1. C++ BASICS

1.1 Data Types

Type specifier	Equivalent types	Width in bytes		
		C++ standard[2]	Unix / Win 64	Unix / Win 32
bool		1	1	1
char[1]		1	1	1
signed char				
unsigned char				
wchar_t		1	4 / 2	4 / 2
char16_t		2	2	2
char32_t		4	4	4
short	short int	2	2	2
	signed short			
	signed short int			
unsigned short	unsigned short int			
int	signed	2	4	4
	signed int			
unsigned int	unsigned			
long	long int	4	8 / 4	4
	signed long			
	signed long int			
unsigned long	unsigned long int			
long long	long long int	8	8	8
	signed long long			
	signed long long int			
unsigned long long	unsigned long long int			
size_t (unsigned)		2	8	4
float		4	4	4
double		8	8	8

[1] type for character representation which can be most efficiently processed on the target system (equivalent to either signed char or unsigned char)

[2] in the C++ standard "at least" values are specified

Type	Size in bytes	Format	Value range
char	1	signed	-128 to 127
		unsigned	0 to 255
integral	2	signed	-32768 to 32767
		unsigned	0 to 65535
	4	signed	$\pm 2.147 \cdot 10^{9}$
		unsigned	0 to $4.294 \cdot 10^{9}$
	8	signed	$\pm 9.223 \cdot 10^{18}$
		unsigned	0 to $18.446 \cdot 10^{18}$
floating point	4	IEEE-754	$\pm 3.402 \cdot 10^{\pm 38}$ (~ 7 digits)
	8	IEEE-754	$\pm 1.797 \cdot 10^{\pm 308}$ (~15 digits)

1.2 Very Important Commands

Create Assembler code:
```
  g++ -S -std=c++11 -O3 -Wall -lrt -o main.s main.cpp
```

$\frac{d}{dt} \Rightarrow$

2. MULTITHREADING

2.1 Threads and Processes [2b, 2]

Process:
- Execution sequence within the OS, i.e. a program
- Relatively expensive to create
- Independent resources, state (by default)
- Immune to many concurrency issues

Thread:
- Execution sequence within the process
- main() is the first thread
- Cheap to create
- Shared resources, state
- Difficult to use correctly

Pros:	compute faster, unblocking (work is done while waiting for events to occur outside the CPU)
Cons:	synchronisation overhead, programming discipline, harder to reason about, harder to debug

During execution of a multi-threaded program threads get spawned and joined dynamically.

Thread States

Running	Currently executing. #running threads $\leq$ #CPU cores
Ready	Prepared to run whenever processor time can be allocated to it.
Blocked	Paused until some resource (other than processor) is allocated to it.
Terminated	Finished execution but OS resources not yet deallocated.

3. C++11: THREADS

```
g++ -std=c++11 -O3 -Wall -pthread -lrt -o main main.cpp
```

Launch a thread:

```
  std::thread t(function, arg1, arg2, …);
```

Using a C++11 lambda function:

```
  std::thread t( [](){std::cout << “Hello world!\n”;} );
```

Join a thread:

```
  t.join();
```

3.1 Properties of Threads [2b, 10-11]

Movable/non-copyable types
Cannot copy/assign from lvalues:

```
  std::thread x,y; x=y; // error!
```

Can place in containers:

```
  std::vector<std::thread> v(10);
```

Can copy/assign from rvalues:

```
  pool[3] = std::thread(f);
```

Can pass to/return from functions:

```
  std::thread t = make_thread();
  do_something( make_thread() );
```

Can swap:

```
  x.swap(y);
  swap(x,y);
```

Detaching and destroying threads
The detach() member function lets the thread run on, but the object no longer refers to it.

3.2 Example: Threaded Simpson Integration [2b, 12-13]

```
#include <thread>
double func(double x) { return x * std::sin(x); }
int main()
{
  double a; // lower bound of integration
  double b; // upper bound of integration
  unsigned int nsteps; // # of subintervals for integration
  double result1; // the integral of the first half
  // spawn a thread for the first half of the interval
  std::thread t( [&] () {
       result1 = simpson(func,a,a+(b-a)/2.,nsteps/2);} );
  // locally integrate the second half
  double result2 = simpson(func,a+(b-a)/2.,b,nsteps/2);
  t.join(); // wait for the thread to join
  std::cout << result1 + result2 << std::endl;
  return 0;
}
```

4. C++11: FUTURES

Futures hold future return values of a function called asynchronously in a thread.

4.1 Example: Async. Simpson Integration [2b, 15-16]

```
#include <thread>
#include <future>
// create a packaged task
std::packaged_task<double()>
     pt(std::bind(simpson,func,a,a+(b-a)/2.,nsteps/2));
// get the future return value
std::future<double> fi = pt.get_future();
std::thread t (std::move(pt)); // launch the thread
double result2 = simpson(func,a+(b-a)/2.,b,nsteps/2);
// wait for the task to finish and the future to be ready
fi.wait(); // optional, when necessary get will wait
std::cout << result2 + fi.get() << std::endl;
t.join();
```

5. C++11: ASYNCHRONOUS FUNCTION CALLS

```
// even easier: launch an asynchronous function call
// force it to be asynchronous and thus in a seperate thread
std::future<double> fi = std::async(std::launch::async,
                         simpson,func,a,a+(b-a)/2.,nsteps/2);
// locally integrate the second half
double result = simpson(func,a+(b-a)/2.,b,nsteps/2);
std::cout << result + fi.get() << std::endl;
```

5.1 Example: Calculating π through a Series [2b, 20]

```
// sum terms [i-j) of the power series for pi/4
void sumterms(long double& sum, std::size_t i, std::size_t j)
{
  sum = 0.0;
  for (std::size_t t = i; t < j; ++t)
    sum += (1.0 - 2* (t % 2)) / (2*t + 1);
}

int main() {
  // decide how many threads to use
  std::size_t const nthreads = std::max(1u,
                    std::thread::hardware_concurrency());
  std::vector<std::thread> threads(nthreads);
  std::vector<long double> results(nthreads);
  for (unsigned i = 0; i < nthreads; ++i)
    threads[i] = std::thread(sumterms, std::ref(results[i]),
                         i * step, (i+1) * step);
  for (std::thread& t : threads)
    t.join();
  long double pi = 4*std::accumulate(results.begin(),
                                results.end(), 0.);
  cout << "pi=" << std::setprecision(18)<< pi << endl;
  return 0;
}
```

6. C++11: MUTEXES & LOCKS

6.1 Basic Principles [2b, 23-24]

Thread safety: Mutexes are used to serialise access
A mutex is either **locked by one thread** or **unlocked**.

When a thread asks to lock a mutex:

- If the mutex is unlocked, it becomes locked and the thread proceeds.
- If the mutex is locked, the thread is blocked until the lock is released and reallocated to the locking thread.

Protocol – threads agree to:

- acquire a lock on the mutex before accessing the data
- release the lock when done accessing the data

Locks:

- Movable/non-copyable. Expresses ownership of a thread.
- Forgetting to unlock will cause the next thread that locks it to wait forever.
- Use RAII (resource acquisition is initialisation) lock objects to avoid this: constructor locks (acquires) the mutex, destructor unlocks (releases) it
- One lock object should never be accessed by multiple threads!

6.2 Example: Calculating π through a Series [2b, 25]

```
// sum terms [i-j) of the power series for pi/4
void sumterms(std::pair<long double, std::mutex>& result,
              std::size_t i, std::size_t j)
{
  long double sum=0.;
  for (std::size_t t = i; t < j; ++t)
    sum += (1.0 - 2* (t % 2)) / (2*t + 1);
  std::lock_guard<std::mutex> l (result.second);
  result.first += sum;
}

std::vector<std::thread> threads(nthreads);
// let us just use a single result
std::pair<long double, std::mutex> result;
result.first = 0.;
for (unsigned i = 0; i < nthreads; ++i)
  threads[i] =std::thread(sumterms, std::ref(result),
                          i * step, (i+1) * step);
```

6. C++11: MUTEXES & LOCKS (CONT.)

6.3 unique_lock [2b, 28-29]

The unique_lock is more flexible and allows deferring the lock.

Lock the lock:
```
  unique_lock<mutex> l(m);
```

Adopt the lock state:
```
  unique_lock<mutex> l(m,std::adopt_lock);
```

Do not lock yet:
```
  unique_lock<mutex> l(m, std:: defer_lock);
```

Try to lock:
```
  unique_lock<mutex> l(m, std:: try_to_lock);
```

Try to lock with timeout:
```
  unique_lock<mutex> l(m,abs_time);
```

Functions
```
  l.owns_lock(); // returns whether it is locked
  if(l) ... // tests whether it is locked

  l.try_lock(); // tries to lock and returns whether is
                   succeeded
  l.try_lock_for(rel_time); // tries to lock with timeout
  l.try_lock_until(abs_time) // tries to lock with timeout
  l.lock(); // locks the lock
  l.unlock();
  std::lock(l1,l2,...); // lock multiple locks at once
```

6.4 Mutex and Lock Types [2b, 30]

Basic mutex types:
- mutex
- recursive_mutex: allows multiple locking by the same thread
- timed_mutex: allows time-outs in lock attempts
- recursive_timed_mutex: both of the above

Lock types:
- lock_guard
- unique_lock

6.5 Deadlock [2b, 32-33]

Scenario:
Mutexes 1 and 2, unlocked
Thread A locks mutex 1, A still running
Thread B locks mutex 2, B still running
Thread A locks mutex 2, A waits (for B)
Thread B locks mutex 1, B waits (for A)

→ we need to lock both in the same order → std::lock()

7. C++11: RANDOM NUMBER GENERATORS

7.1 Example [3b, 3]

```
#include <random>
#include <iostream>
int main()
{
  std::mt19937 mt; // create an engine
  // create four distributions
  std::uniform_int_distribution<int> uint_d(0,10);
  std::uniform_real_distribution<double> ureal_d(0.,10.);
  std::normal_distribution<double> normal_d(0.,4.);
  std::exponential_distribution<double> exp_d(1.);
  // create random numbers:
  std::cout << uint_d(mt) << "\n";
  std::cout << ureal_d(mt) << "\n";
  std::cout << normal_d(mt) << "\n";
  std::cout << exp_d(mt) << "\n";
  return 0;
}
```

7.2 Random Number Engines [3b, 4]

Linear congruential generators:
- minstd_rand0
- minstd_rand

Mersenne twisters:
- mt19937
- mt19937_64

Other generators:
- ranlux24
- ranlux48
- knuth_b

7.3 Seeding Random Generators [3b, 5]

Seeding by an integer:
```
  std::mt19937 mt; // create an engine
  mt.seed(42); // seed the generator
```

Seeding from a seed sequence:
```
  // create a vector of seeds
  int N = ....;
  std::vector<int> seeds(N);
  // fill the vector, e.g. by an LCG generator minstd_rand
  ...
  // create a seed sequence and use it to seed a generator
  std::seed_seq seq(seeds.begin(), seeds.end());
  mt.seed(seq);
```

7.4 Random Distributions [3b, 6-7]

Uniform distributions:
- uniform_int_distribution<T>
- uniform_real_distribution<T>
- generate_canonical<T> // uniform real numbers in [0,1)

Bernoulli distributions:
- bernoulli_distribution<T>
- binomial_distribution<T>
- negative_binomial_distribution<T>
- geometric_distribution<T>

Sampling distributions:
- discrete_distribution<T>
- piecewise_constant_distribution<T>
- piecewise_linear_distribution<T>

Poisson distributions:
- poisson_distribution<T>
- exponential_distribution<T>
- gamma_distribution<T>
- weibull_distribution<T>
- extreme_value_distribution<T>

Normal distributions:
- normal_distribution<T>
- lognormal_distribution<T>
- chi_squared_distribution<T>
- cauchy_distribution<T>
- fisher_f_distribution<T>
- student_t_distribution<T>

```
std::vector<int> data(nterms);
std::generate(data.begin(),data.end(),std::bind(dist,mt));
```

8. C++11: POLYMORPHIC FUNCTION OBJECTS

A runtime polymorphic function object constructible from any compatible
- function pointer
- member function pointer
- function object
- lambda function

Declaration:
```
  std::function<Result(Arg1,Arg2,...)>
```

Example:
```
  double simpson(std::function<double(double)> f,
                 double a, double b, unsigned int N)
```

9. C++11: AUTO KEYWORD

The auto keyword tells the compiler to deduce the type of a variable from the initialiser argument:

```
  auto x = 3.141+5;
```

9.1 Example [3b, 9]

```
#include <functional>
int f(int x) { return x+1;}
int main()
{
  int (*p1)(int) = f; // function pointer
  auto p2 =f; // easier function pointer with auto
  // or here we could just have used std::function
  std::function<int(int)> p3=f;
  std::cout << (*p1)(42) << std::endl;
  std::cout << (*p2)(42) << std::endl;
  std::cout <<    p3(42) << std::endl;
}
```

