HPC Topics

Introduction

Motivation parallel computing – constraints for serial computers

- Transmission speeds (speed of light 30cm / nanosecond, transmission limit of copper wire 9cm / nanosecond)
- Limits to miniaturization
- Economic limitations
- Energy limits (cooling)

Computer architecture

von Neumann: Memory, Control Unit, Arithmetic logic unit, Input / Output

Logical organization of parallel computer: SISD (normal singlecore computer), SIMD (vector machine), MISD, MIMD (most modern supercomputers)

Terminology

1. A **node** contains one or more sockets.
2. A **socket** holds one processor. Each socket has its own L3 Cache, which is shared among the cores.
3. A **processor** contains one or more (CPU) cores. Each core has individual L1 and L2 caches.
4. The **cores** perform FLOPS.

- **Task**: A logically discrete section of computational work. A task is typically a program or program-like set of instructions that is executed by a processor. A parallel program consists of multiple tasks running on multiple processors.
- **Pipelining**: Breaking a task into steps performed by different processor units, with inputs streaming through, much like an assembly line; a type of parallel computing.
- **Symmetric Multi-Processor (SMP):** Hardware architecture : multiple processors share a single address space and access to all resources; shared memory computing.
- **Shared memory**: Independent processors share a common memory. Cache coherency. UMA, NUMA. UMA disadvantage: Additional CPUs increase traffic geometrically.
- **Distributed memory**: Need a communication network. No cache coherency.
- **Parallel overhead**: Task start-up time , synchronizations , data communications , software overhead imposed by parallel compilers, libraries, tools, operating system, etc. , task termination time

Parallel program design

- **Domain decomposition**: block, cyclic
- **Communication**: Latency, bandwidth, synchronous vs. asynchronous, point-to-point vs. collective
- **Synchronization**: Barrier, Lock
- **Data dependencies**: There is a data dependency, if the order of execution affects the correctness of the program.

- **Load balancing**: Even partition of domain or loops, use a task scheduler for tasks like adaptive refinement, N-body simulations, etc.
- **Granularity**: Coarse: relatively large amount of work is done between communication events. Fine: low amount of work. Fine granularity helps resolving load imbalances, but too fine granularity introduces communication overhead.
- **I/O**: Use parallel file system, write large chunks of data, use local disk space.

Week 2

- **Speedup**: S(p) = T(1) / T(p). S(p) <= p, S(p) < 1 may be accepted due to memory reasons.
- **Efficiency**: S(p) / p <= 1.
- **Amdahl's law**: S(p) <= 1/s, where s is the amount of serial work. If only 0.1% is serial, the code never scales beyond 1000 cores.
- **Strong scaling**: Keep problem size N constant, as number of cores c increases. Compare speedup.
- **Weak scaling**: Increase N with c. Compare speedup or throughput.
- **Throughput**: N(c)/T(c)

Monte Carlo Integration

- **Simpson's rule**: Newton-Cotes formula of third order. Error is order $O(M^{-4})$, where M is the number of sampling points. For d dimensions, there are N = M^d points, which leads to an error of $O(N^{-\frac{4}{d}})$. For big dimensions, this is very inefficient.
- **Monte Carlo integration**: $\langle f \rangle = \frac{1}{M}\sum_{i=1} f(x_i)$. X_i are random numbers.
- **Statistical error** (Root mean square error, uncorrelated samples): $\Delta = \sqrt{\frac{Var\, f}{M}} \propto M^{-1/2}$ In d > 8 dimensions Monte Carlo scales better than Simpson.
- **Statistical error for correlated samples:** $\Delta^2 = \frac{Var\, X}{N}(1 + 2\tau_x), \tau_x = \sum_t \frac{E[X_i X_{i+t}] - E[X]^2}{Var\, X}$ integrated autocorrelation time
- **True estimator for mean**: $\bar{X} = \frac{1}{N}\sum_{i=1}^{N} X_i$
- **Estimator for variance**: $Var\, X \approx \frac{N}{N-1}\left(\frac{1}{N}\sum_{i=1}^{N} X_i^2 - \bar{X}^2\right)$
- **Importance sampling:** Choose points not uniformly but with probability p(x)

Random numbers

- Linear congruential: $x_{n+1} = (ax_n + c)\ mod\ m$
- Lagged Fibonacci: $x_n = x_{n-p} \times x_{n-q} mod\ M$, where × denotes an arbitrary binary function.
- Mersenne Twister
- Well generator
- In C++11, you feed the **engine** with a **seed**, and you use the random numbers from the engine to get various **distributions.**

Non-uniform random numbers

- Given: uniform random number u
- Sought: non-uniform random number x

- In general: $x = f^{-1}(u)$
- Uniform distribution in [a,b[: $x = a + (b - a)u$
- Exponential distribution: $x = -\frac{1}{\lambda}\ln(1 - u)$
- Normally distributed numbers: **Box-Muller method**. Two normally distributed numbers can be obtained from two uniform ones: $n_1 = \sqrt{-2\ln(1 - u_1)}\sin(2\pi u_2)$, $n_2 = \sqrt{-2\ln(1 - u_1)}\cos(2\pi u_2)$
- **Rejection method**: Get non-uniform random number by simple function h that bounds f.
    - Choose an h-distributed random number x.
    - Choose a uniform random number $0 \leq u < 1$
    - Accept x if $u < \frac{f(x)}{h(x)}$, otherwise reject.

Markov chain Monte Carlo (MCMC)

- **Markov chain**: MC is a random process usually characterized as memoryless: the next state depends only on the current state and not on the sequence of events that preceded it.
- **Ergodicity**: Every state is potentially reachable from any state. (Whole space can be sampled)
- **MCMC**: If direct sampling from a probability distribution is difficult, a distribution proportional to the original distribution can be used to obtain random samples. The random samples are generated using a process which depends only from the current sample. The random samples therefore constitute a Markov chain.

Metropolis algorithm

1. Initialization: pick an initial state x at random
2. Randomly pick a state x' according to $g(x \to x')$
3. Accept the state according to $A(x \to x')$ If not accepted, that means that $x = x'$, and so there is no need to update anything. Else, the system transits to x'
4. Go to 2 until T states were generated
5. Save the state x, go to 2.

- $g(x \to x')$ is the proposal distribution, e.g. a Gaussian.
- $A(x \to x')$ is the acceptance distribution.
    - Generally, it is chosen $A(x \to x') = \min\left(1, \frac{P(x')g(x' \to x)}{P(x)g(x \to x')}\right)$
    - $P(x)$ is the probability for the system to be in state x
- The transition probability is therefore: $P(x \to x') = g(x \to x')A(x \to x')$

- **Detailed balance condition**: $P(x)P(x \to x') = P(x')P(x' \to x)$ The detailed balance condition states that each state transition is equally likely for both directions, which is a sufficient condition for stability.
    - The acceptance distribution must be symmetric w.r.t x and x' to fulfill the detailed balance condition.

- T mitigates **the autocorrelation effect**: Since MC tend to favor states in the vicinity (depending on the proposal distribution) autocorrelation is inherent to MCMC. By discarding T state changes before taking a state, the autocorrelation can be minimized.
- **Equilibration**: To definitively "forget" the initial state, the first 100 or 1000 states must be discarded.
- Simple rejection methods often suffer from the curse of dimensionality. MCMC methods don't have this problem to such an extent.
- **Boltzmann weight**: To model a canonical configuration, the acceptance ratio is chosen to be the Boltzmann weight $A = \min\left(1, e^{-\beta(E_{new}-E_{old})}\right)$
- **Parallelization of MC simulations:**
    - **Parallelize updates:** +: Effective speedup for individual simulations. +: Update is parallelizable very well (only issue is RNG) -: Double buffering needed
    - **Multiple independent simulations:** +: No communication at all needed. -: Large serial part at beginning due to equilibration. -: Having multiple independent simulations might not be very useful.
- **Binning analysis**: Take averages of consecutive measurements: Averages become less correlated and naïve error estimates converge to real error. However, you lose computational resolution. Sometimes works, sometimes not.
- **Jackknife analysis**: Evaluate the function on all and all but one segment.

$$\langle U \rangle \approx U_0 - (M-1)\,(\bar{U} - U_0)$$

$$\Delta U \approx \sqrt{\frac{M-1}{M}\sum_{i-1}^{M}(U_i - \bar{U})^2}$$

Week 1

Threads

Thread states: Running, ready, waiting, terminated

std::thread: movable, but non-copyable

```
int main()
{
    int nthreads = std::thread::hardware_concurrency();
    std::vector<double> results(nthreads);
    std::vector<std::thread> threads(nthreads);
    for(int i = 0; i < nthreads; i++){
        thread[i] = std::thread([](int i){std::cout << "This is thread
no." << i << std::endl;});
        //thread[i] = std::thread(some_function,i);
    }
    for(std::thread& t : threads)
        t.join();
}
```

std::Future<T>: Hold return values of a function called asynchronously in a thread. (Otherwise, a variable would have to be declared locally and passed to the function, which is clumsier according to M. Troyer)

```
std::future<double> fi = std::async(std::launch::async, some_function,
param1, param2, …); //launch asynchronous fn call in new thread
double result2 = fn(param1, param2, …);
double totalresult = result2 + fi.get(); //fi.get() will wait for result
t.join();
```

Source: Lecture 2b, "Simpson's rules using asynchronous calls"

std::this_thread: Holds information about the running thread.

