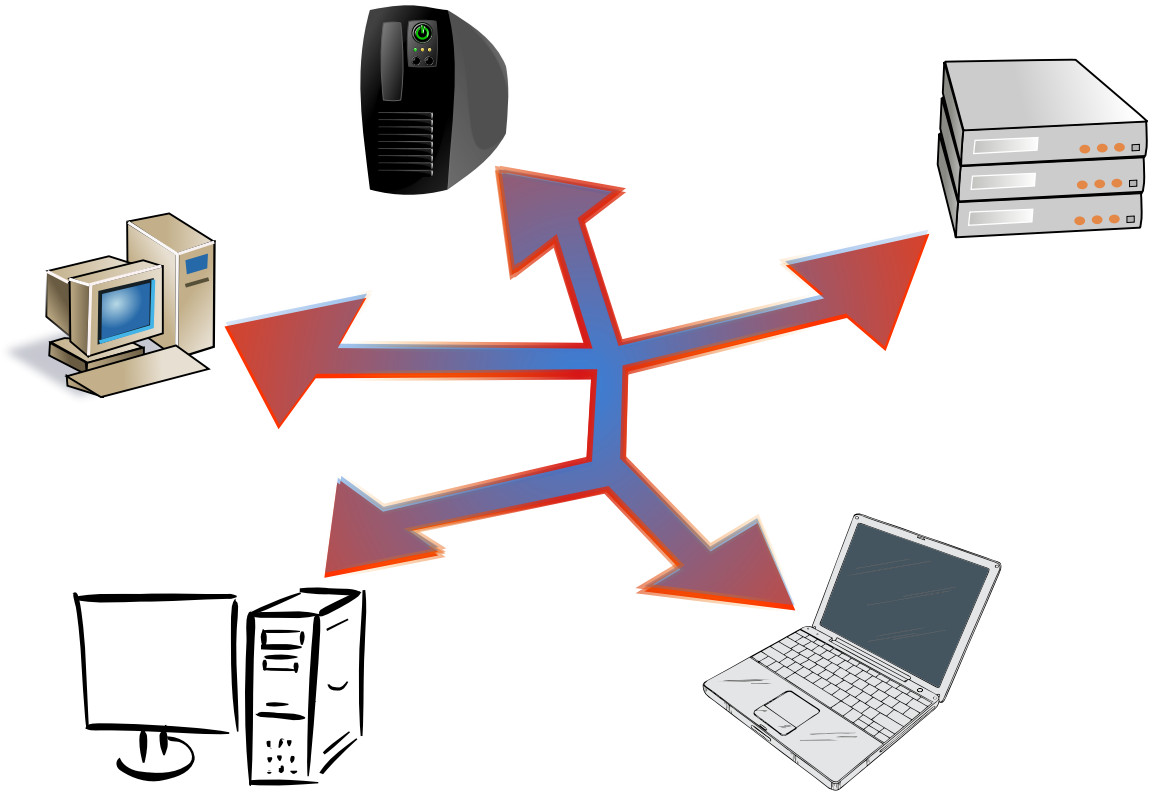
Hyper-threading (e.g., Pentium 4)

Parallelism and Parallelism again



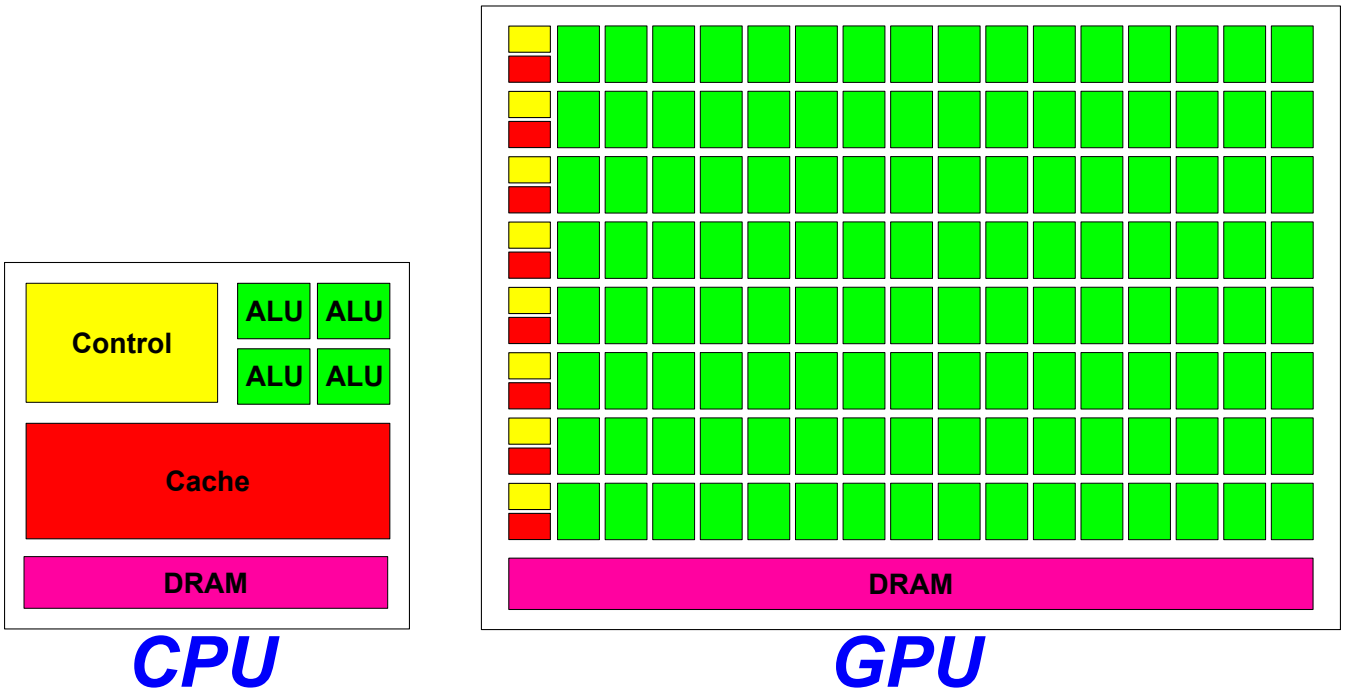
Chip Multiprocessor (CMP) (e.g., Intel Core 2)

Parallelism and Parallelism again



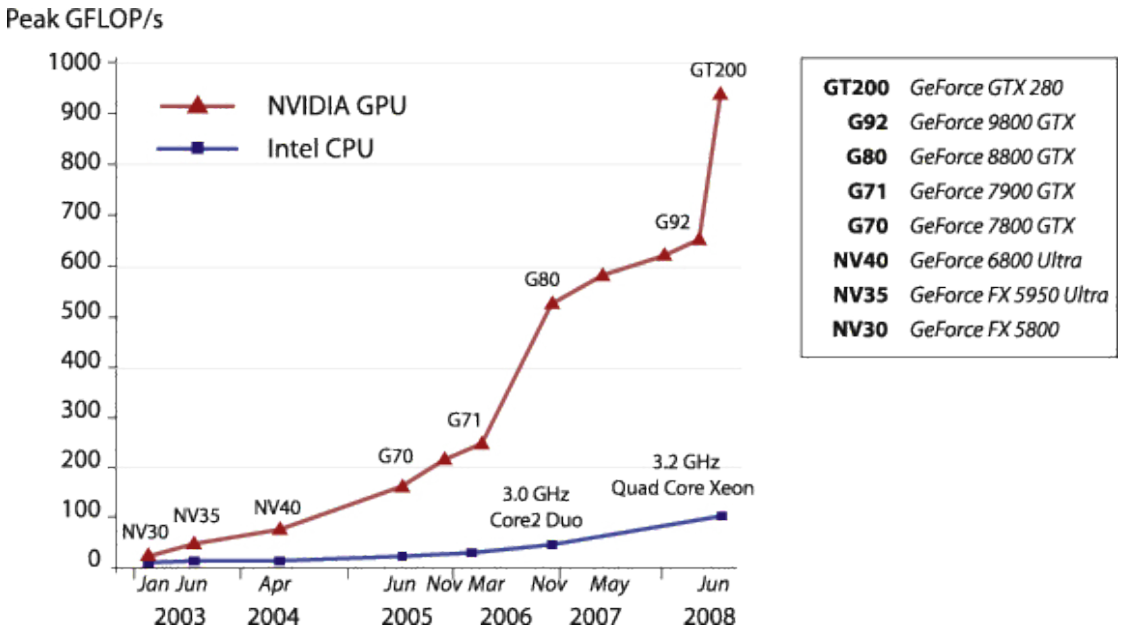
Homogeneous (e.g., Beowulf cluster) or heterogeneous (e.g. SETI@home) distributed computing

Parallelism and Parallelism again



GPGPU (massive data parallelism on graphics cards)

Parallelism and Parallelism again

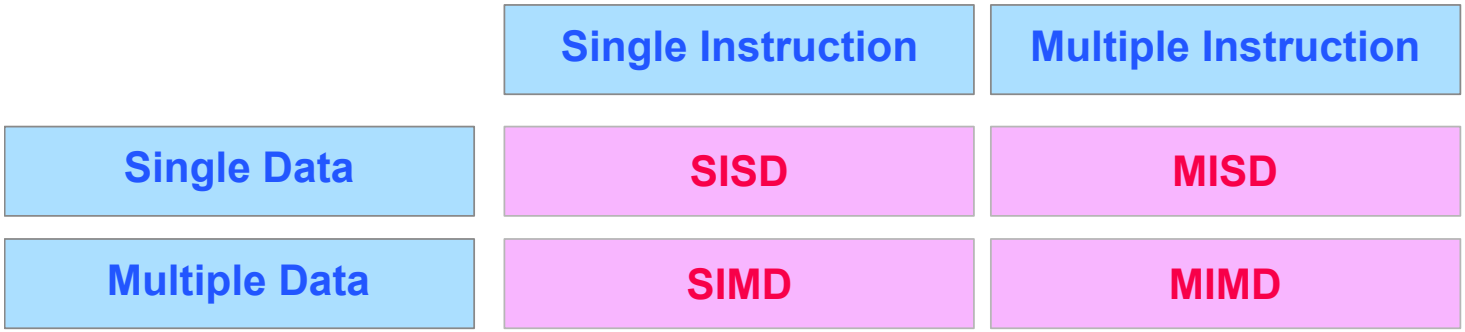


GPGPU (massive data parallelism on graphics cards)

GPU vs. CPU

- For equal performances, GPUs:
  - are nine times less bulky
  - require seven times less power
  - cost six times less
- Counterstrike:
  - Manycore processors (Intel *Many Integrated Core Architecture* Knights Corner, 50 cores)
  - Hybrid processors GPU + CPU (Intel Sandy Bridge)

Flynn's taxonomy



SISD Single core running only one thread of instructions

SIMD One thread of instructions executes on several data simultaneously (e.g., Intel SSE)

MISD Fault tolerant computer

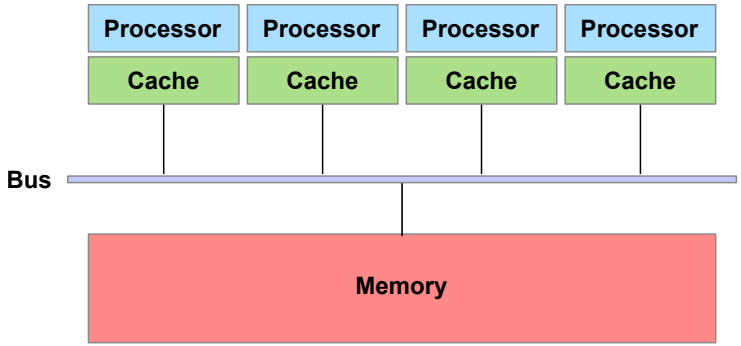
MIMD Distributed computing

Taxonomy of architectures

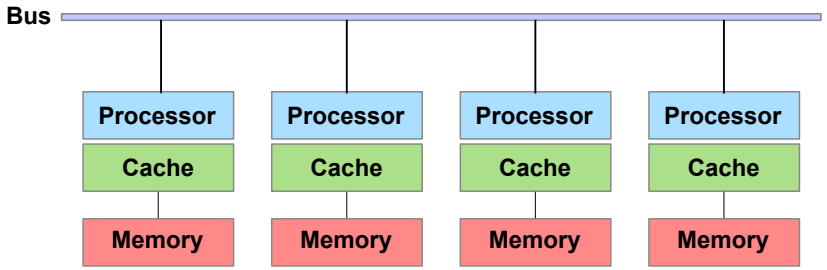
- Shared memory architecture
- Distributed memory architecture
- Mixed architecture (shared/distributed)
- Hybrid architecture (CPU/GPU)

Shared memory

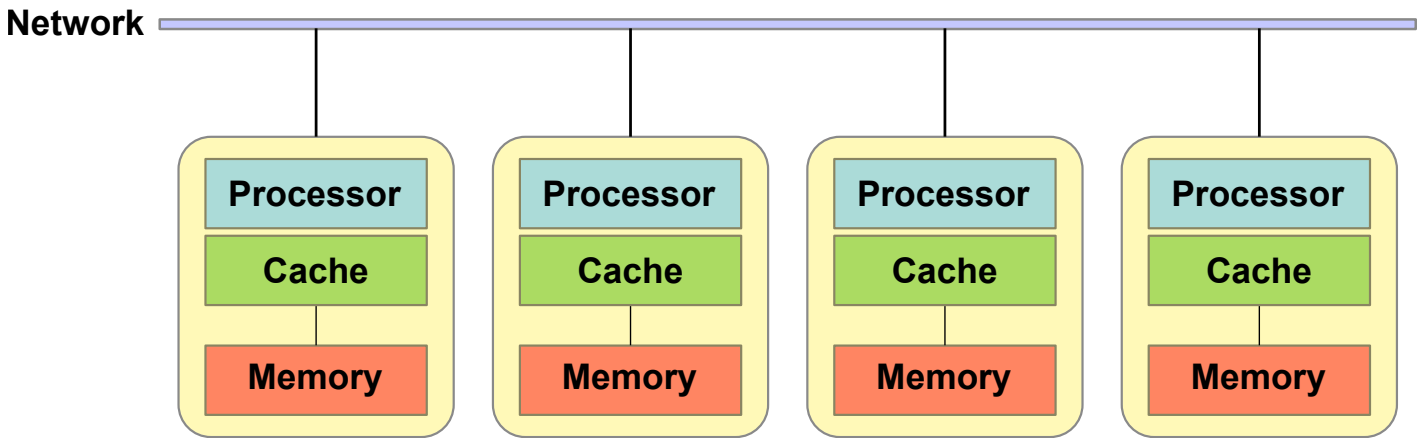
- *Uniform Memory Access* (UMA) architecture
  - One central memory
  - Access time is the same for all processors



- *Non Uniform Memory Access* (NUMA) architecture
  - One memory per processor
  - All processors can access all the available memory
  - Access time depends on location



Distributed memory



- Each processor has its own memory
- Processors are connected in a network
- Access to non local memory through explicit exchange of messages
- Overall performances depend on network performances

Outline

1. Parallel programming: introduction and theoretical framework
2. Multithreading (C++11 Standard Thread Library/Intel Threading Building Blocks)
3. Shared memory multiprocessing (OpenMP)
4. Distributed memory multiprocessing (MPI)
5. GPGPU: Massively parallel programming (OpenCL and CUDA)

Content may be adjusted/modified at all times

Introduction to parallelism

Metrics

Clock frequency F. Determines duration of a cycle

- Computational power $\propto F$
- power dissipation $\propto F^3$

FLOPC. *# FLoating-point Operations per Cycle*

FLOPS. *# FLoating-point Operations per Second*

Peak performance R_{peak}.

$$R_{\text{peak}} = \text{FLOPC} \times F \times N_c$$

with N_c the number of cores

Putting things in perspective

- Desktop computer: 10–100 gigaFLOPS
- GPU: 0.1–1 teraFLOPS
- Supercalculator: 10 teraFLOPS
- TOP500 TITAN: 20 petaFLOPS
- Tianhe-2: 100 petaFLOPS by 2015 (?)

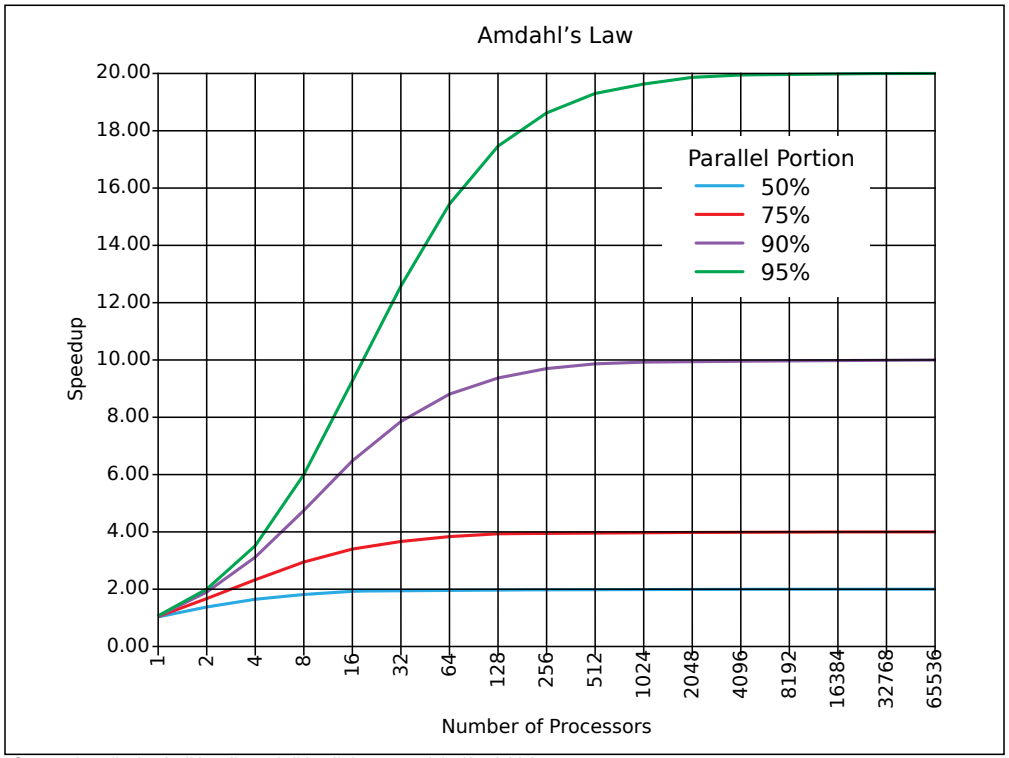
kiloFLOPS	10^3
megaFLOPS	10^6
gigaFLOPS	10^9
teraFLOPS	10^{12}
petaFLOPS	10^{15}
exaFLOPS	10^{18}

Amdahl's law

- P: percentage of program that can be parallelized
- n: number of processors/cores

Speedup:

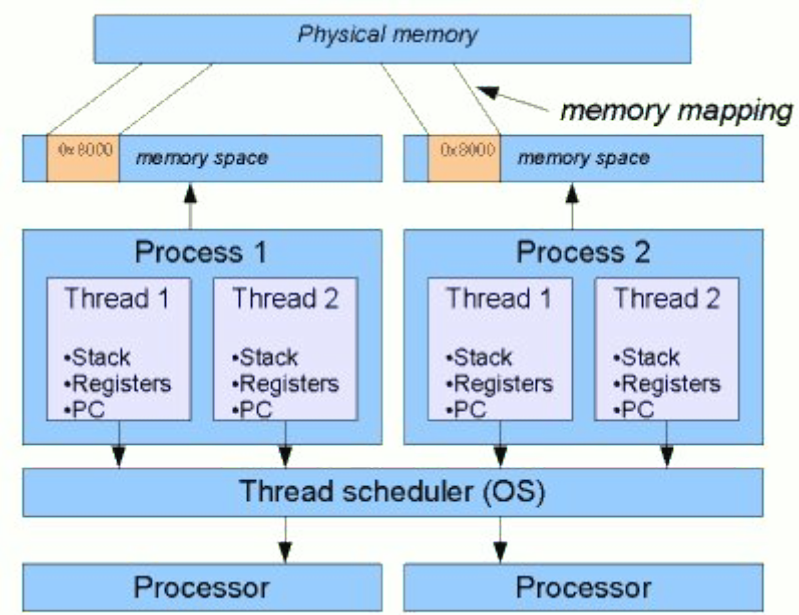
$$S(n) = \frac{1}{(1-p) + \frac{p}{n}}$$



Source: http://upload.wikimedia.org/wikipedia/commons/e/ea/AmdahlsLaw.svg

Shared memory parallel programming

- Programming through *threads* (light processes)
  - *Process*: Unit of execution recognized by the OS. Each process has its own memory space
  - *Thread*: Threads of a process share memory space, environment variables, ...



- Threads execute *concurrently*

Multithreading consistency problem

```
Global int i = 0

Thread 1:                          Thread 2:
  i = i + 1                          i = i - 1
```

Value of `i` when both threads return?

Multithreading consistency problem

```
Global int i = 0

Thread 1:                          Thread 2:
  i = i + 1                          i = i - 1
```

Value of `i` when both threads return?