10. C++11: BIND

10.1 Example

```
#include <random>
#include <iostream>
#include <functional>

void f(int n1, int n2, int n3, const int& n4, int n5)
{
    std::cout << n1 << ' ' << n2 << ' ' << n3 << ' ' << n4
                    << ' ' << n5 << '\n';
}

int g(int n1)
{
    return n1;
}

struct Foo {
    void print_sum(int n1, int n2)
    {
        std::cout << n1+n2 << '\n';
    }
    int data = 10;
};

int main()
{
    using namespace std::placeholders;
    // demonstrates argument reordering and pass-by-reference
    int n = 7;
    auto f1 = std::bind(f, _2, _1, 42, std::cref(n), n);
    n = 10;
    f1(1, 2, 1001); // 1 is bound by _1, 2 is bound by _2,
                       1001 is unused

    // nested bind subexpressions share the placeholders
    auto f2 = std::bind(f, _3, std::bind(g, _3), _3, 4, 5);
    f2(10, 11, 12);

    // common use case: binding a RNG with a distribution
    std::default_random_engine e;
    std::uniform_int_distribution<> d(0, 10);
    std::function<int()> rnd = std::bind(d, e);
    for(int n=0; n<10; ++n)
        std::cout << rnd() << ' ';
    std::cout << '\n';

    // bind to a member function
    Foo foo;
    auto f3 = std::bind(&Foo::print_sum, foo, 95, _1);
    f3(5);

    // bind to member data
    auto f4 = std::bind(&Foo::data, _1);
    std::cout << f4(foo) << '\n';
}
```

Output:

```
  2 1 42 10 7
  12 12 12 4 5
  1 5 0 2 0 8 2 2 10 8
  100
  10
```

10.2 Example

```
#include <functional>
#include <iostream>
void f(int& n1, int& n2, const int& n3) {
    std::cout << "In function: " << n1 << ' ' << n2 << ' '
              << n3 << '\n';
    ++n1; // increments copy of n1 stored in the function obj.
    ++n2; // increments the main()'s n2
    // ++n3; // compile error
}
int main() {
    int n1 = 1, n2 = 2, n3 = 3;
    std::function<void()> bound_f = std::bind(f, n1,
                                  std::ref(n2), std::cref(n3));
    n1 = 10;
    n2 = 11;
    n3 = 12;
    std::cout << "Before function: " << n1 << ' ' << n2
              << ' ' << n3 << '\n';
    bound_f();
    std::cout << "After function: " << n1 << ' ' << n2 << ' '
              << n3 << '\n';
}
```

Output:

```
  Before function: 10 11 12
  In function: 1 11 12
  After function: 10 12 12
```

11. C++11: LAMBDA FUNCTIONS

11.1 Lambda Functions [3b, 14-18]

Hello World example:

```
  // create a function and store a pointer to it in f
  auto f = []() {std::cout << "Hello world!\n";};
  // call the function
  f();
```

Simpson integration:

```
  double a=3.4;
  // create a lambda function; [=] indicates that the variable
  // a should be used inside the lambda
  auto f = [=] (double x) { return std::exp(a*x); };
  std::cout << simpson(f,0.,1.,100) << std::endl;
```

The [] indicate a lambda function, and how variables from the enclosing scope should be captured inside the lambda.

`[]`	Capture nothing
`[&]`	Capture any referenced variable by reference
`[=]`	Capture any referenced variable by making a copy
`[=, &foo]`	Capture any referenced variable by making a copy, but capture variable foo by reference
`[bar]`	Capture bar by making a copy, do not copy anything else
`[this]`	Capture the this pointer of the enclosing class

12. C++11: CONDITION VARIABLES

Blocks a thread until some condition might be satisfied. Always used with a mutex to ensure the condition sees only non-broken invariants.

Always enter it with a locked lock. Always call in a loop that checks the condition at the end, to see whether the notification condition is still valid.

Two types:

- condition_variable: optimized version, needs to be used with unique_lock<mutex>
- condition_variable_any: can be used with any lock

12.1 Example [3b, 19-22]

```
#include <condition_variable>
#include <mutex>
#include <thread>
#include <queue>
#include <chrono>

int main() {
  std::queue<int> produced_nums;
  std::mutex m;
  std::condition_variable cond_var;
  bool done = false;
  bool notified = false;

  std::thread producer([&]() {
    for (int i = 0; i < 5; ++i) {
      std::this_thread::sleep_for(std::chrono::seconds(1));
      std::unique_lock<std::mutex> lock(m);
      std::cout << "producing " << i << '\n';
      produced_nums.push(i);
      notified = true;
      cond_var.notify_one();
    }
    notified = true;
    done = true;
    cond_var.notify_one();
  });

  std::thread consumer([&]() {
    std::unique_lock<std::mutex> lock(m);
    while (!done) {
      while (!notified) {  // loop to avoid spurious wakeups
        cond_var.wait(lock);
      }
      while (!produced_nums.empty()) {
        cout << "consuming " << produced_nums.front() << '\n';
        produced_nums.pop();
      }
      notified = false;
    }
  });

  producer.join();
  consumer.join();
}
```

(Possible) Output:

```
producing 0
consuming 0
producing 1
consuming 1
producing 2
consuming 2
producing 3
consuming 3
producing 4
consuming 4
```

13. C++11: BARRIER CLASS

13.1 Barrier Class [3b, 23]

Synchronisation between threads

- avoid it whenever possible since it serialises and slows down the code
- is sometimes unavoidable: wait for all threads to finish between update steps in a Monte Carlo simulation or integration of a PDE

```
class barrier
{
private:
  mutable std::mutex m_mutex;
  std::condition_variable m_cond;
  unsigned int const m_total;
  unsigned int m_count;
  unsigned int m_generation;
public:
  barrier(unsigned int count)
  : m_total(count)
  , m_count(count)
  , m_generation(0)
  {
    assert(count != 0);
  }

  void wait()
  {
    std::unique_lock<std::mutex> lock(m_mutex);
    unsigned int gen = m_generation;
    // decrease the count
    if (--m_count==0) {
      // if done reset to new generation of wait
      // and wake up all threads
      m_count = m_total;
      m_generation++;
      m_cond.notify_all();
    }
    else
      while (gen == m_generation)
        m_cond.wait(lock);
  }
};
```

14. C++11: ONCE ROUTINES

14.1 Once Routines [3b, 24]

- Executed once, no matter how many invocations
- No invocation will complete until the one execution finishes
- Typical use: initialisation of static and function-static data

```
std::once_flag printonce_flag;
void printonce() {
  std::cout << "This should be printed only once\n";
}

int main() {
  std::vector<std::thread> threads;
  for (int n = 0; n < 10; ++n)
    threads.push_back(std::thread([&]() {
                  std::call_once(printonce_flag,printonce);}));
  for (std::thread& t : threads)
    t.join();
  return 0;
}
```

15. CACHE TRASHING

15.1 Bad Example [4a, 2-4]

```
// sum terms [i-j) of the power series for pi/4
void sumterms(long double& sum, std::size_t i, std::size_t j)
{
  sum = 0.0;
  for (std::size_t t = i; t < j; ++t)
    sum += (1.0 - 2* (t % 2)) / (2*t + 1);
}
```

A thread invalidates the cache for other threads and sum has to be reloaded.

15.2 Better Example [4a, 5-6]

Use a thread-local variable for the summation:

```
// sum terms [i-j) of the power series for pi/4
void sumterms(long double& result, size_t i, size_t j) {
  long double sum = 0.0;
  for (std::size_t t = i; t < j; ++t)
    sum += (1.0 - 2* (t % 2)) / (2*t + 1);
  result = sum;
}
```

15.3 NUMA Effects [4a, 7-8]

The thread touching (not allocating) the memory first decides which part of the memory it gets placed in.

16. C++11: ATOMICS

16.1 Atomics [4a, 13-16]

A standard increment x++ is 3 operations, and this can cause race conditions:

- load x into register
- increment the value in the register
- write the value back to memory

Example:

```
std::atomic<int> countzeros = ATOMIC_VAR_INIT(0);
// int countzeros=0; would cause a race condition
std::vector<std::thread> threads(nthreads);
for (unsigned i = 0; i < nthreads; ++i)
  threads[i] = std::thread([&,i]() {
    for (int j = i * step;
    j < static_cast<unsigned>((i+1) * step); ++j)
      if (data[j]==0)
        ++countzeros;
  });
for (std::thread& t : threads)
  t.join();
```

16.2 Atomic Flags [4a, 17]

An atomic flag has two important member functions:

- clear(): sets the flag to false
- test_and_set(): sets the flag to true and returns the previous value

Example:

```
#include <atomic>
std::atomic_flag lock = ATOMIC_FLAG_INIT;
void f(int n) {
  for(int cnt = 0; cnt < 100; ++cnt) {
    while(lock.test_and_set()) // acquire lock
      ; // spin
    std::cout << "Output from thread " << n << '\n';
    lock.clear(); // release lock
  }
}
int main() {
  std::vector<std::thread> v(10);
  for (int n = 0; n < 10; ++n)
    v[n] = std::thread(f, n);
  for (auto& t : v) {
    t.join();
}
```

(Possible) Output:

```
  Output from thread 2
  Output from thread 6
  Output from thread 7
  ...<exactly 1000 lines>...
```

16.3 Memory Ordering [4a, 18-24]

Atomics without a specified memory order (as in the examples above) have two consequences:

- updates are atomic
- no reordering of loads and stores takes place

The second condition might be too restrictive and can be relaxed by specifying the memory order to be used for an operation:

memory_order_relaxed	No constraints on reordering of memory accesses around the atomic variable.
memory_order_consume	No reads in the current thread dependent on the value currently loaded can be reordered before this load. This ensures that writes to dependent variables in other threads that release the same atomic variable are visible in the current thread.
memory_order_acquire	No reads in the current thread can be reordered before this load. This ensures that all writes in other threads that release the same atomic variable are visible in the current thread.
memory_order_release	No writes in the current thread can be reordered after this store. This ensures that all writes in the current thread are visible in other threads that acquire the same atomic variable.
memory_order_acq_rel	No reads in the current thread can be reordered before this load as well as no writes in the current thread can be reordered after this store. The operation is read-modify-write operation. It is ensured that all writes in another threads that release the same atomic variable are visible before the modification and the modification is visible in other threads that acquire the same atomic variable.
memory_order_seq_cst	Sequential ordering. The operation has the same semantics as acquire-release operation, and additionally has sequentially-consistent operation ordering.

Other description:

memory_order_relaxed	All reorderings are okay.
memory_order_consume	Potentially weaker form of memory_order_acquire that enforces ordering of the current load before other operations that are data-dependent on it (for instance, when a load of a pointer is marked memory_order_consume, subsequent operations that dereference this pointer won't be moved before it.
memory_order_acquire	Guarantees that subsequent loads are not moved before the current load or any preceding loads.
memory_order_release	Preceding stores are not moved past the current store or any subsequent stores.
memory_order_acq_rel	Combines the two previous guarantees.

16.4 Member Functions of Atomic Variables [4a, 19]

The atomics have the operators:

```
  ++, --, +=, -=, &=, |=, ^= // bitwise and/or/xor
```

Member functions:

- **store, load:**
  assignment and conversion with optional specification of memory order
- **exchange:**
  sets the value and returns the old value, with optional specification of memory order
- **compare_exchange_weak(expected,desired)**
  compare_exchange_strong(expected,desired):
  compares the value of the atomic object with non-atomic argument *expected* and performs atomic exchange with *desired* if equal or atomic load if not. Returns true if the arguments were equal. The weak version may spuriously fail to detect equal values.
- **fetch_add, fetch_sub, fetch_and, fetch_or, fetch_xor:**
  implement +=, -=, &=, |= and ^= with optional specification of memory order

16.5 Example: Memory Ordering [4a, 21-22]

```
std::atomic<std::string*> ptr;
int data;
void producer() {
  std::string* p = new std::string("Hello");
  data = 42;
  ptr.store(p, std::memory_order_release);
}
void consumer() {
  std::string* p2;
  while (!(p2 = ptr.load(std::memory_order_consume)))
    ;
  assert(*p2 == "Hello"); // never fires
  assert(data == 42); // may or may not fire
  // use memory_order_acquire and data won't fire
}
int main() {
  std::thread t1(producer);
  std::thread t2(consumer);
  t1.join(); t2.join();
}
```

16.6 Example: compare_exchange_strong

```
#include <atomic>

std::atomic<int>  ai;
int  tst_val= 4;
int  new_val= 5;
bool exchanged= false;

void valsout() {
  cout << "ai= " << ai << "  tst_val= " << tst_val
       << "  new_val= " << new_val
       << "  exchanged= " << std::boolalpha << exchanged
       << "\n";
}

int main() {
  ai= 3;
  valsout();

  // tst_val != ai   ==>  tst_val is modified
  exchanged= ai.compare_exchange_strong( tst_val, new_val );
  valsout();

  // tst_val == ai   ==>  ai is modified
  exchanged= ai.compare_exchange_strong( tst_val, new_val );
  valsout();

  return 0;
}
```

Output:

```
  ai= 3  tst_val= 4  new_val= 5  exchanged= false
  ai= 3  tst_val= 3  new_val= 5  exchanged= false
  ai= 5  tst_val= 3  new_val= 5  exchanged= true
```

16.7 Atomic Barrier

17. OPENMP

```
g++ -std=c++11 -O3 -Wall -lrt -fopenmp -o main main.cpp
```

OpenMP provides semi-automatic parallelisation through compiler directives, environment variables and function calls.

17.1 Basic Example [5, 3-5]

```
#include <omp.h>
int main() {
  #pragma omp parallel
  {
    // now we execute this block in multiple threads
    #pragma omp critical
    std::cout << "I am thread " << omp_get_thread_num()
              << " of " << omp_get_num_threads()
              << " threads." << std::endl;
  }
}
```

The "critical" directive lets only one thread run the statement at any given time. Without it, the output would be messed up.