Thread safety

Problem: Race conditions due to uncontrolled read/write access to shared variables.

- **Mutex**: At any time, a mutex is either locked or unlocked by one thread. Before, accessing the data, the mutex is locked. If it is already locked, the process is blocked until the mutex is freed.
- **Lock**: A lock (lock_guard, unique_lock) locks the mutex.
- **RAII**: Resource allocation is initialization. Constructor locks mutex, destructor unlocks mutex

```
void some_fn(std::pair<long double, std::mutex>& result){
    std::lock_guard<std::mutex> l (result.second);
    Result.first = sum;
}
int main(){
    std::pair<long double, std::mutex> result;
    …
    threads[i] = std::thread(some_fn, std::ref(result));
    …
    for(std::thread& t : threads)
         t.join();
}
```

- **Deadlocks**: Typically occur if multiple mutexes need to be locked by threads.
    - Solution 1: Introduce lock order on mutexes.
    - Solution 2: `std::lock(Lockable& l1, …, Lockable& lN)` locks multiple locks simultaneously.

Synchronization is expensive; whenever possible the need for locks, barriers or other synchronization should be avoided!

C++11 features

Std::bind: The function template `bind` generates a forwarding call wrapper for `f`. Calling this wrapper is equivalent to invoking `f` with some of its arguments bound to `args`. This is useful especially for functions which have parameters and "running variables"

```
using namespace std::placeholders; // for _1
double expax(double a, double x){ return std::exp(a*x);}
double a = 3.4;
auto f = std::bind(expax, a, _1);
for(int i = 0; i < 100; i++)
    std::cout << f(i*0.01) << std::endl;
```

Lambda functions: Sort of "on-the-fly" functions

```
double expax(double a, double x){ return std::exp(a*x);}
double a = 3.4;
for(int i = 0; i < 100; i++)
    std::cout << [=](double x){return expax(a*x);} << std::endl;
```

Name capture specification

[]	Capture nothing
[&]	Capture any referenced variable by reference
[=]	Capture any referenced variable by making a copy
[=, &foo], [bar], etc.	

Condition_variable: The `condition_variable` class is a synchronization primitive that can be used to block a thread, or multiple threads at the same time, until:

- a notification is received from another thread
- a timeout expires, or
- a spurious wakeup occurs

Any thread that intends to wait on `std::condition_variable` has to acquire a std::unique_lock first. The **wait** operations atomically release the mutex and suspend the execution of the thread. When the condition variable is **notified**, the thread is awakened, and the mutex is reacquired.

Barrier: Is a synchronization point between all threads. The process can continue only when all threads have reached the barrier.

Performance issues

Cache thrashing: A thread invalidates the cache for other threads. If multiple threads access the same cache, it can be catastrophic if a thread invalidates this cache line all the time. => Avoid cache accesses by using thread-local variables.

```
//sumterms is called by every thread
void sumterms(double & sum){
for(int i=0; i<100; i++)
    sum += i; //Not good. Sum is accessed 100 times, producing 100 cache
misses in the worst case.
}

void sumterms(double & sum){
    double tmp;
    for(int i=0; i<100; i++)
          tmp += i;
    sum = tmp;//Only one cache miss.
}
```

NUMA (Non-uniform memory access): Scales better than UMA, but memory latency depends on where the data is allocated in memory.

First touch policy: One thread allocates memory, and then every thread initializes its part of the memory.

UMA

```
    //
    // Allocation and init centralized from "Main" thread
    //
    float * const x1 = new float[size_per_thread * n_threads];
    float * const x2 = new float[size_per_thread * n_threads];
    std::cout<< "init = plain";
    for(std::size_t i=0; i < size_per_thread*n_threads; ++i) {
        x1[i] = 1.0;
        x2[i] = 1.1;
    }
```

NUMA

```
    //
    // Allocation and init local in different threads
    //
    float * const x1 = new float[size_per_thread * n_threads];
    float * const x2 = new float[size_per_thread * n_threads];
    std::cout<< "init = numa";
    for(unsigned int i=0; i < n_threads; ++i)
        threads.push_back( std::thread( [=]() {
                    for(std::size_t j=i*size_per_thread; j <
(i+1)*size_per_thread; ++j) {
                        x1[j] = 1.0;
                        x2[j] = 1.1;
                    } } )
        );
    for(std::thread& t : threads)
        t.join();

    threads.clear();
```

Atomic operations

Problem: The compiler can reorder memory accesses to optimize code as long as the semantics doesn't change. Therefore, any dependencies between memory accesses within threads must be atomic.

```
int x = 0;
std::thread t1([&](){x++;});
std::thread t2([&](){x++;});
t1.join(); t2.join();
std::cout << x << std::endl; //Output either 1 or 2
```

std::atomic<T>: Provides atomic (=> thread-safe) and lock-free updates. Moreover, No memory reordering is allowed.

```
std::atomic<int> x = ATOMIC_VAR_INIT(0);
std::thread t1([&](){x++;});
std::thread t2([&](){x++;});
t1.join(); t2.join();
std::cout << x << std::endl; //Output 2
```

Memory ordering: Memory ordering can be relaxed when using the member functions of std::atomic. Member functions are store, load, exchange, fetch_add, fetch_sub, fetch_and, fetch_or, fetch_xor. Important memory

orderings are memory_order_relaxed (no constraints), _acquire (no reordering before this load), _release (no reordering after this write)

Spinning lock: A spinning lock is good if a thread is expected to wait only a short time. If the thread is kept waited by the OS, it usually waits much longer.

std::atomic_flag : `clear()` sets the flag atomically to false . `test_and_set()` sets the flag atomically to true and returns the previous value. `test_and_set()` is volatile.

```
std::atomic_flag lock = ATOMIC_FLAG_INIT;
void f(){//Threaded function
while(lock.test_and_set()){//If the flag is free (i.e. false), test_and_set returns false and goes to cout. Otherwise test_and_set returns true and the thread spins until another process clears the lock.
    ;
}
std::cout << "Thread-safe output" << std::endl;
lock.clear;
}
```

OpenMP

Synchronization directives

#pragma omp **master**	The block is performed only by the master thread
#pragma omp **single** ([copyprivate (list)])	The block is performed only by on single thread. Need not be the master
#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)	The omp flush directive identifies a point at which the compiler ensures that all threads in a parallel region have the same view of specified objects in memory. An implied flush directive appears in conjunction with the following directives: • omp barrier • Entry to and exit from omp critical. • Exit from omp parallel. • Exit from omp for. • Exit from omp sections. • Exit from omp single.
#pragma omp **parallel** (clauses)	Allowed clauses are: if, private, shared, copyin, firstprivate, reduction, num_threads

#pragma omp **section** (clauses)	Every section gets assigned to a different thread. Allowed clauses are: private, firstprivate, lastprivate, reduction, nowait
#pragma omp **parallel section** (clauses)	Allowed clauses: The union of the allowed clauses of parallel and section
#pragma omp **parallel for** (clauses)	• Loop control variable must be an integer, pointer or C++ random access iterator • Loop condition must be a simple comparison with a constant. • The increment must be constant. Allowed clauses are the allowed clauses of parallel and nowait, ordered, collapse, schedule

OpenMP clauses

If (scalar_expression)	Only parallelize if expression is true. Useful to stop parallelization if work is too little.
Private (list)	Variables are thread-private
Shared (list)	
Copyin (list)	Initialize private variables from master thread
Firstprivate(list)	Combination of private and copyin
Lastprivate (list)	Variable is written back to master thread in last section.
Copyprivate (list)	Copies the listed variables to all other threads. (Broadcast)
Reduction (operator: list)	Allowed reduce operations: + - * / & ^ | && ||. The operators may not be overloaded
Num_threads (integer)	
Nowait	There is no implicit barrier at the end of the parallel block. Useful for multiple independent blocks.
Ordered	The same execution order as in the sequential case can be enforced. Warning: slow!
Collapse (integer)	Nested loops can be collapsed to one to simplify parallelization
Schedule (type [,chunk])	Specify parallelization schedule. There are five types: STATIC, DYNAMIC, GUIDED, RUNTIME, AUTO

Examples

```
long double sum = 0;
#pragma omp parallel for reduction(+: sum)
for(std::size_t t = 0; t < nterms; ++t)
    sum += t;

std::cout << „Sum = „ << t << std::endl;
```

```
#pragma omp parallel
{
    #pragma omp for nowait
```

```
    for(int i=1; i<n; i++)
        b[i] = (a[i] + a[i-1]) * 0.5;

    #pragma omp for nowait
    for(int i=1; i<m; i++)
        y[i] = sqrt(z[i]);
}
```

```
#pragma omp parallel for collapse(2)
for(int i=0; i<n; i++)
    for(int j=0; j<n; j++)
        a[i][j] = b[i][j]*c[i][j];
```

Important OpenMP environment variables

OMP_PROC_BIND	If set to `true`, OpenMP threads should not be moved among threads, if set to `false` they may be moved. Important on NUMA architectures
OMP_WAIT_POLICY	If set to ACTIVE, waiting threads spin actively, if set to PASSIVE they sleep (waste no CPU resources but take longer time to wake up)
OMP_NESTED	Set to TRUE or FALSE to enable / disable nested parallelization. NP occurs if a parallel region calls a function that contains a parallel region itself.

