HPCSE HS16 Tanja Baumann baumanta@student.ethz.ch

Theory

Adapted from: Alexander Jöhl

Layout of a cluster: Cluster > Node (+Memory) > Processor (CPU) > Core
Physical limits for faster serial computers: Transmission speed, limit to miniaturization, Economic limits (expensive to go faster), Energy limits (cooling)
Flop/s: Floating point operations per second (Mega 10^6, Giga, Terra, Peta, Exa, Zetta,Yotta)
Nominal Peak Performance R_{Peak} (PP): v: SIMD width in # of doubles, n: # of cores

```
PP[Flop/s] = f_Processor[Hz = cycles/s] * c[Flop/cycle] * v[ ] * n[ ]
```

Benchmark PP: xo = s[0] -> x1 = a*x0+b -> x2 = a*x1+b -> … -> s[0] = xn (count flops & measure time)
Performance metrics:
- Time to solution
- Speedup (p) = T (1)/T (p) where: T (p): time on p processors
- Strong scaling: Keep problem size const. & increase # processor y=x
- Weak scaling: const. work per core. Increase prob. size with p Effici.
- Strong/ Weak scaling Efficiency (p) = Speedup (p)/p (*100%)

Amdahl's Law: Speedup is limited by serial fraction s of code: Formula assumes Load balance. 90-times faster -> Speedup = 90. Speedup = T_old/T_new

$$T_{new} = \frac{Time\ Affected}{\#processors} + Time\ Unaffected$$

$$Speedup = \frac{1}{(1 - Fraction\ Time\ Affected) + \frac{Fraction\ Time\ Affected}{\#\ processors}}$$

Roofline Model

Operational Intensity: Operations per byte of DRAM Traffic. Code dependent.
Roofline Plot: log-log scale, relates Performance f [Flop/s] to Operational Intensity r [Flop/Byte]

- Performance of kernel k limited by DRAM bandwidth: $f(r_k) = r_k * PB$
- Performance of kernel k limited by compute power : $f(r_k) = PP$

```
PB[GB/s] = f_Memory[GHz] * channelsize[bits] * #channels/8[bits/byte]
         = f_Memory[GHz]* #channels * w[bit/(cycle*channel)]*0.125[byte/bit]
         = f_Memory[GHz]* #channels * 64bit * 0.125[byte/bit]
```

Benchmark PB: copy one array into another: b[i] = a[i], i=[0,N]-> measure time & compute memory access [GB] = (10^-9)*2*sizeof(float)*N

Euler

Enter Euler cluster from Terminal

```
ssh –Y baumanta@euler.ethz.ch
```

Enter Euler folder from Nautilus

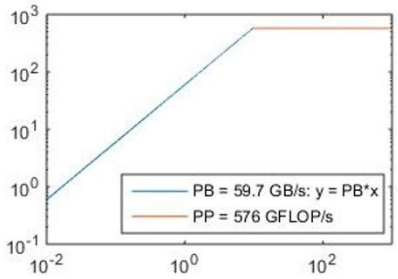
```
Crtl L & type: sftp://baumanta@euler.ethz.ch/cluster/home/baumanta
```

Run on Euler

```
bsub –W 00:10 –n 24 ./run.cpp
```

Gnuplot

```
gnuplot
set term png
set output ''pic.png''
plot ''output.txt'' using 1:2 with lines
```



C++11 Threads

To compile use (gcc-4.7): #of threads possible: std::thread::hardware_concurrency()

```
g++ -std=c++11 –pthread... or: clang++ -std=c++11 –stdlib=libc++...
```

To use parallel computing launch a thread, which executes a function:

```
#include <thread>
std::thread t (func, argf1, argf2, ...);
```

Can assign/copy from rvalues:

```
result[3] = std::thread(f, argf1, argf2, ...);
```

Threads can be placed in containers (for more than 1 added thread):

```
std::vector<std::thread> t(10);
t[1]=([](){std::cout << "Hello world!\n";}); //Lambda func.
```

Lambda Functions

[&] take everything by reference, [=] take everything by value
[ ] more relevant, ( ) parameter list, as in function arguments
-> lambda func. takes parameters at the moment when it is evaluated, not when it is defined!
Capture y by reference [&y], if y is changed after func. def. the new value (at execution time) will be used.
Capture y by value [=y] will use the value at func. def. time for execution.