17.2 Synchronisation Directives [5, 11]

#pragma omp master	The block is performed only by the master thread.
#pragma omp single	The block is performed only by one single thread.
#pragma omp critical [(*name*)]	If a thread is already in the (named) section, all other threads entering it will wait.
#pragma omp barrier	All threads wait until every thread has called the barrier.
#pragma omp atomic	The following update operation is atomic.
#pragma omp threadprivate (*list*)	Declares the listed variables to be thread-private, i.e. every thread gets its own copy.
#pragma omp flush (*list*)	Makes sure that all (or all listed) variables are written back to memory. This ensures that all threads then have the same value of these variables.

Example:

```
double result = 0.;
#pragma omp parallel
{
  int i = omp_get_thread_num();
  int n = omp_get_num_threads();
  // integrate just one part in each thread
  double r = simpson(func,a+i*delta,a+(i+1)*delta,nsteps/n);
  #pragma omp atomic
  result += r;
}
```

17.3 Clauses for omp parallel [5, 12-14]

if (*scalar_expression*)	Only parallelise if the expression is true. Can be used to stop parallelisation if the work is too little.
private (*list*)	The specified variables are thread-private.
shared (*list*)	The specified variables are shared among all threads.
default (shared | none)	Unspecified variables are shared or not.
copyin (*list*)	Initialise private variables from the master thread.
firstprivate (*list*)	A combination of private and copyin.
reduction (*operator*: *list*)	Perform a reduction on the thread-local variables and assign it to the master thread.
num_threads (*integer-expression*)	Set the number of threads.

Example:

```
double result; // no need to initialise result
#pragma omp parallel reduction(+:result)
{
  int i = omp_get_thread_num();
  int n = omp_get_num_threads();
  double delta = (b-a)/n;
  // integrate just one part in each thread
  result = simpson(func,a+i*delta,a+(i+1)*delta,nsteps/n);
}
```

Allowed reduction operations:

```
  +, -, *, /, &, ^, |, &&, ||
```

17.4 Sections [5, 15-16]

Each section gets assigned to a different thread:

```
#pragma omp parallel shared(result)
{
  #pragma omp sections reduction(+:result)
  {
    #pragma omp section
    {
      result = simpson(func,a,a+0.5*(b-a),nsteps/2);
    }
    #pragma omp section
    {
      result = simpson(func,a+0.5*(b-a),b,nsteps/2);
    }
  }
}
```

Additional clause for sections:

lastprivate (*list*)	The value which the listed variables (or all private variables) have in the last section gets copied back to the master thread.

17.5 Master and Single Directive [5, 17-18]

Only the master thread (number 0) will execute the code:

```
#include <omp.h>
int main() {
  #pragma omp parallel
  #pragma omp master
  cout << "Only thread " << omp_get_thread_num() << " of "
       << omp_get_num_threads() << " is printing.\n";
  return 0;
}
```

Using single, only a single thread will execute the code, but this thread is not necessarily the master.

The single directive takes an optional clause:

copyprivate (*list*)	Copies the listed variables to all other threads. This is useful to e.g. broadcast input parameters that have been read by a single thread.

17.6 For Directive [5, 19-26]

Example: Calculating π through a series

```
int main() {
  unsigned long const nterms = 100000000;
  long double sum=0.;
  #pragma omp parallel for reduction(+:sum)
  for (std::size_t t = 0; t < nterms; ++t)
    sum += (1.0 - 2* (t % 2)) / (2*t + 1);
  std::cout << "pi=" << 4.*sum <<  std::endl;
  return 0;
}
```

The for directive only works if

- the loop control variable is an integer, pointer or C++ random access iterator
- the loop condition is a simple binary comparison (<, <=, !=, > or >=) with a constant
- the increment is x++, ++x, x--, --x, x+= *inc*, x-= *inc*, x=x + *inc*, or x= x - *inc* with a constant increment *inc*

The for directive takes additional optional clauses:

nowait	There is no implicit barrier at the end of the for. Useful, e.g. if there are two for loops in a parallel section.
ordered	The same ordering as in the serial code can be enforced. (slow)
collapse (*n*)	Collapse n nested loops into one and parallelise it.
schedule (*type* [, *chunk*])	Specify the schedule for loop parallelisation (see below).

The schedule clause specifies how the iterations get divided onto threads. The type can be:

STATIC	Loop iterations are divided into fixed chunks and assigned statically.
DYNAMIC	Loop iterations are divided into fixed chunks and assigned dynamically whenever a thread finished with a chunk.
GUIDED	Like dynamic but with decreasing chunk sizes. The chunk parameter defines the minimum block size.
RUNTIME	Decide at runtime depending on the OMP_SCHEDULE environment variable.
AUTO	Decided by compiler and/or runtime system.

Example: ordered

```
int main() {
  #pragma omp parallel for ordered
  for (int i=0; i < 100; ++i) {
    // do some (fake) work
    int j=i;
    #pragma omp ordered
    std::cout << "Hello from the " << j << "-th iteration\n";
  }
}
```

17.7 Environment Variables [5, 27]

The behaviour of the code can be controlled at runtime using environ. variables:

OMP_NUM_THREADS	The maximum number of threads to be used in parallelisation of one parallel region.
OMP_DYNAMIC	Set to TRUE or FALSE to enable or disable dynamic adjustment of the number of threads.
OMP_PROC_BIND	Supported since OpenMP 3.0. Set to TRUE to bind threads to processors and disable migration to other processors. Important on NUMA architectures.
OMP_NESTED	Supported since OpenMP 3.0. Set to TRUE or FALSE to enable or disable nested parallelisation. Nested parallelism occurs when a function containing a parallel region is called from another parallel region.
OMP_STACKSIZE	Supported Since OpenMP 3.0. Controls the stack size of non-Master threads. Be careful: if the stack gets exhausted a program may segfault or just continue running with corrupted data.
OMP_MAX_ACTIVE_LEVELS	Supported since OpenMP 3.0. Limits the number of nested levels.
OMP_THREAD_LIMIT	Supported since OpenMP 3.0. Limits the total number of threads.
OMP_WAIT_POLICY	If set to ACTIVE waiting threads spin actively, if set to PASSIVE they sleep.

17.8 Runtime Library [5, 28]

`void omp_set_num_threads(int n)`	Sets the number of threads to be used.
`int omp_get_num_threads()`	Gets the number of currently running threads.
`int omp_get_max_threads()`	Gets the maximum number of threads that can be used for one parallel region.
`int omp_get_thread_num()`	Get the id of the calling thread
`int omp_get_thread_limit()`	Gets the maximum number of threads used for nested parallel region.
`int omp_get_num_procs()`	Gets the number of processors available
`int omp_in_parallel()`	Returns true if called from within a parallel region
`void omp_set_dynamic(int n)`	Sets dynamic adjustment of threads with the given number as maximum. This overrides the environment variable.
`int omp_get_dynamic()`	Returns true if dynamic scheduling is enables
`double omp_get_wtime()`	A portable wallclock timing routine, returns time in seconds. The time is not synchronized across threads to be fast.
`double omp_get_wtick()`	Returns the number of seconds between successive clock ticks

17.9 Functions for Nested Parallelism [5, 29]

`void omp_set_nested(int l)`	Enables nested parallelism if called with true as argument
`int omp_get_nested()`	Returns true if nested parallelism is available
`void omp_set_max_active_levels (int l)`	Sets the maximum number of nested parallel regions allowed. If parallel regions are nested deeper, the additional ones will be inactive and only be run by one thread.
`int omp_get_max_active_levels()`	Gets the maximum number of nested parallel regions allowed
`int omp_get_level()`	Returns the number parallel regions enclosing the call
`int omp_get_active_level()`	Returns the number of active parallel regions (run by more than one thread) enclosing the call
`int omp_get_ancestor_thread_num(int l)`	Returns the thread number of the ancestor of the current thread in a higher level. Returns -1 if the level is invalid.
`int omp_get_team_size(int level)`	Returns the size of the thread team to which the ancestor at the given level belongs. Returns -1 if the level is invalid.

17.10 Functions for Loop Schedules [5, 30]

Functions to set runtime loop schedules:

```
  void omp_set_schedule(omp_sched_t kind, int modifier)
  void omp_get_schedule(omp_sched_t * kind, int * modifier)
```

The schedules are defined by an enum:

```
  typedef enum omp_sched_t {
    omp_sched_static = 1,
    omp_sched_dynamic = 2,
    omp_sched_guided = 3,
    omp_sched_auto = 4,
  } omp_sched_t;
```

The modifier is the chunk size for those schedules which take a chunk size.

17.11 Locking Functions [5, 31]

OpenMP also contains locks that need to be manually created, locked, unlocked, and destroyed. The nested lock allows recursive locking by the same thread.

Initialise a lock variable:
```
  void omp_init_lock(omp_lock_t *lock)
  void omp_init_nest_lock(omp_nest_lock_t *lock)
```

Destroy a lock variable:
```
  void omp_destroy_lock(omp_lock_t *lock)
  void omp_destroy_nest_lock(omp_nest_lock_t *lock)
```

Set (lock) the lock: (thread suspends until the lock can be obtained)
```
  void omp_set_lock(omp_lock_t *lock)
  void omp_set_nest__lock(omp_nest_lock_t *lock)
```

Release the lock:
```
  void omp_unset_lock(omp_lock_t *lock)
  void omp_unset_nest_lock(omp_nest_lock_t *lock)
```

Test whether a lock is available: (if lock is set true is returned, otherwise false)
```
  int omp_test_lock(omp_lock_t *lock)
```

This is ugly, thus let's create C++ objects using RAII - we can then never forget to initialise, destroy, or unlock.

17.12 A C++ Conforming OpenMP Mutex [5, 32-35]

std::lock_guard<Mutex> template requires a mutex m of type Mutex to model the BasicLockable concept:

Expression	Requires	Effects
m.lock()		Blocks until a lock can be obtained for the current execution agent. If an exception is thrown, no lock is obtained.
m.unlock()	The current execution agent should hold the lock m.	Releases the lock held by the execution agent. Throws no exceptions.

Code main.cpp:
```
#include <mutex>
#include "omp_mutex.hpp"
int main() {
  omp_mutex m;
  #pragma omp parallel for
  for (int i=0; i < 100; ++i) {
    {
      std::lock_guard<omp_mutex> lock(m);
      cout << "Hello from the " << i << "-th iteration\n";
    }
  }
}
```

Code omp_mutex.hpp:
```
#include <omp.h>
class omp_mutex {
public:
  omp_mutex() { omp_init_lock(&_mutex); }
  ~omp_mutex() { omp_destroy_lock(&_mutex); }
  void lock() { omp_set_lock(&_mutex); }
  void unlock() { omp_unset_lock(&_mutex); }
private:
  omp_lock_t _mutex;
};

class omp_recursive_mutex {
public:
  omp_recursive_mutex() { omp_init_nest_lock(&_mutex); }
  ~omp_recursive_mutex() { omp_destroy_nest_lock(&_mutex); }
  void lock() { omp_set_nest_lock(&_mutex); }
  void unlock() { omp_unset_nest_lock(&_mutex); }
private:
  omp_nest_lock_t _mutex;
};
```

17.13 Tasks [5, 36-46]

Spawning threads dynamically is expensive. Tasks are more lightweight:
- new tasks get put onto a task queue
- idle threads pull tasks from the queue

Spawns tasks and puts them into a queue for the threads to work on:
```
  #pragma omp task [clause ...]
```

if (*scalar_expression*)	Only parallelise if the expression is true. Can be used to stop parallelisation if the work is too little.
private (*list*)	The specified variables are thread-private
shared (*list*)	The specified variables are shared among all threads
default (shared | none)	Unspecified variables are shared or not
firstprivate (*list*)	Initialise private variables from the master thread
mergeable	If specified allows the task to be merged with others
untied	If specified allows the task to be resumed by other threads after suspension. Helps prevent starvation but has unusual memory semantics: after moving to a new thread all private variables are that of the new thread
final (*scalar_expression*)	If the expression is true this has to be the final task. All dependent tasks are included into it.

Wait for all dependent tasks:
```
  #pragma omp taskwait
```

Yield the thread to another task:
```
  #pragma omp taskyield
```

Check at runtime whether this is a final task: (returns true if it is a final task)
```
  int omp_in_final()
```

Example 1: too many calls to fibonacci

```
int main() {
  int n;
  std::cin >> n;
  #pragma omp parallel shared(n)
  {
    std::cout << fibonacci(n) << std::endl;
  }
}

int fibonacci(int n) {
  int i, j;
  if (n<2)
    return n;
  else {
    #pragma omp task shared(i) firstprivate(n)
    i = fibonacci(n-1);
    #pragma omp task shared(j) firstprivate(n)
    j = fibonacci(n-2);
    return i + j;
  }
}
```

Example 2: still with problems (main modified)

```
#pragma omp parallel shared(n)
{
  #pragma omp single nowait
  std::cout << fibonacci(n) << std::endl;
}
```

- i and j get added before the tasks are done
- when i and j are written the variables no longer exist

Example 3: working (main as in 2, fibonacci modified)

```
#pragma omp task shared(i) firstprivate(n)
i = fibonacci(n-1);
#pragma omp task shared(j) firstprivate(n)
j = fibonacci(n-2);
#pragma omp taskwait
return i + j;
```

Example 4: optimised using the final clause

```
#pragma omp task shared(i) firstprivate(n) untied final (n<=5)
i = fibonacci(n-1);
#pragma omp task shared(j) firstprivate(n) untied final (n<=5)
j = fibonacci(n-2);
#pragma omp taskwait
return i + j;
```

Now it will not spawn tasks for n<=5.