OpenMP Tasks

Tasks are a construct that allow parallelizing unstructured data. New tasks are put on a task queue; idle threads pull tasks from the queue.

#pragma omp task (clauses). Allowed clauses are listed below.

#pragma omp taskwait: Wait for all tasks that are generated by the current task.

If (scalar_expression)	
Private (list)	
Shared (list)	
Firstprivate (list)	
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**:Thread-private variables are not copied.
Final (scalar_expression)	If expression is true, this has to be a *final* task. A *final* task is a task that forces all of its descendant tasks to become included tasks.

Example

```
int fibonacci(int n){
    int i,j;
    if(n<2)
```

```
            return n;
    else{
            #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;
}

int main(){
    int n;
    std::cin >> n;

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

Remarks

In main: #pragma omp parallel. Otherwise program wouldn't run in multiple threads.

In main: #pragma omp single nowait. Single: Only one call to Fibonacci needed. Nowait: Other threads don't need to wait until Fibonacci(n) is finished. Couldn't use multiple threads otherwise.

In Fibonacci: final(n<=5). No new tasks are spawned for n<=5

Intel TBB (Thread Building Blocks)

Threading library by Intel, this is based on functions rather than pragmas.

- parallel_for, parallel_do, parallel_while
- Features thread-local data and thread-safe data structures.
- Provides tasks as an abstraction over threads.
    - 10-100 times faster than creating a thread.
    - Harder to create and manage

N-body problem

In short, an N-body problem consists of N individual particles which interact with each other according to Newton's equations of motion. The properties of a concrete system are determined by the form of potential energy between two particles.

Newton's equations of motion

Newton's equations of motion $\frac{d\vec{x}_i}{dt} = \vec{v}_i$

$$\frac{d\vec{v}_i}{dt} = \frac{\vec{F}_i}{m_i} = -\frac{1}{m_i}\nabla_{x_i} V(\vec{x}_i, \dots, \vec{x}_N)$$

Total potential energy in terms of 2-body interaction. r is the distance between to particles

$$V(\vec{x}_i, \dots, \vec{x}_N) = \frac{1}{2}\sum_{i \neq j} V_{ij}(r_{ij}),$$

Kinetic energy

$$E_{kin}(\vec{v}_1, \dots, \vec{v}_N) = \sum_i \frac{m_i}{2}|\vec{v}_i|^2$$

Total energy. The total energy can be separated into a term dependent only on positions and a term only dependent on velocities.

$$E_{tot}(\vec{x}_i, \dots, \vec{x}_N, \vec{v}_1, \dots, \vec{v}_N) = E_{kin}(\vec{v}_1, \dots, \vec{v}_N) + V(\vec{x}_1, \dots, \vec{x}_N)$$

Concrete potential energies

- Coulomb interaction (q is the charge)

$$V_{ij}(r_{ij}) = \frac{q_i q_j}{r_{ij}}$$

- Gravity (m is the mass)

$$V_{ij}(r_{ij}) = -G\,\frac{m_i m_j}{r_{ij}}$$

- **Lennard-Jones** (long range van-der-Waals attraction and short range hardcore repulsion)

$$V_{ij}(r_{ij}) = 4\epsilon_{ij}\left[\left(\frac{\sigma_{ij}}{r}\right)^{12} - \left(\frac{\sigma_{ij}}{r}\right)^{6}\right]$$

Ergodicity / System observables

Long-time average of the dynamics is the same as the ensemble average, i.e. we can integrate over the ensemble instead of integrating over all times.

$$\lim_{T\to\infty}\frac{1}{T}\int_0^T A(t)dt = \frac{\int A(\vec{x}_1, \dots, \vec{x}_N) w(\vec{x}_1, \dots, \vec{x}_N, \vec{v}_1, \dots, \vec{v}_N) d\vec{x}_1 \cdots d\vec{x}_N d\vec{v}_1 \cdots d\vec{v}_N}{\int w(\vec{x}_1, \dots, \vec{x}_N, \vec{v}_1, \dots, \vec{v}_N) d\vec{x}_1 \cdots d\vec{x}_N d\vec{v}_1 \cdots d\vec{v}_N}$$

The weights for a canonical ensemble are

$$w = \exp\left(-\beta V(\vec{x}_1, \dots, \vec{x}_N)\right)\exp\left(-\beta E_{kin}(\vec{v}_1, \dots, \vec{v}_N)\right)$$

Therefore, the ergodic hypothesis simplifies to

$$\lim_{T\to\infty}\frac{1}{T}\int_0^T A(t)dt = \frac{A(\vec{x}_1, \dots, \vec{x}_N)\exp\int\left(-\beta V(\vec{x}_1, \dots, \vec{x}_N)\right)d\vec{x}_1 \cdots d\vec{x}_N}{\exp\int\left(-\beta V(\vec{x}_1, \dots, \vec{x}_N)\right)d\vec{x}_1 \cdots d\vec{x}_N}$$

Integration of Newton's equations of motion

Verlet algorithm

$$\vec{x}_i(t+\Delta t) = \vec{x}_i(t) + \vec{v}_i(t)\Delta t + \frac{\Delta t^2}{2}\vec{a}_i(t) + O(\Delta t^3)$$

$$\vec{v}_i(t+\Delta t) = \vec{v}_i(t) + \frac{\Delta t}{2}\left(\vec{a}_i(t) + \vec{a}_i(t+\Delta t)\right) + O(\Delta t^2)$$

Initial configuration of positions and velocities.

A total energy is prescribed. In principle, positions and velocities can be set arbitrarily.

Rescale velocities. To match the prescribed energy, the velocities are rescaled. During the run, velocities are rescaled to compensate for roundoff errors.

$$\lambda = \sqrt{\frac{E_{tot} - V}{E_{kin}}}$$
$$\vec{v}_i \leftarrow \lambda \vec{v}_i$$

Boundary conditions

- No boundaries. Useful for bounded systems (crystal), pointless for gas.
- Hard walls. Particles are reflected at walls.
- Periodic boundary conditions. Summation of long-range forces over infinitely many periodic images needed.

Potential energy over infinitely many periodic images

$$V_{ij}(\vec{r}) = \sum_n \sum_i \frac{q_i q_j}{|\overrightarrow{r_n} - \vec{r}|}$$

$\overrightarrow{r_n}$ is the distance between the original cell and periodic image n.

Ewald summation.

- For long-range forces such as Coulomb, the potential energy sum can't be truncated
- Rewrite as a sum of two rapidly converging sums.
- Short range part summed in real space
- Long range part summed in Fourier space

Truncation of potential energy

- Assume a power law decay of potential energy $V(r) \propto r^{-a}$
- $\int dr \propto r$, therefore $\iiint V(r)dr \propto r^{d-a}$
- Truncation is possible, if a > d, d = dimensions

Truncation of Lennard-Jones potential (cutoff-radius $r_c = 2.5\,\sigma$)

$$V_{LJ_{trunc}}(r) = \begin{cases} V_{LJ}(r) - V_{LJ}(r_c), & r \le r_c \\ 0, & r > r_c \end{cases}$$

Domain subdivision (Cell-Lists) Only particles in neighboring cells are taken into account

Computational cost without cell-list: $O(N^2)$
Computation cost with cell lists: $9cN$

How to parallelize?
Have particles have a cell index or have cells have particle indices?

Constant temperature: Relation of temperature to kinetic energy: $\langle E_{kin} \rangle = \frac{G}{2} k_B T$, where G is the degrees of freedom of a particle times the number of particles.
To change the ensemble from constant energy to constant temperature, a thermostat, e.g. Nosé-Hoover thermostat can be used. To that end, a friction term is added to the acceleration.

Dense linear algebra

BLAS (Basic Linear Algebra Subprograms)

- Level 1: Scalar and vector operations. Scale as $O(1)$ or $O(N)$
- Level 2: Matrix-vector operations. Scale as $O(N^2)$
- Level 3: Matrix-matrix operations. Scale as $O(N^3)$

Calling BLAS functions. BLAS is Fortran library. To call it from C++, some things have to be considered.

Fortran DDOT function

```
DOUBLE PRECISION FUNCTION
DDOT(N,DX,INCX,DY,INCY)
INTEGER INCX,INCY,N
DOUBLE PRECISION DX(*),DY(*)
```

Translates to the following C++ prototype

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

- Fortran is case-insensitive
- Extern "C" because parameters don't belong to function signature.
- Pass all arguments by reference, and C-style arrays as pointers.

Array storage

- Fortran indices start at 1, C++ indices at 0.
- Fortran stores arrays in column-major order, while C++ uses row-major.

Leading dimension (or increments)

- Tells the program at which position in memory the next column begins.
- It might be that the leading dimension is not the same as the number of rows, if we are computing only on a submatrix or if the matrix Is in aligned storage, i.e. one column = one cacheline.

Packed storage formats

- For banded or triangular matrices, only their diagonals must be stored.

Parallelizing BLAS calls

A simple matrix-vector multiply consists of two nested loops.