- `i=0`
- `i=1`
- `i=-1`

- Critical section
- Mutual exclusion
- Ensuring mutual exclusion: *semaphores*

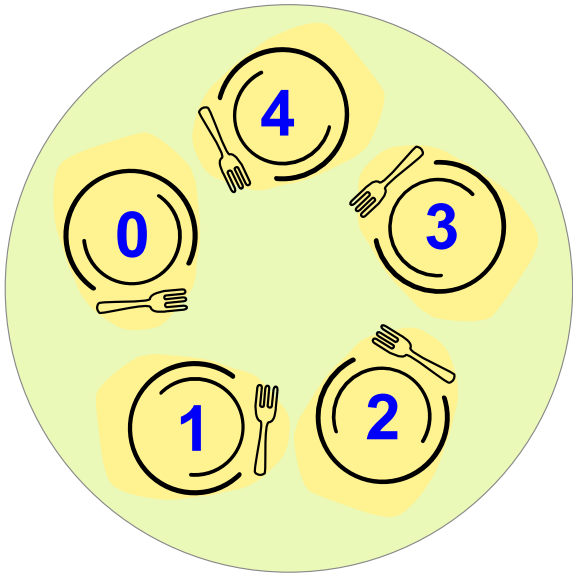
Semaphores

- Invented by E. Dijkstra in 1965
- Object with one integer attribute, one waiting list, and two methods guaranteed to run *atomically*:
  - `P()` (*proeberen* —"test"): wait for attribute to become strictly greater than 0, then decrement it
  - `V()` (*verhogen* —"increment"): increments attribute by 1

Example:

```
Global int i = 0
semaphore mutex(1)

Thread 1:
  P(mutex)
  i = i + 1
  V(mutex)

Thread 2:
  P(mutex)
  i = i - 1
  V(mutex)
```

Example: the dining philosophers



Five philosophers around a table:

- They think
- They eat spaghetti

Constraints:

- A fork can be held by only one philosopher at a time
- No *deadlock* shall occur
- No philosopher shall starve waiting for a fork
- More than one philosopher must be allowed to eat at the same time

Write a concurrent program that allows each philosopher to think and eat

Dining philosophers

```
def philosopher(i):
    while True:
        think()
        get_forks(i)
        eat()
        put_forks(i)

def left(i): return i
def right(i): return (i+1) % 5

forks = [Semaphore(1) for i in range(5)]

def get_forks(i):
    P(forks[right(i)])
    P(forks[left(i)])
def put_down_forks(i):
    V(forks[right(i)])
    V(forks[left(i)])
```

Dining philosophers

```
def philosopher(i):
    while True:
        think()
        get_forks(i)
        eat()
        put_forks(i)

def left(i): return i
def right(i): return (i+1) % 5

forks = [Semaphore(1) for i in range(5)]

def get_forks(i):
    P(forks[right(i)])
    P(forks[left(i)])
def put_down_forks(i):
    V(forks[right(i)])
    V(forks[left(i)])
```

WRONG! Why?

Dining philosophers

```
def philosopher(i):
    while True:
        think()
        get_forks(i)
        eat()
        put_forks(i)

def left(i): return i
def right(i): return (i+1) % 5

forks = [Semaphore(1) for i in range(5)]

def get_forks(i):
    P(forks[right(i)])
    P(forks[left(i)])
def put_down_forks(i):
    V(forks[right(i)])
    V(forks[left(i)])
```

WRONG! Why? Deadlock.

Dining philosophers

```
state = ['thinking'] * 5
sem = [Semaphore(0) for i in range(5)]
mutex = Semaphore(1)

def get_fork(i):
  P(mutex)
  state[i] = 'hungry'
  test(i)
  V(mutex)
  P(sem[i])

def put_down_fork(i):
  P(mutex)
  state[i] = 'thinking'
  test(right(i))
  test(left(i))
  V(mutex)

def test(i):
  if   state[i] == 'hungry' and
       state (left (i)) != 'eating' and
       state (right (i)) != 'eating':
     state[i] = 'eating'
     V(sem[i])
```

Dining philosophers

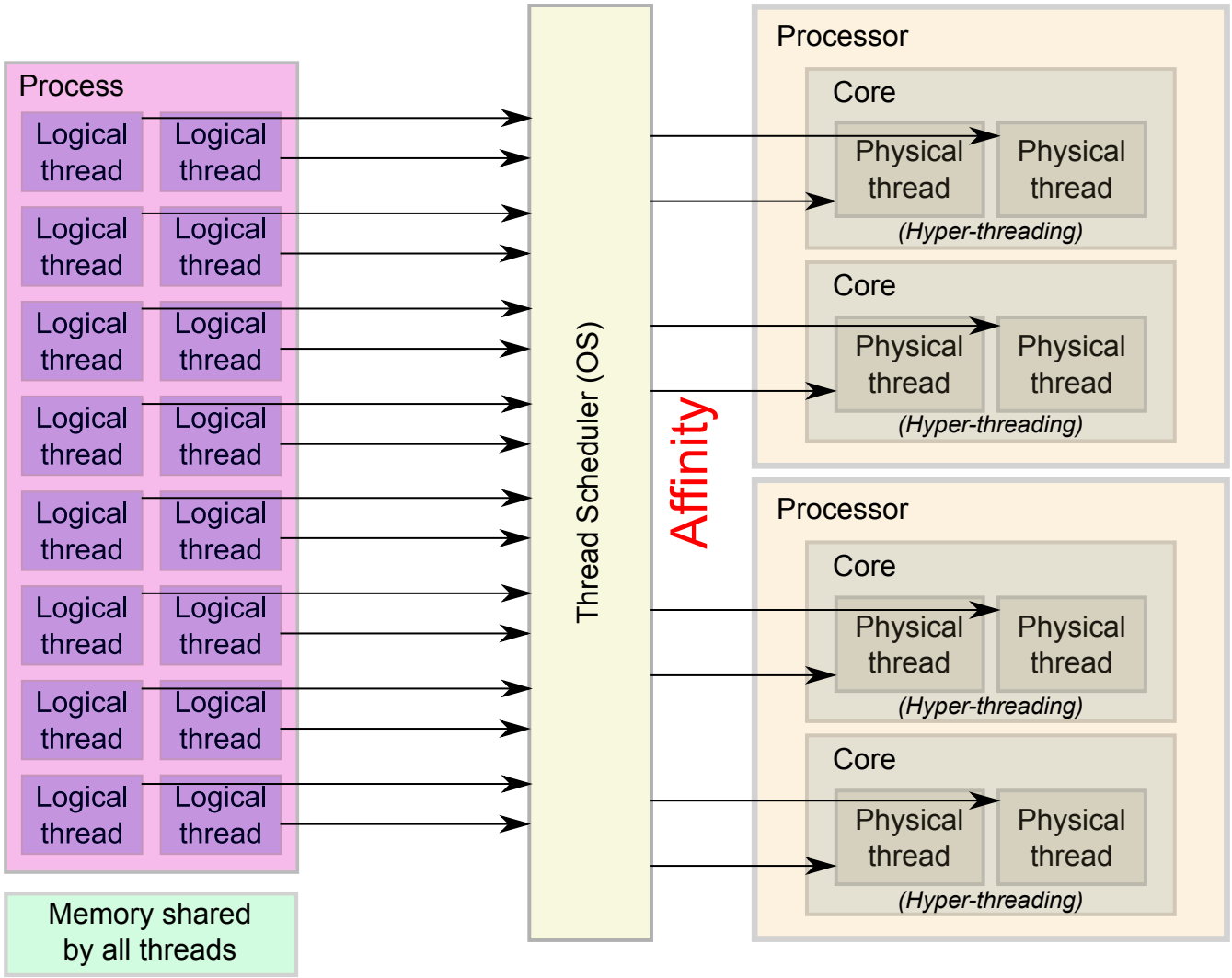
```
state = ['thinking'] * 5
sem = [Semaphore(0) for i in range(5)]
mutex = Semaphore(1)

def get_fork(i):
  P(mutex)
  state[i] = 'hungry'
  test(i)
  V(mutex)
  P(sem[i])

def put_down_fork(i):
  P(mutex)
  state[i] = 'thinking'
  test(right(i))
  test(left(i))
  V(mutex)

def test(i):
  if   state[i] == 'hungry' and
       state (left (i)) != 'eating' and
       state (right (i)) != 'eating':
     state[i] = 'eating'
     V(sem[i])
```

No deadlock. Starvation possible

Lehman & Rabin, POPL '81: no deterministic solution satisfying all constraints with same code for all philosophers

Multithreading

Shared memory concurrent programming



Multithreading in C++

Libraries:

- POSIX Threads
- Boost Thread Library
- C++11 Standard Thread Library (inspired from Boost)
- Intel Threading Building Blocks

Bibliography:

- *C++ Concurrency in Action: Practical Multithreading*. Anthony Williams. Manning Publications Co., 2012
- *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. James Reinders. O'Reilly, 2007

Multithreading

The C++11 Standard Thread Library

Do You Know C++?

```
#include <cstdlib>
#include <cmath>
#include <iostream>
#include <vector>

double f(double x)
{
  if (fabs(x) < 1e-10) {
    return 0.0;
  } else {
    return x;
  }
}

int main(void)
{
  std::vector<double> T(10);

  for (int i=0; i<10; ++i) {
    T[i] = f(T[i]);
  }

  return EXIT_SUCCESS;
}
```

Maybe not... the new one

```
#include <cstdlib>
#include <cmath>
#include <iostream>
#include <vector>

double f(double x)
{
  if (fabs(x) < 1e-10) {
    return 0.0;
  } else {
    return x;
  }
}

int main(void)
{
  std::vector<double> T(10);

  for (int i=0; i<10; ++i) {
    T[i] = f(T[i]);
  }

  return EXIT_SUCCESS;
}
```

```
#include <cstdlib>
#include <cmath>
#include <iostream>
#include <vector>

int main(void)
{
  std::vector<double> T(10);

  for (auto& v : T) {
    v = [](double x) {
      if (fabs(x) < 1e-10) {
        return 0.0;
      } else {
        return x;
      }
    }(v);
  }

  return EXIT_SUCCESS;
}
```

Maybe not... the new one

```cpp
#include <cstdlib>
#include <cmath>
#include <iostream>
#include <vector>

double f(double x)
{
  if (fabs(x) < 1e-10) {
    return 0.0;
  } else {
    return x;
  }
}

int main(void)
{
  std::vector<double> T(10);

  for (int i=0; i<10; ++i) {
    T[i] = f(T[i]);
  }

  return EXIT_SUCCESS;
}
```

```cpp
#include <cstdlib>
#include <cmath>
#include <iostream>
#include <vector>

int main(void)
{
  std::vector<double> T(10);

  for (auto& v : T) {
    v = [](double x) {
      if (fabs(x) < 1e-10) {
        return 0.0;
      } else {
        return x;
      }
    }(v);
  }

  return EXIT_SUCCESS;
}
```

- C++11: many new features for a better/cleaner language
- Compile with "`-std=c++11`" or "`-std=c++0x`"
- Update your knowledge: C++11 Draft Standard, What's new?, C++11 at Wikipedia

Why Use Threads?

- Separation of concerns
  - Separating unrelated parts (e.g., GUI vs. kernel of application)
  - Number of logical threads related to application design, not to number of physical threads
- Performances
  - Speeding up computation (e.g., solving linear equations)
  - Number of logical threads must match number of physical threads (true parallelism)

"Hello World!" with C++11 Threads

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

using namespace std;

struct Hello {
  Hello(uint16_t v) : id(v) {}
  void operator()(void) {
    cout << "Hello from Thread "
         << id << endl;
  }
  uint16_t id;
};

int main(void) {
  vector<thread> allthreads;
  for (uint16_t i = 0; i < 10; ++i) {
    allthreads.push_back(thread(Hello(i)));
  }
  for (auto& t : allthreads) {
    t.join();
  }
  cout << "Hello from main" << endl;
}
```

"Hello World!" with C++11 Threads

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

using namespace std;

struct Hello {
  Hello(uint16_t v) : id(v) {}
  void operator()(void) {
    cout << "Hello from Thread "
         << id << endl;
  }
  uint16_t id;
};

int main(void) {
  vector<thread> allthreads;
  for (uint16_t i = 0; i < 10; ++i) {
    allthreads.push_back(thread(Hello(i)));
  }
  for (auto& t : allthreads) {
    t.join();
  }
  cout << "Hello from main" << endl;
}
```

```
Hello from Thread Hello from Thread Hello
from Thread Hello from Thread 4012
Hello from Thread Hello from Thread

Hello from Thread 8
5
6

Hello from Thread 7
Hello from Thread 3
Hello from Thread 9
Hello from main
```

```
g++ -std=c++11 -pthread -o hello hello.cpp
```

"Hello World!" with C++11 Threads

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

using namespace std;
mutex display_mutex;

struct Hello {
  Hello(uint16_t v) : id(v) {}
  void operator()(void) {
    display_mutex.lock();
    cout << "Hello from Thread "
         << id << endl;
    display_mutex.unlock();
  }
  uint16_t id;
};

int main(void) {
  vector<thread> allthreads;
  for (uint16_t i = 0; i < 10; ++i) {
    allthreads.push_back(thread(Hello(i)));
  }
  for (auto& t : allthreads) {
    t.join();
  }
  cout << "Hello from main" << endl;
}
```

"Hello World!" with C++11 Threads

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

using namespace std;
mutex display_mutex;

struct Hello {
  Hello(uint16_t v) : id(v) {}
  void operator()(void) {
    display_mutex.lock();
    cout << "Hello from Thread "
         << id << endl;
    display_mutex.unlock();
  }
  uint16_t id;
};

int main(void) {
  vector<thread> allthreads;
  for (uint16_t i = 0; i < 10; ++i) {
    allthreads.push_back(thread(Hello(i)));
  }
  for (auto& t : allthreads) {
    t.join();
  }
  cout << "Hello from main" << endl;
}
```

```
Hello from Thread 0
Hello from Thread 3
Hello from Thread 2
Hello from Thread 1
Hello from Thread 5
Hello from Thread 6
Hello from Thread 4
Hello from Thread 7
Hello from Thread 8
Hello from Thread 9
Hello from main
```

"Hello World!" with C++11 Threads

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

using namespace std;
mutex display_mutex;

struct Hello {
  Hello(uint16_t v) : id(v) {}
  void operator()(void) {
    { // Critical section
      lock_guard<mutex> guard(display_mutex);
      cout << "Hello from Thread "
           << id << endl;
    }
  }
  uint16_t id;
};

int main(void) {
  vector<thread> allthreads;
  for (uint16_t i = 0; i < 10; ++i) {
    allthreads.push_back(thread(Hello(i)));
  }
  for (auto& t : allthreads) {
    t.join();
  }
  cout << "Hello from main" << endl;
}
```

```
Hello from Thread 0
Hello from Thread 3
Hello from Thread 2
Hello from Thread 1
Hello from Thread 5
Hello from Thread 6
Hello from Thread 4
Hello from Thread 7
Hello from Thread 8
Hello from Thread 9
Hello from main
```

- `lock_guard`: locks mutex argument during construction and unlocks it upon destruction
- *Ressource Acquisition is Initialization* (RAII idiom)
- Guarantees *exception-safe* code

Creating Threads With C++11

- Include `<thread>` header
- Create a thread and launch it concurrently:

```
thread a_thread(f);
```

  Body `f` is:

  - a function;
  - a function object (`operator()` method). Copied into local storage for thread;
  - a lambda expression.

- Decide the relation parent/child:
  - Default: `std::terminate()` is called when the `thread` object is destroyed
  - `join()`: wait for the child to finish
  - `detach()`: sever the relation parent/child (*daemon thread*)
- Number of physical threads: `thread::hardware_concurrency()`
- Thread identifier: `thread::id get_id()` method

Parent/Child Relation

```
#include <cstdint>
#include <cmath>
#include <iostream>
#include <thread>

using namespace std;

void f(void)
{
  for (int i = 0; i < 100000; ++i) {
    cout << i << " ";
  }
  cout << endl;
}

int main(void)
{
  {
    thread t(f);
    cout << "End of block" << endl;
  }

  cin.ignore(); // wait for key pressed
  cout << "End of program" << endl;
}
```

```
End of block
0terminate called without an active exception
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38
39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56
57 58 59 60 61 62 63 64 65 66 67 68 69 70 71
72 Aborted
```

Parent/Child Relation

```
#include <cstdint>
#include <cmath>
#include <iostream>
#include <thread>

using namespace std;

void f(void)
{
  for (int i = 0; i < 100000; ++i) {
    cout << i << " ";
  }
  cout << endl;
}

int main(void)
{
  {
    thread t(f);
    t.join(); // Wait for end of t
    cout << "End of block" << endl;
  }

  cin.ignore(); // wait for key pressed
  cout << "End of program" << endl;
}
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
19 20 21 22 23 24 25 26 ...
99992 99993 99994 99995 99996 99997 99998 99999
End of block
End of program
```

Parent/Child Relation

```
#include <cstdint>
#include <cmath>
#include <iostream>
#include <thread>

using namespace std;

void f(void)
{
  for (int i = 0; i < 100000; ++i) {
    cout << i << " ";
  }
  cout << endl;
}

int main(void)
{
  {
    thread t(f);
    cout << "End of block" << endl;
    t.join(); // Wait for end of t
  }

  cin.ignore(); // wait for key pressed
  cout << "End of program" << endl;
}
```

```
End of block
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
19 20 21 22 23 24 25 26 27 28 29 30 31 ...
99992 99993 99994 99995 99996 99997 99998 99999
End of program
```

Parent/Child Relation

```
#include <cstdint>
#include <cmath>
#include <iostream>
#include <thread>

using namespace std;

void f(void)
{
  for (int i = 0; i < 100000; ++i) {
    cout << i << " ";
  }
  cout << endl;
}

int main(void)
{
  {
    thread t(f);
    t.detach(); // Daemonize t
    cout << "End of block" << endl;
  }

  cin.ignore(); // wait for key pressed
  cout << "End of program" << endl;
}
```

```
End of block
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
19 20 21 22 23 24 25 26 27 28 29 30 31 ...
99992 99993 99994 99995 99996 99997 99998 99999
End of program
```

Sharing Data with C++11 Threads

- All threads share the memory of the process
- Local variables of the *body* private to each thread
- No trouble if all threads only *read* global data
- Trouble: *race conditions*
- Protection of shared data: *mutexes*

```
// Global mutex
std::mutex exclusive;

{ // Critical section
  mutex.lock();
  // Do stuff on global variable exclusively
  // [...]
  mutex.unlock();
} // End of critical section: mutex unlocked
```

Note: a thread shall not try to lock a mutex it already owns (cf. `recursive_mutex`)

- `mutex` methods:
  - `lock()`: wait if mutex owned by another thread
  - `unlock()`: free the mutex
  - `try_lock()`: check whether the mutex is already owned by somebody

Synchronizing C++11 Threads

Condition variable: block threads until a condition is met

```
#include <iostream>

#define _GLIBCXX_USE_NANOSLEEP
#include <condition_variable>
#include <thread>
#include <chrono>

using namespace std;

condition_variable cv;
mutex mut;
int i = 0;

void waits(void)
{
  unique_lock<mutex> lock(mut);
  cerr << "Waiting... \n";
  cv.wait(lock, [](){return i == 1;});
  cerr << "...finished waiting.\n";
}
```

```
void signals(void)
{
  this_thread::sleep_for(chrono::seconds(1));
  cerr << "Notifying...\n";
  cv.notify_all();
  this_thread::sleep_for(chrono::seconds(1));
  i = 1;
  cerr << "Notifying again...\n";
  cv.notify_all();
}

int main(void)
{
  thread t1(waits), t2(waits),
    t3(waits), t4(signals);
  t1.join();
  t2.join();
  t3.join();
  t4.join();
  return EXIT_SUCCESS;
}
```

Atomically releases `lock`, blocks the current thread, and adds it to the list of threads waiting on the condition if it is not met. The thread will be unblocked and the condition retested when `notify_all()` or `notify_one()` is executed.

Asynchronous Computation

```
#include <iostream>
#include <string>
#include <future>

using namespace std;

string what_is_the_answer_to_life(void)
{
  return string("42");
}

int main(void)
{
  future<string> answer = async(what_is_the_answer_to_life);

  // Doing unrelated stuff
  // [...]

  cout << "The answer to life is: " << answer.get() << endl;

}
```

- `what_is_the_answer_to_life()` launched asynchronously
- Main thread blocks on `answer.get()` until the answer is available

Reflections on Using Threads