Threads can be passed to or returned form functions.
The threads are terminated (or joined to master thread) by:

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

Results by functions in other threads can only be used after the corresponding thread is terminated.
Can use futures to hold future values (first argument ensures different thread):

```
std::future<double> fi = std::async(std::launch::async, func, argf1, argf2, )
```

Get the future value with (waits automatically until thread is done):

```
result = fi.get();
```

For references in function heads:

```
result = std::thread(f, std::ref(argf1), ...)
```

Results from references can only be used after corresponding thread is terminated.

Race Conditions, Mutex

Threads agree to aquire a lock on the mutex before accessing data & unlocking it when done.
Example, Mutex for specific variable:

```
#include <mutex>
main()
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*del,(i+1)*del);
...
```

```
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;}
```
(l unlocked by running out of scope)

Other example:

```
struct collab{
collab() : partner(0) {}
~collab() { decouple(); }
void couple(collab* new_partner);
void decouple();
private:
collab* partner;
std::mutex gate;
};

typedef std::lock_guard<std::mutex> guard;

struct lock2{
lock2(std::mutex& a, std::mutex& b): l0( a ),l1( b ){}
guard l0, l1;
};

void collab::couple(collab* other){
decouple();
other->decouple();
//lock2 g(gate,other->gate);
```
(bad, deadlock possible!)
```
guard g1(gate, defer_lock);
guard g2(other->gate,defer_lock);
std:lock(g1,g2
```
(better!)
```
if (partner || other->partner) return;
partner = other;
other->partner = this;}
```

Types of mutexes and locks:

- unique_lock<mutex> l(m); // locks the lock
- unique_lock<mutex> l(m,std::adopt_lock); // adopts the lock state
- unique_lock<mutex> l(m, std:: defer_lock); // does not lock yet
- unique_lock<mutex> l(m, std:: try_to_lock); // tries to lock
- unique_lock<mutex> l(m,abs_time); // tries to lock, with timeout

Functions:

- l.owns_lock(); // returns whether It is locked
- if (l) … // tests whether locked
- l.try_lock(); // tries to lock and returns whether it 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); std::lock(l1,l2,l3); … // lock multiple locks at the same time

Cache Thrashing

Create local variables in Thread functions to do the calculation and write only final result in the result[i] variable. If calculations (e.g. sum) are done directly as result[i] +=… in a loop, then one thread invalidates the cache of the others, result has to be reloaded all the time.

Barrier

```
#ifndef HPCSE_BARRIER_HPP
#define HPCSE_BARRIER_HPP
#include <thread>
#include <mutex>
#include <cassert>
#include <limits>

class barrier{
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 gen of wait
            m_count = m_total;
            m_generation++;
        }
        else {
            lock.unlock();
            while (true) {
                lock.lock();
                if (gen != m_generation)
                    break;
                lock.unlock();}
        }
}
unsigned int num_waiting() const{
        std::unique_lock<std::mutex> lock(m_mutex);
        return m_count;
}
private:
    mutable std::mutex m_mutex;
    unsigned long const m_total;
    unsigned long m_count;
    unsigned long m_generation;
};

#endif //HPCSE_BARRIER_HPP
```

Atomics Types:

Updates on atomic types look like a single operation → no race condition and faster than mutexes.

```
#include <atomic>
std::atomic<int> value = ATOMIC_VAR_INIT(0);
```

They can have the following operations: ++, --, +=, -=, &=, |=, ^=.

Random Numbers

Real random numbers hard to obtain. → Use pseudorandom number generators. They are deterministic, but look randomlike enough to use. Use different generators and compare results to be sure.

Generating independent sequences:

Leapfrog: elements are given cyclically: n cores, element i is given to core i mod n

Sequential Splitting: each core gets a block (sequentially, nonoverlapping)

Independent sequences: LFG generator can produces independent sequences depending on seed.