```
for(int i=0; i<M; i++)
    for(int j=0; j<N; j++)
          y[i] = A(i,j)*x[j];
```

- In general it is better to **parallelize the outer loop**, because parallelizing the inner loop causes too many threads to be spawned.
- The best thing however is to **use BLAS calls** as much as possible.

LU factorization

1. Search absolute largest pivot (idamax). Store the pivot.
2. Swap the row with the largest pivot with the first row (dswap)
3. Get coefficient by inverting the pivot.
4. Scale the leftmost nonzero column by the coefficient and store it in the left-triangular matrix (dscal).
5. Divide all rows by the coefficient and subtract the first row from the others. (daxpy). Store the coefficients.

- LU factorization can be implemented using BLAS 2-calls, where daxpy is called multiple times within one elimination step.
- For peak performance this could be replaced by a BLAS 3-call, dger.

- Whenever possible high-level BLAS calls should be used, as $O(N^3)$ computations are made on $O(N^2)$ data.

Sparse linear algebra

Most operations can be solved in $O(N)$ using iterative methods.

1-d diffusion equation as sparse matrix problem

$$f(x_i, t + \Delta t) = f(x_i, t) + \frac{c\Delta t}{\Delta x^2}[f(x_{i+1}, t) + f(x_{i-1}, t) - 2f(x_i, t)]$$

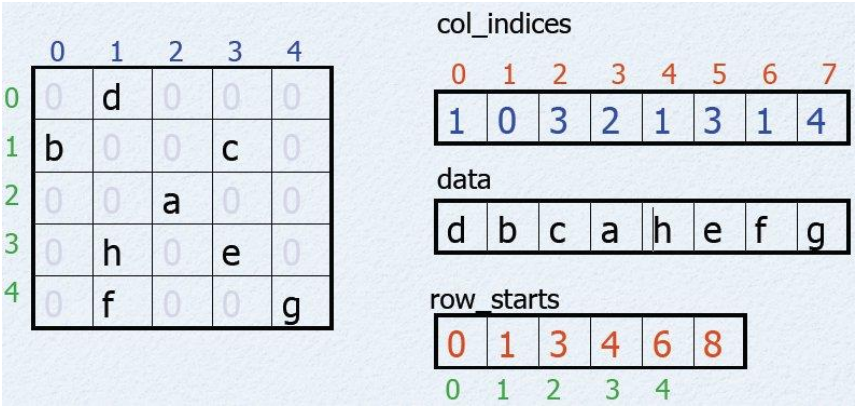
- This equation can be rewritten as sparse matrix-vector multiply, i.e. $f(x_i, t + \Delta t) = Mf(x_i, t)$.
- However, it is in general not a very wise idea to solve it this way, since the **matrix is highly repetitive** and therefore much **memory is wasted**. Additionally, unnecessary and **expensive memory reads** from this matrix are done.

The power method

- The power method computes the eigenvector with the largest eigenvalue of a matrix.
- Algorithm: Multiply an arbitrary vector many times with the matrix and normalize it. If it doesn't change anymore, it's finished.

Compressed storage formats

- Dictionary of keys: Associative array mapping an index pair (I,j) to a value.
  - Fast for building the matrix iteratively, slow access later
- List of lists: Stores one list per row, containing column index and value of the nonzero entries.
- Coordinate list: List of triples (column, row, value) sorted by column and row.
- Compressed sparse row (CSR): Vectors containing column index and value of nonzero-entries as well as vector which indicates for every row which index in the data vector is the first in the row.
  - As a consequence, the row-starts vector has ascending values.
  - Efficient memory usage, inefficient for building the matrix iteratively.



- Similar to CSR, CSC (Compressed sparse column) is also possible.

Parallelizing sparse matrix operations

- In CSR representation, a matrix-vector multiplication requires looping over rows.

```
#pragma omp parallel for
for(int row = 0; row < dimension(); row++)
    for(int i=row_starts[row]; i!= row_starts[row+1]; i++)
         y[row] += data[i] * x[col_indices[i]];
         /*Update for a transposed matrix
          * #pragma omp atomic
          * y[col_indices[i]] += data[i] * x[col_indices[i]];
          */
```

- For a transposed matrix, an atomic update is required.
- On the contrary, in CSC representation, the transposed matrix-vector multiply is safe and fast, while the regular matrix-vector multiply needs an atomic update.

Vectorization with SIMD instructions

- SIMD units contain vector registers.
    - For SSE, these are 128-bit registers, which can store 16 bytes or 2 doubles, 4 floats, 2 64-bit integers etc.
    - For AVX, these are 256-bit registers, which can store twice as much as the SSE can.
    - **SSE and AVX operations should never be mixed**. If AVX registers are available, then the SSE registers are just the lower bits of the same AVX registers. If an AVX operation fills the upper bits, a subsequent SSE operation would store these upper bits to memory and a later AVX operation would read these bits again from memory. Solution: _mm256_zeroupper clears the upper bits. Call it before switching from AVX to SSE.
- SIMD vector operations act on all values in the register simultaneously. That is, for doubles, you can get a speedup of 2, for floats a speedup of 4 by using SIMD vector operations compared to regular operations.
- SSE/AVX documentation: http://software.intel.com/en-us/avx Intel Intrinsics Guide

Caches

- Caches are fast intermediate buffers which have limited space. Access to caches is significantly faster than access to memory (DRAM).

Sandy bridge Intel CPU	Size	Access time in cycles
L1 cache (per core)	2 x 32 kB	4-5
L2 cache (per core)	256 kB	12-19
L3 cache (shared among cores)	3-20 MB	30-50
Memory (DRAM)	Many GB	≈ 300

- DRAM-addresses are mapped to cache blocks. This mapping can be direct (an address can be cached only by one specific block), n-way associative or fully associative (an address can be cached by any block).
- Data that is requested from memory is stored in the caches until it needs to be evicted because other data has been loaded.
- Data written to memory is written to cache first, and only when it needs to be evicted it is effectively written to memory.

- **Prefetching**: If some data is used in the near future, the CPU can be instructed to already load the data into cache.
- **Reasons and solutions for cache misses**:
    - Capacity (data has been relocated since cache is completely full). Solution: Bigger cache.
    - Conflict (in direct or n-way associative caches, it is possible that the associated cache block is already full, while other cache blocks would still have space). Solution: Fully associative cache
    - Compulsory (When data is read for the first time it is most likely not yet in cache). Solution: Prefetching
- If the CPU requests a word (4 bytes) from memory, a full cache line (64 bytes) is read from memory and the requested word is sent to the CPU. The cache lines have fixed positions in the cache, i.e. the requested word might not be at the start of a cache line.
- **Data alignment**: If an array is stored across a cache line boundary, two instead of just one memory read is necessary. Therefore, it is possible to align data structures to start at fixed positions in the cache, making sure that the data structure fits into one cache line. C++11 provides the alignas(int numBytes) specifier data types and aligned allocators for C++ containers. **SSE registers need 16-byte alignment, AVX registers need 32-byte alignment.**

Loop dependencies

```
for(in j=0; j<N-1; j++)
    a[j] = a[j-5] + 20;
```

There is a dependency, since a[j] relies on a[j-5]. However, the loop can be parallelized when just four consecutive elements are processed simultaneously.

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

//Call to saxpy
saxpy(10,2.3, x, x+1)
```

In this example, aliasing introduces a dependency, which is not apparent at the first glimpse. To profit from automatic vectorization, a function parameter can be declared alias-free by the C-modifier `restrict` or in C++ with g++ the `__restrict__` keyword or the keyword `restrict` with iCC using the compiler switch -restrict

Example using SIMD instructions

```
#include <x86intrin.h>
void sscal(int n, float a, float* x){
    //load the scale factor four times into a SSE-register
    __m128 x0 = _mm_set1_ps(a);

    int ndiv4 = n/4;
    //loop over chunks of 4 values
    for(int j=0; j<ndiv4; ++j){
         __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
    int j = ndiv4*4;
    switch(n-j){//n-j == 3 always
    case 3: x[i+2] *= a;//no break statement, but intended fallthrough
    case 2: x[i+1] *= a;
    case 1: x[i]   *= a;
    }
}
```

MPI

- Wrapper compiler: mpicc, mpic++
- Executor: mpiexec –np number_of_processes
- Processes are numbered within **communicators**. One process may belong to many communicators.
- Numbered processes within a communicator are referred to as **ranks**.