```
#include <iostream>
using namespace std;

int fib(int n) {
  if (n <= 1) {
    return n;
  }
  return fib(n-1) + fib(n-2);
}

int main(void) {
  cout << fib(30) << endl;
}
```

```
#include <iostream>
#include <future>
using namespace std;

int fib(int n) {
  if (n <= 1) {
    return n;
  }
  auto fib1 = async([n]{
      return fib(n-1);});
  int fib2 = fib(n-2);
  return fib1.get() + fib2;
}

int main(void) {
  cout << fib(30) << endl;
}
```

Reflections on Using Threads

```
#include <iostream>
using namespace std;

int fib(int n) {
  if (n <= 1) {
    return n;
  }
  return fib(n-1) + fib(n-2);
}

int main(void) {
  cout << fib(30) << endl;
}
```

```
> time fib
832040

real    0m0.024s
user    0m0.020s
sys     0m0.000s
```

```
#include <iostream>
#include <future>
using namespace std;

int fib(int n) {
  if (n <= 1) {
    return n;
  }
  auto fib1 = async([n]{
      return fib(n-1);});
  int fib2 = fib(n-2);
  return fib1.get() + fib2;
}

int main(void) {
  cout << fib(30) << endl;
}
```

```
> time parallel_fib
832040

real    0m3.186s
user    0m3.080s
sys     0m0.110s
```

Choose what to parallelize wisely...

The C++ Standard Thread Library

- Bird's eye view of the library
- Many features left aside (e.g., `atomic`)
- Many problems overlooked (e.g., protecting containers when references/pointers are allowed)
- For details:
  - *C++ Concurrency in Action: Practical Multithreading*. Anthony Williams. Manning Publications Co., 2012
  - Multi-threading Library for Standard C++ (Revision 1)
- Standard Thread Library: very basic building blocks. Applications require a lot of work
- Additional facilities: *Intel Threading Building Blocks*

Multithreading

Intel Threading Building Blocks

Framework

C++11 Standard Thread Library:

- Parallel program written in terms of logical threads (code parallelism)
- Up to the programmer to decide the number of logical threads wrt. physical threads

Intel Threading Building Blocks (TBB):

- Parallel program written in terms of *tasks* independently of ressources available
- TBB runtime maps tasks to physical threads efficiently (load balancing)
- Portability: Linux, Windows, Mac OS, Solaris
- Targets *scalability* of performances (data parallelism)
- Usable with other threading packages
- Open Source / Licensed

Intel TBB Components

- Parallel algorithms
  - `parallel_for`, `parallel_reduce`, `parallel_while`, `pipeline`, `parallel_sort`, `parallel_scan`
- Concurrent containers
  - `concurrent_hash_map`, `concurrent_queue`, `concurrent_vector`
- Synchronization primitives
  - Mutexes and atomic operations
- Memory allocator
  - `tbb_allocator`, `cache_aligned_allocator`, `scalable_allocator`
- Task Scheduler

What is a Task?

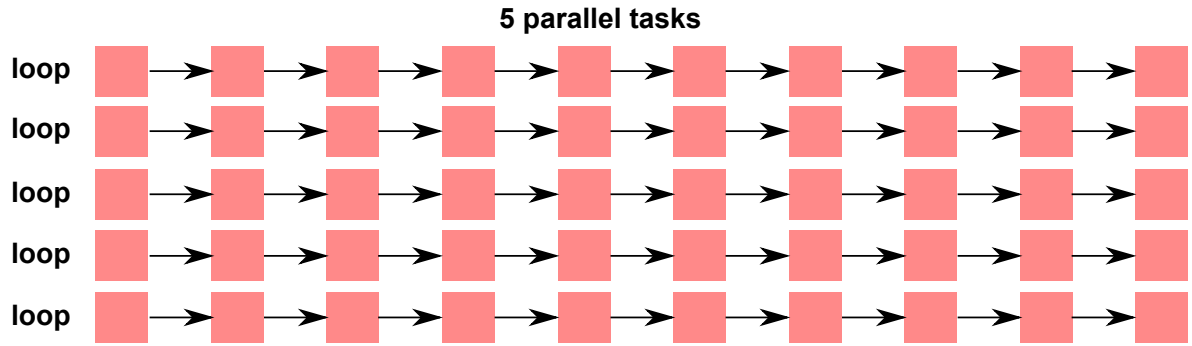
Task: light-weight thread (starting/stopping a task $\approx$ 18 times faster than a thread on Linux)

- Intel TBB parallel algorithms map tasks onto threads automatically
- Intel TBB Task Scheduler manages the physical thread pool
  - A task is mapped onto a thread until it finishes (no preemption—but look out for the OS scheduler!)
  - A task shall not block (the thread is not reassigned)

parallel_for (1/2)

Application of an operation on a set of values independently

```
// g++ -Wall -o para_for para_for.cpp -ltbb
#include <iostream>
#include <vector>
#include <tbb/blocked_range.h>
#include <tbb/parallel_for.h>
using namespace std;
using namespace tbb;
typedef blocked_range<int> range;
class addV {
public:
  addV(vector<double>& a,
       double v) : T(a), val(v) { }
  void operator()(const range& r) const{
    for (int i = r.begin();
         i != r.end(); ++i) {
      T[i] += val;
    }
  }
private:
  vector<double>& T;
  double val;
};

int main(void) {
  const int sz = 50;
  vector<double> T(sz,0.0);

  parallel_for(range(0,sz,10), addV(T,4.5));
}
```



parallel_for (2/2)

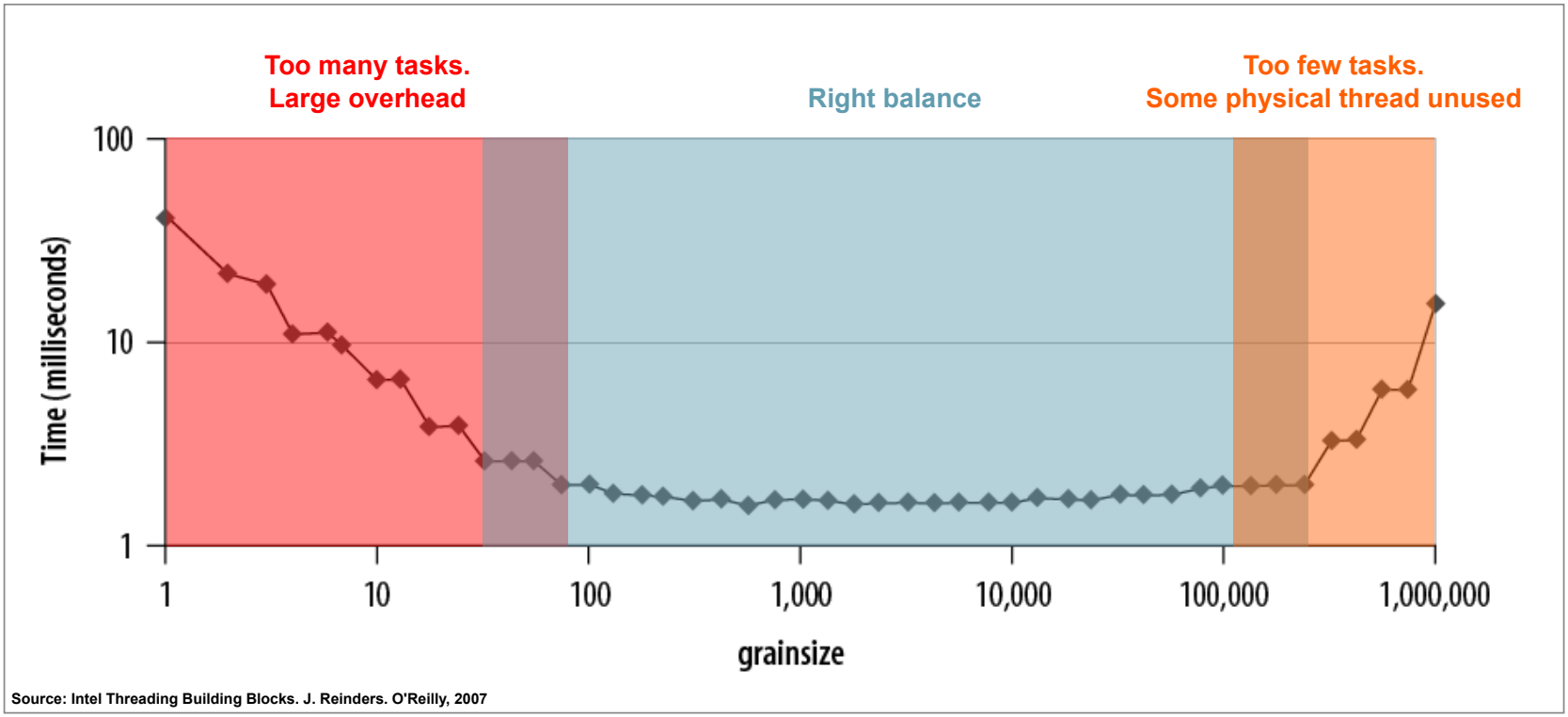
Requirements for the body "Body":

- `Body::Body(const Body&)` (Copy constructor)
- `Body::~Body()` (Destructor)
- `void Body::operator()(Range& subrange) const` (Apply the body to *subrange*)

Granularity

- Parameter to amortize parallel scheduling overhead
- How many object should be treated sequentially? (size of the subrange)
- Minimum threshold for parallelization (grain too high $\rightsquigarrow$ sequential program)
- Rule of thumb: body should contain 10000 to 100000 machine instructions
- Setting the right granularity: delicate balance to strike $\Rightarrow$ test!

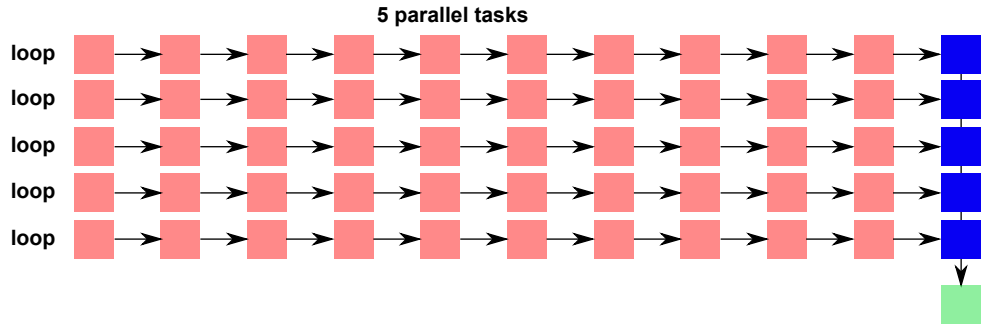
Computing a[i] = b[i]*c for 1,000,000 values



Source: Intel Threading Building Blocks. J. Reinders. O'Reilly, 2007

parallel_reduce (1/2)

```
#include <iostream>
#include <vector>
#include <limits>
#include <tbb/blocked_range.h>
#include <tbb/parallel_reduce.h>
using namespace std;
using namespace tbb;
typedef blocked_range<int> range;
const double inf =
  numeric_limits<double>::infinity();
class Min {
public:
  Min(vector<double>& a): mini(inf),
                          T(a) {}
  Min(Min& o, split): mini(inf),
                      T(o.T) {}
  void operator()(const range& r) {
    for (int i = r.begin();
         i != r.end(); ++i) {
      if (mini > T[i]) {
        mini = T[i];
      }
    }
  }
  void join(const Min& o) {
    if (o.mini < mini) {
      mini = o.mini;
    }
  }
  double mini;
private:
  vector<double>& T;
};

int main(void) {
  const int sz = 50;
  vector<double> T;

  // Filling T

  Min m(T);
  parallel_reduce(range(0,sz,10), m);

  cout << m.mini << endl;
}
```



parallel_reduce (2/2)

Requirements for the body "Body":

- `Body::Body(const Body&, split)` (Splitting constructor)
- `Body::~Body()` (destructor)
- `void Body::operator()(Range& subrange)` (Accumulate results from subrange)
- `void Body::join(Body& rhs)` (Merge results of *rhs* into the result of `this`)

OpenMP

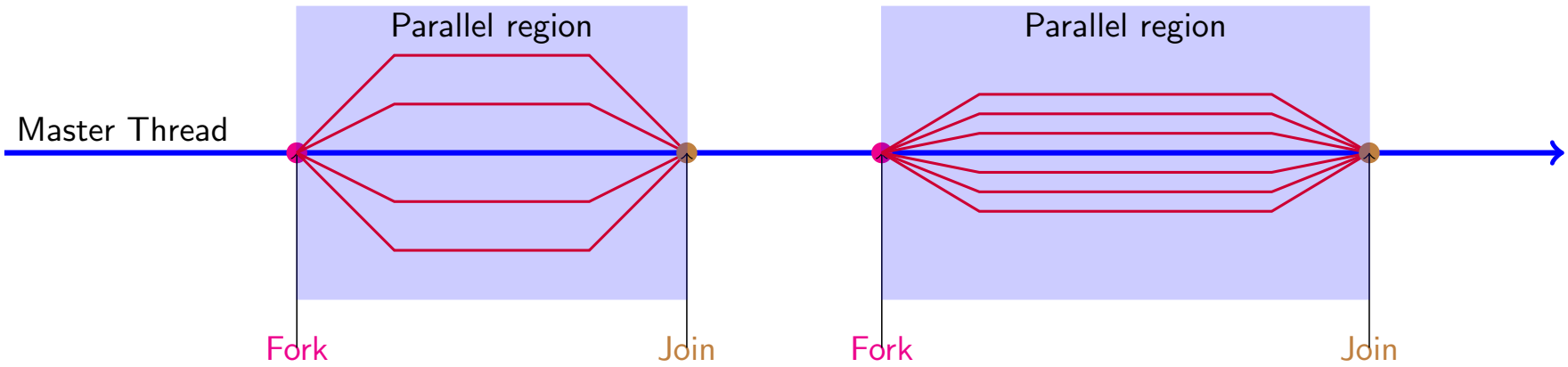
What is OpenMP?

- Multi-platform compiler extension for Fortran/C/C++
- Handled by not-for-profit consortium since 1997
- Shared memory parallel programming
- Task parallelism and data parallelism
- No fundamental modification of sequential algorithms
- Support: GCC, Visual C++, Intel C++, ...

GCC Version	*OpenMP Specs.*
gcc 4.2	OpenMP 2.5
gcc 4.4	OpenMP 3.0
gcc 4.7	OpenMP 3.1

- Bibliography:
  - *Programming with OpenMP*. Intel Software College, 2008
  - *Using OpenMP: Portable Shared Memory Parallel Programming*. B. Chapman, G. Jost, R. van der Pas. The MIT Press, 2008

The OpenMP Execution Model



- Master Thread forks Worker Threads and waits for their completion

Hello World! without OpenMP

```
#include <iostream>

using namespace std;

int main(void)
{
  for (int i = 0; i < 4; ++i) {
    cout << "Hello from Loop " << i << endl;
  }
}
```

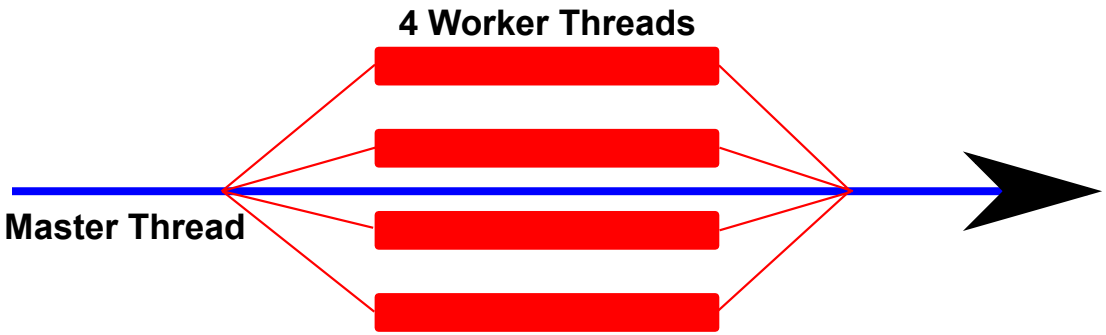
Hello World! without OpenMP

```
#include <iostream>

using namespace std;

int main(void)
{
  for (int i = 0; i < 4; ++i) {
    cout << "Hello from Loop " << i << endl;
  }
}
```

```
./hello
Hello from Loop 0
Hello from Loop 1
Hello from Loop 2
Hello from Loop 3
```

Hello World! with OpenMP

```
// Compile with:
//     g++ -fopenmp -o hello-omp-1 hello-omp-1.cpp

#include <iostream>
#include <omp.h>

using namespace std;

int main(void)
{
#pragma omp parallel for
  for (int i = 0; i < 4; ++i) {
    cout << "Hello from Thread " << omp_get_thread_num() << endl;
  }
}
```

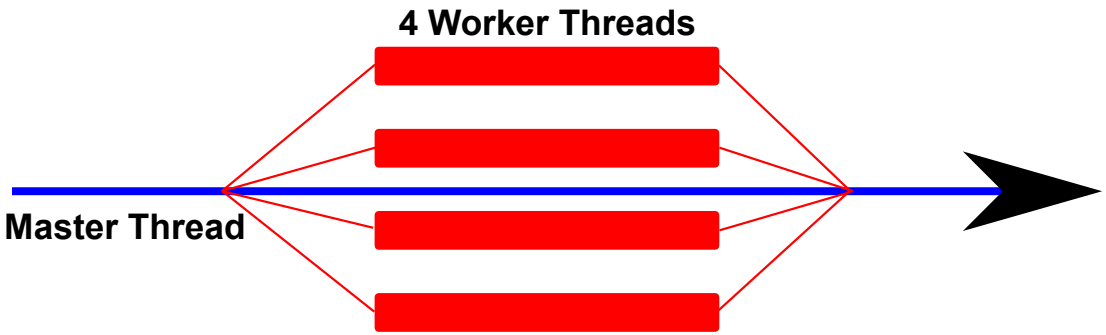


Synchronization point at end of group following #pragma

Hello World! with OpenMP

```
// Compile with:
//     g++ -fopenmp -o hello-omp-1 hello-omp-1.cpp

#include <iostream>
#include <omp.h>

using namespace std;

int main(void)
{
#pragma omp parallel for
  for (int i = 0; i < 4; ++i) {
    cout << "Hello from Thread " << omp_get_thread_num() << endl;
  }
}
```

```
> ./hello-omp-1
Hello from Thread Hello from Thread 02
Hello from Thread 3
Hello from Thread 1
```



Synchronization point at end of group following #pragma

Hello World! with OpenMP

```
// Compile with:
//  g++ -std=gnu++11 -pthread -fopenmp -o hello-omp-2 hello-omp-2.cpp

#include <iostream>
#include <omp.h>
#include <mutex>

using namespace std;

int main(void)
{
  {
    mutex mut; // Mutex with limited scope
#pragma omp parallel for
    for (int i = 0; i < 4; ++i) {
      mut.lock();
      cout << "Hello from Thread " << omp_get_thread_num() << endl;
      mut.unlock();
    }
  }
}
```

Hello World! with OpenMP

```
// Compile with:
//  g++ -std=gnu++11 -pthread -fopenmp -o hello-omp-2 hello-omp-2.cpp

#include <iostream>
#include <omp.h>
#include <mutex>

using namespace std;

int main(void)
{
  {
    mutex mut; // Mutex with limited scope
#pragma omp parallel for
    for (int i = 0; i < 4; ++i) {
      mut.lock();
      cout << "Hello from Thread " << omp_get_thread_num() << endl;
      mut.unlock();
    }
  }
}
```

```
> ./hello-omp-2
Hello from Thread 0
Hello from Thread 2
Hello from Thread 1
Hello from Thread 3
```

Hello World! with OpenMP

```
// Compile with:
//     g++ -fopenmp -o hello-omp-3 hello-omp-3.cpp

#include <iostream>
#include <omp.h>

using namespace std;

int main(void)
{
#pragma omp parallel for
    for (int i = 0; i < 4; ++i) {
#pragma omp critical
      cout << "Hello from Thread " << omp_get_thread_num() << endl;
    }
}
```

Hello World! with OpenMP

```
// Compile with:
//     g++ -fopenmp -o hello-omp-3 hello-omp-3.cpp

#include <iostream>
#include <omp.h>

using namespace std;

int main(void)
{
#pragma omp parallel for
    for (int i = 0; i < 4; ++i) {
#pragma omp critical
      cout << "Hello from Thread " << omp_get_thread_num() << endl;
    }
}
```

```
> ./hello-omp-3
Hello from Thread 0
Hello from Thread 3
Hello from Thread 1
Hello from Thread 2
```

Hello World! with OpenMP

```
// Compile with:
//     g++ -fopenmp -o hello-omp-4 hello-omp-4.cpp

#include <iostream>
#include <omp.h>

using namespace std;

int main(void)
{
#pragma omp parallel for ordered
    for (int i = 0; i < 4; ++i) {
#pragma omp ordered
      cout << "Hello from Thread " << omp_get_thread_num() << endl;
    }
}
```

Hello World! with OpenMP

```
// Compile with:
//     g++ -fopenmp -o hello-omp-4 hello-omp-4.cpp

#include <iostream>
#include <omp.h>

using namespace std;

int main(void)
{
#pragma omp parallel for ordered
    for (int i = 0; i < 4; ++i) {
#pragma omp ordered
      cout << "Hello from Thread " << omp_get_thread_num() << endl;
    }
}
```

```
> ./hello-omp-4
Hello from Thread 0
Hello from Thread 1
Hello from Thread 2
Hello from Thread 3
```

`#pragma omp ordered`: block executed in the order of the sequential iterations

Using OpenMP

- Incremental addition of parallelism to code
- Addition of `#pragmas` before *structure blocks*
- Structured block: C++ statement with only one point of entry/exit
- Implicit barrier (synchronization point) at the end of block

#pragma omp parallel

#pragma omp parallel *[clause[[,] clause] . . .]*
Structured-block

- Create a team of OMP_NUM_THREADS threads
- *Structured-block* executed concurrently by each thread
- Master thread waits for all threads to finish

```
#include <iostream>
#include <omp.h>

using namespace std;

int main(void)
{
  omp_set_num_threads(1);
#pragma omp parallel
  cout << "A structured block" << endl;
}
/* Output:
   A structured block
*/
```

#pragma omp parallel

#pragma omp parallel *[clause[[,] clause] . . .]*
Structured-block

- Create a team of `OMP_NUM_THREADS` threads
- *Structured-block* executed concurrently by each thread
- Master thread waits for all threads to finish

```
#include <iostream>
#include <omp.h>

using namespace std;

int main(void)
{
  omp_set_num_threads(4);
#pragma omp parallel
  cout << "A structured block" << endl;
}
/* Output:
   A structured block
   A structured block
   A structured block
   A structured block
*/
```

API for threads

- `omp_get_num_threads()`. Get the number of threads in the current team of workers
- `omp_set_num_threads()`. Set the number of threads for the future teams of workers
- `omp_get_max_threads()`. The maximum number of threads available to perform the next `parallel` pragmas
- `omp_get_thread_num()`. Current thread id (integer)

#pragma omp for

#pragma omp for *[clause[[,] clause] . . .]*
for-loop

- Pragma to appear in a "`#pragma omp parallel`" block (or use "`#pragma omp parallel for`"
- Master thread divides iterations among threads

```
#include <iostream>
#include <omp.h>

using namespace std;

int main(void)
{
  omp_set_num_threads(8);

  int T[8];
#pragma omp parallel
#pragma omp for
  for (int i = 0; i < 8; ++i) {
    T[i] = 0;
  }
}
```

Variable scopes

- Default rule:
  - Data declared outside parallel blocks is shared
  - Data declared inside parallel blocks is private to each thread
- Overriding the default rule: *clauses* `private`, `firstprivate`, `lastprivate`, `shared`, ...