Stochastic Seeding: Use a generator to fill the seed blocks.
Usually get random numbers u ~uniform[0,1[. Usually have to get other distributions from u:

- **Uniform in interval $[a, b]$:** $x = a + (b - a)u$
- **Invert distribution function**:

  Exponential distribution: $f(x) = \lambda \exp(-\lambda x) \rightarrow x = -\frac{1}{\lambda}\ln(1-u)$

  Normal distribution: (get 2 normally distributes numbers from 2 uniform ones)

  $f(x) = \frac{1}{\sqrt{2\pi}}\exp(-x^2) \rightarrow n_1 = \sqrt{-2\ln(1-u_1)}\sin(2\pi u_2), n_2 = \sqrt{-2\ln(1-u_1)}\cos(2\pi u_2)$
- **Rejection method**:

  Two generators, one with distribution h that bounds f: $f(x) < \lambda h(x)$ and one uniform with interval [0,1[; Accept x if $u < f(x)/(\lambda\, h(x))$ otherwise get a new pair. Needs good guess $h(x)$ to be efficient

Random Number Generators: Include header:

```
#include <random>
```

How to use it:

```
std::mt19937 mt; // create an engine (Mersenne Twister)
mt.seed(42); // seed the generator //or: std::mt19937 mt(42);
std::uniform_int_distribution<int> uint_d(0,10);
randnumber = uint_d(mt);
// for arrays:
double x* new double[N];              //array
double A* new double[N*N];            //matrix
std::generate_n(x,N,([& uint_d,& mt] () {return uint_d(mt);}));
std::generate_n(A,N*N,...) //use Lambda functions
```

Auto Keyword: tells compiler to deduce type of variable by initializer argument:

```
auto value = 0.1; //compiler recognizes it as double
```

Saves typing of complicated types (eg. vector of vector of struct).

OpenMP

To compile use:

```
g++ -fopenmp ...
```

Include Header:

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

To make a part of the code parallel:

```
#pragma omp parallel
{...}*
```

List of **#pragma omp**: (* Barrier at the end)

- master — performed only by master thread
- single* — performed only by one single thread
- critical ([name]) — if a thread is already in a sec "name" all others will wait
- barrier* — all threads wait until the last one has called the barrier
- atomic — the following update operation is atomic (only certain operations)
- threadprivate (list) — listed variables are thread-private (every thread gets own copy)
- flush (list) — listed vars are written back to memory (all thr. have same value)
- sections* — each section gets assigned to different thread (def inside parallel)

Optional clauses:

- if (scalar_expr.) — Only parallelize if the expression is true.
- private (list) — The specified variables are thread-private, init. val undef.
- shared (list) — The specified variables are shared among all threads
- default (shared | none) — Unspecified variables are shared or not
- copyin (list) — Initialize 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. Allowed operations: +, -, *, &, ^, |, &&, ||, min, max
- num_threads (integer-expr.) — Set the number of threads

Examples:

```
#pragma omp parallel private(i) shared (n) if (n>10)  {...}
```

```
double result;
double a = 0;
double b = 20;
#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);}
```

Parallelize for Loops:
#pragma parallel for*
Additional directives:

- 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
- collapse (n) — Collapse n nested loops into one and parallelize it
- schedule (*type* [,chunk]) — Specify the schedule for loop parallelization

  Types: STATIC — Loop iterations divided into fixed chunks & assigned statically
  DYNAMIC — Loop iterations divided into fixed chunks & 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 dep. on OMP_SCHEDULE environment variable
  AUTO — Decided by compiler and/or runtime system

Certain functions:

```
int omp_get_num_threads(); //current number of threads
void omp_set_num_threads(); //set number of threads
int omp_get_thread_num(); //number of this thread
double omp_get_wtime();       // gets abs time. Delta = t2-t1
```

Environment variables (in terminal):

```
OMP_NUM_THREADS=4  //run with 4 threads
./a.out
```

```
OMP_PROC_BIND = TRUE; //binds threads to processors (NUMA!)
```

False Sharing: (e.g. in a tight for loop.)
Occurs when: multiple processors update data in same cache line. And this happens frequently.

```
for (int i = 0; i < N; i++) {
        #pragma omp parallel for
        for (int j = 0; j < N; j++) {
                A[i*N+j] = some_value(i, j);}} //False Sharing
```

NUMA and First-Touch:
Memory affinity is not decided by the memory allocation but by the initialization. First-touch principle: memory mapped to the NUMA domain that first touches the data.

```
#pragma omp parallel for
for(i=0;i<N;i++) {
a[i] = 1.0; b[i] = 2.0; c[i] = 0.0;}
#pragma omp parallel for
for(i=0;i<N;i++) {
a[i] = b[i] + d * c[i];}
```