Point-to-point message passing

A **message** generally consists of:

Body			Envelope			
Buffer	Count	MPI Datatype	Source / Destination	Tag	Communication	Status

Message passing example

```
int main(int argc, char**argv){
    MPI_Init(&argc, &argv);
    int num;

    MPI_Comm_rank(MPI_COMM_WORLD,&num);

    if(num==0) //master, receiver
        MPI_Status status;
        char txt[100];
        MPI_Recv(txt, 100, MPI_CHAR, 1, 42, MPI_COMM_WORLD, &status);
        std::cout << txt << "\n";
    }
    else{// worker, sender
        std::string text="Hello World!";
        MPI_Send(cont_cast<char*>(text.c_str()), text.size()+1,
MPI_CHAR, 0, 42, MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}
```

Sending and receiving

- There are different send and receive methods. All have the same parameters but return at different points:

    - MPI_Ssend. Synchronous send: Returns when destination has started to receive the message.
    - MPI_Bsend. Buffered send: Returns after making a copy of the buffer. Destination **might not have started** receiving.
    - MPI_Send. Can be synchronous or buffered, depending on message size.
    - MPI_Rsend. Ready send: Optimized send if the user can guarantee that the destination is ready to receive (is already executing MPI_Recv).
    - MPI_Recv. Blocking receive: Returns only when the message has been received.
- **Deadlock** threat through blocking sends and receives:

```
if(num==0){
    MPI_Ssend(…)
    MPI_Recv(…)
}
else{
    MPI_Ssend(…)
    MPI_Recv(…)
}
```

Both send first and won't continue until the other starts receiving, which will never happen. Solutions:

- MPI_Sendrecv
- Asynchronous sending and receiving

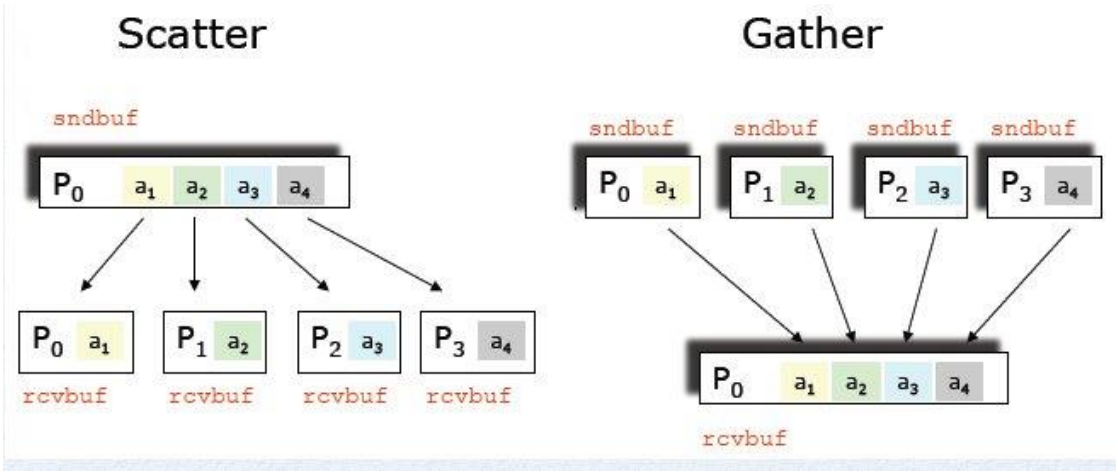
Asynchronous sending and receiving

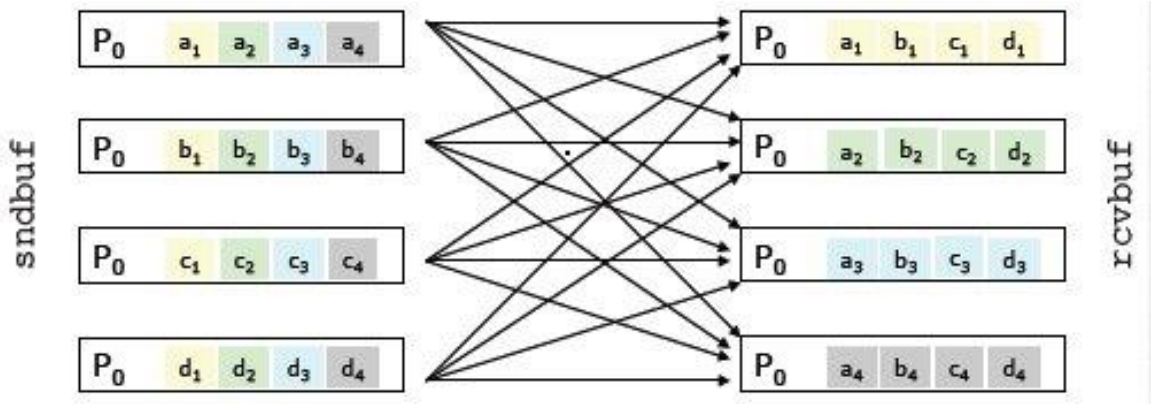
- MPI_Issend, MPI_Ibsend, MPI_Isend, MPI_Irecv. Analogous to synchronous counterparts. They return immediately, while communication is still going on.
- When using asynchronous sending and receiving, it may be necessary to wait or test for completion.
- MPI_Wait, MPI_Waitall, MPI_Waitany, MPI_Waitsome. These functions only return when the corresponding communication requests have finished.
- MPI_Test, MPI_Testall, MPI_Testany, MPI_Testsome. These functions set a number which is passed as a parameter.
- Asynchronous sending and receiving allows overlaying communication and computation. E.g. compute interior while boundary cells are yet to be communicated.

Collective communication

- **MPI_Reduce**: Performs a reduction on data in *sendbuf* and places the results in *recvbuf* on the root rank. The keyword MPI_IN_PLACE can be used as *sendbuf* for the root rank.
- **MPI_Allreduce**: Performs the same reduction, but places the result in *recvbuf* on all ranks.
- **MPI_Broadcast**: Broadcasts data in the *buffer* from a specified rank to all others.
- **MPI_Gather**: Gathers data from the *sendbuf* buffers into a *recvbuf* buffer on the specified rank. *recvbuf*, *recvcnt* and *recvtype* are significant only on the receiving rank. The size of the sendbuffers needs to be the same on all ranks.
- **MPI_Gatherv**: The size of the sendbuffers may vary among ranks.
- **MPI_Allgather**: Same as MPI_Gather, but all ranks receive. Semantically the same as a MPI_Gather followed by a MPI_Bcast
- **MPI_Allgatherv**: Guess what?!

- **MPI_Scatter**: Scatters data from the *sendbuf* buffer on the *root* rank into *recvbuf* buffers on the other ranks. Each rank receives the same portion from *sendbuf*.
- **MPI_Scatterv**: Each rank may receive different buffer sizes.
- **MPI_Reduce_scatter**: MPI_Reduce followed by a MPI_Scatter, optimized version.
- **MPI_Alltoall**: All ranks scatter and gather.
- **MPI_Barrier**: Waits until all ranks have called it.





Sending non-contiguous data

- Packing into buffer
  - **MPI_Pack, MPI_Unpack, MPI_Pack_Size**.
  - Allocate a large enough buffer with the help of MPI_Pack_Size. For every item, call MPI_Pack which copies the value into the buffer, then broadcast using datatype **MPI_PACKED**, receivers call MPI_Unpack subsequently to extract the individual items.
- Sending items bitwise in a struct
  - Put all items into a struct
  - Send it using datatype **MPI_BYTE**. Dangerous since it assumes a homogeneous machine with identical integer and floating point formats.
- Building a custom MPI datatype. The key method in this example is **MPI_Type_create_struct**.

```
// define a struct for the items
struct parms {
double a; // lower bound of integration
```

```
double b; // upper bound of integration
int nsteps; // number of subintervals for integration
};
parms p;

//Getting addresses of struct items
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

//Describing the struct through sizes, offsets and types
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); //use it as MPI_Datatype
```

- Other MPI type creators are: MPI_Type_contiguous, MPI_Type_vector, MPI_Type_create_subarray, MPI_Type_indexed.

Groups and Communicators

- **Groups** are a collection of ranks, for which there are some useful functions, such as union, intersection, difference or selectively choosing ranks. Groups can be used as input parameter to create a communicator.
- **Communicators** provide access to ranks and define the scope of a communication function.
    - **MPI_Comm_split**: Fills *newcomm* with ranks from *comm* which have the same *color*. Useful for creating column or row communicators. All ranks must call this function.
    - **MPI_Comm_create**: Creates a new communicator based on a group. All ranks must call this function.

Topologies

- **MPI_Cart_create**: Create comm_cart, a cartesian communicator, based on comm_old and an array specifying the dimensions of the Cartesian communicator.
- **MPI_Dims_create**: Yields an automatic splitting in approximately equal dimensions.
- **MPI_Cart_shift**: Neighboring (x- and y-direction) ranks can be obtained.
- **MPI_Cart_rank**: Gets rank of given coordinate.

Distributed linear algebra

In general, there are five ways to distribute matrices: Block-row, block-column, cyclic-row, cyclic-column or block-cyclic. Cyclic-row and cyclic-block make not so much sense in terms of memory locality.

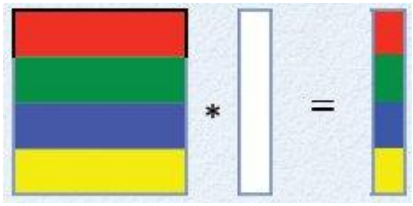
In distributed linear algebra, a block of a matrix is assumed to reside on a given node and the corresponding block of the result has to reside on the same node.

Important design principles:

- Maximize reuse of data in cache ➔ Choose block sizes accordingly (Typically 32 x 32).
- Maximize overlapping of computation and communication. ➔ Choose block-cyclic data layout.

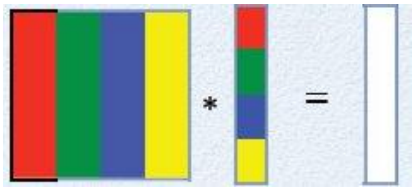
Parallel gemv (A*x = y)

Block-row distribution



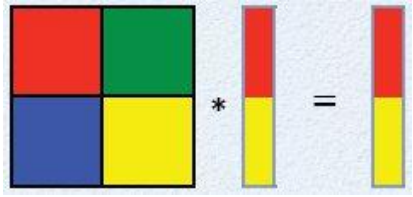
1. Allgather of x. Each rank needs the whole vector.
2. Local dgemv
3. No further communication needed

Block-column distribution



1. No communication prior to computation
2. Local dgemv
3. **Reduction** along rows to final result vector. Final result vector needs to be **scattered**.