```
#include <iostream>
#include <iterator>
#include <omp.h>
#include <vector>
#define STRINGI(T) #T,T
using namespace std;

void displayT(const char* n,
              const vector<int>& T) {
  cout << n << ": ";
  copy(T.begin(), T.end(),
       ostream_iterator<int>(cout, " "));
  cout << "\n";
}

int main(void) {
  vector<int> T1(4,0), T2(4,0);

  omp_set_num_threads(5);
#pragma omp parallel firstprivate(T2)
  {
#pragma omp critical
      cout << "*";
    vector<int> T3(4,0);
#pragma omp for
    for (int i = 0; i < 4; ++i) {
      ++T1[i]; ++T2[i]; ++T3[i];
                cout << "+";
    }
#pragma omp critical
    displayT(STRINGI(T3));
  }
  displayT(STRINGI(T1));
  displayT(STRINGI(T2));
}
```

Variable scopes

- Default rule:
  - Data declared outside parallel blocks is shared
  - Data declared inside parallel blocks is private to each thread
- Overriding the default rule: *clauses* `private`, `firstprivate`, `lastprivate`, `shared`, ...

```
#include <iostream>
#include <iterator>
#include <omp.h>
#include <vector>
#define STRINGI(T) #T,T
using namespace std;

void displayT(const char* n,
              const vector<int>& T) {
  cout << n << ": ";
  copy(T.begin(), T.end(),
       ostream_iterator<int>(cout, " "));
  cout << "\n";
}

int main(void) {
  vector<int> T1(4,0), T2(4,0);

  omp_set_num_threads(5);
#pragma omp parallel firstprivate(T2)
  {
#pragma omp critical
      cout << "*";
    vector<int> T3(4,0);
#pragma omp for
    for (int i = 0; i < 4; ++i) {
      ++T1[i]; ++T2[i]; ++T3[i];
                cout << "+";
    }
#pragma omp critical
    displayT(STRINGI(T3));
  }
  displayT(STRINGI(T1));
  displayT(STRINGI(T2));
}
```

```
**++**+*+T3: 1 0 0 0
T3: 0 1 0 0
T3: 0 0 0 0 // <-!
T3: 0 0 0 1
T3: 0 0 1 0
T1: 1 1 1 1

T2: 0 0 0 0
```

Clauses

- `private`. Each thread has its own copy
- `shared`. All threads share the same copy
- `firstprivate`. Initialize the private copy with value available before thread

```
#include <iostream>
#include <omp.h>

using namespace std;

int main(void) {
  double a = 3.125, b = 6.25;

  omp_set_num_threads(1);
#pragma omp parallel private(a)
  cout << a << " " << b << endl;

#pragma omp parallel firstprivate(a)
  cout << a << " " << b << endl;
}
/* Output:
      0 6.25
      3.125 6.25
*/
```

- `lastprivate`. "Last value" computed is assigned to global variable

```
#include <iostream>
#include <omp.h>

using namespace std;

int main(void) {
  int v;

#pragma omp parallel for lastprivate(v)
  for (int i = 4; i > 0; --i) {
    v = i;
  }
  cout << v << endl;
}
/* Output:
      1
*/
```

Mutual exclusion

- `#pragma omp critical` *`[(name)]`*. Mutual exclusion on all regions with same *name*

```
#include <iostream>
#include <omp.h>

using namespace std;

int main(void)
{
  int i = 0;
  omp_set_num_threads(4);
#pragma omp parallel shared(i)
  {
#pragma omp critical (section_1)
    i += 1;
#pragma omp critical (section_2)
    i += 1;
  }
  cout << i << endl;
}
```

- `#pragma omp atomic`. Exclusion on write statement

```
#include <iostream>
#include <omp.h>

using namespace std;

int main(void)
{
  int i = 0;
  omp_set_num_threads(4);
#pragma omp parallel shared(i)
  {
#pragma omp atomic
    i += 1;
#pragma omp atomic
    i += 1;
  }
  cout << i << endl;
}
```

- Locks

Mutual exclusion

- `#pragma omp critical` *`[(name)]`*. Mutual exclusion on all regions with same *name*

```
#include <iostream>
#include <omp.h>

using namespace std;

int main(void)
{
  int i = 0;
  omp_set_num_threads(4);
#pragma omp parallel shared(i)
  {
#pragma omp critical (section_1)
    i += 1;
#pragma omp critical (section_2)
    i += 1;
  }
  cout << i << endl;
}
```

7

- `#pragma omp atomic`. Exclusion on write statement

```
#include <iostream>
#include <omp.h>

using namespace std;

int main(void)
{
  int i = 0;
  omp_set_num_threads(4);
#pragma omp parallel shared(i)
  {
#pragma omp atomic
    i += 1;
#pragma omp atomic
    i += 1;
  }
  cout << i << endl;
}
```

- Locks

Mutual exclusion

- `#pragma omp critical` *`[(name)]`*. Mutual exclusion on all regions with same *name*

```
#include <iostream>
#include <omp.h>

using namespace std;

int main(void)
{
  int i = 0;
  omp_set_num_threads(4);
#pragma omp parallel shared(i)
  {
#pragma omp critical (section_1)
    i += 1;
#pragma omp critical (section_2)
    i += 1;
  }
  cout << i << endl;
}
```

7

- `#pragma omp atomic`. Exclusion on write statement

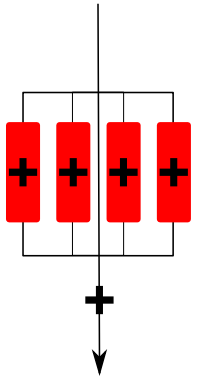
```
#include <iostream>
#include <omp.h>

using namespace std;

int main(void)
{
  int i = 0;
  omp_set_num_threads(4);
#pragma omp parallel shared(i)
  {
#pragma omp atomic
    i += 1;
#pragma omp atomic
    i += 1;
  }
  cout << i << endl;
}
```

8

- Locks

Reduction

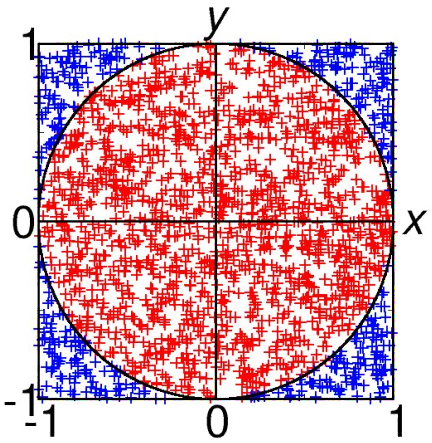
```
#include <iostream>
#include <omp.h>

using namespace std;

int main(void)
{
  omp_set_num_threads(4);
  int sum = 0;
#pragma omp parallel for reduction(+:sum)
  for (int i = 1; i<= 100000; ++i) {
    sum += i;
  }
  cout << sum << endl;
}
```



- Format:
  `#pragma omp parallel reduction(op:var, var, ...)`
- Private copies of global *var* (initialized to *neutral value*)
- All private copies combined with operator *op* in global *var*
- Reduction operators: $+$, $-$, $\times$, &, ^, |, &&, ||, min, max

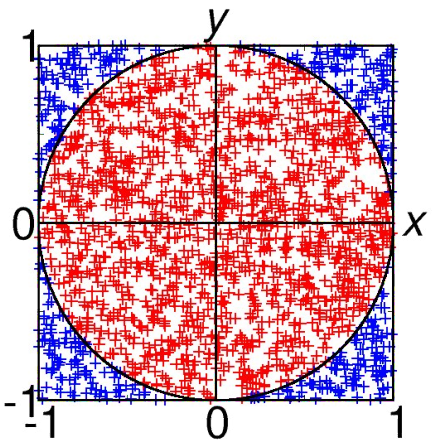
Computing π with a Monte Carlo method

```
#include <iostream>
#include <cstdlib>
#include <cmath>

using namespace std;

int main(void)
{
  unsigned int nlaunches = 10000000;
  srand48(10);
  unsigned int nhits = 0;

  for (unsigned int i = 0; i < nlaunches; ++i) {
    double x = drand48();
    double y = drand48();
    nhits +=(sqrt(x*x + y*y) < 1.0);
  }
  double pi = double(nhits)/double(nlaunches)*4;
  cout << pi << endl;
}
```



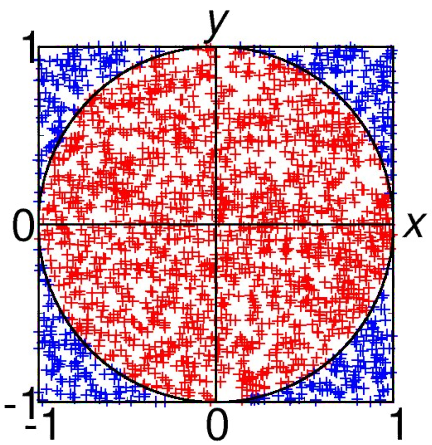
Computing π with a Monte Carlo method

```
#include <iostream>
#include <cstdlib>
#include <cmath>

using namespace std;

int main(void)
{
  unsigned int nlaunches = 10000000;
  srand48(10);
  unsigned int nhits = 0;

#pragma omp parallel for reduction(+:nhits)
  for (unsigned int i = 0; i < nlaunches; ++i) {
    double x, y;
    x = drand48();
    y = drand48();
    nhits += (sqrt(x*x + y*y) < 1.0);
  }
  double pi = double(nhits)/double(nlaunches)*4;
  cout << pi << endl;
}
```



Computing π with a Monte Carlo method

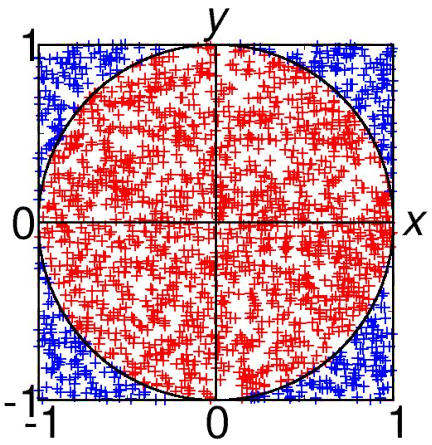
```
#include <iostream>
#include <cstdlib>
#include <cmath>

using namespace std;

int main(void)
{
  unsigned int nlaunches = 10000000;
  srand48(10);
  unsigned int nhits = 0;

#pragma omp parallel for reduction(+:nhits)
  for (unsigned int i = 0; i < nlaunches; ++i) {
    double x, y;
    x = drand48();
    y = drand48();
    nhits += (sqrt(x*x + y*y) < 1.0);
  }
  double pi = double(nhits)/double(nlaunches)*4;
  cout << pi << endl;
}
```



Problem: drand48() not thread-safe

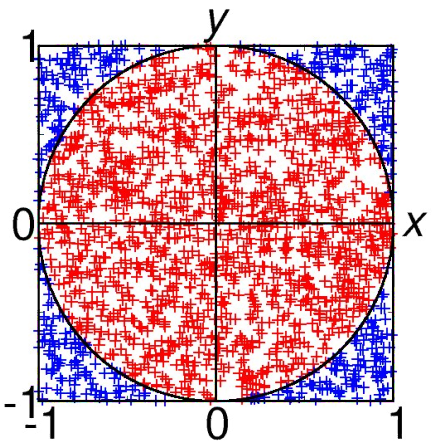
Computing π with a Monte Carlo method

```
#include <iostream>
#include <cstdlib>
#include <cmath>

using namespace std;

int main(void)
{
  unsigned int nlaunches = 10000000;
  drand48_data buf;
  srand48_r(10,&buf);
  unsigned int nhits = 0;

#pragma omp parallel for reduction(+:nhits), firstprivate(buf)
  for (unsigned int i = 0; i < nlaunches; ++i) {
    double x, y;
    drand48_r(&buf,&x);
    drand48_r(&buf,&y);
    nhits += (sqrt(x*x + y*y) < 1.0);
  }
  double pi = double(nhits)/double(nlaunches)*4;
  cout << pi << endl;
}
```



Computing π with a Monte Carlo method

```
#include <iostream>
#include <cstdlib>
#include <cmath>

using namespace std;

int main(void)
{
  unsigned int nlaunches = 10000000;
  drand48_data buf;
  srand48_r(10,&buf);
  unsigned int nhits = 0;

#pragma omp parallel for reduction(+:nhits), firstprivate(buf)
  for (unsigned int i = 0; i < nlaunches; ++i) {
    double x, y;
    drand48_r(&buf,&x);
    drand48_r(&buf,&y);
    nhits += (sqrt(x*x + y*y) < 1.0);
  }
  double pi = double(nhits)/double(nlaunches)*4;
  cout << pi << endl;
}
```

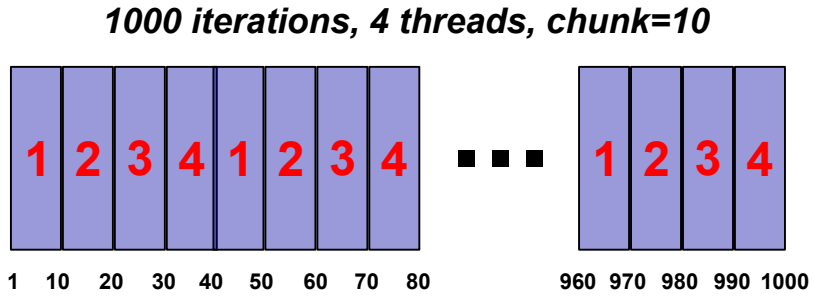


Problem: all random generators initialized with same seed

Scheduling

`schedule` clause in "`omp for`" pragma:

- `schedule(static, [chunk])`.
  - Chunks assigned round-robin statically to threads



  Plus: Low overhead / Minus: load-imbalance
- `schedule(dynamic, [chunk])`.
  - Each thread request chunks as soon as finished with previous one
  - Plus: reduces load-imbalance / Minus: high overhead
- `schedule(guided, [chunk])` and `schedule(runtime, [chunk])`
  - Statically/dynamically defined strategies

Parallelism and parallelism again

- Previous constructs : data parallelism
  - Same code executed in parallel on different cores
- OpenMP also offers *task parallelism*
  - Different tasks performed in parallel
  - #pragma omp sections
  - #pragma omp task

#pragma omp sections

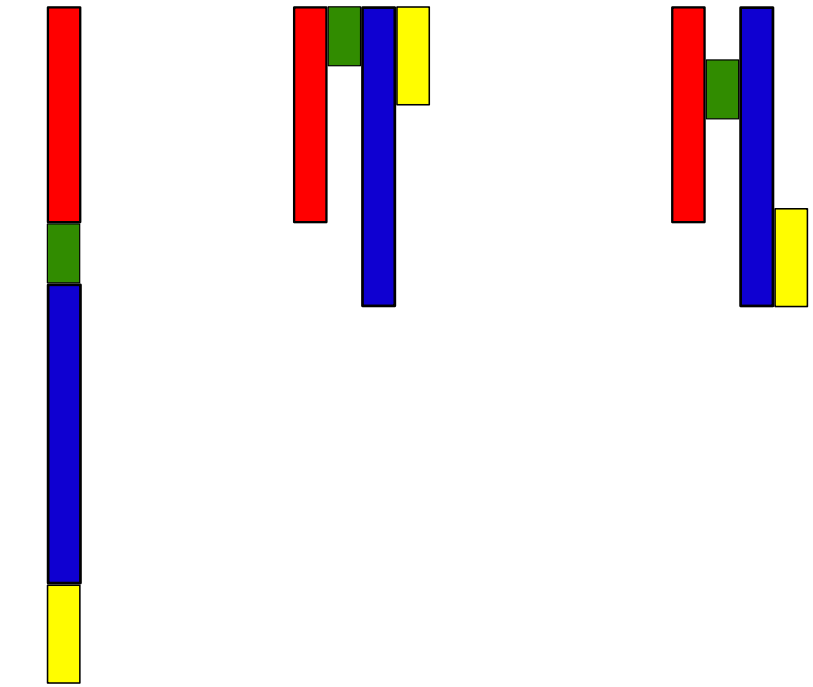
```
#include <omp.h>

int main(void)
{
  int res, tmp1, tmp2;

#pragma omp parallel
#pragma omp sections
  {
#pragma omp section
    tmp1 = do_lengthy_stuff();
#pragma omp section
    tmp2 = do_other_lengthy_stuff();
  }
  res = tmp1 + tmp2;
}
```

- Sections executed in parallel
- Synchronization point at the end of the "`sections`" block

#pragma omp task

- Tasks are independent units of work
- Threads assigned to each task
- Tasks may be deferred or executed immediately (compare with C++11 `future/async`) by decision of runtime system
- Completion of taks guaranteed at:
  - Thread/task barrier
  - directive `#pragma omp barrier`
  - directive `#pragma omp taskwait`

Sections vs. Tasks



Sections

- Barrier at the end
- Some threads idle

Tasks

- No implicit barrier
- Reuse of threads

Using Tasks (1/2)

```
#include <iostream>
#include <omp.h>

using namespace std;

int main(void)
{
  omp_set_num_threads(2);
#pragma omp parallel
  {
    cout << " How ";
#pragma omp task
    cout << " are ";
#pragma omp task
    cout << " you? ";
  }
  cout << endl;
}
```

```
> ./task
 How How are you? you? are
```

Using Tasks (1/2)

```
#include <iostream>
#include <omp.h>

using namespace std;

int main(void)
{
  omp_set_num_threads(2);
#pragma omp parallel
#pragma omp single
  {
    cout << " How ";
#pragma omp task
    cout << " are ";
#pragma omp task
    cout << " you? ";
  }
  cout << endl;
}
```

```
> ./task
 How are you?
> ./task
 How you? are
```

Using Tasks (1/2)

```
#include <iostream>
#include <omp.h>

using namespace std;

int main(void)
{
  omp_set_num_threads(2);
#pragma omp parallel
#pragma omp single
  {
    cout << " How ";
#pragma omp task
    cout << " are ";
#pragma omp task
    cout << " you? ";
  }
  cout << endl;
}
```

```
> ./task
 How are you?
> ./task
 How you? are
```

`#pragma omp single`: only one thread executes the code. Other threads wait for the completion of the block

Using Tasks (2/2)

```
#include <iostream>
#include <omp.h>

using namespace std;

int fib(int n)
{
  if (n <= 1) {
    return n;
  }
  int fibn1, fibn2;
#pragma omp task shared(fibn1)
    fibn1 = fib(n-1);
#pragma omp task shared(fibn2)
    fibn2 = fib(n-2);
#pragma omp taskwait
  return fibn1 + fibn2;
}

int main(void)
{
  int result;
#pragma omp parallel
#pragma omp single
  result = fib(30);
  cout << result << endl;
}
```

Using Tasks (2/2)

```
#include <iostream>
#include <omp.h>

using namespace std;

int fib(int n)
{
  if (n <= 1) {
    return n;
  }
  int fibn1, fibn2;
#pragma omp task shared(fibn1)
    fibn1 = fib(n-1);
#pragma omp task shared(fibn2)
    fibn2 = fib(n-2);
#pragma omp taskwait
  return fibn1 + fibn2;
}

int main(void)
{
  int result;
#pragma omp parallel
#pragma omp single
  result = fib(30);
  cout << result << endl;
}
```

`#pragma omp taskwait`: the two subtasks at the same level wait for each other

Going further (1/2)

- Many other constructs and clauses
  - `omp master`, `nowait`, `flush`, ...
- Possibility to conditionally parallelize code
  - Macro `_OPENMP`
  - Clause `if`