Memory

Locality: Temporal (Loop), Spatial (sequential access to array elements)
Hit: we look in next higher level memory for the entity. If found -> Hit, otherwise Miss.
Hit-rate: percentage of entities found in next higher level memory.
Effective Access Time (EAT) = Hit-rate*(Cache Access Time) + (1-Hit-rate)* (Main Mem AT)

Vectorization, SIMD (Single Instruction Multiple Data)

SSE (Streaming SIMD Extensions), AVX (Advanced Vector Extensions)
XXM-register (for SSE) are 128 bit (16 bytes) and can store: 2 doubles, 4 floats, 2 64bit int, ect. YMM-register (for AVX) are 256 bit (32 bytes), overlap with XMM-registers.
SIMD instructions act on whole register "packaged floating point instruction".
[A1 | A2 | A3 | …] +[B1 | B2 | B3 | …] = [A1+B1 | A2+B2 | A3+B3 | …]
Alignment:
SSE-registers: 16byte-alignment, AVX-registers: 32-byte alignment, Cache-line: 64-byte

```
float alignas(16) tmp[4];
#include<malloc.h>
space = (double*)malloc(N*sizeof(float));
if(space==NULL){return 1;}
free(space) //frees memory
or: _aligned_malloc(size,64);
```

Loops can only be vectorized if there are no dependencies between iterations, or if they are far enough apart. E.g. a[i] = a[i-p] + a[i-q]; is OK for vectors of length min(p,q).
Compile with (depending on the CPU): find out what is supported: cat /proc/cpuinfo

```
g++ -msse3 or -msse4 or -maxv or -mtune=native (loads all)
```

Include header (loads all supported headers):

```
<x86intrin.h>
```

Data Types: (2 underscores at beginning)

__m128	4 floats	__m256	8 floats	__m512	16 floats
__m128d	2 doubles	__m256d	4 doubles	__m512d	8 doubles
__m128i	Int of any size	__m256i	Int of any size	__m512i	Int of any size

SSE instructions are named as: _mm_name_type
AVX instructions are named as: _mm256_name_type
Careful when mixing SSE with AVX instr.! Call `_mm256_zeroupper( )` to clear the upper bits before switching from AVX to SSE.
Types:

type	Length (bits)	description	type	Length (bits)	description
ss	32	A single float	pi8	64	8 8bit int
ps	32,128 or 256	4,8,16 floats	pi16	64	4 16bit int
sd	64	A single double	pi32	64	2 32bit int
pd	32,128 or 256	4,8,16 doubles	epi8	128,256 or 512	16,32,64 8bit int
si64	64	Any integers	epi16	128,256 or 512	8,16,32 16bit int
si128	128	Any integers	epi32	128,256 or 512	4,8,16 32bit int
si256	256	Any integers	epi64	128,256 or 512	2,4,8 64bit int

Instructions: (Load/Store)

name	type	description
set1	all	Sets all elements to given value
set	all	Sets each element to different value
setr	all	Set in reverse order
setzero	pd, ps, si64, si128, si256	Set to zero
load1	pd, ps	Load single value into each elem. of register
broadcast	pd, ps	Same as load1 but much faster (AVX only!)
load	pd, ps, ss, sd, si128, si256	Load values from memory into 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 to memory
storeu	pd, ps, ss, sd, si128, si256	Store v. from register to unalign. mem (slow!)
stream	pd, ps, pi, si128, si256	Store values into memory bypassing the cache

Prefetch Instructions (slides on SIMD)
Instructions: (Aritmethic)

name	Explanation	name	Explanation
add, sub	+, -	ceil	round up
mul, div	*, /	floor	round down
addsub	- on even, + on odd elem.	round	round (allows specif.)
min	min	rcp	reciprocal (inverse)
max	max	or, xor	bitwise |,^
sqrt	sqrt	and, andnot	bitwise &, &!
cmpeq,cmpneq	x==y, x!=y	cmplt, cmple	x<y, x<=y
cmpgt, cmpge	x>y, x>=y	test_all_zeros	test if all bits are 0 (only i128)

Example

```
std::vector<float,Allocator> x;
_mm256_load_ps(&x[i]);
__m256 res =...;
_mm256_store_ps(&x[i],res);
```