Block-cyclic distribution



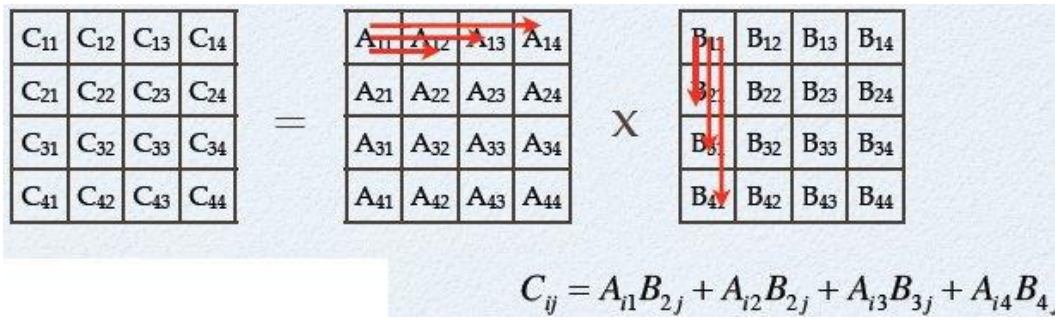
Vectors stored on diagonal blocks.

1. **Broadcast** vector parts along their columns
2. Local dgemv
3. **Reduce** along rows.

Parallel gemm (A*B = C)

Block-row and block-column are out of question, because the full matrix B is needed on every rank of A, although not the whole matrix B is needed on every rank ➔ Inefficient.

Ideal: Block-cyclic layout and all ranks send only one block at a time while computing on two other blocks.



A_{ij} is needed on all rows i, B_{ij} is needed on all columns j. Individual blocks need not be scattered to all ranks before computation, rather every rank can start with receiving its “neighbor” and compute one local dgemm and then receive its “neighbor of neighbor” and so on.

- Blocks are cyclically passed from rank to rank within one row and one column respectively.

- Communication and computation are always overlapped.

Hybrid MPI

Hybrid MPI = Combining MPI processes and multithreading. Many variants:

- One MPI process per node
- One MPI process per socket (avoids NUMA issues)
- Multiple MPI processes per socket, each with threads

Boost.MPI

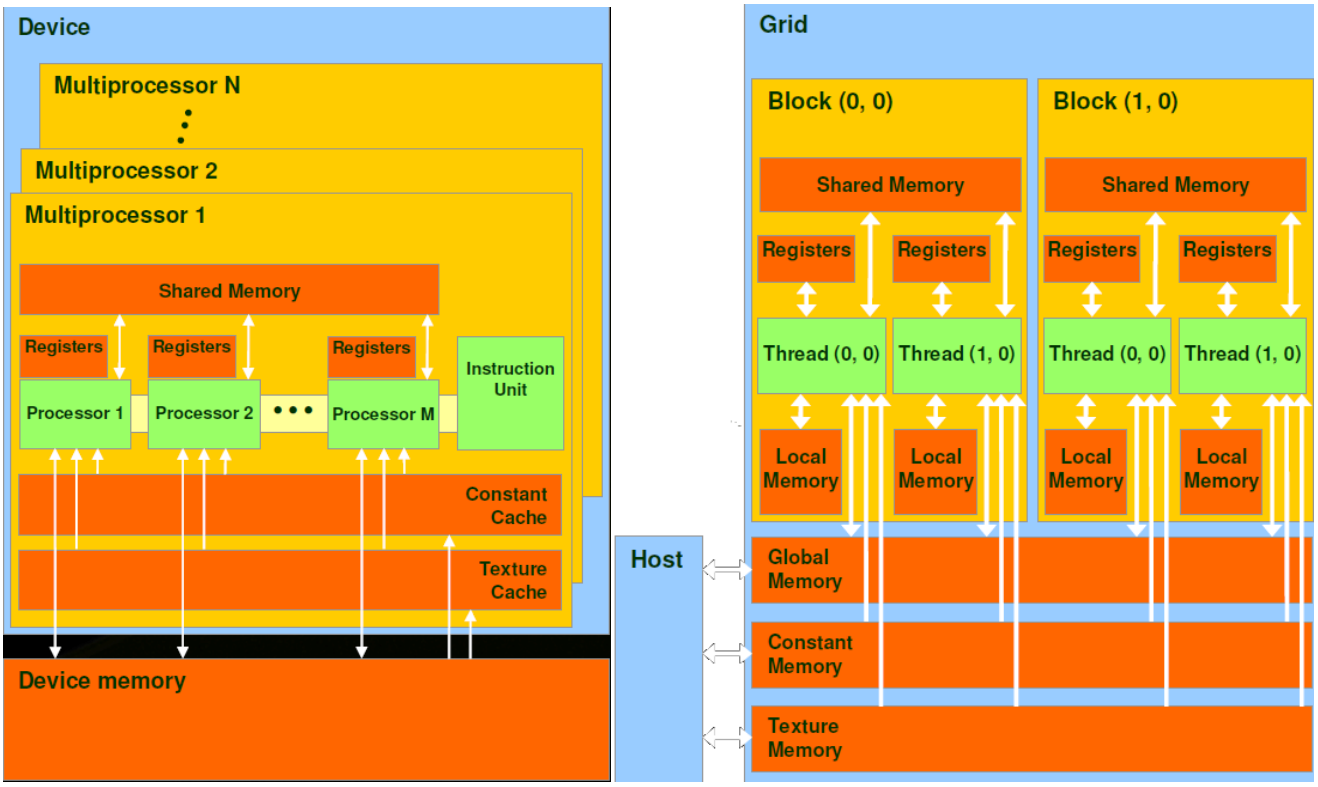
C++ API that provides more functionality than C API for MPI.

- Simplified calls (return values instead of passing output parameters, classes etc.)
- Allows messages to contain custom datatypes, pointers, linked lists, trees, etc. thanks to Boost.Serialization
- Arbitrary function objects possible for reduction.
- Packaging much easier thanks to serialization function.

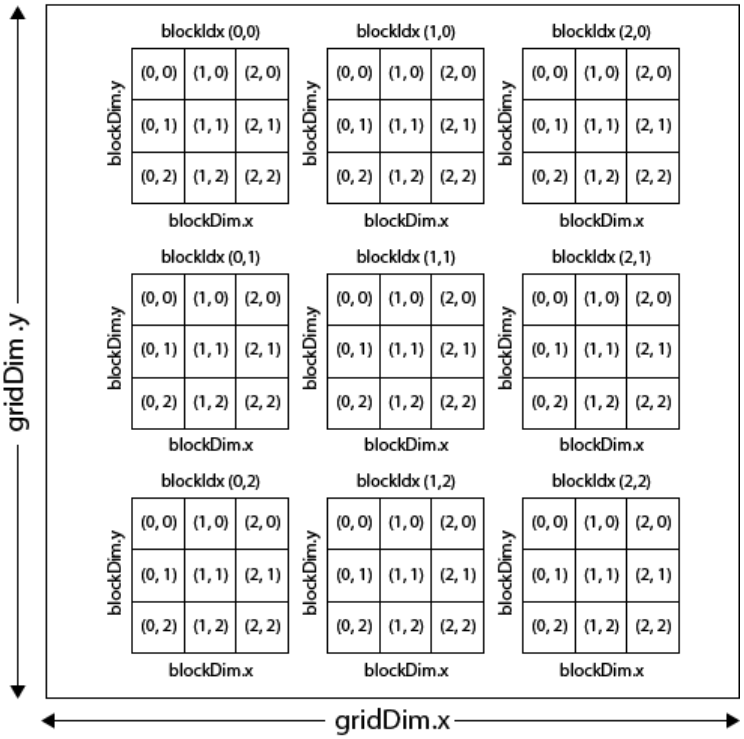
CUDA / GPUs

Hardware component	Memory	Logical unit
GPU (Device)	Global memory Constant memory Texture memory	Grid
Streaming multiprocessor (SM)	Shared memory	Blocks
Core	Local memory Registers	Warp / Threads

- Grid: Collection of blocks.
- Block: Collection of threads.
- Threads are grouped in a functional SIMD unit of 32, called a warp.
- Each block is executed by only one SM.
- A SM can execute several blocks concurrently.



CUDA Grid




```
x = threadIdx.x + blockIdx.x * blockDim.x
y = threadIdx.y + blockIdx.y * blockDim.y
offset = x + y * blockDim.x * gridDim.x
```

As of today, a **warp** contains 32 threads. These threads get executed in **lock-step**. Each thread in a warp executes the same instruction on different data, i.e. threads in a warp are **synchronized**. A negative consequence of this fact is **warp divergence**. If the program contains an if-else block in the following fashion:

```
if(tid%2==0)//tid: thread ID
    doA();
else
    doB();
```

doA and doB won't be executed in parallel, but serially, since threads in a warp are synchronized at every line of a program.