```
#include <iostream>
#if _OPENMP
#  include <omp.h>
#endif
using namespace std;

int main(void)
{
  int n;
  int sum = 0;
  cout << "# elements? ";
  cin >> n;
#pragma omp parallel for if(n > 1000), reduction(+:sum)
  for (int i = 0; i < n; ++i) {
    sum += i;
  }
  cout << sum << endl;
}
```

- Refer to documentation (links on madoc)

Going further (2/2)

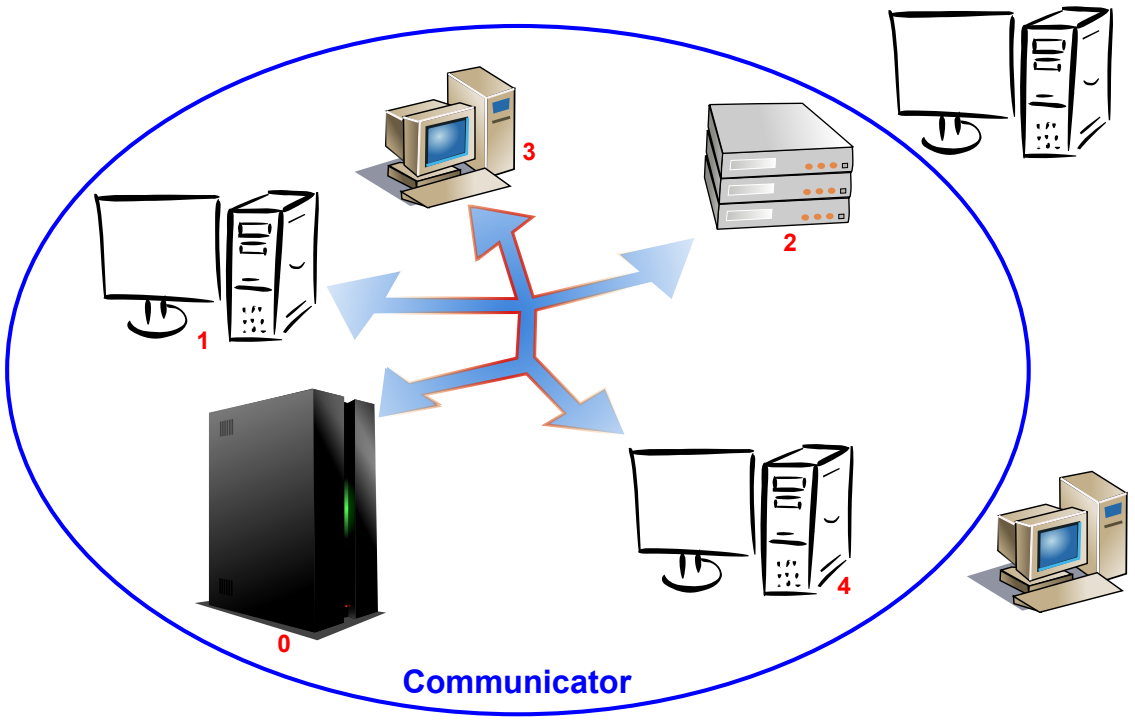
- OpenMP for shared-memory fine-grained parallelism
- Applications have corse-grained/fine-grained parallelism
- Solution: MPI for distributed memory coarse-grained level, and OpenMP for shared memory fine-grained level

MPI

What is MPI?

- *Message-Passing Interface* (MPI) "standardized" in 1994
- API, not implementation. Many implementations:
  - MPICH, LAM, OpenMPI, ...
- Multi-language: C, C++, FORTRAN (+ numerous bindings)
  - Caveat: C++ interface less used than C (see also: Boost.MPI)
- Distributed computing on clusters of (heterogeneous) computers
- Documentation:
  - *MPI: The Complete Reference*. Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra. The MIT Press, 1991
  - *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. William Gropp, Ewing Lusk, Anthony Skjellum. The MIT Press, 1994
  - . William Gropp, Ewing Lusk, Rajeev Thakur. The MIT Press, 1999

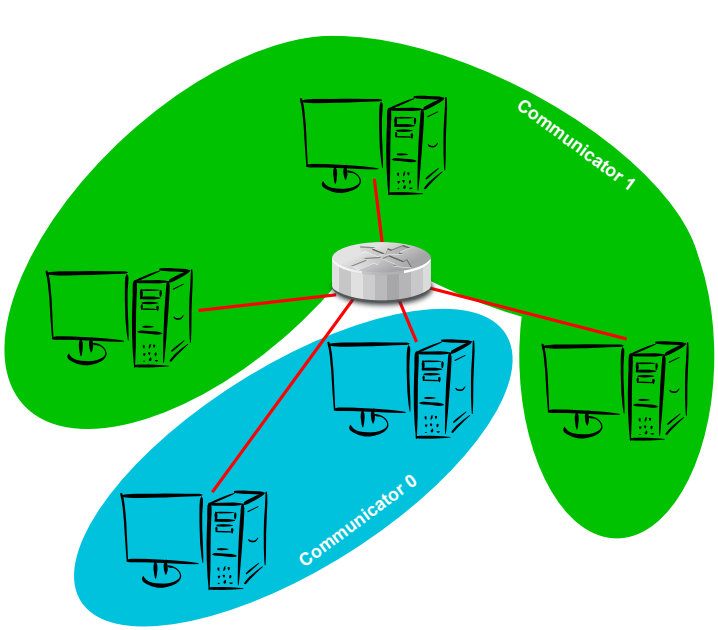
The MPI Computing Model

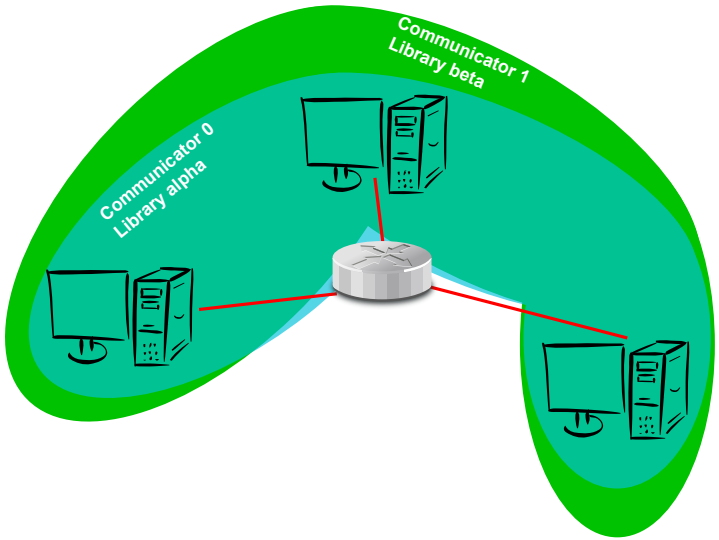


- Independent *processes*
- Private data
- Communication by passing messages between processes
- Good algorithm design: minimize communications
  - Latency (Cost first call)
  - Bandwidth

Communicators

- Communicator: set of processes that can communicate with each other

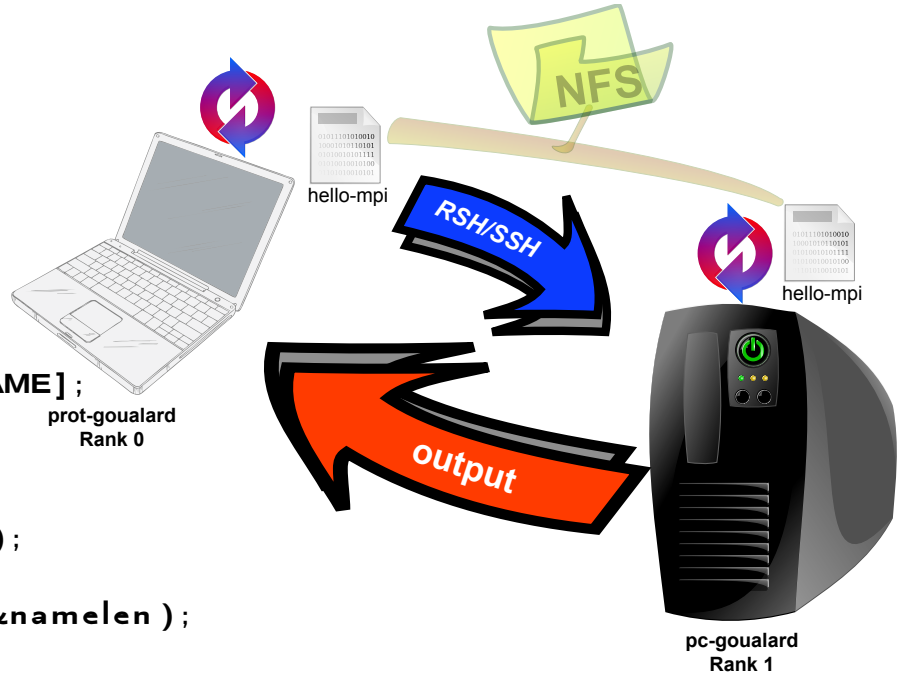




- `MPI_Init()`: definition of a communicator (`MPI_COMM_WORLD`)
- Each process in `MPI_COMM_WORLD` has a unique *rank*
  - Rank 0 is the server
  - Only Rank 0 can get input from `stdin`
- Information on `MPI_COMM_WORLD`:
  - Number of processes: `MPI_Comm_size()`
  - Rank of current process: `MPI_Comm_rank()`

"Hello World!" with MPI

```
#include <iostream>
#include <mpi.h>

using namespace std;

int main(int argc, char *argv[]) {
  int numprocs, rank, namelen;
  char processor_name[MPI_MAX_PROCESSOR_NAME];

  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(processor_name, &namelen);

  cout << "Process " << rank << " on "
       << processor_name << " out of " << numprocs << endl;

  MPI_Finalize();
}
```



- `mpic++ hello-mpi.cpp -o hello-mpi`
- `mpirun -H prot-goualard,pc-goualard -n 5 hello-mpi`

```
Process 0 on prot-goualard out of 5
Process 2 on prot-goualard out of 5
Process 4 on prot-goualard out of 5
Process 1 on pc-goualard out of 5
Process 3 on pc-goualard out of 5
```

Signatures of basic functions

```
int MPI_Init(int* argc_p,      // Address of argc from main()
             char ***argv_p);  // Address of argv from main()
```

```
int MPI_Finalize(void);    // Frees all MPI ressources
```

```
int MPI_Get_processor_name(char *name,      // Name of actual node
                           int *resultlen); // Length of name string
```

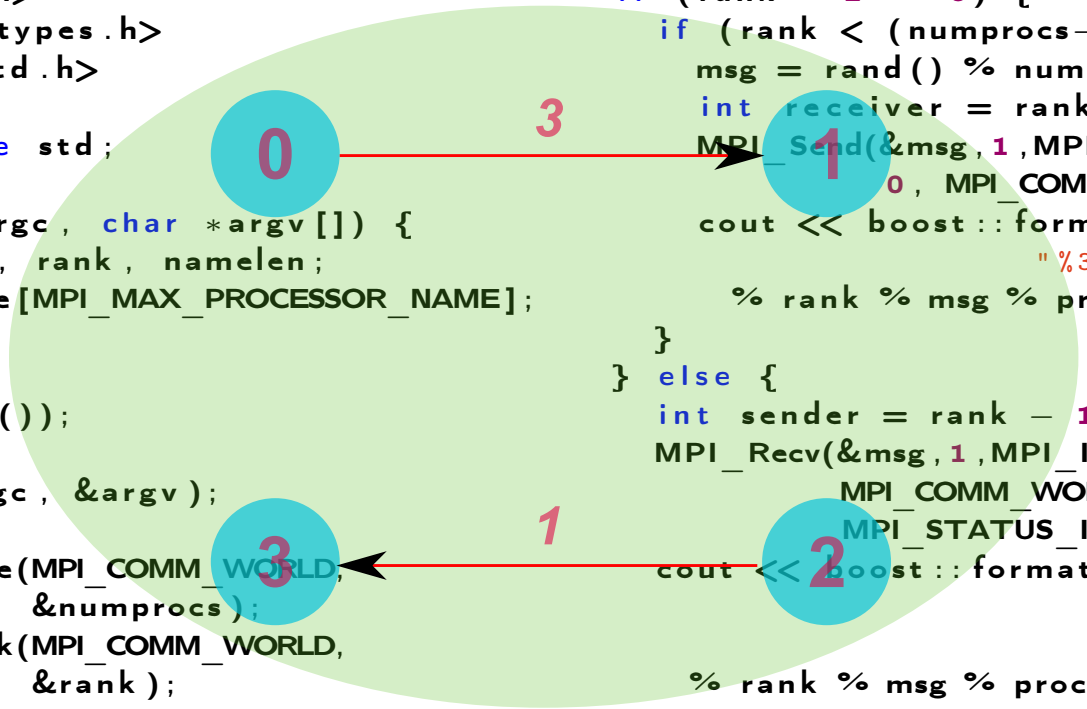
```
int MPI_Comm_size(MPI_Comm comm,   // Communicator
                  int* comm_sz_p); // Number of processes in comm
```

```
int MPI_Comm_rank(MPI_Comm comm,   // Communicator
                  int* my_rank_p); // Rank of current process
```

Return values: error code

Point-to-point communication

```
#include <iostream>
#include <boost/format.hpp>
#include <mpi.h>
#include <sys/types.h>
#include <unistd.h>

using namespace std;

int main(int argc, char *argv[]) {
  int numprocs, rank, namelen;
  char procname[MPI_MAX_PROCESSOR_NAME];
  int msg;

  srand(getpid());

  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD,
                &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,
                &rank);
  MPI_Get_processor_name(procname,
                         &namelen);

  // Each process with even rank sends
  // a random value to the process
  // with rank immediately above, if
  // it exists
  if (rank % 2 == 0) {
    if (rank < (numprocs-1)) {
      msg = rand() % numprocs;
      int receiver = rank + 1;
      MPI_Send(&msg,1,MPI_INT, receiver,
               0, MPI_COMM_WORLD);
      cout << boost::format("Process %1% on "
                            "%3% sent value %2%")
        % rank % msg % procname << endl;
    }
  } else {
    int sender = rank - 1;
    MPI_Recv(&msg,1,MPI_INT, sender, 0,
             MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    cout << boost::format("Process %1% on "
                          "%3% received "
                          "value %2%")
      % rank % msg % procname << endl;
  }
  MPI_Finalize();
}
```




```
mpiexec -H prot-goualard,orwell -wdir /export/mpi -n 4 send-receive
```

```
Process 2 on prot-goualard sent value 1
Process 3 on orwell received value 1
Process 0 on prot-goualard sent value 3
Process 1 on orwell received value 3
```

Sending/Receiving

```
int MPI_Send(void *buf, // Address of data
             int count, // Number of data
             MPI_Datatype datatype, // Type of data
             int dest, // Rank of recipient
             int tag,  // Message tag
             MPI_Comm comm); // Communicator concerned
```

```
int MPI_Recv(void *buf, // Address of data
             int count, // Maximum number of data allowed
             MPI_Datatype datatype, // Type of data
             int source, // Rank of sender
             int tag,    // Message tag
             MPI_Comm comm, // Communicator concerned
             MPI_Status *status); // Status object
```

- `MPI_Send()` may or may not block
- No *overtaking*
- Sender's rank replaceable by `MPI_ANY_SOURCE`
- Tag: differentiate messages (or `MPI_ANY_TAG`)
- Investigating message beforehand : `MPI_Probe()`

Message Passing Status

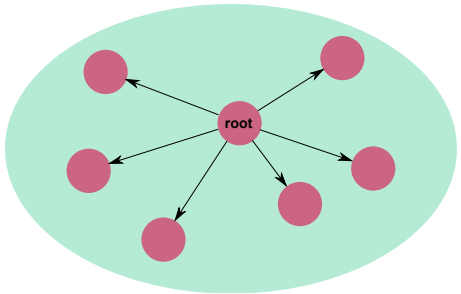
```
int MPI_Get_count(MPI_Status *status,    // Status object of exchange
                  MPI_Datatype datatype, // Type of data
                  int *count)            // Number of data received
```

- Type `MPI_Status` has fields:
  - `MPI_SOURCE`: Rank of the sender
  - `MPI_TAG`: Tag of the message
  - `MPI_ERROR`: error raised during the exchange

MPI Data types

MPI data type	C data type
MPI_BYTE	Unstructured data
MPI_CHAR	char
MPI_UNSIGNED_CHAR	unsigned char
MPI_SHORT	short int
MPI_UNSIGNED_SHORT	unsigned short int
MPI_INT	int
MPI_UNSIGNED	unsigned int
MPI_LONG	long int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_PACKED	

Broadcasting

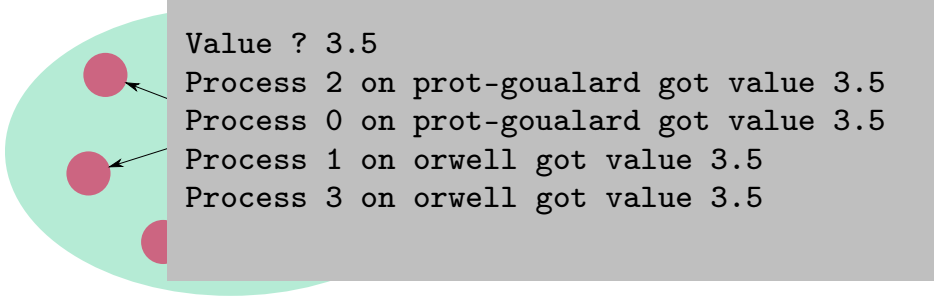



```
#include <iostream>
#include <boost/format.hpp>
#include <mpi.h>

using namespace std;

int main(int argc, char *argv[]) {
  int numprocs, rank, namelen;
  char procname[MPI_MAX_PROCESSOR_NAME];
  double value;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(procname,
                         &namelen);

  if (rank == 0) {
    cout << "Value ? ";
    cin >> value;
  }

  // Barrier on Broadcast
  MPI_Bcast(&value,1,MPI_DOUBLE,0,
            MPI_COMM_WORLD);

  cout << boost::format("Process %1% on "
                        "%2% got value %3%")
    % rank % procname
    % value << endl;
  MPI_Finalize();
}
```

```
int MPI_Bcast( void *buffer, int count, MPI_Datatype datatype, int root,
               MPI_Comm comm)
```

- "Output" for `root` and "input" for other processes

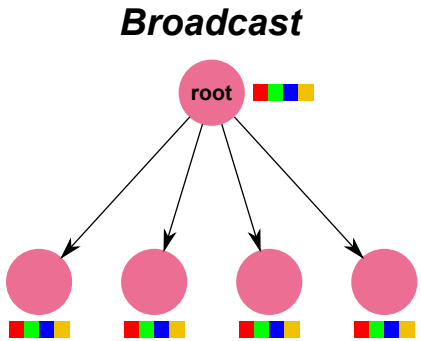
Broadcasting

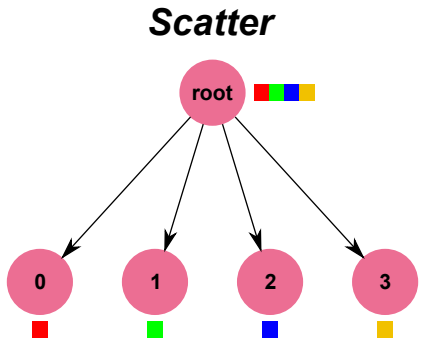



```
#include <iostream>
#include <boost/format.hpp>
#include <mpi.h>

using namespace std;

int main(int argc, char *argv[]) {
  int numprocs, rank, namelen;
  char procname[MPI_MAX_PROCESSOR_NAME];
  double value;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(procname,
                         &namelen);

  if (rank == 0) {
    cout << "Value ? ";
    cin >> value;
  }

  // Barrier on Broadcast
  MPI_Bcast(&value,1,MPI_DOUBLE,0,
            MPI_COMM_WORLD);

  cout << boost::format("Process %1% on "
                        "%2% got value %3%")
    % rank % procname
    % value << endl;
  MPI_Finalize();
}
```

- "Output" for `root` and "input" for other processes

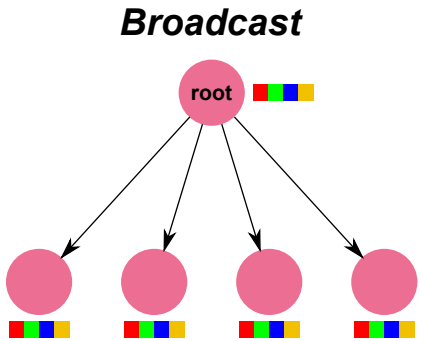
Scattering



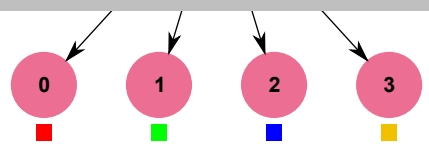