18. VECTORISATION WITH SIMD INSTRUCTIONS

Single Instruction Multiple Data:
- SSE (Streaming SIMD Extensions)
- AVX (Advanced Vector Extensions)

18.1 SIMD Registers and Operations [11, 3-4]

SIMD units contain vector registers:
- 128-bit registers for SSE, XMM0 – XMM15
- 256-bit registers for AVX, YMM0 – YMM15, overlapping the XMM registers

The SSE XMM register can store:
- 2 doubles
- 4 floats
- 2 64-bit integer
- 4 32-bit integer
- 8 16-bit integer
- 16 bytes

AVX register can store 8 float or 4 double, integer to come with AVX2.

Operations
SIMD vector operations act on all values in the vector at once.

Advantages:
- 1 instruction instead of 4 (in case of floats and SSE)
- memory access can be optimised

19. CACHES

19.1 Caches [11, 7-9]

Usual PC configuration (in brackets: access time in cycles):
- many GB of slow but cheap DRAM (~300)
- 2-20 MB of fast L3-cache (30-50)
- 256-512 kB of faster L2-cache per core (12-19)
- 2x32 – 2x64 kB of fastest L1-cache per core (instruction and data cache, 4-5)

Data that is read is stored in the caches and kept there until it needs to be evicted because new data is loaded.

Data written to memory is written to the cache and only further to memory if it needs to be evicted (or if we need to synchronize memory access between cores).

Problems reusing memory will run faster!

How a cache works
CPU requests a word (e.g. 4 bytes) from memory:
- a full cache line (nowadays typically 64 bytes) is read from memory and stored in the cache
- the first word is sent to the CPU

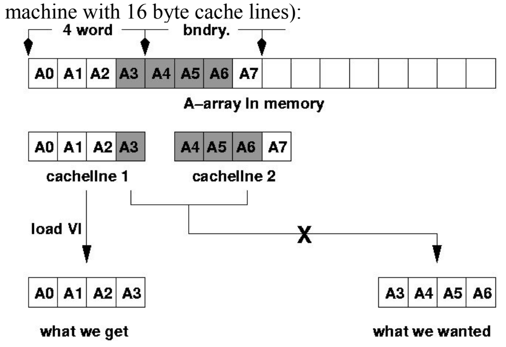
CPU requests another word from memory:
- cache checks whether it has already read that part as part of the previous cache line
- if yes, the word is sent quickly from cache to CPU
- if not, a new cache line is read

Once the cache is full, the oldest data is overwritten.

19.2 Data Alignment [11, 10-13]

To achieve optimal speed data should be aligned on cache line boundaries.

Consider what happens if we load one value that is not at the start of a cache line (on an old machine with 16 byte cache lines):



Alignment:
- SSE registers are 16 bytes and need 16-byte alignment
- AVX register are 32 bytes and need 32-byte alignment
- it is even better to align on cache line boundaries: 64 bytes on modern CPUs

Aligned memory can be allocated:
- on POSIX systems by calling posix_memalign
- on Windows systems by calling _aligned_malloc
- alignment specifier in the declaration

Alignment specifiers:

C++03 with GCC:	`float __attribute__((aligned(32))) sse[8];`
C++03 with MSVC:	`float __declspec(align(32)) sse[8];`
C++11:	`float alignas(32) sse[8];`

(alignas not supported by g++ 4.7 and MSVC11)

Example:
```
struct alignas(32) avx_double {
  double data[4];
};
template <class T>
struct alignas(32) avx_t {
  T data[32/sizeof(T)];
};
```

Allocating aligned data with allocators
For C++ containers we need an aligned allocator. Recall the usually ignored second template parameter of standard containers:
```
  template< class T, class Alloc = std::allocator<T> >
  class vector;
```

Allocators are used to allocate and free memory for a container.

20. DEPENDENCIES

20.1 Dependencies [11, 14-15]

Look at every variable in the loop and check whether it might be written or read by another loop iteration. If so there is a dependency.

Some dependencies can be removed by introducing additional variables:
```
  for (int i=0; i<N-1; i++) {
    x = (b[i] + c[i])/2;
    a[i] = a[i+1] + x;
  }
```
can be rewritten as:
```
  for (int i=0; i<N-1; i++)
    a2[i] = a[i+1];
  for (int i=0; i<N-1; i++) {
    x = (b[i] + c[i])/2;
    a[i] = a2[i] + x;
  }
```

21. SSE / AVX

```
g++ -std=c++11 -O3 -Wall -lrt -fopenmp -msse -o main main.cpp
```

Possible switches:

```
  -mmmx -msse -msse2 -msse3 -mssse3 -msse4.1 -msse4.2 -msse4 -mavx -maes
```

21.1 Intrinsics: Header Files [11, 16]

Include the appropriate header:

MMX	<mmintrin.h>	SSE4.1	<smmintrin.h>
SSE	<xmmintrin.h>	SSE4.2	<nmmintrin.h>
SSE2	<emmintrin.h>	SSE4A	<ammintrin.h>
SSE3	<pmmintrin.h>	AES	<wmmintrin.h>
SSSE3	<tmmintrin.h>	AVX	<immintrin.h>

or use the header **<x86intrin.h>** that is available with some compilers to load all headers available depending on the target platform.

21.2 Data Types and Naming Scheme [11, 17-18]

`__m128`	4 floats	`__m256`	8 floats
`__m128d`	2 doubles	`__m256d`	4 doubles
`__m128i`	integers of any size	`__m256i`	ints of any size (AVX2)

SSE and AVX instructions have a certain naming scheme:

- SSE operations: _mm_*name*_*type*
- AVX operations: _mm256_*name*_*type*

type	length in bits	description
ss	32	1 float
ps	128 or 256	4 or 8 floats
sd	64	1 double
pd	128 or 256	2 or 4 doubles
si64	64	any integers
si128	128	any integers
si256	256	any integers
pi8	64	8 8-bit integers
pi16	64	4 16-bit integers
pi32	64	2 32-bit integers
epi8	128 or 256	16 or 32 8-bit integers
epi16	128 or 256	8 or 16 16-bit integers
epi32	128 or 256	4 or 8 32-bit integers
epi64	128 or 256	2 or 4 64-bit integers

Operations on types shorter than a full register will not modify the higher bits.

256 bit integer operations will only be available in AVX2.

21.3 Example: sscal [11, 19-21]

Multiply a vector by a scalar, assuming aligned data:

```
void sscal(int n, float a, float* x) {
  // load the scale factor four times into a register
  __m128 x0 = _mm_set1_ps(a);
  int ndiv4 = n/4;
  // loop over chunks of 4 values
  for (int i=0; i<ndiv4; ++i) {
    __m128 x1 = _mm_load_ps(x+4*i); // aligned (fast) load
    __m128 x2 = _mm_mul_ps(x0,x1); // multiply
    _mm_store_ps(x+4*i,x2); // store back aligned
  }
  // do the remaining entries
  for (int i=ndiv4*4 ; i< n ; ++i)
    x[i] *= a;
}
```

21.4 Load / Store Instructions [11, 22]

instruction	types	description
set1	all	sets all elements to a given value
set	all	set each element to a different value
setr	all	set in reverse order
setzero	pd, ps, si64, si128, si256	set to zero
load1	pd, ps	load a single value into each element of the register
broadcast	pd, ps	like load1 but much faster (AVX only)
load	pd, ps, ss, sd, si128, si256	load values from memory into a register
loadr	pd, ps	load values in reverse order
loadu	pd, ps, ss, sd, si128, si256	load unaligned values from memory (slow!)
streamload	si128	load integer values bypassing the cache
store	pd, ps, ss, sd, si128, si256	store values from register into memory
storeu	pd, ps, ss, sd, si128, si256	store values from register into unaligned memory (slow!)
stream	pd, ps, pi, si128, si256	store values into memory bypassing the cache

21.5 Prefetch [11, 23]

Prefetch instruction can be used to hint that some data will be used later and should already be fetched into the cache since they will soon be used:

```
  void _mm_prefetch (char const *p, int hint)
```

hint	description
_MM_HINT_T0	prefetch into L1 (and L2 and L3) cache. Use for integer data.
_MM_HINT_T1	prefetch into L2 (an L3) cache. Use for floating point data.
_MM_HINT_T2	prefetch into L3 cache. Use if cache line is not reused much.
_MM_HINT_NTA	prefetch into L2 but not L3 cache. Use if the data is needed only once

Example

```
// loop over chunks of 4 values
for (int i=0; i<ndiv4; ++i) {
  // prefetch data for two iterations later
  _mm_prefetch((char*) y+4*i+8,_MM_HINT_NTA );
  __m128 x1 = _mm_load_ps(x+4*i); // aligned (fast) load
  __m128 x2 = _mm_mul_ps(x0,x1); // multiply
  _mm_store_ps(x+4*i,x2); // store back aligned
}
```

21.6 Arithmetic Floating Point Instructions [11, 24]

instruction	description
add, sub	+, -
addsub	- on even + on odd elements
mul, div	*, /
ceil	ceil, round up
floor	floor, round down
round	round, allows specification of rounding policy
min	min
max	max
rcp	reciprocal (inverse)
sqrt	sqrt
rsqrt	reciprocal (inverse) square root
and, andnot	bitwise &, &!
or, xor	bitwise |, ^

21.7 Arithmetic Integer Instructions [11, 25]

instruction	description
add, adds	+. adds is saturated add: assigns maximum/minimum if overflow or underflow
sub, subs	-, subs is saturated sub: assigns maximum/minimum if overflow or underflow
avg	rounded average of x and y: (x+y+1)/2
mul	*, multiplies low words into result of twice the size - ignores every second input value
mullo	*, low word of product (result has twice the number of bits)
mulhi	*, high word of product (result has twice the number of bits)
sign	transfers sign of one integer to another and sets it to zero if “sign” is 0
min, max	min, max
and, andnot	&, &!
or, xor	|, ^
sll, slli	<<, the version ending in i needs an integer constant shift
srl, srli	>> for unsigned integers, shifting in 0 bits
sra, srai	>> for signed integer, shifting in the sign bit

21. SSE / AVX (CONT.)

21.8 Comparison Instructions [11, 26]

instruction	types	description
cmpeq, cmpneq	all	x==y , x!=y
cmpgt, cmpge	all	x>y, x>=x
cmplt, cmple	all	x<y, x<=y
cmpngt, cmpnge	float	!(x>y), !(x>=x)
cmpnlt, cmpnle	float	!(x<y), !(x<=y)
cmpord, cmpunord	float	tests whether the number are ordererd or unordered (e.g. if NaN)
test_all_ones	i128	test if all bits are 1
test_all_zeros	i128	test if all bits are 0
test_mix_ones_zeros	i128	test if either all are 0 or all are 1

21.9 Example: Dot Product [11, 29]

```
float sdot(int n, float* x, float* y) {
  // set the total sum to 0, one sum per vector element
  __m128 x0 = _mm_set1_ps(0.);
  // we assume alignment
  assert(((std::size_t)x) % 16 == 0 &&
         ((std::size_t)y) % 16 == 0);
  // loop over chunks of 4 values
  int ndiv4 = n/4;
  for (int i=0; i<ndiv4; ++i) {
    __m128 x1 = _mm_load_ps(x+4*i); // aligned (fast) load
    __m128 x2 = _mm_load_ps(y+4*i); // aligned (fast) load
    __m128 x3 = _mm_mul_ps(x1,x2); // multiply
    x0 = _mm_add_ps(x0,x3); // add
  }
  // store the 4 partial sums back to aligned memory
  float alignas(16) tmp[4];
  _mm_store_ps(tmp,x0);
  // do the reduction over the vector elements by hand
  float sum = tmp[0]+tmp[1]+tmp[2]+tmp[3];
  // do the remaining entries
  for (int i=ndiv4*4 ; i< n ; ++i)
    sum += x[i]*y[i];
  return sum;
}
```

21.10 Mixing SSE and AVX [11, 31-33]

Be careful when mixing SSE and AVX instructions: if the higher bits of the AVX registers are nonzero they get stored to memory if you call an SSE instruction and get reloaded when you cal an AVX instruction. SLOW!!!

Solution: call _mm256_zeroupper to clear the upper bits before switching from AVX to SSE.

```
void sscal(int n, float a, float* x) {
  // load the scale factor four times into a register
  __m128 x0 = _mm_broadcast_ss(&a); // an AVX instruction!
  _mm256_zeroupper();
```

22. AUTOMATIC VECTORISATION

22.1 Automatic Vectorisation with g++ [11, 34]

Turn vectorisation on:
```
  -ftree-vectorize
```

Generate vectorisation reports:
```
  -ftree-vectorizer-verbose=n
```

n	description
0	No output at all.
1	Report vectorized loops.
2	Also report unvectorized "well-formed" loops and respective reason.
3	Also report alignment information (for "well-formed" loops).
4	Like level 3 + report for non-well-formed inner-loops.
5	Like level 3 + report for all loops.
6	Print all vectorizer dump information.