Typical workflow

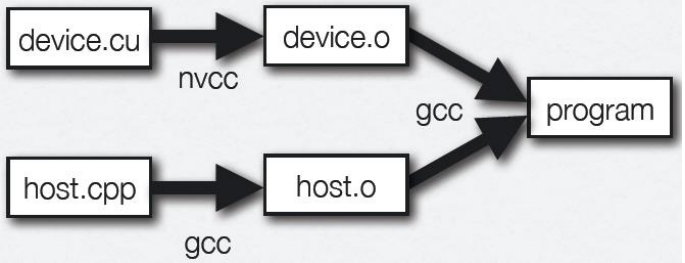
```
#define N 1024

int main(){
    //0. allocate memory on CPU
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;

    //1. allocate memory on GPU
    cudaMalloc( (void**) &dev_a, N*sizeof(int) );
    cudaMalloc( (void**) &dev_b, N*sizeof(int) );
    cudaMalloc( (void**) &dev_c, N*sizeof(int) );

    //fill arrays a and b on CPU
    for(int i=0; i<N; i++){
        a[i] = -i;
        b[i] = i*i;
    }

    //2. copy arrays a and b from host to GPU
    cudaMemcpy(dev_a, a, N*sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, N*sizeof(int), cudaMemcpyHostToDevice);

    //3. launch kernel (on 1024/48 = 16 blocks and 48 threads per block)
    add<<<N/48,48>>>(dev_a, dev_b, dev_c);

    //4. copy array c from GPU back to host
    cudaMemcpy(c, dev_c, N*sizeof(int), cudaMemcpyDeviceToHost);

    //display results
    for(int j=0; j<N; j++)
        std::cout << c[j] << std::endl;

    //5. free memory on GPU
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);
}

__global__ void add(int *a, int *b, int *c){
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
```

```
    if(tid < N) //each thread performs only one addition
        c[tid] = a[tid] + b[tid];
}
```

Restrictions on the usage of device pointers

- Device pointers cannot be dereferenced by the host.
- Operations on device pointers on the host as well as transfers are ok.

Compilation of CUDA-Code



Different types of GPU memory

Memory	Declaration	Location	Cached	Speed	Access	Scope
Register	`int var`	On-Chip	N/A	Fast	Read/Write	One thread
Local	`int var[N]`	Off-Chip	No	Slow	Read/Write	One thread
Shared	`__shared__ int var`	On-Chip	N/A	Fast	Read/Write	All threads in a block
Global	`__device__ int var` (or via cudaMalloc)	Off-Chip	No	Slow	Read/Write	Grid + Host
Constant	`__constant__ int var`	Off-Chip	Yes	Fast	Read	Grid + Host
Texture	`texture<int> var`	Off-Chip	Yes	Fast	Read	Grid + Host

Memory coalescence

Global memory is accessed in chunks of 32 / 64 / 128 bytes. Threads in the same warp perform a coalesced access, that is, **consecutive threads access consecutive addresses.** Multiple threads cannot access global memory in parallel. Instead, a half-warp makes a “group” access to global memory. (The mechanism resembles the concept of a cache line.)

Example: Assume the threads 0,1,2,3 access an array of values, stored in addresses 0x0, 0x4, 0x8, 0xc. A good coalesced access pattern would be thread 0 accesses 0x0, thread 1 0x4, thread 2 0x8 and thread 3 0xc. A bad access pattern would be thread 0 accesses 0x4, thread 1 0xc, thread 2 0x0, thread 3 0x4.

Bank conflict

Within **shared memory**, **parallel memory accesses from multiple threads** are possible. Moreover, the addresses of the shared memory are consecutively divided in banks. For example, addresses 0 to 4 are in bank 1, 5 to 8 in bank 2, … , 33 to 36 again in bank 1 an d so on. That means that two threads could access the same bank for different addresses, e.g. thread 1 could opt for address 0, while thread 0 requests address 33 which come both through bank 1. Therefore, the data access can’t be performed in parallel anymore, which is called a bank conflict.

Occupancy

Active warps / Total warps. High number ensures high throughput.

Streams

```
int main(){
    //1. Check if overlap is supported, otherwise streaming pointless
    cudaDeviceProp prop;
    int whichDevice;
    cudaGetDevice(&whichDevice);
    cudaGetDeviceProperties(&prop, whichDevice);
    if(!prop.deviceOverlap){
         std::cout << "Device does not support overlap of computation and
copying, so no speedup from streams" << std::endl;
    }

    //2. Initialize streams
    cudaStream_t stream0, stream1;
    cudaStreamCreate(&stream0);
    cudaStreamCreate(&stream1);

    int *host_a, *host_b;
    int *dev_a0, *dev_b0; //associated with stream0
    int *dev_a1, *dev_b1; //associated with stream1

    //3. Allocate memory on GPU
    cudaMalloc( (void**)&dev_a0, N*sizeof(int));
    cudaMalloc( (void**)&dev_a1, N*sizeof(int));
    … //same for b0, b1

    //4. Allocate page-locked "pinned" memory on host
    cudaHostAlloc((void**)&host_a, N*sizeof(int), cudaHostAllocDefault);
    … // same for host_b

    //for (int j=0; j<N; j++){
         host_a[j] = rand();
         host_b[j] = rand();
    }

    //5.1 Copy locked memory asynchronously to device
    //Switch at each command between streams (breadth-first style),
otherwise copying and computation cannot be overlapped
    cudaMemcpyAsync(dev_a0, host_a, N*sizeof(int), cudaMemcpyHostToDevice,
stream0);
    cudaMemcpyAsync(dev_a1, host_a, N*sizeof(int), cudaMemcpyHostToDevice,
stream1);
    cudaMemcpyAsync(dev_b0, host_b, N*sizeof(int), cudaMemcpyHostToDevice,
stream0);
    cudaMemcpyAsync(dev_b1, host_b, N*sizeof(int), cudaMemcpyHostToDevice,
stream1);

    //5.2 Call kernels

    kernel<<<N/256, 256, 0, stream0>>>(dev_a0, dev_b0, dev_c0);
```

```
    kernel<<<N/256, 256, 0, stream1>>>(dev_a1, dev_b1, dev_c1);

    //5.3 Copy back

    cudaMemcpyAsync(host_c, dev_c0, N*sizeof(int), cudaMemcpyDeviceToHost,
stream0);
    cudaMemcpyAsync(host_c, dev_c1, N*sizeof(int), cudaMemcpyDeviceToHost,
stream1);

    //6. Synchronize streams
    cudaStreamSynchronize(stream0);
    cudaStreamSynchronize(stream1);

    //7. Destroy streams
    cudaStreamDestroy(stream0);
    cudaStreamDestroy(stream1);

    //8. Free memory
    cudaFreeHost(host_a);
    cudaFreeHost(host_b);
    cudaFree(dev_a0);
    cudaFree(dev_a1);
    cudaFree(dev_b0);
    cudaFree(dev_b1);
}
```

- Asynchronous functions return, even if the function has not terminated or even started yet. They basically put the task into a queue and return.
- Streams allow a task-based parallelism by overlapping host-device transfers with computation.
- To handle asynchronous memory access on the host, page-locked “pinned” memory is mandatory. Pinned memory cannot be relocated out of physical memory, therefore a direct memory access (DMA) is possible. However, pinned memory vastly reduces the available memory for other programs.
- Cuda schedules tasks in a copy engine and a kernel engine. The tasks are executed serially. Therefore, it is important to queue the tasks in a breadth-first fashion, i.e. first all inputs are transferred, then all kernels are executed and then the results are transferred back.
  In a depth-first way, the input of the second stream would have to wait for the result of the first stream to be finished, which is only the case when the kernel of stream1 has finished. So, computation and data transfer would not be overlapped.

CUDA Event API

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start,0);

//do some work on GPU

cudaEventRecord(stop,0);
cudaEventSychronize(stop);//ensure that all asynchronous tasks have
finished
```

```
float elapsedTime;
cudaEventElapsedTime( &elapsedTime, start, stop);
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Further useful comments about CUDA

- cudaDeviceProp is a struct that contains various information about a GPU.
- cudaChooseDevice(&int deviceID, &cudaDeviceProp prop) returns the deviceID which fulfills the properties sought by prop.
- __device__ modifier: Function or variable callable only from GPU.
- __global__ modifier: Function or variable callable from GPU and CPU.
- __shared__ memory is attractive to serve as a "cache" or buffer for a block.
- In reductive functions, the occupancy declines drastically to the end. In the end, only two of potentially many threads can be used. Therefore it makes sense to return a "half-result" to the CPU which should compute the remaining reduction.
- Use __constant__ memory for values which do not change. They can be set from the host via `cudaMemcpytoSymbol(const char* symbol, const void* src, size_t count, size_t offset=0, enum cudaMemcpyKind kind=cudaMemcpyHostToDevice)`
- A single read from constant memory can be broadcast to other threads in the same half-warp, saving up to 15 reads. Moreover, constant memory is cached.
- If the threads need different data from the constant memory, the requests are serialized. This is worse than with global memory, since global memory can handle parallel data accesses.

Roofline model

- Abstraction model for performance analysis.
- Hardware abstraction
  - Peak performance [GFLOP/s]
  - Memory bandwidth [GB/s]
- Software abstraction
  - Operational intensity [GFLOP/GB] = #floating point operations associated to an operation / #Bytes read / written on this operation.
- Plot GFLOP/s vs. GFLOP/GB
- **Maximal achievable performance**: min(Memory bandwidth * Operational intensity, Peak performance)
- Multithreading, Vectorization, etc. increase peak performance
- #available NUMA nodes increases memory bandwidth
- Caches can reduce #Bytes read, therefore increase operational intensity.