```
#include <iostream>
#include <boost/format.hpp>
#include <mpi.h>

using namespace std;

int main(int argc, char *argv[]) {
  int numprocs, rank, namelen;
  char procname[MPI_MAX_PROCESSOR_NAME];
  const int SZ = 8;
  double array[SZ] { 3.5, 1.25, 5.75, 8.0,
      2.75, 1.5, -1.0, 9.0 };

  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD,
                &numprocs);

  const int localSZ = SZ/numprocs;
  double myslice[localSZ];

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(procname,
                         &namelen);
  // Barrier on Scatter
  MPI_Scatter(array,localSZ,MPI_DOUBLE,
              myslice, localSZ, MPI_DOUBLE,
              0, MPI_COMM_WORLD);

  cout << boost::format("Process %1% on "
                        "%2% got values: ")
    % rank % procname;
  for (int i = 0; i < localSZ; ++i) {
    cout << myslice[i] << " ";
  }
  cout << endl;
  MPI_Finalize();
}
```

Scattering

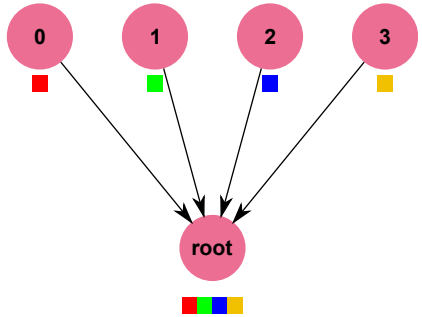



```
Process 2 on prot-goualard got values: 2.75 1.5
Process 1 on orwell got values: 5.75 8
Process 0 on prot-goualard got values: 3.5 1.25
Process 3 on orwell got values: -1 9
```



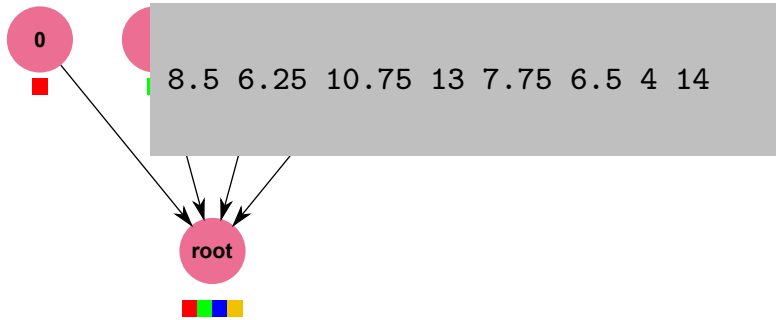

```
#include <iostream>
#include <boost/format.hpp>
#include <mpi.h>

using namespace std;

int main(int argc, char *argv[]) {
  int numprocs, rank, namelen;
  char procname[MPI_MAX_PROCESSOR_NAME];
  const int SZ = 8;
  double array[SZ] { 3.5, 1.25, 5.75, 8.0,
      2.75, 1.5, -1.0, 9.0 };

  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD,
                &numprocs);

  const int localSZ = SZ/numprocs;
  double myslice[localSZ];

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(procname,
                         &namelen);
  // Barrier on Scatter
  MPI_Scatter(array,localSZ,MPI_DOUBLE,
              myslice, localSZ, MPI_DOUBLE,
              0, MPI_COMM_WORLD);

  cout << boost::format("Process %1% on "
                        "%2% got values: ")
    % rank % procname;
  for (int i = 0; i < localSZ; ++i) {
    cout << myslice[i] << " ";
  }
  cout << endl;
  MPI_Finalize();
}
```

Gathering




```
#include <iostream>
#include <boost/format.hpp>
#include <mpi.h>

using namespace std;

int main(int argc, char *argv[]) {
  int numprocs, rank, namelen;
  char procname[MPI_MAX_PROCESSOR_NAME];
  const int SZ = 8;
  double array[SZ] { 3.5, 1.25, 5.75, 8.0,
      2.75, 1.5, -1.0, 9.0 };

  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD,
                &numprocs);

  const int localSZ = SZ/numprocs;
  double myslice[localSZ];

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(procname,
                         &namelen);

  // Barrier on Scatter
  MPI_Scatter(array,localSZ,MPI_DOUBLE,
              myslice, localSZ, MPI_DOUBLE,
              0, MPI_COMM_WORLD);

  for (int i = 0; i < localSZ; ++i) {
    myslice[i] += 5.0;
  }
  // Barrier on Gather
  MPI_Gather(myslice, localSZ, MPI_DOUBLE,
             array, localSZ, MPI_DOUBLE,
             0, MPI_COMM_WORLD);

  if (rank == 0) {
    for (int i = 0; i < SZ; ++i) {
      cout << array[i] << " ";
    }
    cout << endl;
  }

  MPI_Finalize();
}
```

Gathering




```
#include <iostream>
#include <boost/format.hpp>
#include <mpi.h>

using namespace std;

int main(int argc, char *argv[]) {
  int numprocs, rank, namelen;
  char procname[MPI_MAX_PROCESSOR_NAME];
  const int SZ = 8;
  double array[SZ] { 3.5, 1.25, 5.75, 8.0,
      2.75, 1.5, -1.0, 9.0 };

  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD,
                &numprocs);

  const int localSZ = SZ/numprocs;
  double myslice[localSZ];

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(procname,
                         &namelen);

  // Barrier on Scatter
  MPI_Scatter(array,localSZ,MPI_DOUBLE,
              myslice, localSZ, MPI_DOUBLE,
              0, MPI_COMM_WORLD);

  for (int i = 0; i < localSZ; ++i) {
    myslice[i] += 5.0;
  }
  // Barrier on Gather
  MPI_Gather(myslice, localSZ, MPI_DOUBLE,
             array, localSZ, MPI_DOUBLE,
             0, MPI_COMM_WORLD);

  if (rank == 0) {
    for (int i = 0; i < SZ; ++i) {
      cout << array[i] << " ";
    }
    cout << endl;
  }

  MPI_Finalize();
}
```

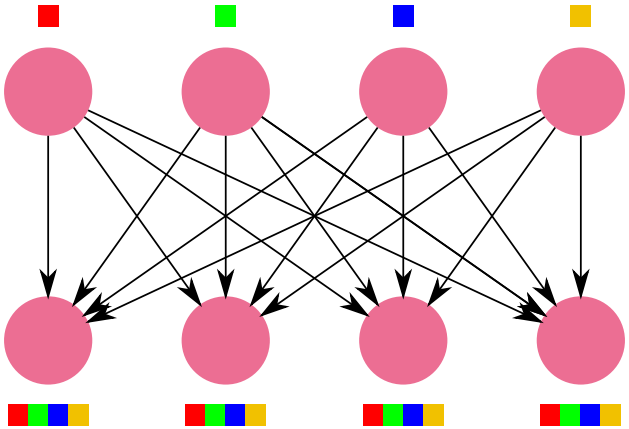
Signatures for Collective Communications

```
int MPI_Bcast(void *buffer, // Address of message
              int count,    // Number of values in message
              MPI_Datatype datatype, // Type of message values
              int root, // Rank of broadcaster
              MPI_Comm comm);
```

```
int MPI_Scatter(void *sendbuf, // Address of message to send
                int sendcnt,   // Number of values to send to each process
                MPI_Datatype sendtype, // Type of values sent
                void *recvbuf, // Address of buffer receiving message
                int recvcnt,   // Number of elements to receive
                MPI_Datatype recvtype, // Type of values received
                int root, // Rank of scatterer
                MPI_Comm comm);
```

```
int MPI_Gather(void *sendbuf, // Address of message to send
               int sendcnt, // Number of values in sendbuf
               MPI_Datatype sendtype, // Type of values sent
               void *recvbuf, // Address of buffer receiving all messages
               int recvcnt, // Number of elements to receive from each process
               MPI_Datatype recvtype, // Type of values received
               int root, // Rank of gatherer
               MPI_Comm comm)
```

Many-to-Many Communication

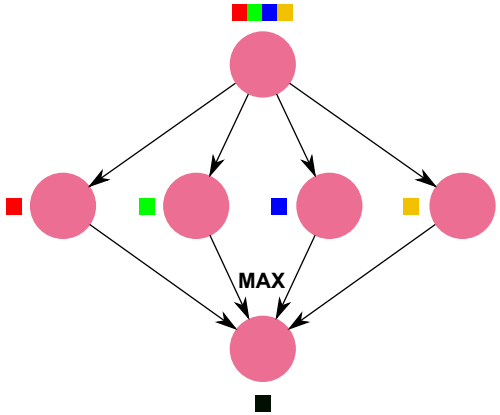


```
int MPI_Allgather(void *sendbuf, // Address of message to send
                  int sendcount, // Number of values in sendbuf
                  MPI_Datatype sendtype, // Type of values to send
                  void *recvbuf, // Address of buffer receiving message
                  int recvcount, // Number of values received from each process
                  MPI_Datatype recvtype, // Type of values received
                  MPI_Comm comm);
```

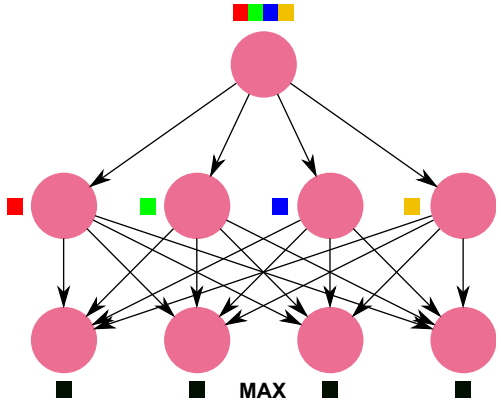
- `MPI_Allgather()` functionally equivalent to `MPI_Gather()` followed by `MPI_Bcast()`

Reductions

MPI_Reduce()



MPI_Allreduce()




```
#include <iostream>
#include <mpi.h>

using namespace std;

int main(int argc, char *argv[]) {
  int numprocs, rank, namelen;
  char procname[MPI_MAX_PROCESSOR_NAME];
  const int SZ = 8;
  double array[SZ] { 3.5, 1.25, 5.75, 8.0,
      2.75, 1.5, -1.0, 9.0 };

  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD,
                &numprocs);
  const int localSZ = SZ/numprocs;
  double myslice[localSZ];
  double local_maximum;
  double maximum;

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);   }
  MPI_Get_processor_name(procname, &namelen);
```

```
// Barrier on Scatter
MPI_Scatter(array,localSZ,MPI_DOUBLE,
            myslice, localSZ, MPI_DOUBLE,
            0, MPI_COMM_WORLD);
// Finding local maximum
local_maximum = myslice[0];
for (int i = 1; i < localSZ; ++i) {
  if (local_maximum < myslice[i]) {
    local_maximum = myslice[i];
  }
}
// Combining local maximums
MPI_Reduce(&local_maximum, &maximum, 1,
           MPI_DOUBLE,
           MPI_MAX, 0, MPI_COMM_WORLD);
if (rank == 0) {
  cout << "Maximum: " << maximum << endl;
}

MPI_Finalize();
```

Reduction Operators

Operator	Meaning
`MPI_MAX`	Maximum
`MPI_MIN`	Minimum
`MPI_SUM`	Sum
`MPI_PROD`	Product
`MPI_LAND`	Logical and
`MPI_BAND`	Bitwise and
`MPI_LOR`	Logical or
`MPI_BOR`	Bitwise or
`MPI_LXOR`	Logical xor
`MPI_BXOR`	Bitwise xor

Data Communication

- Exchange of primitive types or arrays of primitive types
- Aggregation of values of homogeneous or heterogeneous types (for better performances):
  - Packed types (`MPI_Pack()`, `MPI_Unpack()`)
  - Derived datatype (`MPI_Get_address()`, `MPI_Type_create_struct()`)

Non-blocking communications

MPI offers asynchronous routines:

- `MPI_Isend()`/`MPI_Irecv()`
- `MPI_Wait()`

```
#include <iostream>
#include <boost/format.hpp>
#include <mpi.h>
#include <sys/types.h>
#include <unistd.h>

using namespace std;

int main(int argc, char *argv[]) {
  int numprocs, rank, msgOUT, msgIN;
  MPI_Request req;
  MPI_Status status;

  srand(getpid());

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD,
                &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,
                &rank);

  msgOUT = rand() % 100;
  cout << boost::format("Process %1% "
                        "has value %2%")
    % rank % msgOUT << endl;
  MPI_Barrier(MPI_COMM_WORLD);
```

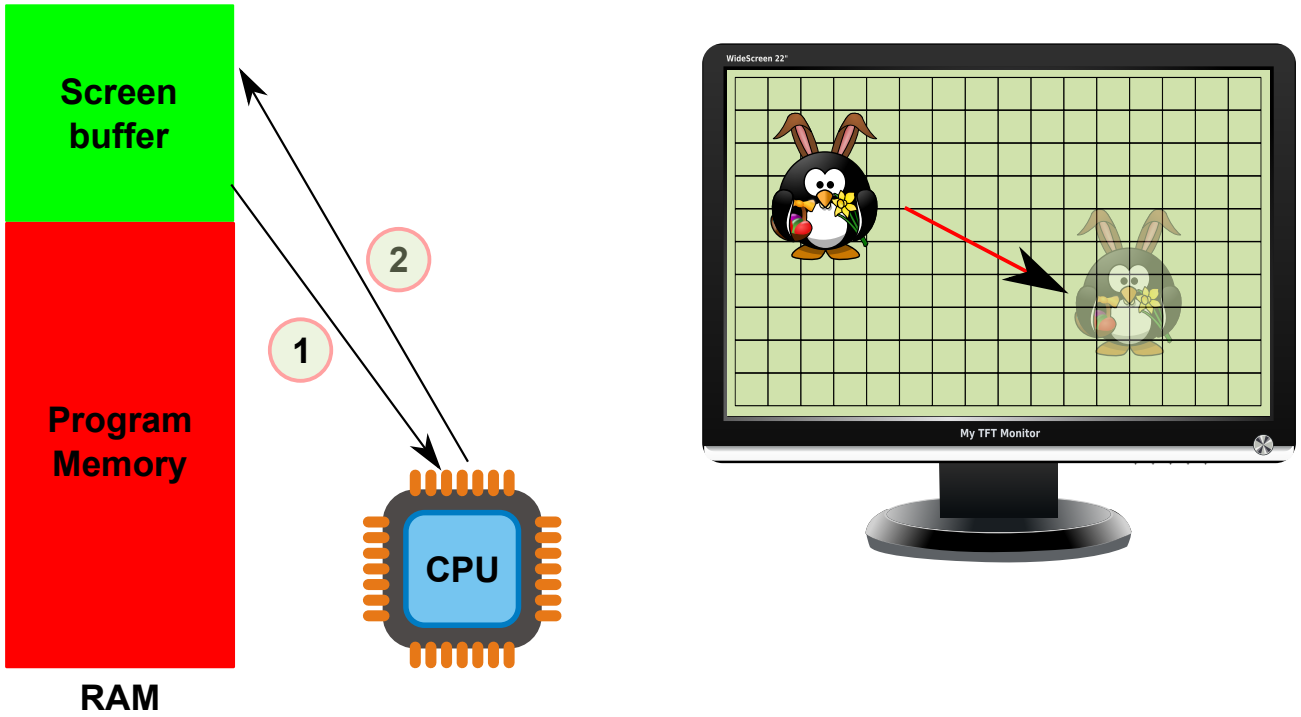
```
  // Each even process exchanges its value
  // with the next one
  // Assuming an even number of processes
  if (rank % 2 == 0) {
    MPI_Isend(&msgOUT,1,MPI_INT,
              rank+1,0,MPI_COMM_WORLD,&req);
    MPI_Irecv(&msgIN,1,MPI_INT,
              rank+1,0,MPI_COMM_WORLD,&req);
  } else {
    MPI_Isend(&msgOUT,1,MPI_INT,
              rank-1,0,MPI_COMM_WORLD,&req);
    MPI_Irecv(&msgIN,1,MPI_INT,
              rank-1,0,MPI_COMM_WORLD,&req);
  }
  MPI_Wait(&req,&status);