22.2 Aliasing [11, 36-37]

SAXPY operation:
```
void saxpy(int n, float a, float* x, float* y) {
  for (int i=0; i<n; ++i)
    y[i] += a*x[i];
}
```

Consider the following call:
```
  float x[1000];
  saxpy( 999, 1., x, x+1)
```

The loop becomes:
```
  for (int i=0; i<n; ++i)
    x[i+1] += a*x[i];
```

We have potential dependencies! No optimization or vectorization is actually possible unless we prevent aliasing.

23. MPI

Numbering: *p* processes are numbered by integer "ranks" 0 to *p*-1.

Wrapper compiler:
```
mpic++ -std=c++11 -O3 -Wall -lrt -o main main.cpp
```
Show added compiler options:
```
mpicc -showme:compile          mpicc --showme:link
```

Launch a MPI program:
```
mpiexec -np number_of_processes executable [options]
mpiexec -np 4 ./main
```

23.1 MPI Environment [12, 13-15]

```
#include <mpi.h>
int main(int argc, char** argv) {
  MPI_Init(&argc, &argv); // initialise the environment
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  cout << "I am rank " << rank << " of " << size << endl;
  MPI_Finalize(); // clean up at the end
  return 0;
}

int MPI_Abort( MPI_Comm comm, int errorcode );
// terminates all processes with the given error code
int MPI_Initialized( int *flag )
// sets the flag to true if MPI has been initialized
int MPI_Finalized( int *flag )
// sets the flag to true if MPI has been finalized
```

23.2 MPI Data Types [12, 18]

C++ datatype	MPI datatype
char	MPI_CHAR
signed char	MPI_SIGNED_CHAR
unsigned char	MPI_UNSIGNED_CHAR
int	MPI_WCHAR
long	MPI_SHORT
long long	MPI_INT
wchar_t	MPI_LONG
short	MPI_LONG_LONG
unsigned short	MPI_UNSIGNED_SHORT
unsigned int	MPI_UNSIGNED
unsigned long int	MPI_UNSIGNED_LONG
unsigned long long	MPI_UNSIGNED_LONG_LONG
float	MPI_FLOAT
double	MPI_DOUBLE
long double	MPI_LONG_DOUBLE
bool	MPI_BOOL
	MPI_BYTE
	MPI_PACKED

23.3 MPI: Sending and Receiving Messages [12, 19+23-26]

```
int MPI_Send(void* buf, int count, MPI_Datatype type,
             int dest, int tag, MPI_Comm comm);

int MPI_Recv(void* buf, int count, MPI_Datatype type,
             int source, int tag, MPI_Comm comm,
             MPI_Status* status);
```

MPI_Recv matches a message sent by MPI_Send if tag, source and dest match.

Wildcards for MPI_Recv: MPI_ANY_TAG, MPI_ANY_SOURCE

```
int MPI_Ssend(void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm)
// synchronous send: returns when the destination has started
   to receive the message

int MPI_Bsend(void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm)
// buffered send: returns after making a copy of the buffer.
   The dest. might not yet have started to recv. the message

int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
// standard send: can be synchronous or buffered, depending on
   message size

int MPI_Rsend(void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm)
// ready send: an optimized send if the user can guarantee
   that the dest. has already posted the matching receive

int MPI_Sendrecv
  (void *sendbuf, int sendcount, MPI_Datatype sendtype,
   int dest, int sendtag, void *recvbuf, int recvcount,
   MPI_Datatype recvtype, int source, int recvtag,
   MPI_Comm comm, MPI_Status *status)
```

23.4 MPI: Nonblocking Send and Receive [12, 32]

```
int MPI_Issend(void *buf, int count, MPI_Datatype datatype,
               int dest, int tag, MPI_Comm comm,
               MPI_Request *request)

int MPI_Ibsend(void *buf, int count, MPI_Datatype datatype,
               int dest, int tag, MPI_Comm comm,
               MPI_Request *request)

int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm,
              MPI_Request *request)

int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm,
              MPI_Request *request)
```

MPI_Request object can be used to test for completion.

23.5 MPI: Probing for Messages [12, 22]

```
int MPI_Probe(int source, int tag, MPI_Comm comm,
              MPI_Status *status)
// wait for a matching message to arrive

int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,
               MPI_Status *status)
// check if a message has arrived
// flag is nonzero if there is a message waiting

int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype,
                  int* count)
// gets the number of elements in the message
```

Fields in MPI_Status: MPI_SOURCE, MPI_TAG, MPI_ERROR

23.6 MPI: Buffered Send Example [12, 27]

```
int main(int argc, char** argv) {
  MPI_Status status;
  int num;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&num);
  double ds=3.1415927; // to send
  double dr; // to receive
  int tag=99;
  // allocate a buffer and attach it to MPI
  int buffer_size = sizeof(double) + MPI_BSEND_OVERHEAD;
  char* buffer = new char[buffer_size];
  MPI_Buffer_attach(buffer,buffer_size);
  if(num==0) {
    MPI_Bsend(&ds,1,MPI_DOUBLE,1,tag,MPI_COMM_WORLD);
    MPI_Recv (&dr,1,MPI_DOUBLE,1,tag,MPI_COMM_WORLD,&status);
  }
  else {
    MPI_Bsend(&ds,1,MPI_DOUBLE,0,tag,MPI_COMM_WORLD);
    MPI_Recv (&dr,1,MPI_DOUBLE,0,tag,MPI_COMM_WORLD,&status);
  }
  // detach the buffer, making sure all sends are done
  MPI_Buffer_detach(buffer,&buffer_size);
  delete[] buffer;
  MPI_Finalize();
  return 0;
}
```

23.7 MPI: Waiting for Completion [12, 33]

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
// waits for the communication to finish and fills in status

int MPI_Waitall(int count, MPI_Request array_of_requests[],
                MPI_Status array_of_statuses[])
// waits for all given communications to finish and fills in
   the statuses

int MPI_Waitany(int count, MPI_Request array_of_requests[],
                int *index, MPI_Status *status)
// waits for one of the given communications to finish, sets
   the index to indicate which one and fills in the status

int MPI_Waitsome(int incount, MPI_Request array_of_requests[],
                 int *outcount, int array_of_indices[],
                 MPI_Status array_of_statuses[])
// waits for at least one of the given communications to
   finish, sets the number of communication requests that have
   finished, their indices and status
```

23.8 MPI: Testing for Completion and Cancellation [12, 34]

```
int MPI_Test(MPI_Request *request, int *flag,
             MPI_Status *status)
// tests if the communication is finished. Sets flag to 1 and
   fills in the status if finished or sets the flag to 0

int MPI_Testall(int count, MPI_Request array_of_requests[],
                int *flag, MPI_Status array_of_statuses[])
// test whether all given communications are finished. Sets
   flag to 1 and fills in the status aray if all are finished
   or sets the flag to 0 if not all are finished.

int MPI_Testany(int count, MPI_Request array_of_requests[],
                int *index, int *flag, MPI_Status *status)
// test whether one of the given communications is finished.
   Sets flag to 1 and fills in the index and status if one
   finished or sets the flag to 0 if none is finished.

int MPI_Testsome(int incount, MPI_Request array_of_requests[],
                 int *outcount, int array_of_indices[],
                 MPI_Status array_of_statuses[])
// tests whether some of the given communications is finished,
   sets the number of communication requests that have
   finished, their indices and statuses.

int MPI_Cancel(MPI_Request *request)
```

23.9 MPI: Diffusion Example [12, 35]

```
for (int t=0; t<iterations; ++t) {
  // first start the communications
  if (rank % 2 == 0) {
    MPI_Isend(&density[1],1,MPI_DOUBLE,left,0,MPI_COMM_WORLD,
              &reqs[0]);
    MPI_Irecv(&density[0],1,MPI_DOUBLE,left,0,MPI_COMM_WORLD,
              &reqs[1]);
    MPI_Isend(&density[local_N-2],1,MPI_DOUBLE,right,0,
              MPI_COMM_WORLD,&reqs[2]);
    MPI_Irecv(&density[local_N-1],1,MPI_DOUBLE,right,0,
              MPI_COMM_WORLD,&reqs[3]);
  }
  else {
    MPI_Irecv(&density[local_N-1],1,MPI_DOUBLE,right,0,
              MPI_COMM_WORLD,&reqs[0]);
    MPI_Isend(&density[local_N-2],1,MPI_DOUBLE,right,0,
              MPI_COMM_WORLD,&reqs[1]);
    MPI_Irecv(&density[0],1,MPI_DOUBLE,left,0,
              MPI_COMM_WORLD,&reqs[2]);
    MPI_Isend(&density[1],1,MPI_DOUBLE,left,0,
              MPI_COMM_WORLD,&reqs[3]);
  }
  // do calculation of the interior
  for (int i=2; i<local_N-2;++i)
    newdensity[i] = density[i] + coefficient *
                 (density[i+1]+density[i-1]-2.*density[i]);
  // wait for the ghost cells to arrive
  MPI_Waitall(4,reqs,status);
  // do the boundaries
  newdensity[1] = density[1] + coefficient *
               (density[2]+density[0]-2.*density[1]);
  newdensity[local_N-2] = density[local_N-2] + coefficient *
                       (density[local_N-1]+
                        density[local_N-3]-
                     2.*density[local_N]);
  // and swap
  density.swap(newdensity);
}
```

24. MPI: COLLECTIVE COMMUNICATION

24.1 MPI: Collective Reductions [12, 38-42]

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int root,
               MPI_Comm comm);
// performs a reduction using the operation op on the data in
   sendbuf and places the results in recvbuf on the root rank.
   if MPI_IN_PLACE is specified as sendbuf then the data to be
   reduced is assumed to be in the recvbuf and will be
   replaced on the root rank

int MPI_Allreduce(void *sendbuf, void *recvbuf, int count,
      MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
// performs a reduction using the operation op on the data in
   sendbuf and places the results in recvbuf on all ranks
   if MPI_IN_PLACE is specified as sendbuf then the data to be
   reduced is assumed to be in the recvbuf and will be
   replaced by the reduction
```

op	**description**	**op**	**description**
MPI_MAX	Maximum	MPI_LAND	Logical AND
MPI_MIN	Minimum	MPI_BAND	Bitwise AND
MPI_SUM	Sum	MPI_LOR	Logical OR
MPI_PROD	Product	MPI_BOR	Bitwise OR
MPI_MAXLOC	Max. and location	MPI_LXOR	Logical XOR
MPI_MINLOC	Min. and location	MPI_BXOR	Bitwise XOR

Example: Parallelising the sum of π

```
int size, rank;
MPI_Comm_size(MPI_COMM_WORLD,&size);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
long double sum=0.;
long double localsum=0.;
long double const step = (nterms+0.5l) / size;
// do just one piece on each rank
unsigned long start = rank * step;
unsigned long end = (rank+1) * step;
for (std::size_t t = start; t < end; ++t)
  localsum += (1.0 - 2* (t % 2)) / (2*t + 1);
// now collect all to the master (rank 0)
MPI_Reduce(&localsum, &sum, 1, MPI_LONG_DOUBLE, MPI_SUM,
           0, MPI_COMM_WORLD);
if (rank==0) // only one prints
  cout << "pi=" << setprecision(18) << 4.*sum << endl;
```

Use **MPI_IN_PLACE** to avoid local and global sums:

```
  MPI_Reduce(rank == 0 ? MPI_IN_PLACE : &sum, &sum, 1,
             MPI_LONG_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Broadcast:

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
              int root, MPI_Comm comm )
// broadcast the data from the root rank to all others
```

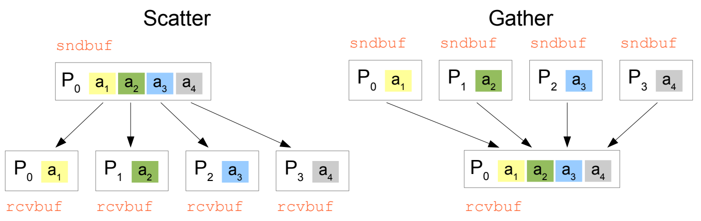
24.2 MPI: Packing [12, 43-]

```
int MPI_Pack(void *inbuf, int incount, MPI_Datatype datatype,
             void *outbuf, int outcount, int *position,
             MPI_Comm comm)
// packs the data given as input into the outbuf buffer
   starting at a given position. outcount is the size of the
   buffer and position gets updated to point to the first free
   byte after packing in the data. An error is returned if the
   buffer is too small.

int MPI_Unpack(void *inbuf, int insize, int *position,
               void *outbuf, int outcount,
               MPI_Datatype datatype, MPI_Comm comm)
// unpack data from the buffer starting at given position into
   the buffer outbuf. position is updated to point to the
   location after the last byte read

int MPI_Pack_size(int incount, MPI_Datatype datatype,
                  MPI_Comm comm, int *size)
// returns in size an upper bound for the number of bytes
   needed to pack incount values of type datatype. This can be
   used to determine the required buffer size
```

Example:

```
int size_double, size_int;
MPI_Pack_size(1, MPI_DOUBLE, MPI_COMM_WORLD,&size_double);
MPI_Pack_size(1, MPI_INT, MPI_COMM_WORLD,&size_int);
int buffer_size = 2*size_double+size_int;
char* buffer = new char[buffer_size];
int pos=0;
MPI_Pack(&a, 1, MPI_DOUBLE, buffer, buffer_size, &pos, MPI_COMM_WORLD);
MPI_Pack(&b, 1, MPI_DOUBLE, buffer, buffer_size, &pos, MPI_COMM_WORLD);
MPI_Pack(&steps, 1, MPI_INT, buffer, buffer_size, &pos, MPI_COMM_WORLD);
MPI_Bcast(buffer, buffer_size, MPI_PACKED, 0, MPI_COMM_WORLD);
delete[] buffer;
```