Principal Component Analysis (PCA)

- PCA is a linear projection that minimizes the mean squared distance between the data points and the projection points.

- PCA is the orthogonal projection of data onto a lower dimensional linear space, such that the variance of the projected data is maximized (Large variance in projected data => a tendency can be seen, you can draw a line. Small variance in projected data => The points are around the same spot, there is no tendency)
- PCA: Find a basis that **minimizes noise** and **maximizes the signal**.

Task

Given a $m \times n$ datamatrix X, where n is the number of observations and m is the number of dimensions, we seek to find a principal component matrix P, such that $PX = Y$ where Y is a stretch-rotation of X. P is chosen such that the covariance matrix for Y is diagonal.

Derivation

- Covariance: $Cov(S,T) = \sum_{i=1}^{n} \frac{(S_i - \bar{S})(T_i - \bar{T})}{n-1}$, where S and T are vectors of length n.
  - A large covariance means both dimensions increase and decrease together.
  - Covariance zero means both dimensions are independent from each other.
  - A negative covariance means while one dimension increases, the other one decreases.
  - Very high or very low covariance between two datasets indicates redundancy between the datasets.
- Covariance matrix: $\begin{bmatrix} Cov(S,S) & Cov(S,T) \\ Cov(T,S) & Cov(T,T) \end{bmatrix}$
  - The covariance matrix is symmetric and can be extended arbitrarily to more dimensions.
  - The covariance matrix of the input data X is calculated as $Cov(X) = \frac{1}{n-1} XX^T$, where the mean of X is assumed to be zero.
  - m-dimensional data results in a $m \times m$ covariance matrix.
- Properties of the covariance matrix for Y, $Cov(Y)$
  - No redundancy between individual datasets (columns of Y)
    ➔ Off diagonals of $Cov(Y)$ should be zero.
  - Maximal signal, i.e. maximal variance wished
    ➔ Diagonal of $Cov(Y)$ should be nonzero.
  - $Cov(Y)$ must be diagonalized
- Diagonalization of $Cov(Y)$

$$
\begin{aligned}
Cov(Y) &= \frac{1}{n-1} YY^T \\
&= \frac{1}{n-1} (PX)(PX)^T \\
&= \frac{1}{n-1} P(XX^T)P^T
\end{aligned}
$$

- XX^T is a symmetric matrix. Symmetric matrices can be diagonalized and their eigenvectors are orthonormal.
- **Choice of $\boldsymbol{P}$**: P is chosen such that each column p_i is an eigenvector of XX^T, i.e. $\boldsymbol{XX^T = P^{-1}DP}$, where D is a diagonal matrix. Therefore:

$$Cov(Y) = \frac{1}{n-1} P(P^{-1}DP)P^T$$
$$= \frac{1}{n-1} D$$

since $P^T = P^{-1}$ for orthonormal matrices and $PP^{-1} = I$.

- To sum up, the choice $XX^T = P^{-1}DP$ leads to a covariance matrix of the final data Y that is non-zero only on the diagonal. Such a covariance matrix guarantees the final data Y to have no redundancy between individual datasets and to have maximal signal strength.

Algorithm

1. $\tilde{X} = X - \bar{X}$ //Subtract the mean from the datamatrix.
2. Calculate the eigenvectors $\vec{v}_i$and eigenvalues λ_i of the covariance matrix $\frac{1}{n-1}\tilde{X}\tilde{X}^T$
3. Sort the eigenvectors after their eigenvalue in descending order. Fill P with the sorted eigenvectors. These are the principal components of X. The eigenvector with the largest eigenvalue is the most important principal component of X
4. Principal components with low eigenvalues can be clipped. The remaining vectors are called the feature matrix $\tilde{P}$. The significance of this clipping can be measured by an eigenvalue-measure . $S = \frac{\sum_{i=0}^{r}\lambda_i}{\sum_{i=0}^{m}\lambda_i}$, where r eigenvectors are considered for the feature matrix. The bigger S, the less information has been lost.
5. Final data: $Y = \tilde{P}\tilde{X}$
6. Best possible reconstruction of original data: $X \approx \left(\tilde{P}^{-1}Y\right) + \bar{X}$.
   The reconstruction is perfect if $\tilde{P} = P$, i.e. no principal components were discarded.

Fast multipole method and vorticity-velocity relation

Green's function

A Green's function $G(x|x')$ is a solution of an ODE where the RHS is set to $\delta(x - x')$

Green's functions to the Poisson equation: $\nabla^2\Psi = \rho(\vec{x}) \coloneqq \delta(x - x')$	
1D	$G(x\|x') = -\frac{1}{2}\|x - x'\|$
2D	$G(x\|x') = \frac{1}{2\pi}\log\left(\frac{a}{x - x'}\right)$
3D	$G(x\|0) = \frac{1}{4\pi x}$

For general RHS $\rho(x)$, the solution can be found using the convolution rule in Fourier space.

	Real space	Fourier space
Special RHS	$h(x) = \delta(x)$	1
Solution to any ODE with this RHS	$G(x)$	$\hat{G}(k) \cdot 1$
General RHS	$h(x)$	$\hat{h}(k)$
Solution by convolution	$\int_{-\infty}^{\infty} G(x - y)h(y)dy$	$\hat{G}(k) \cdot \hat{h}(k)$

The solution for a general RHS is useless, since it involves an integral.

Compressible 2D flows

The Poisson's equation for compressible 2D flows can be expressed using point vortices.

$$\nabla^2\Psi = -\omega = \sum_{n=1}^{N} \Gamma_n \delta(x - x_n)$$

The velocity field $V(z)$ is $V(z) = \ rot\ \Psi$. $V(z) = u - iv$

$$V(z = x + iy) = \ -\frac{i}{2\pi}\sum_{n=1}^{N}\frac{\Gamma_n}{z - z_n}$$

Fast multipole method

The sum needs to be evaluated for all vortices and has therefore a complexity of $O(N^2)$. Using the fast multipole method, the sum can be evaluated in only $O(N \log N)$.

The multipole of order P is

$$V(z) = \ -\frac{i}{2\pi}\sum_{k=0}^{P}\frac{\alpha_k}{(z - z_M)^{k+1}}$$

$$\alpha_k = \sum_{m=1}^{M} \Gamma_m (z_m - z_M)^k$$

- z_M is the center of a multipole
- z_m is a particle
- M is the number of particles belonging to a cluster.
- $\left(\frac{R_M}{R}\right)^{P+1}$ is the error, where R_M is the cluster size, R is the distance to the cluster.

Barnes-Hut algorithm

For efficiency, the clusters should be large if the distance from a particle to other particles is big. Therefore, we need a hierarchy of clusters, sorted in a tree.

Such a tree could be constructed with the following rule:

1. Start with one node representing the whole domain.
2. Divide the domain in four subdomains and add them as children.
3. For every child, do step 2, unless the subdomain contains 3 or less particles. In this case, add the particles as leaves.
4. From leaves to root, compute the multipole expansion at every node O(pN)

To evaluate the interactions, start at the coarsest box and traverse it to finer boxes. If a box is distant enough from the particle of interest, take its multipole expansion, otherwise proceed to its child boxes. If the children are particles then compute direct particle-particle interaction.

For every particle, the tree has to be traversed to compute the velocity. Therefore, O(N log N)

Numerical solution to diffusion equation

Diffusion equation

$$\frac{\partial c(x,t)}{\partial t} = D\frac{\partial^2 c(x,t)}{\partial x^2}$$

Finite differences

Forward difference in time: $\frac{c^{n+1}-c^n}{\Delta t} \approx \frac{\partial c}{\partial t}$

Centered differences in space: $\frac{c_{i+1}-2c_i+c_{i-1}}{\Delta x^2} \approx \frac{\partial^2 c}{\partial x^2}$

Crank-Nicolson-Method (2D)

$$\frac{u_{ij}^{n+1}-u_{ij}^{n}}{\Delta t} = \frac{1}{2}\left(\delta_x^2+\delta_y^2\right)\left(u_{ij}^{n+1}+u_{ij}^{n}\right)$$

where δ_p is the central difference operator for the *p*-coordinate

- Central difference in space
- Trapezoidal rule for time advancement
- Unconditionally stable, but expensive, because it leads to a broad-banded matrix.

Von Neumann stability analysis

- Assume solution is of the form $c_i^n = \rho^n \exp(ikx_j)$
- Insert this into difference stencil and see what relationship you get between Δx and Δt

Alternating direction implicit

Step 1: $$\frac{u_{ij}^{n+1/2}-u_{ij}^{n}}{\Delta t/2} = \left(\delta_x^2 u_{ij}^{n+1/2}+\delta_y^2 u_{ij}^{n}\right)$$

Step 2: $$\frac{u_{ij}^{n+1}-u_{ij}^{n+1/2}}{\Delta t/2} = \left(\delta_x^2 u_{ij}^{n+1/2}+\delta_y^2 u_{ij}^{n+1}\right).$$

where δ_p is the central difference operator for the *p*-coordinate

- Unconditionally stable
- Step 1: N systems (one for each i) of M tridiagonal systems have to be solved.
- Step 2: M systems (one for each j) of N tridiagonal systems have to be solved.
- Overall cost: O(2MN)