  if (rank % 2 == 0) {
    cout << boost::format("Process %1% and "
                          "%2% exchanged "
                          "%3% and %4%")
      % rank % (rank+1) % msgOUT
      % msgIN << endl;
  }
  MPI_Finalize();
}
```

Going Further with MPI

- Six/seven routines enough for most applications
- More than 100 routines
- Refer to documentation (links on *madoc*)

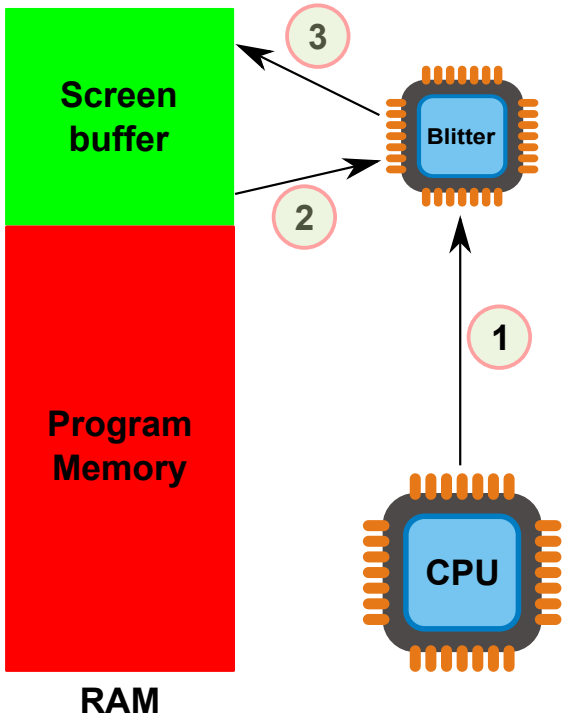
General-Purpose computing on Graphics Processing Units (GPGPU)

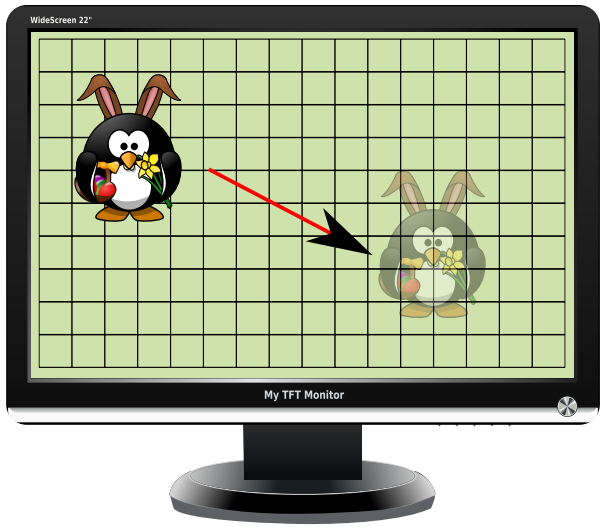
The Graphics Processing Unit (GPU)



- Heavy use of CPU for image rendering

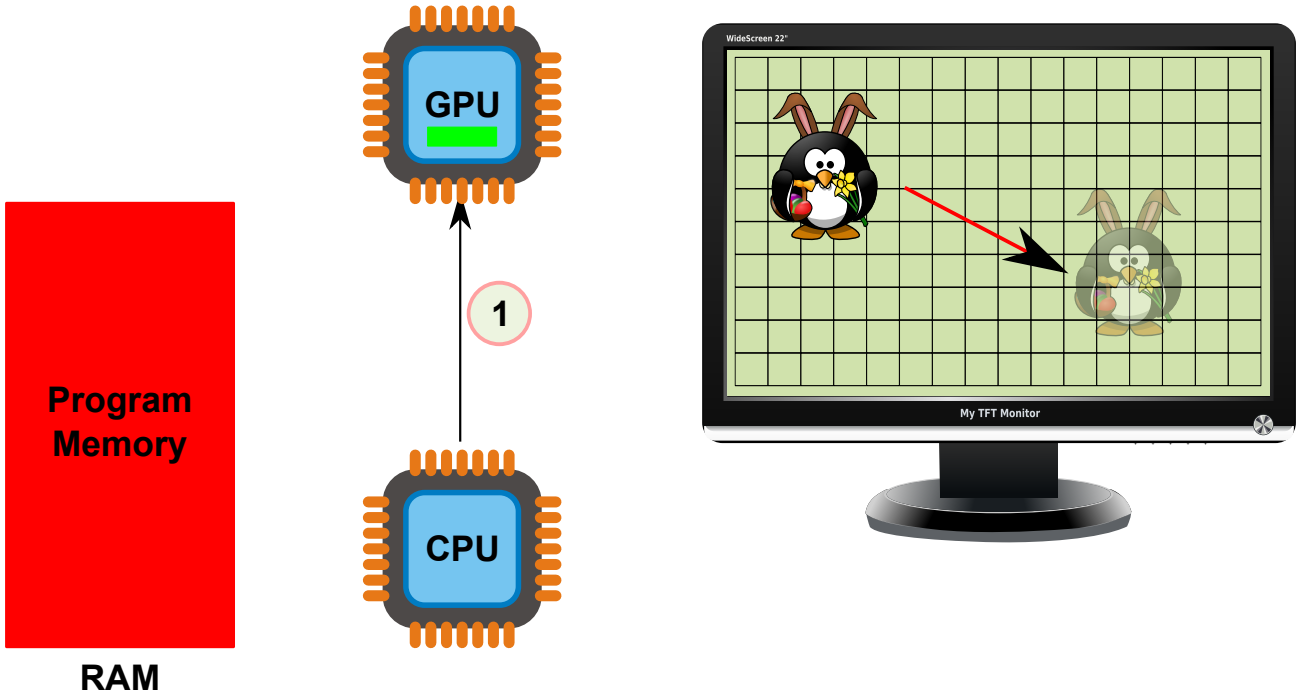
The Graphics Processing Unit (GPU)





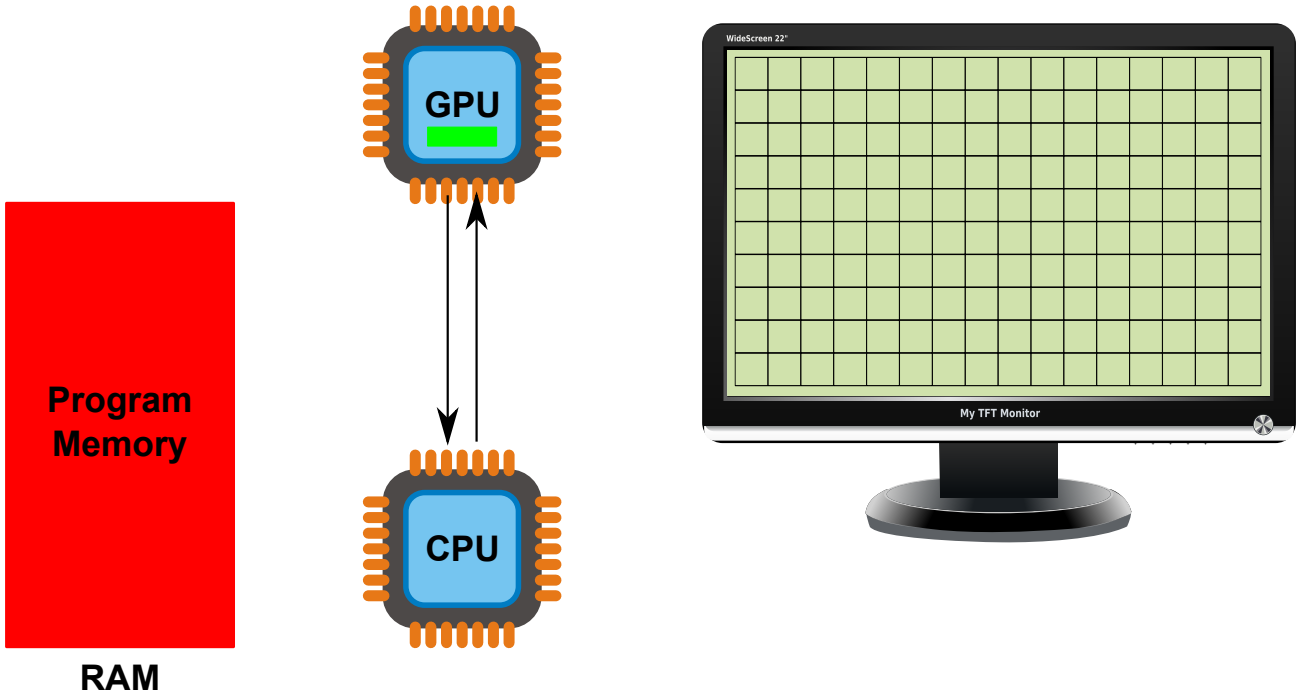
- Data movement by *Blitter*

The Graphics Processing Unit (GPU)



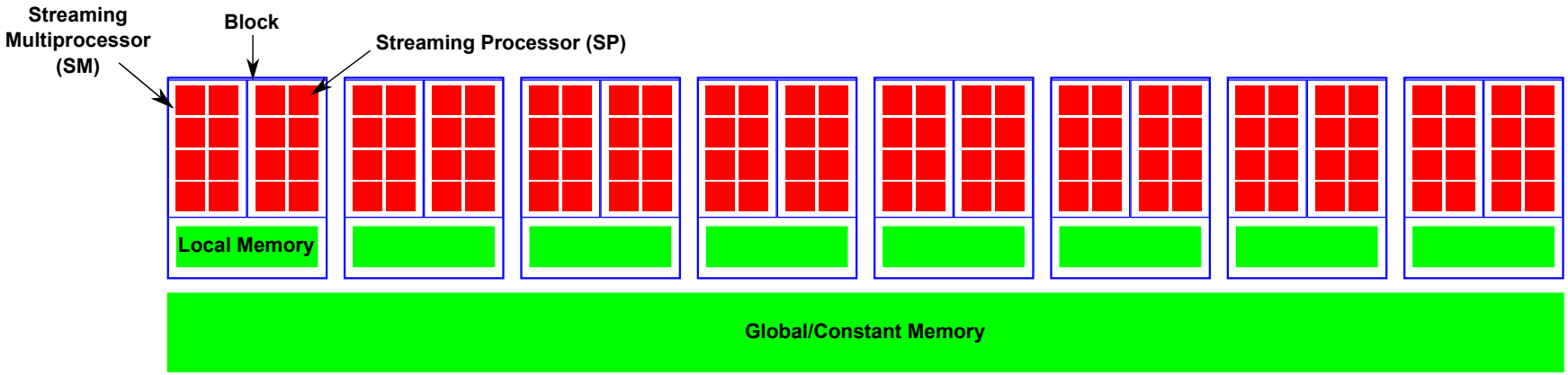
- Image computation by GPU (vector graphics + rasterization)
- 2D and 3D acceleration

The Graphics Processing Unit (GPU)



- GPGPU: GPU used as external computing device

Inside a modern GPU (1)

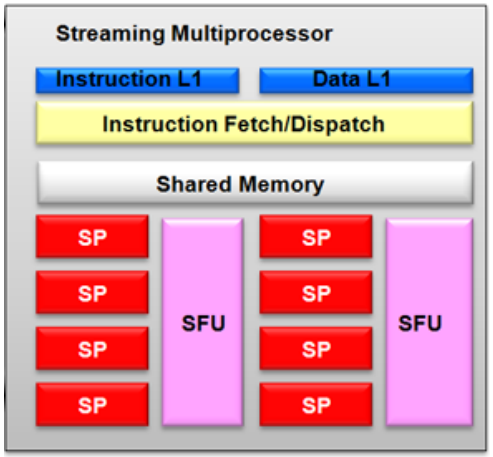


Modern GPUs: many-core chips

- Massive data-parallelism
- Task parallelism (to some extent)
- Many *very cheap* threads
- Streaming processors: applying same operations on each pixel of an image (SIMD)

Inside a modern GPU (2)

- Streaming Multiprocessor:



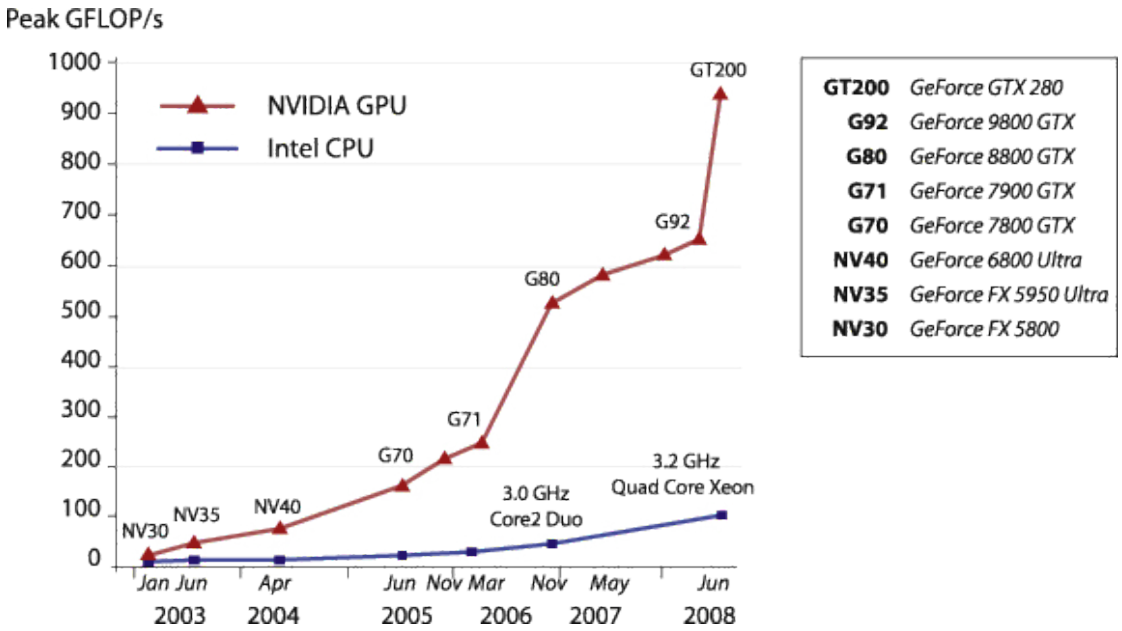
Source: http://media.bestofmicro.com/Q/Z/111131/original/020.png

- Streaming Processor: *Single Instruction Multiple Threads* (SIMT)
  - MAD and multiplier
  - Execution of same instructions simultaneously on different data
  - At least 4 threads per SP (32 threads per SM of 8 SPs)
- Execution of threads in *warps* of 32 threads per SM
- Global memory latency very high
- Special Function Unit (SFU): computation of transcendental functions and such

GPGPU?

General-Purpose computing on Graphics Processing Unit

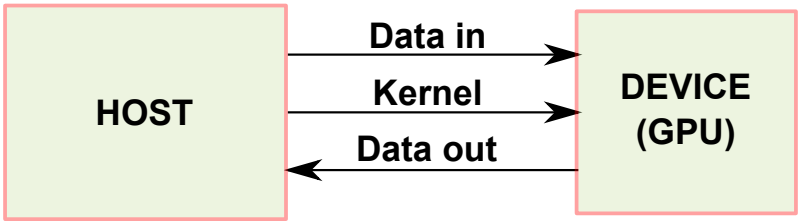
- Manipulation of images embarrassingly parallel
- Graphics Cards: dedicated data parallel hardware
- Strong economic incentive for more powerful GPUs: games



History of GPGPU:

- 2002: abuse of GPUs to perform non graphics computation
- 2006: CUDA for GPGPU on NVIDIA GPUs
- 2008: OpenCL for programming CPUs, GPUs, DSPs, ...
- 2008: Microsoft DirectCompute and C++AMP, for MS Windows

Computing with a GPU



1. Initialize Graphics Card
2. Allocate memory in host and device
3. Copy data from host to device memory
4. Launch *kernel* on device
5. Copy data from device to host memory
6. Deallocate memory

Analogy with MPI (cluster of cores on a single computer)

NVIDIA CUDA *vs.* OpenCL

NVIDIA:

- ☺ More mature
- ☺ Offers some features tailored specifically to NVIDIA GPUs for maximum performances
- ☺ More libraries (cudpp)
- ☹ Limited to NVIDIA GPUs

OpenCL:

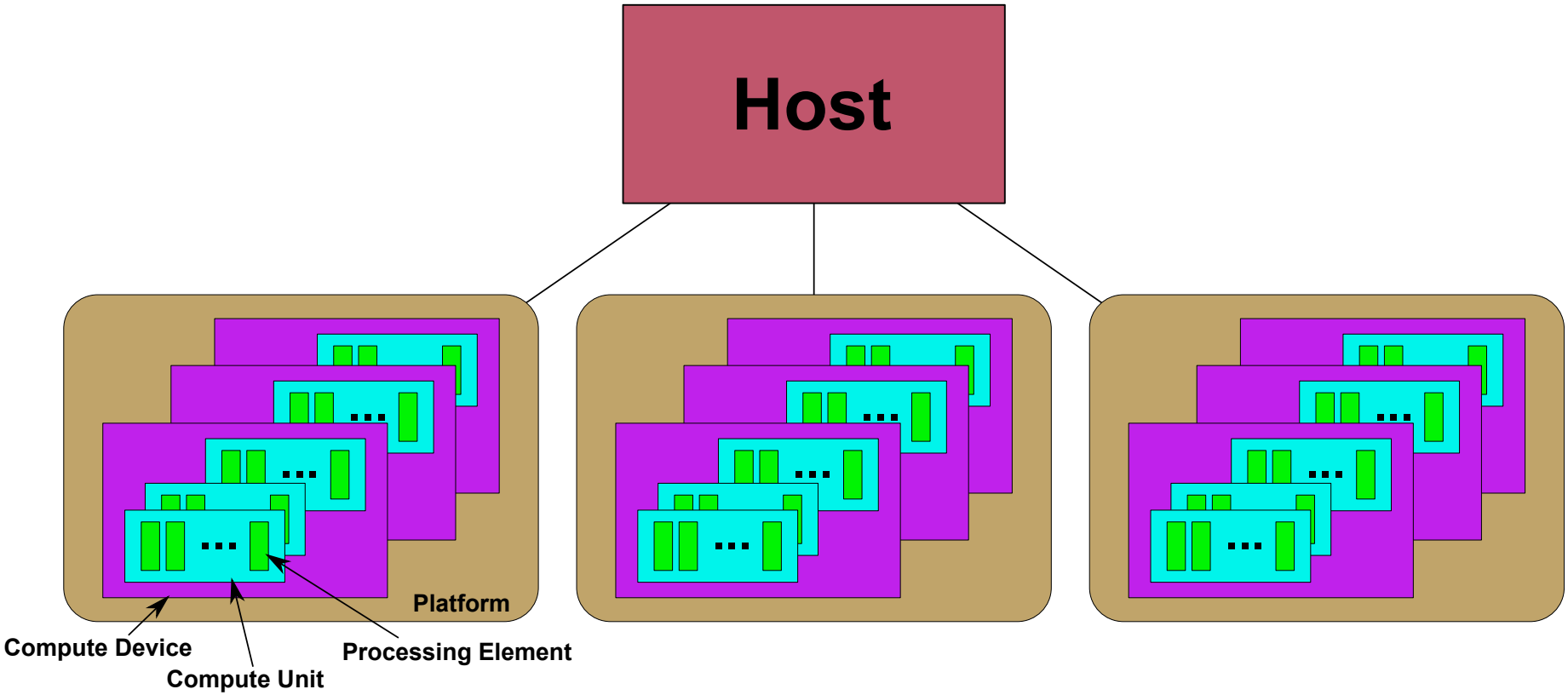
- ☹ Rougher around the edges (but improving)
- ☺ Open standard available for many *devices*:
  - CPUs (SIMD SSE instructions)
  - GPUs from several vendors
  - Hardware accelerators
  - ...
- ☺ Allows heterogeneous parallel computing (CPU+GPU)

- Both frameworks are here to stay
- For this course: *exclusive focus on OpenCL*
- OpenCL knowledge easily reusable to learn CUDA

OpenCL

- *Open Computing Language*
- Non-proprietary standard initiated by Apple, Inc. in 2008
- C/C++ framework to program CPUs, GPUs, and more from various vendors
- OpenCL implementation:
  - *Installable Client Driver* (ICD): interface to devices from a vendor
  - ICD Loader: agnostic library to load the ICD of any vendor

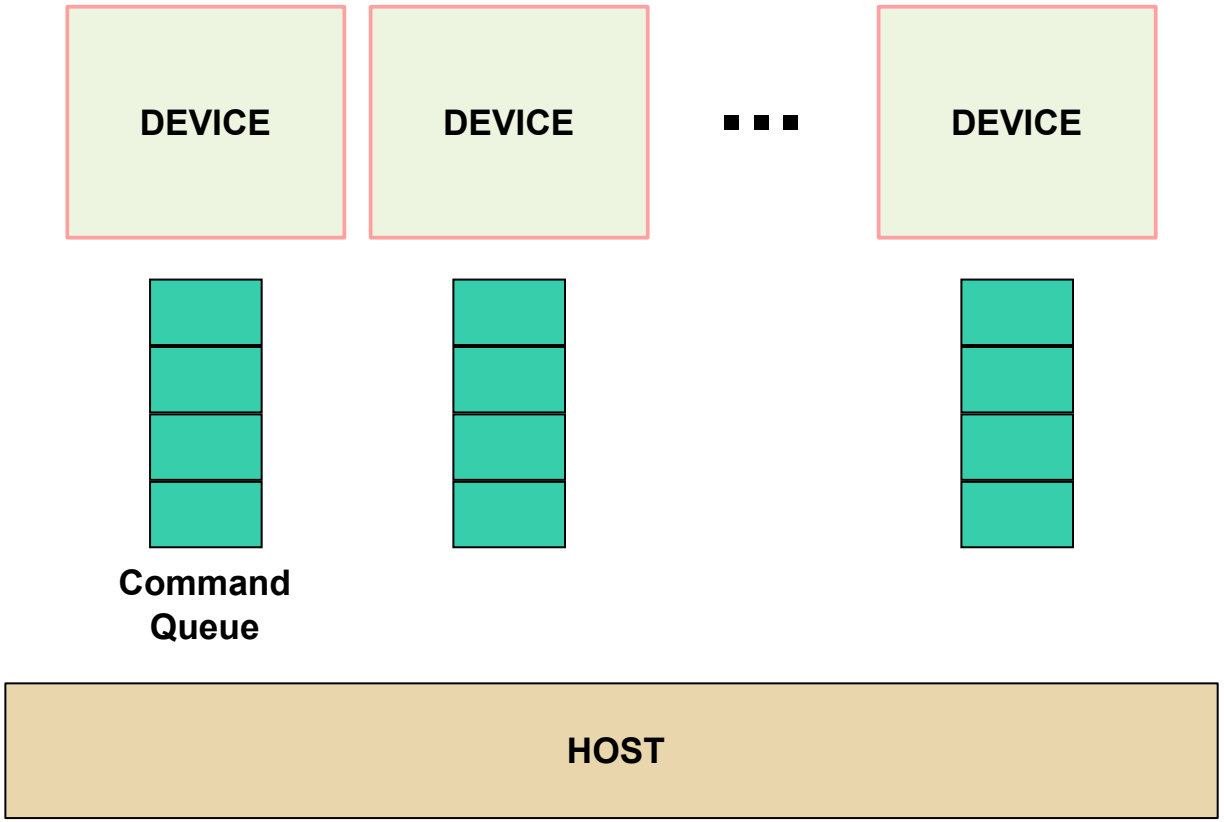
OpenCL Platform Model



GPU example:

- Host: CPU
- Platform: GPU Vendor OpenCL implementation
- Compute Device: GPU
- Compute Unit: Streaming multiprocessor
- Processing Element: Streaming processor

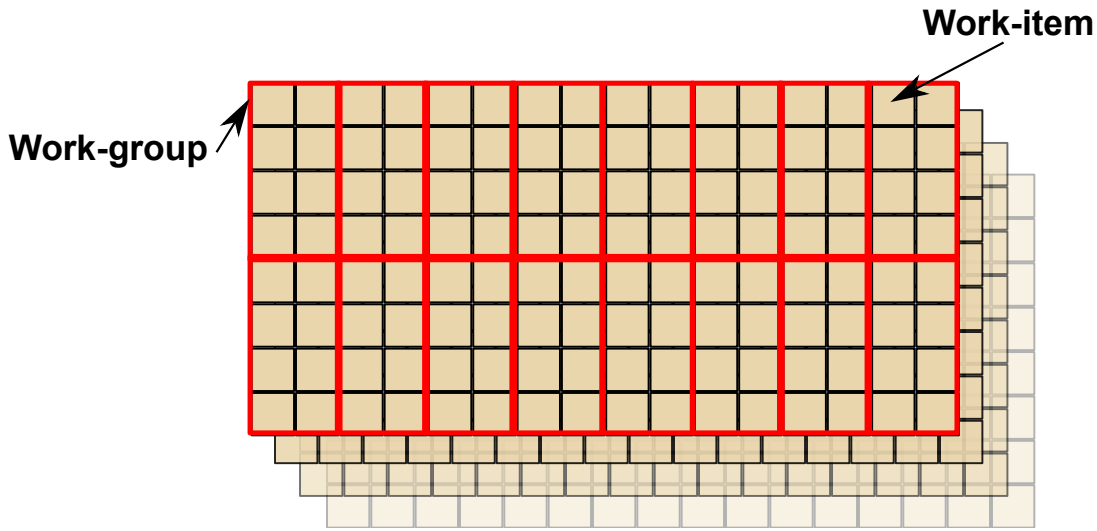
OpenCL Execution Model



- Code and data exchanged through a queue (in-order or out-of-order)
- Task parallelism through use of multiple queues

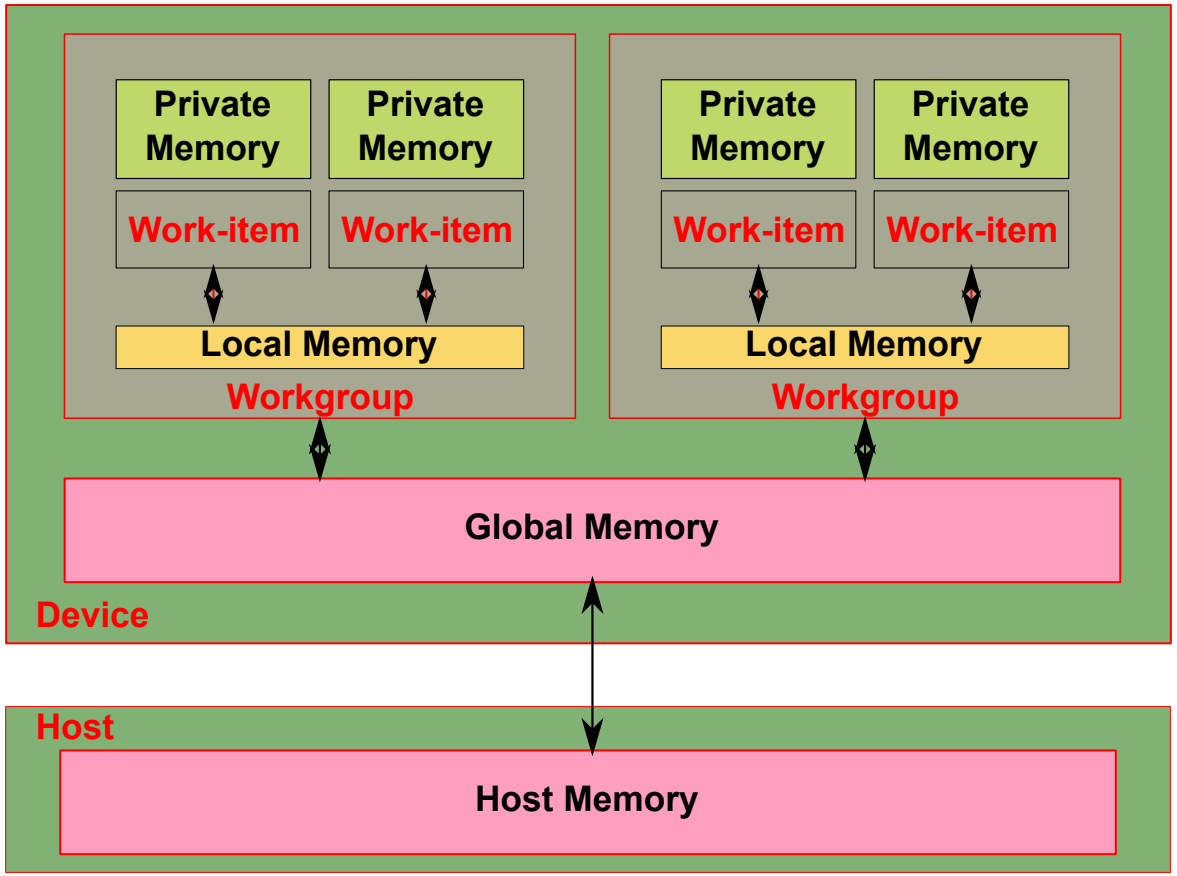
Partitioning the work

- Kernel executed across a global domain of *work-items*
- Work-items grouped into local *work-groups*
- Synchronization between work-items in the same work-group only (barriers, memory fences)
- 1D, 2D, 3D domains



- Work-groups must be independent from each others (scaling & scheduling)
- Calls to OpenCL to know the best partition depending on device used

OpenCL Memory Model



Explicit memory management:

- host $\leftrightarrow$ global memory $\leftrightarrow$ local memory

Memory Consistency

Relaxed consistency memory model:

- Load/store consistency in private memory
- Consistency in local memory across work-items in a workgroup at a barrier (but not otherwise)
- Consistency in global memory across workgroups at a barrier (but not otherwise)

OpenCL Parlance

- Platform. OpenCL implementation (e.g., NVIDIA SDK, AMD SDK, . . .)
- Host. System executing C/C++ code sending kernels to devices
- Device. Receives tasks and data from host and processes them (e.g., CPU, GPU, . . .)
- Program. Collection of kernels and other functions
- Kernel. Code that is executed on a *device*
- Context. Set of devices on the same platform working together and sharing memory
- Work-group. Set of work-items assigned to a Compute Unit (SM)
- Work-item. Unit of concurrent execution (thread) on a Streaming Processor

Programming with OpenCL

1. Choose one or several platforms depending on availability and possibilities
2. Choose one or several devices depending on availability and possibilities
3. Create a context
4. Create command queues (per device)
5. Compile programs
6. Create kernels
7. Allocate memory on devices
8. Transfer data to devices
9. Execute
10. Transfer results back to host
11. Free memory on devices

Querying Platform Information