24.3 MPI: Scatter and Gather [12, 46]



scatter: splits a vector over the other ranks
gather: collects data from the other ranks into a big buffer

24.4 MPI: Gather Operations [12, 47]

MPI_IN_PLACE can be used for the sendbuf

```
int MPI_Gather
  (void *sendbuf, int sendcnt, MPI_Datatype sendtype,
   void *recvbuf, int recvcnt, MPI_Datatype recvtype,
   int root, MPI_Comm comm)
// gathers data from the sendbuf buffers into a recvbuf buffer
   on the root rank recvbuf, recvcnt and recvtype are
   significant only on the root rank
// Note: the sendcnt needs to be the same on all ranks

int MPI_Gatherv
  (void *sendbuf, int sendcnt, MPI_Datatype sendtype,
   void *recvbuf, int *recvcnts, int *displs,
   MPI_Datatype recvtype, int root, MPI_Comm comm)
// similar to MPI_Gather but the sendcnt values can differ
   from rank to rank the root node thus gets an array of
   recvcnts and of displacements displs. The displacements
   specify where the data from each rank starts in the buffer

int MPI_Allgather(void *sendbuf, int sendcnt,
                  MPI_Datatype sendtype, void *recvbuf,
                  int recvcnt, MPI_Datatype recvtype,
                  MPI_Comm comm)
// similar to MPI_Gather, but the data is gathered at all
   ranks and not just a root it is semantically the same as an
   MPI_Gather followed by MPI_Bcast

int MPI_Allgatherv(void *sendbuf, int sendcount,
                   MPI_Datatype sendtype, void *recvbuf,
                   int *recvcounts, int *displs,
                   MPI_Datatype recvtype, MPI_Comm comm)
// similar to MPI_Gatherv, but the data is gathered at all
   ranks and not just a root it is semantically the same as an
   MPI_Gatherv followed by MPI_Bcast
```

24. MPI: COLLECTIVE COMMUNICATION (CONT.)

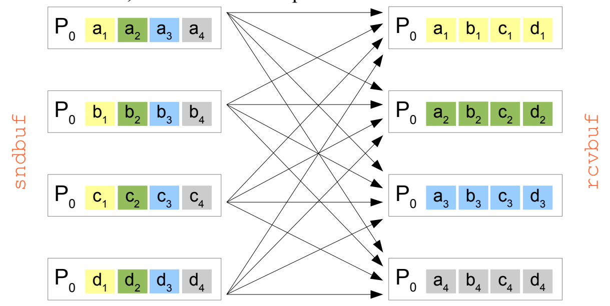
24.5 MPI: Scatter Operations [12, 48]

MPI_IN_PLACE can be used for the recvbuf

```
int MPI_Scatter(void *sendbuf, int sendcnt,
                MPI_Datatype sendtype, void *recvbuf,
                int recvcnt, MPI_Datatype recvtype, int root,
                MPI_Comm comm)
// scatters data from the sendbuf buffer on the root rank into
   recvbuf buffers on the other ranks. Each rank gets a
   corresponding junk of the data sendbuf, sendcnt and
   sendtype are significant only on the root rank
// Note: recvcnt needs to be the same on all ranks

int MPI_Scatterv( void *sendbuf, int *sendcnts, int *displs,
                  MPI_Datatype sendtype, void *recvbuf,
                  int recvcnt, MPI_Datatype recvtype,
                  int root, MPI_Comm comm)
// similar to MPI_Scatter but the sendcnt values can differ
   from rank to rank the root node thus spcifies an array of
   recvcnts and of displacements displs. The displacements
   specify where the data for each rank starts in the buffer

int MPI_Reduce_scatter(void *sendbuf, void *recvbuf,
                       int *recvcnts, MPI_Datatype datatype,
                       MPI_Op op, MPI_Comm comm)
// optimized version of an MPI_Reduce followed by an
   MPI_Scatter, MPI_IN_PLACE can be used for the sendbuf
```

24.6 MPI: All-to-All and Barrier [12, 46]

MPI_Alltoall: *n*-th rank sends *k*-th portion of its data to rank *k* and receives *n*-th portion from node *k*, like a matrix transpose. Slow!


sndbuf
P_0 a_1 a_2 a_3 a_4
P_0 b_1 b_2 b_3 b_4
P_0 c_1 c_2 c_3 c_4
P_0 d_1 d_2 d_3 d_4
P_0 a_1 b_1 c_1 d_1
P_0 a_2 b_2 c_2 d_2
P_0 a_3 b_3 c_3 d_3
P_0 a_4 b_4 c_4 d_4
rcvbuf

```
int MPI_Alltoall(void *sendbuf, int sendcount,
                 MPI_Datatype sendtype, void *recvbuf,
                 int recvcount, MPI_Datatype recvtype,
                 MPI_Comm comm)
```

MPI barrier: waits for all ranks to call it, used for synchronisation

```
int MPI_Barrier( MPI_Comm comm )
```

25. MPI: USER DEFINED DATA TYPES

For simpson integration we have to broadcast three parameters. By now, three options, none was ideal:

- 3 broadcasts: wasteful since 3 communications
- packing it into a buffer: wasteful since it involves copying
- sending a struct bitwise: dangerous since it assumes homogeneity

Solution: MPI data types

25.1 MPI: Building a MPI Data Type [12, 52-55]

```
int MPI_Type_create_struct
  (int count, int blocklengths[], MPI_Aint offsets[],
   MPI_Datatype types[], MPI_Datatype *newtype)
// builds an MPI data type for a data type for a general data
   structure given by types, counts (blocklengths) and their
   offsets relative to the start of the data structure

int MPI_Get_address(void *location, MPI_Aint *address)
// converts a pointer to the integer type used internally by
   MPI to store pointers

int MPI_Type_commit(MPI_Datatype *datatype)
// commits the data type: finished building, can now be used.
int MPI_Type_free(MPI_Datatype *datatype)
// frees the data type, releasing any allocated memory
```

Example:

```
struct parms { // define a struct for the parameters
  double a; // lower bound of integration
  double b; // upper bound of integration
  int nsteps; // number of subintervals for integration
};
parms p;
// describe the struct through sizes, offsets and types
// the safe way getting addresses
MPI_Aint p_lb, p_a, p_nsteps, p_ub;
MPI_Get_address(&p, &p_lb); // start of the struct is the
                               lower bound
MPI_Get_address(&p.a, &p_a); // address of the first double
MPI_Get_address(&p.nsteps, &p_nsteps); // address of the
                                          integer
MPI_Get_address(&p+1, &p_ub); // start of the next struct is
                                 the upper bound
int blocklens[] = {0, 2, 1, 0};
MPI_Datatype types[] = {MPI_LB, MPI_DOUBLE, MPI_INT, MPI_UB};
MPI_Aint offsets[] = {0, p_a-p_lb, p_nsteps-p_lb, p_ub-p_lb};
MPI_Datatype parms_t;
MPI_Type_create_struct(2, blocklens, offsets, types,&parms_t);
MPI_Type_commit(&parms_t);
// read the parameters on the master rank
if (rank==0) std::cin >> p.a >> p.b >> p.nsteps;
// broadcast the parms now using our type
MPI_Bcast(&p, 1, parms_t, 0, MPI_COMM_WORLD);
// and now free the type
MPI_Type_free(&parms_t);
```

25.2 MPI: Receiving a List Into a Vector [12, 56]

One can give absolute addresses as offsets if passing **MPI_BOTTOM**:

```
if(num==0) {
  // receive data into a vector and print it
  std::vector<int> data(10);
  MPI_Status status;
  MPI_Recv(&data[0], 10, MPI_INT, 1, 42, MPI_COMM_WORLD,
           &status);
  for (int i=0; i < data.size(); ++i)
    std:: cout << data[i] << "\n";
}
else {
  // fill a list with the numbers 0-9 and send it
  std::list<int> data;
  for (int i=0; i<10; ++i)
    data.push_back(i);
  std::vector<MPI_Datatype> types(10,MPI_INT);
  std::vector<int> blocklens(10,1);
  std::vector<MPI_Aint> offsets;
  for (int& x : data) {
    MPI_Aint address;
    MPI_Get_address(&x, &address); // use absolute addresses
    offsets.push_back(address);
  }
  MPI_Datatype list_type;
  MPI_Type_create_struct(10, &(blocklens[0]), &offsets[0],
                         &types[0] ,&list_type);
  MPI_Type_commit(&list_type);
  MPI_Send(MPI_BOTTOM, 1, list_type, 0, 42, MPI_COMM_WORLD);
  MPI_Type_free(&list_type);
}
```

25. MPI: USER DEFINED DATA TYPES (CONT.)

25.3 MPI: Arrays and Vectors [12, 57-58]

```
int MPI_Type_contiguous(int count, MPI_Datatype old_type,
                        MPI_Datatype *new_type_p)
// build an MPI datatype for a contiguous array

int MPI_Type_vector(int count, int blocklength, int stride,
                    MPI_Datatype old_type,
                    MPI_Datatype *newtype_p)
// build an MPI datatype for a vector array of blocklength
   contiguous entries that are spaced at a given stride.
   stride is like the leading dimension in BLAS and specifies
   the distance between blocks

int MPI_Type_create_hvector(int count, int blocklen,
                            MPI_Aint stride,
                            MPI_Datatype old_type,
                            MPI_Datatype *newtype_p)
// like MPI_Type_vector but now the stride is given in bytes
```

Example: data types for rows and columns of a matrix

```
hpc12::matrix<double,hpc12::column_major> a(4,4);
MPI_Datatype row, col;
MPI_Type_contiguous(4, MPI_DOUBLE, &col);
MPI_Type_vector (4, 1, 4, MPI_DOUBLE, &row);
MPI_Type_commit(&row);
MPI_Type_commit(&col);
// use them
// ...
// and finally free them
MPI_Type_free(&row);
MPI_Type_free(&col);
```

25.4 MPI: Subarrays [12, 59]

```
int MPI_Type_create_subarray
  (int ndims, int sizes[], int subsizes[], int starts[],
   int order, MPI_Datatype oldtype, MPI_Datatype *newtype)
// build an MPI datatype for a subarray of a larger array:
// ndims: number of dimensions
// sizes: extent of the full array in each dimension
// subsizes: extent of the subarray in each dimension
// starts: starting index of the subarray
// order: array storage order, can be either of MPI_ORDER_C
          or MPI_ORDER_FORTRAN
```

e.g. for 2D diffusion

25.5 MPI: Indexed Data Types [12, 60]

For sending just some elements of an array. For example, send some particles to a different cell list.

```
int MPI_Type_indexed(int count, int blocklens[],
                     int indices[], MPI_Datatype old_type,
                     MPI_Datatype *newtype)
// build an MPI datatype selecting specific entries from a
   contiguous array. Starting a at each of the given indices a
   number of elements given in the corresponding entry of
   blocklens is chosen.

int MPI_Type_create_hindexed
  (int count, int blocklens[], MPI_Aint displacements[],
   MPI_Datatype oldtype, MPI_Datatype *newtype)
// same as MPI_Type_indexed but now instead of indices the
   displacement in bytes from the start of the array is
   specified

int MPI_Type_create_indexed_block
  (int count, int blocklength, int array_of_displacements[],
   MPI_Datatype oldtype, MPI_Datatype *newtype)
// same as MPI_Type_indexed but with constant sized blocks
```

26. MPI: GROUPS AND COMMUNICATORS

We want to split the ranks into groups and build a new communicator for each group. We can then do collective operations within a group instead of within all ranks.

26.1 MPI: Communicators [15, 5]

```
int MPI_Comm_rank( MPI_Comm comm, int *rank )

int MPI_Comm_size( MPI_Comm comm, int *size )

int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2,
                     int *result)
// compares two communicators to test is they are the same,
// i.e. they have the same ranks in the same order

int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
// duplicates a communicator. This is a collective
// communication that needs to be called by all ranks.

int MPI_Comm_free(MPI_Comm *comm)
// frees a communicator

int MPI_Comm_split(MPI_Comm comm, int color, int key,
                   MPI_Comm *newcomm)
// splits a communicator into subcommunicators. Ranks with the
// same color are grouped together and sorted within each
// group by key. This is a collective communication that needs
// to be called by all ranks.

int MPI_Comm_create(MPI_Comm comm, MPI_Group group,
                    MPI_Comm *newcomm)
// creates a new communicator based on group that is a
// subgroup of the ranks in comm. This function allows more
// flexible creation of subcommunicators than MPI_Comm_split.
// This is a collective communication that needs to be called
// by all ranks.
```

26.2 MPI: Simpson Integration Example [15, 3-4]

```
double parallel_simpson(MPI_Comm comm, parms p) {
  // get the rank and size for the current communicator
  int size, rank;
  MPI_Comm_size(comm,&size);
  MPI_Comm_rank(comm,&rank);
  // integrate just one part on each rank
  double delta = (p.b-p.a)/size;
  double result = simpson(func,p.a+rank*delta,p.a+
                          (rank+1)*delta,p.nsteps/size);
  // collect the results to all ranks
  MPI_Allreduce(MPI_IN_PLACE, &result, 1, MPI_DOUBLE,
                MPI_SUM, comm);
  return result;
}

int main(int argc, char** argv) {
  MPI_Init(&argc,&argv);
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  // we want to do three integrals at once
  parms p[3];
  ...
  // split the ranks into three groups
  int which = rank % 3;
  MPI_Comm comm;
  MPI_Comm_split(MPI_COMM_WORLD, which, rank, &comm);
  // do the integration in each group
  double result = parallel_simpson(comm,p[which]);
  // only the master for each group prints
  int grouprank;
  MPI_Comm_rank(comm, &grouprank);
  if (grouprank==0)
  cout << "Integration " << which << " results in " << result;
  // free the type and the new communicator
  MPI_Comm_free(&comm);
  MPI_Finalize();
  return 0;
}
```

26. MPI: GROUPS AND COMMUNICATORS (CONT.)

26.3 MPI: Groups [15, 6-7]

```
int MPI_Group_rank(MPI_Group group, int *rank)
int MPI_Group_size(MPI_Group group, int *size)
// are similar to the corresponding communicator functions

int MPI_Group_translate_ranks(MPI_Group group1, int n,
            int *ranks1, MPI_Group group2, int *ranks2)
// translates ranks between group: given a set of ranks1 in
// group1 it sets their ranks in group2 in the array ranks2,
// or sets them to MPI_UNDEFINED if no correspondence exists

int MPI_Group_compare(MPI_Group group1, MPI_Group group2,
                    int *result)

int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
// extracts the group from a communicator

int MPI_Group_free(MPI_Group *group)

int MPI_Group_union(MPI_Group group1, MPI_Group group2,
                  MPI_Group *newgroup)
int MPI_Group_intersection(MPI_Group group1, MPI_Group group2,
                         MPI_Group *newgroup)
int MPI_Group_difference(MPI_Group group1, MPI_Group group2,
                       MPI_Group *newgroup)
// newgroup is the union, intersection, or difference of the
// given groups
```

Selectively choosing ranks:

```
int MPI_Group_incl(MPI_Group group, int n, int *ranks,
                 MPI_Group *newgroup)
// create a newgroup containing only the given ranks of a
// group

int MPI_Group_excl(MPI_Group group, int n, int *ranks,
                 MPI_Group *newgroup)
// create a newgroup containing all except the given ranks of
// a group

int MPI_Group_range_incl(MPI_Group group, int n,
                       int ranges[][3], MPI_Group *newgroup)
// create a newgroup containing only the given ranges of ranks
// of a group

int MPI_Group_range_excl(MPI_Group group, int n,
                       int ranges[][3], MPI_Group *newgroup)
// create a newgroup containing all except the given ranges of
// ranks of a group
```

Ranges are given as triples (first, last, stride), and a range includes the ranks

$$\text{first}, \text{first}+\text{stride}, \text{first}+2\cdot\text{stride}, \ldots, \text{first}+\left(\frac{\text{last}-\text{first}}{\text{stride}}\right)\text{stride}$$

27. MPI: TOPOLOGIES

To find the rank of the neighbour (cells) use MPI topologies.

Two main types:

- Cartesian topology: each process is "connected" to its neighbours in a virtual grid. Nodes are labeled by cartesian indices, boundaries can be cyclic.
- Graph topology: an arbitrary connection graph

27.1 MPI: Cartesian Topologies [15, 11]

To work with a regular mesh with row-major ordering we create a cartesian communicator

```
  int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims,
                int *periods, int reorder, MPI_Comm *comm_cart)
```

comm_old	the original communicator
ndims	number of dimensions
dims	integer array specifying the number of processes in each dimension
periods	integer array of boolean values whether the grid is periodic in that dimension
reorder	boolean flag whether the processes may be reordered
comm_cart	a new cartesian grid communicator

To get automatic splitting into approximately equal counts in each dimension use

```
  int MPI_Dims_create(int nnodes, int ndims, int *dims)
  // fills in the dims array to be the best fit of arranging
  // nnodes ranks to form an ndims dimensional array
```

27.2 MPI: Cartesian Communicator Example [15, 13]

```
int main(int argc, char** argv) {
  MPI_Init(&argc,&argv);
  int size, rank;
  MPI_Status status;
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  int nums[3] = {0,0,0};
  int periodic[3] = {false, false, false};
  // split the nodes automatically
  MPI_Dims_create(size, 3, nums);
  if (rank==0)
    cout << "We create a " << nums[0] << "x" << nums[1] << "x"
         << nums[2] << " arrangement.\n";
  // now everyone creates a a cartesian topology
  MPI_Comm cart_comm;
  MPI_Cart_create(MPI_COMM_WORLD, 3, nums, periodic, true,
                  &cart_comm);
  MPI_Comm_free(&cart_comm);
  MPI_Finalize();
}
```

27.3 MPI: Cartesian Communicator Functions [15, 14-16]

The neighbours are obtained by

```
  int MPI_Cart_shift(MPI_Comm comm, int direction,
                   int displacement, int *source, int *dest)
  // gives the ranks shifted in the dimension given by
  // direction by a certain displacement, where the sign of
  // displacement indicates the direction. It returns both the
  // source rank from which the current rank can be reached by
  // that shift and the dest rank that is reached from the
  // current rank by that shift.
```

3D Example:

```
  int left, right, bottom, top, front, back, newrank;
  MPI_Comm_rank(cart_comm,&newrank);
  MPI_Cart_shift(cart_comm, 0, 1, &left, &right);
  MPI_Cart_shift(cart_comm, 1, 1, &bottom, &top);
  MPI_Cart_shift(cart_comm, 2, 1, &front, &back);
  cout << "Rank " << rank << " has new rank " << newrank
       << " and neighbors " << left << ", " << right
       << ", " << top << ", " << bottom
       << ", " << front << ", " << back << endl;
```

Get number of dimensions:

```
  int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
```

Get the cartesian topology information:

```
  int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims,
                 int *periods, int *coords)
  // retrieves information about the cartesian topology
  // associated with a communicator. The arrays are allocated
  // with maxdims dimensions. dims and periods are the numbers
  // used when creating the topology. coords are the
  // dimensions of the current rank.
```

Get the rank of a given coordinate:

```
  int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
```

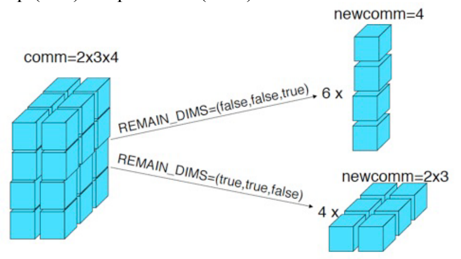
Get the coordinate of a given rank:

```
  int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims,
                    int *coords)
```

One can split the cartesian communicator into sub grid communicators for columns, rows, planes, ...

```
  int MPI_Cart_sub(MPI_Comm comm, int *remain_dims,
                 MPI_Comm *comm_new)
```

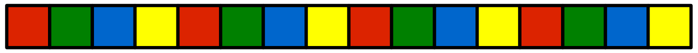
The remain_dims array specifies whether to keep the processes along a direction joined in a group (true) or split them (false).



28. DISTRIBUTED LINEAR ALGEBRA

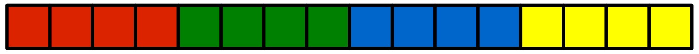
28.1 Distributed Vector Storage [15, 19]

Cyclic distribution:



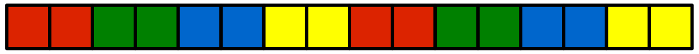
- element i stored on rank rank(i) = i mod p
- local index of element i is local(i) = floor(i/p)

Block distribution:



- element i stored on rank rank(i) = floor(i/b), where b = ceil(N/p)
- local index of element i is local(i) = i mod b

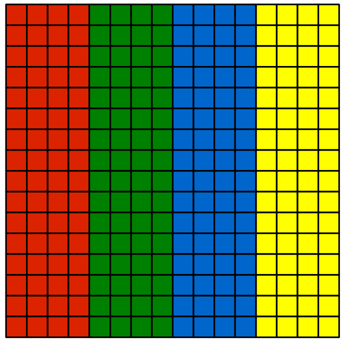
Block-cyclic distribution:



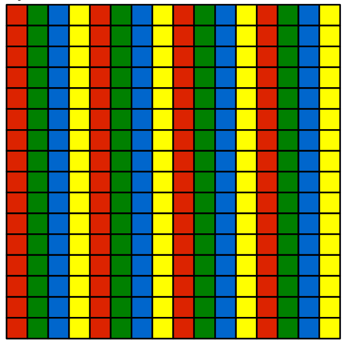
Code for distributed vector on slides 20-21.

28.2 Distributed Matrix Storage [15, 22-24]

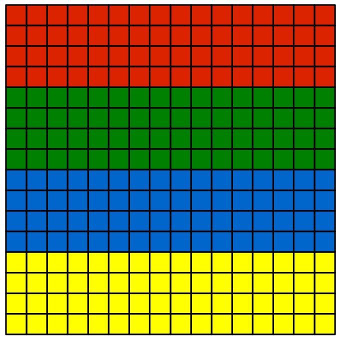
Block column distribution



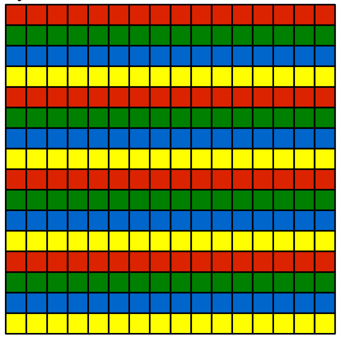
Cyclic column distribution



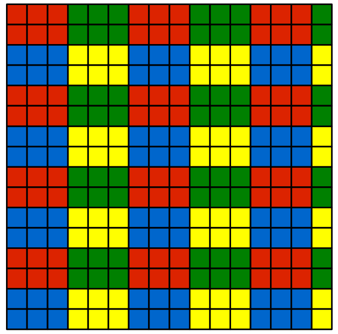
Block row distribution



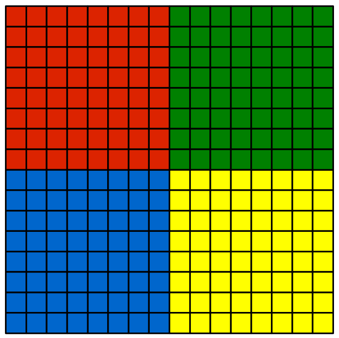
Cyclic row distribution



Block cyclic distribution (3x2)



Block cyclic distribution (perfect fit)



Code and more details for distributed matrix operations on slides 25-29.

28.3 Sparse Matrix-Vector Multiplication [15, 30-33]

Let

- vector dimension N
- sparsity $a \rightarrow aN$ non-zeroes per row or column
- number of ranks $p \rightarrow$ block size $b=N/p$

Block row distributions need to gather vector to every rank:
N numbers collected to every rank.
Block column distribution can send a sparse result vector:
baN numbers sent from every rank.
Block column uses less communication data if $aN<p$. But sparse data produces overhead. Hence, you have to time it.

Recall the matrix multiplications. Three versions, neither of which scaled very well (exchange i,j,k for the other versions):

```
for(unsigned int i=0; i < m; ++i)
  for(unsigned int j=0; j < n; ++j)
    for(unsigned int k=0; k < l; ++k)
      c(i,j) += a(i,k) * b(k,j);
```

These BLAS-2 operations perform N operations for N data accesses and are thus limited by memory bandwidth. Recall the roofline model: we need to make more computations per byte that we load from memory.

Blocking of matrix multiplies
The solution: block the operations and do b of these matrix-vector multiplications or vector-vector outer products at once. Data is then reused b times and thus we do bN operations for N memory accesses.

Block row distribution required an all-gather of the full vector.
Block column distribution built a full-sized vector.

- We need the full matrix on one node! Might run out of memory!
- Lots of communication.

Block cyclic distribution needed memory only for a row or column:

- less memory requirements
- less network traffic

28.4 Parallel Matrix Multiplication [15, 34-35]

Matrix multiplication $C = A \cdot B$.
Distribute matrices over the nodes of a parallel computer:

$$\begin{array}{|c|c|c|c|}\hline C_{11} & C_{12} & C_{13} & C_{14} \\ \hline C_{21} & C_{22} & C_{23} & C_{24} \\ \hline C_{31} & C_{32} & C_{33} & C_{34} \\ \hline C_{41} & C_{42} & C_{43} & C_{44} \\ \hline \end{array} = \begin{array}{|c|c|c|c|}\hline A_{11} & A_{12} & A_{13} & A_{14} \\ \hline A_{21} & A_{22} & A_{23} & A_{24} \\ \hline A_{31} & A_{32} & A_{33} & A_{34} \\ \hline A_{41} & A_{42} & A_{43} & A_{44} \\ \hline \end{array} \bullet \begin{array}{|c|c|c|c|}\hline B_{11} & B_{12} & B_{13} & B_{14} \\ \hline B_{21} & B_{22} & B_{23} & B_{24} \\ \hline B_{31} & B_{32} & B_{33} & B_{34} \\ \hline B_{41} & B_{42} & B_{43} & B_{44} \\ \hline \end{array}$$

Need to send data around: $C_{ij} = A_{i1} B_{1j} + A_{i2} B_{2j} + A_{i3} B_{3j} + A_{i4} B_{4j}$
A_{ij} is needed on all rows i, B_{ij} is needed on all columns j.
8
Do an all-gather along rows and columns, then calculate the local C_{ij} .