```
#include <iostream>
#include <CL/cl.h>
using namespace std;

char *cl_get_platform_info(cl_platform_id platform, cl_platform_info param_name) {
  char *info;
  size_t sz;
  clGetPlatformInfo(platform,param_name,0,nullptr,&sz);
  info = new char[sz];
  clGetPlatformInfo(platform,param_name,sz,info,nullptr);
  return info;
}

int main(void) {
  cl_platform_id *platforms;
  cl_uint num_platforms;

  clGetPlatformIDs(0,nullptr,&num_platforms);
  platforms = new cl_platform_id[num_platforms];
  clGetPlatformIDs(num_platforms,platforms,nullptr);

  for (cl_uint i = 0; i < num_platforms; ++i) {
    char *name, *vendor, *profile, *exts;
    name = cl_get_platform_info(platforms[i], CL_PLATFORM_NAME);
    vendor = cl_get_platform_info(platforms[i], CL_PLATFORM_VENDOR);
    profile = cl_get_platform_info(platforms[i], CL_PLATFORM_PROFILE);
    exts = cl_get_platform_info(platforms[i], CL_PLATFORM_EXTENSIONS);
    cout << name << "\n" << vendor << "\n" << profile << endl;
    cout << exts << "\n" << endl;
    delete[] name; delete[] vendor; delete[] profile; delete[] exts;
  }
}
```

Querying Platform Information

```
#include <iost
#include <CL/
using namespa

char *cl_get_
  char *info;
  size_t sz;
  clGetPlatfor
  info = new
  clGetPlatfor
  return info
}

int main(void
  cl_platform_
  cl_uint num_platforms;

  clGetPlatformIDs(0,nullptr,&num_platforms);
  platforms = new cl_platform_id[num_platforms];
  clGetPlatformIDs(num_platforms,platforms,nullptr);

  for (cl_uint i = 0; i < num_platforms; ++i) {
    char *name, *vendor, *profile, *exts;
    name = cl_get_platform_info(platforms[i], CL_PLATFORM_NAME);
    vendor = cl_get_platform_info(platforms[i], CL_PLATFORM_VENDOR);
    profile = cl_get_platform_info(platforms[i], CL_PLATFORM_PROFILE);
    exts = cl_get_platform_info(platforms[i], CL_PLATFORM_EXTENSIONS);
    cout << name << "\n" << vendor << "\n" << profile << endl;
    cout << exts << "\n" << endl;
    delete[] name; delete[] vendor; delete[] profile; delete[] exts;
  }
}
```

```
Intel(R) OpenCL
Intel(R) Corporation
FULL_PROFILE
cl_khr_icd cl_khr_global_int32_base_atomics cl_khr_global_int32_extended_atomics
cl_khr_local_int32_base_atomics cl_khr_local_int32_extended_atomics
cl_khr_byte_addressable_store cl_khr_spir cl_intel_exec_by_local_thread cl_khr_fp64

NVIDIA CUDA
NVIDIA Corporation
FULL_PROFILE
cl_khr_byte_addressable_store cl_khr_icd cl_khr_gl_sharing cl_nv_compiler_options
cl_nv_device_attribute_query cl_nv_pragma_unroll
```

Querying Device Information

```
#include <iostream>
#include <CL/cl.h>
using namespace std;

char *device_info(cl_device_id device,
          cl_device_info param_name) {
  char *info;
  size_t sz;
  clGetDeviceInfo(device,param_name,0,
                  nullptr,&sz);
  info = new char[sz];
  clGetDeviceInfo(device,param_name,sz,
                  info,nullptr);
  return info;
}

int main(void) {
  // [...]
  for (cl_uint i = 0; i < nplatforms; ++i) {
    char *name, *vendor, *profile, *exts;
    // [Get platform information for each
//  platform available]
// Displaying devices of each platform
cl_device_id *devices;
cl_uint num_devices;
clGetDeviceIDs(platforms[i],
                 CL_DEVICE_TYPE_ALL,0,
                 nullptr,&num_devices);
devices = new cl_device_id[num_devices];
clGetDeviceIDs(platforms[i],
        CL_DEVICE_TYPE_ALL,
        num_devices,devices,nullptr);
for (cl_uint j = 0; j < num_devices; ++j)
  char *name = device_info(devices[j],
               CL_DEVICE_NAME);
  char *vendor = device_info(devices[j],
                CL_DEVICE_VENDOR);
}
}
```

Querying Device Information

```
#inclu
#inclu
using

char *

  char
  size
  clGe

  info
  clGe

  retu
}

int ma
  // [
  for
    ch
    //
```

```
Intel(R) OpenCL
Intel(R) Corporation
FULL_PROFILE
cl_khr_icd cl_khr_global_int32_base_atomics cl_khr_global_int32_extended_atomics
cl_khr_local_int32_base_atomics cl_khr_local_int32_extended_atomics
cl_khr_byte_addressable_store cl_khr_spir cl_intel_exec_by_local_thread cl_khr_fp64
Devices: 1
      Intel(R) Core(TM) i7-2630QM CPU @ 2.00GHz
Intel(R) Corporation
cl_khr_icd cl_khr_global_int32_base_atomics cl_khr_global_int32_extended_atomics
cl_khr_local_int32_base_atomics cl_khr_local_int32_extended_atomics
cl_khr_byte_addressable_store cl_khr_spir cl_intel_exec_by_local_thread cl_khr_fp64
8348508160
64
1
1

NVIDIA CUDA
NVIDIA Corporation
FULL_PROFILE
cl_khr_byte_addressable_store cl_khr_icd cl_khr_gl_sharing cl_nv_compiler_options
cl_nv_device_attribute_query cl_nv_pragma_unroll
Devices: 1
GeForce GTX 460M
NVIDIA Corporation
cl_khr_byte_addressable_store cl_khr_icd cl_khr_gl_sharing cl_nv_compiler_options
cl_nv_device_attribute_query cl_nv_pragma_unroll  cl_khr_global_int32_base_atomics
cl_khr_global_int32_extended_atomics cl_khr_local_int32_base_atomics
cl_khr_local_int32_extended_atomics cl_khr_fp64
1609760768
32
1
1
```

Device types and parameters

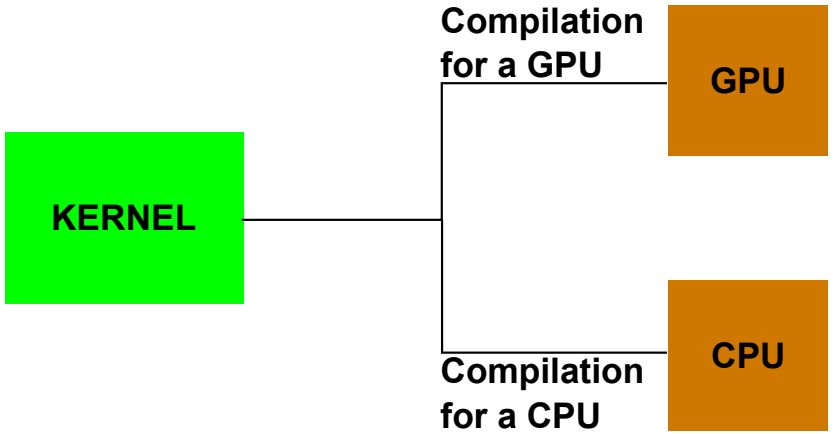
Device type	Meaning
`CL_DEVICE_TYPE_ALL`	All devices of a platform
`CL_DEVICE_TYPE_DEFAULT`	Platform's default type
`CL_DEVICE_TYPE_CPU`	Host processor
`CL_DEVICE_TYPE_GPU`	Graphics processor
`CL_DEVICE_TYPE_ACCELERATOR`	External device

Parameter	Type	Purpose
`CL_DEVICE_NAME`	`char[]`	Name of device
`CL_DEVICE_VENDOR`	`char[]`	Device's vendor
`CL_DEVICE_EXTENSIONS`	`char[]`	Supported extensions
`CL_DEVICE_GLOBAL_MEM_SIZE`	`cl_ulong`	Size of device's global mem.
`CL_DEVICE_ADDRESS_BITS`	`cl_uint`	Size of device's address space
`CL_DEVICE_AVAILABLE`	`cl_bool`	Is the device available?
`CL_DEVICE_COMPILER_AVAILABLE`	`cl_bool`	Is there a compiler for the device?

`cl_ulong`, `cl_uint`, ...: portable data types

Executing kernels

```
__kernel void vecadd(__global float* vecA,
                     __global float* vecB,
                     __global float* result)
{
  int i = get_global_id(0);
  result[i] = vecA[i] + vecB[i];
}
```



- Set kernel arguments (`clSetKernelArg()`)
- Enqueue the kernel in the command queue (`clEnqueueNDRangeKernel()`)

OpenCL example: adding two vectors

- The kernel (File `vecadd.cl`):

```
__kernel void vecadd(__global float* vecA,
                     __global float* vecB,
                     __global float* result)
{
  int i = get_global_id(0);
  result[i] = vecA[i] + vecB[i];
}
```

OpenCL example: adding two vectors

```
#include <iostream>
#include <string>
#include <fstream>
#include <streambuf>
#include <stdexcept>
#include <CL/cl.h>

using namespace std;

int main(void)
{
  try {
    cl_platform_id *platforms;
    cl_uint nplatforms;
    cl_int err;

    const size_t size = 2;
    float vec1[size] = { 3.0f, 2.0f};
    float vec2[size] = {-7.0f, 8.0f};
    float res[size];

    // Detect platforms available
    err = clGetPlatformIDs(0,nullptr,&nplatforms);
    if (err != CL_SUCCESS) throw runtime_error("No platform available!");
    platforms = new cl_platform_id[nplatforms];
    clGetPlatformIDs(nplatforms,platforms,nullptr);
```

OpenCL example: adding two vectors

```
cl_device_id gpu_device;
cl_uint num_gpus;
unsigned int i;
bool gpu_found = false;
// Searching for a platform with a GPU
for (i = 0; i < nplatforms; ++i) {
  err = clGetDeviceIDs(platforms[i],CL_DEVICE_TYPE_GPU,1,nullptr,
                       &num_gpus);
  gpu_found = (err == CL_SUCCESS && num_gpus > 0);
  if (gpu_found) {
    break;
  }
}
if (!gpu_found) throw runtime_error("No GPU available");
clGetDeviceIDs(platforms[i],CL_DEVICE_TYPE_GPU,1,&gpu_device,nullptr);
```

OpenCL example: adding two vectors

```
// Creating the context
cl_context context = clCreateContext(nullptr,1,&gpu_device,nullptr,
                                     nullptr,&err);
if (err != CL_SUCCESS) throw runtime_error("Could not create a context!");
```

OpenCL example: adding two vectors

```
// Creating the program to be run on the GPU
ifstream fic("vecadd.cl");
string program_buffer((std::istreambuf_iterator<char>(fic)),
                      std::istreambuf_iterator<char>());
size_t program_size=program_buffer.size();
const char* buf = program_buffer.c_str();
cl_program program = clCreateProgramWithSource(context,1,
                                                (const char**)(&buf),
                                                &program_size,
                                                &err);
if (err != CL_SUCCESS) throw runtime_error("Could not create a program!");

err = clBuildProgram(program,1,&gpu_device,nullptr,nullptr,nullptr);
if (err != CL_SUCCESS) throw runtime_error("Could not build the program!");

cl_kernel kernel = clCreateKernel(program,"vecadd",&err);
if (err != CL_SUCCESS) {
  throw runtime_error("Could not create the kernel vecadd");
}
```

OpenCL example: adding two vectors

```
// Creating the buffers for exchanging data
cl_mem vec1buf, vec2buf, resbuf;

vec1buf = clCreateBuffer(context,CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                         sizeof(vec1),vec1,&err);
if (err != CL_SUCCESS) {
  throw runtime_error("Could not create a buffer for Vector 1!");
}
vec2buf = clCreateBuffer(context,CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                         sizeof(vec2),vec2,&err);
if (err != CL_SUCCESS) {
  throw runtime_error("Could not create a buffer for Vector 2!");
}
resbuf = clCreateBuffer(context,CL_MEM_WRITE_ONLY,
                        sizeof(res),nullptr,&err);
if (err != CL_SUCCESS) {
  throw runtime_error("Could not create a buffer for the result!");
}

// Creating the argument for the kernel
clSetKernelArg(kernel,0,sizeof(cl_mem),&vec1buf);
clSetKernelArg(kernel,1,sizeof(cl_mem),&vec2buf);
clSetKernelArg(kernel,2,sizeof(cl_mem),&resbuf);
```

OpenCL example: adding two vectors

```
// Creating the command queue for the GPU
cl_command_queue queue = clCreateCommandQueue(context, gpu_device, 0,&err);
if (err != CL_SUCCESS) {
  throw runtime_error("Could not create a command queue!");
}

// 1 dim, 'size' work-items/ 1 workgroup for all
size_t work_size[1] = { size };
// Enqueueing the kernel to execute it on the GPU
//(synchronous or asynchronous---use clFinish() for synchronization)
err = clEnqueueNDRangeKernel(queue,kernel,1,nullptr,
                             work_size,nullptr,0,nullptr,nullptr);
if (err != CL_SUCCESS) {
  throw runtime_error("Could not enqueue the kernel!");
}
// Read the output (synchronous or asynchronous---use clFinish() or events)
err = clEnqueueReadBuffer(queue,resbuf,CL_TRUE,0,sizeof(res),res,0,
                          nullptr,nullptr);
if (err != CL_SUCCESS) {
  throw runtime_error("Could not enqueue the reading command!");
}

// Printing the result
for (size_t i=0; i < size; ++i) {
  cout << res[i] << "\n";
}
```

OpenCL example: adding two vectors

```
    // Freeing all resources
    clReleaseMemObject(vec1buf);
    clReleaseMemObject(vec2buf);
    clReleaseMemObject(resbuf);
    clReleaseKernel(kernel);
    clReleaseCommandQueue(queue);
    clReleaseProgram(program);
    clReleaseContext(context);

} catch (const exception& e) {
    cout << e.what() << endl;
```

Synchronization

- Each queue can execute in-order or out-of-order
- Each device has its queue
- Explicit synchronization between queues
- Synchronization *via* events
- A command accepts as input a list of events and outputs one event
  - Starts executing only when all input events have arrived
  - Sends its event on completion
  - User events (`clCreateUserEvent()`)

Thread computation

- Work-items of a work-group execute concurrently on a SM
- A SM can execute concurrently several work-groups
- A SM executes 32 work-items "doing the same thing" in parallel as a *warp*
- A warp executes one common instruction at a time
- Divergence?

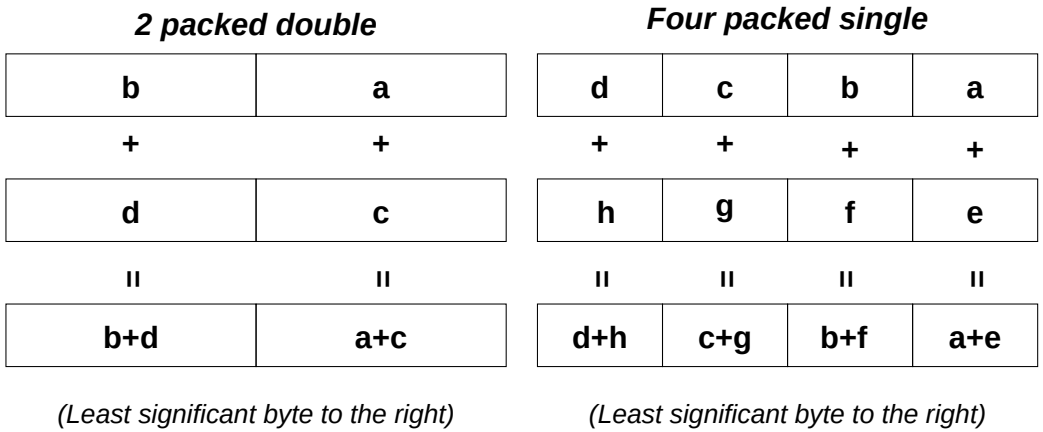
```
if (x < 0.0) {
  y += 3;
} else {
  y *= x;
}
```

- Serial execution of all paths, disabling threads as needed $\Rightarrow$ branches are costly
- Two warps may follow different paths

SIMD *vs.* SIMT

SIMD (e.g., Intel SSE):

- User knows the number of operations performed in parallel
- Code written accordingly



SIMT (e.g., GPU):

- User does not "know/care" for the number of operations executed truly in parallel
- Code written for an ideal processor with infinite cores
- Threads packed in groups and mapped onto cores $\Rightarrow$ great scalability

Using GPGPU

When to use GPGPU:

- Kernels with many threads
- Data exchanges only inside work-groups
- Lot of data-parallelism (few conditionals)
- Huge amount of computation, few memory accesses
- Few synchronizations

End of Lecture