Code example:

```
// prepare row and column communicators like before
...
// allocate a block of the matrix everywhere and fill it in
matrix_type A(block_size,block_size);
matrix_type B(block_size,block_size);
matrix_type C(block_size,block_size);
for (int i=0; i<block_size; ++i)
  for (int j=0; j<block_size; ++j) {
    A(i,j) = i+j+(row+col)*block_size;
    B(i,j) = i+j+(row+col)*block_size;
    C(i,j) = 0.;
  }
// allocate working space for the block row of A
// and the block column of B
vector_type Arow(block_size*block_size*q);
vector_type Bcol(block_size*block_size*q);
// 1. gather rows and columns
MPI_Allgather(A.data(),block_size*block_size,MPI_DOUBLE,
              &Arow[0],block_size*block_size,
              MPI_DOUBLE,row_comm);
MPI_Allgather(B.data(),block_size*block_size,MPI_DOUBLE,
              &Bcol[0],block_size*block_size,
              MPI_DOUBLE,col_comm);
// 2. do all multiplications
for (int i=0; i<q; ++i)
  dgemm_('N','N',block_size,block_size,block_size,1.,
         &Arow[i*block_size*block_size],block_size,
         &Bcol[i*block_size*block_size],block_size,
         1., C.data(),block_size);
```

For more advanced algorithms → slides 36-41

29. PARALLEL SPARSE MATRICES

29.1 CSR Matrix Class [9, 23]

```
// an (incomplete) CSR class
template <class ValueType, class SizeType=std::size_t>
class csr_matrix
{
  typedef ValueType value_type;
  typedef SizeType size_type;

  csr_matrix(size_type s = 0)
  : n_(s)
  , row_starts(s+1)
  {}

  // we are missing functions to actually fill the matrix
  size_type dimension() const { return n_;}

  std::vector<value_type> multiply(std::vector<value_type>
                                   const& x) const;
private:
  size_type n_;
  std::vector<size_type> col_indices;
  std::vector<size_type> row_starts;
  std::vector<value_type> data;
};
```

29.2 Matrix-Vector Multiplication [9, 24]

Use CSR! CSC would require an atomic update to avoid race conditions.

```
template <class ValueType, class SizeType>
std::vector<ValueType>
     csr_matrix<ValueType,SizeType>::multiply
     (std::vector<value_type> const& x) const
{
  assert( x.size()== dimension());
  std::vector<value_type> y(dimension());
  // loop over all rows
  #pragma omp parallel for
  for (size_type row = 0 ; row < dimension() ; ++ row)
    // loop over all non-zero elements of the row
    for (size_type i = row_starts[row] ;
         i != row_starts[row+1] ; ++i)
      y[row] += data[i] * x[col_indices[i]];
  return y;
}
```

29.3 Matrix-Vector Multiplication with the Tranposed [9, 27]

Use CSC! Potential race condition for CSR.

```
template <class ValueType, class SizeType>
std::vector<ValueType>
     csc_matrix<ValueType,SizeType>::multiply
     (std::vector<value_type> const& x) const
{
  assert( x.size()== dimension());
  std::vector<value_type> y(dimension());
  // loop over all columns
  #pragma omp parallel for
  for (size_type col = 0 ; col < dimension() ; ++ col)
    // loop over all non-zero elements of the column
    for (size_type i = col_starts[col] ;
         i != col_starts[col+1] ; ++i)
      y[col] += data[i] * x[row_indices[i]];
  return y;
}
```

30. BLAS

30.1 BLAS: Basic Linear Algebra Subprograms [10, 2-7]

BLAS level 1:
- scalar and vector operations
- scale as O(1) or O(N)

BLAS level 2:
- matrix-vector operations
- scale as O(N^2)

BLAS level 3:
- matrix-matrix operations
- scale worse than O(N^2), often O(N^3)

BLAS is a Fortran library. Important to know:
- Parameter types are not mangled into the function name. Use **extern "C"** in the function declaration.
- Pass scalar arguments by reference.
- Pass C-style arrays as pointers.
- Be careful about how integer types relative. This can depend on compiler options. Typically a Fortran integer is a C int, but it can be a long.

```
g++ -std=c++11 -O3 -Wall -lgfortran -fopenmp -o main main.cpp
```

Do not forget to link against the BLAS library!

Example
Fortan DDOT function:
```
  DOUBLE PRECISION FUNCTION DDOT(N,DX,INCX,DY,INCY)
  INTEGER INCX,INCY,N
  DOUBLE PRECISION DX(*),DY(*)
```

C++ prototype:
```
  extern "C" double ddot_(int& n, double *x, int& incx,
                          double *y, int& incy);
```

Sample usage:
```
  int main() {
    // intialize a vector with ten 1s
    std::vector<double> x(10, 1.);
    // intialize a vector with ten 2s
    std::vector<double> y(10, 2.);
    // calculate the inner product
    int n=x.size();
    int one = 1;
    double d = ddot_(n,&x[0],one,&y[0],one);
    std:: cout << d << "\n"; // should be 20
  }
```

30.2 BLAS: Array Storage [10, 8]

Fortran indices by default start at 1, while C starts at 0.

Fortran stores arrays in column-major order, while C uses row-major order.

Column-major (Fortran)

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

Row-major (C)

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Consequences:
- matrices are typically transposed
- A[i][j] in C is A(j+1,i+1) in Fortran

30.3 BLAS: Increments [10, 9]

The DDOT dot product function takes two pointers and two increments:
```
  DOUBLE PRECISION FUNCTION DDOT(N,DX,INCX,DY,INCY)
  INTEGER INCX,INCY,N
  DOUBLE PRECISION DX(*),DY(*)
```

In arrays the increments are typically 1. The increments exist as arguments to be able to treat columns and rows in matrices as vectors:

0	5	10	15	20
1	6	11	16	21
2	7	12	17	22
3	8	13	18	23
4	9	14	19	24

DX = start of storage + 2
INCX = 5

30.4 BLAS: Naming Conventions [10, 10]

BLAS functions always have one (or two) prefix indicating the type of the arguments and optional return value.

I	int
S	float
D	double
C	std::complex<float>
Z	std::complex<double>

Example: dot product
generic name: _DOT
→ SDOT, DDOT, CDOT, ZDOT

30.5 BLAS-1: Vector Operations [10, 11]

Reduction operations:				
s	$\leftarrow$	$x \cdot y$	inner product	_DOT_
s	$\leftarrow$	$\max\{\lvert x_i \rvert\}$	pivot search	I_AMAX
s	$\leftarrow$	$\lVert x \rVert_2$	norm of a vector	_NRM2
s	$\leftarrow$	$\sum_i \lvert x_i \rvert$	sum of abs	_ASUM

Vector to vector transformations:				
y	$\leftarrow$	x	copy x into y	_COPY
x	$\leftrightarrow$	y	swap	_SWAP
y	$\leftarrow$	$\alpha \cdot x$	scale x	_SCAL
y	$\leftarrow$	$\alpha \cdot x + y$	saxpy	_AXPY

Generate and apply Givens rotation:

Compute rotation:			
$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$	$\rightarrow$	$\begin{bmatrix} r \\ 0 \end{bmatrix} \quad c, s \ni r = \sqrt{a^2 + b^2}$	_ROTG
Apply rotation:			
$\begin{bmatrix} x \\ y \end{bmatrix}$	$\leftarrow$	$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$	_ROT

30.6 BLAS: Matrix Types and Naming Conventions [10, 12]

BLAS2 and BLAS3 support various matrix types, given as two letters after the prefix:

GE	general dense matrix
GB	banded matrix, stored packed
SY	symmetric, stored like a general dense matrix
SP	symmetric, stored packed
SB	symmetric banded, stored packed
HE	hermitian, stored like a general dense matrix
HP	hermitian, stored packed
HB	hermitian banded, stored packed
TR	upper or lower triangular, stored like a general dense matrix
TP	upper or lower triangular, stored packed
TB	upper or lower triangular band matrix, stored packed

Example: DGEMV is matrix-vector multiplicat. for a general matrix of doubles.

30.7 BLAS: Matrix Storage [10, 13-14]

Packed storage for triangular matrices, depending on the UPLO parameter:

UPLO	Dense storage of matrix	Packed storage as array
U	$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ & a_{22} & a_{23} & a_{24} \\ & & a_{33} & a_{34} \\ & & & a_{44} \end{pmatrix}$	$a_{11}\ a_{12}\ a_{22}\ a_{13}\ a_{23}\ a_{33}\ a_{14}\ a_{24}\ a_{34}\ a_{44}$
L	$\begin{pmatrix} a_{11} & & & \\ a_{21} & a_{22} & & \\ a_{31} & a_{32} & a_{33} & \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$	$a_{11}\ a_{21}\ a_{31}\ a_{41}\ a_{22}\ a_{32}\ a_{42}\ a_{33}\ a_{43}\ a_{44}$

Symmetric and hermitian packed formats store only one triangle.

Dense matrix storage

Matrix operations accept four size arguments:

- matrix size: rows and columns of the matrix
- leading dimension: increment between columns

0	5	10	15	20
1	6	11	16	21
2	7	12	17	22
3	8	13	18	23
4	9	14	19	24

number of rows: 3
number of columns: 3
leading dimension: 5
(operation on a submatrix)

30.8 BLAS-2: Matrix-Vector Operations [10, 15]

Matrix times Vector				
x	$\leftarrow$	$\alpha A x + \beta y$	general	_GEMV
			general band	_GBMV
			general hermitian	_HEMV
			hermitian banded	_HBMV
			hermitian packed	_HPMV
			general symmetric	_SYMV
			symmetric banded	_SBMV
			symmetric packed	_SPMV
y	$\leftarrow$	$A x$	triangular	_TRMV
			triangular banded	_TBMV
			triangular packed	_TPMV

Rank one and rank two updates:				
A	$\leftarrow$	$\alpha x y^T + A$	general	_GER_
A	$\leftarrow$	$\alpha x x^* + A$	general hermitian	_HER
			hermitian packed	_HPR
A	$\leftarrow$	$\alpha(x y^* + y x^*) + A$	general hermitian	_HER2
			hermitian packed	_HPR2
A	$\leftarrow$	$\alpha x x^T + A$	general symmetric	_SYR
			symmetric packed	_SPR
A	$\leftarrow$	$\alpha(x y^T + y x^T) + A$	general symmetric	_SYR2
			symmetric packed	_SPR2

Triangular solve:				
x	$\leftarrow$	$A^{-1} x$	triangular	_TRSV
			triangular banded	_TBSV
			triangular packed	_TPSV

30.9 BLAS-3: Matrix-Matrix Operations [10, 16]

Matrix product:				
C	$\leftarrow$	$\alpha A \cdot B + \beta C$	general	_GEMM
			symmetric	_SYMM
			hermitian	_HEMM
B	$\leftarrow$	$\alpha A \cdot B$	triangular	_TRMM
Rank *k* update:				
C	$\leftarrow$	$\alpha A \cdot A^T + \beta C$		_SYRK
C	$\leftarrow$	$\alpha A \cdot A^H + \beta C$		_HERK
C	$\leftarrow$	$\alpha(A \cdot B^T + B \cdot A^T) + \beta C$		_SYRK2
C	$\leftarrow$	$\alpha(A \cdot B^H + B \cdot A^H) + \beta C$		_HERK2
Triangular solve for multiple r.h.s.:				
B	$\leftarrow$	$\alpha A^{-1} \cdot B$	triangular	_TRSM

30.10 BLAS: Transpose Arguments [10, 17]

_GEMV, _GBMV, _T_MV, and _T_SV take arguments indicating whether the matrix should be transposed:

TRANS	**real matrix S, D**	**complex matrix C, Z**
'N' or 'n'	no transpose	no transpose
'T' or 't'	transposed	transposed
'C' or 'c'	transposed	transposed and complex conjugated

Similarly some of the BLAS-3 calls take one or two transpose arguments:
_GEMM, _TRMM, _SYRK, _HERK, _SYRK2, _TRSM

30.11 BLAS: Parallelising _GEMV [10, 19-24]

→ see slides

30.12 BLAS: Linpack-Style LU Factorisation [10, 36-40]

```
void dgefa(hpc12::matrix<double,hpc12::column_major>& a,
           std::vector<int> pivot)
{
  assert(a.num_rows() == a.num_cols());

  pivot.clear();
  int one=1;
  int n=a.num_rows();
  int lda=a.leading_dimension();

  for(int k=0; k < a.num_rows()-1; k++){
    // 1. find the index of the largest element in column k
    // starting at row k
    int nk = n-k;
    int l = idamax(nk,&a(k,k),one) + k;
    pivot.push_back(l); // and save it
    assert( a(l,k) =!0.0); // error if pivot is zero

    // 2. swap rows l and k, starting at column k
    dswap(nk,&a(l,k),lda,&a(k,k),lda);

    // 3. scale the column k below row k by the inverse
    // negative pivot element, to store L in the lower part
    double t = -1./a(k,k);
    int nkm1 = n-k-1;
    dscal(nkm1,t,&a(k+1,k),one);

    // 4. add the scaled k-th row to all rows in the lower
    // right corner
    double alpha=1.;
    dger_(nmk1,nmk1,alpha,&a(k+1,k),one,&a(k,k+1),
          lda,&a(k+1,k+1),lda);
  }
}
```