Example (y = ax+y): restrict is used to tell compiler, that there are no dependencies (x~=y)

```
void saxpy(int n, float a, restrict float* x, restrict float* y){
__m128 x0 = _mm_set1_ps(a); // load the a 4 times into a register
assert(((std::size_t)x) % 16 == 0 && ((std::size_t)y) % 16 == 0);
int ndiv4 = n/4;
for (int i=0; i<ndiv4; ++i) {// loop over chunks of 4 values
        __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(x0,x1); // multiply
        __m128 x4 = _mm_add_ps(x2,x3); // add
        _mm_store_ps(y+4*i,x4);} // store back aligned
for (int i=ndiv4*4 ; i< n ; ++i) // do the remaining entries
y[i] += a*x[i];}
```

Automatic vectorization:

```
g++ -ftree-vectorize ...
```

Generate vectorization reports: -Om -ftree-vectorizer-verbose=n with n = {1,…, 6}

Dense Linear Algebra

BLAS: Basic Linear Algebra Subprograms
Level1: Scalar or Vector operations O(1) or O(N)
Level2: matrix-vector operations O(N^2)
Level3: matrix-matrix operations worse than O(N^2) often O(N^3)

```
g++ -lgfortran and: #include <cblas.h>
```

Example:
Fortran DDOT function (forms dot product of 2 vectors)

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

has the following C++ prototype:

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

It can be called as :

```
std::vector<double> x(10, 1.), y(10, 2.);
```

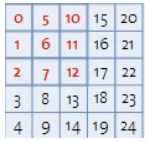
```
// calculate the inner product
int n=x.size();
int one = 1;
double d = ddot_(n,&x[0],one,&y[0],one);
```

Fortran stores matrices column-major, C/C++ row-major -> matrices are typically transposed
Fortran indices start at 1, C/C++ indices start at 0 -> A[i][j] in C++ is A[j+1][i+1] in Fortran

Example: in DDOT the increment exists to treat rows or cols of matrices as vectors
DX = startOfStorage + 2; INCX = 5;
BLAS Prefixes: I (int), S(float), D(double), C(std::conlex<float>), Z(std::complex<double>)
_DOT: gets called as IDOT, SDOT, DDOT,...
BLAS Leve1 functions:

_ _DOT	inner prod.	_COPY	copy x->y	_NRM2	2-Norm
I_AMAX	max(x,y)	_SWAP	swap x->y	_SCAL	scale x

BLAS Level2 functions: Matrix type behind Prefix: GE,GB, SY, TR, ... see slide 12 dense lin.alg
Matrix operations accept 3 arguments: #rows = 3, #cols = 3, leading dimension = 5
For BLAS Level2 & Level3 functions, see slide 15&16 dense lin.alg



Example: Matrix-Matrix multiplication

$$c(i,j) = \sum_{k=0}^{n} a(i,k) * b(k,j)$$

```
for (i=0; i<rows(a); i++){  //loop order i,k,j is better than i,j,k!
        for(k=0; k<n; k++){
                for(j=0; j>cols(b); j++){
                        c(i,j)+= a(i,k) * b(k,j);}}}
```

Sparse Linear Algebra

Many problems can be formulated as sparse matrix problems, e.g. Diffusion 1D:

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

$$\vec{f}(t+\Delta t) = M\vec{f}(t) \quad \textit{where } M \textit{ is tridiagonal (circular)}, 3N-4 \textit{ nonzero entries}$$

Same works for 2D Diffusion, where M is a banded Matrix (less than 5N nonzeros).
Also for the Poisson equation: $\Delta\varphi = f$ *where*: $\Delta\varphi = 4\pi G\rho \quad \rightarrow \quad M\vec{\varphi} = 4\pi G\vec{\rho}$

$$\vec{\varphi}(n+1) = M\vec{\varphi} - \pi(\Delta x)^2 G\vec{\rho}$$

Unstructured grids: Page Rank, $\vec{p} = W^T\vec{p}$ (largest left eigenvector)
Eigenvalues & Eigen vectors from Power Method (iterative):

```
y = y_init              //initial guess
for k= 1,2...
        v = y/norm(y);
        y = A*v;
        theta = v*y;
        if(abs(y-theta*v) < bound){ stop: accept lambda = theta & v}
```

Page Rank Matrix is not really sparse: Random surfer follows the link with probability d, or jumps to a random page with probability (1-d). Therefore zero entries are replaced with small finite values (1-d)/N.

$$\rightarrow M = constantMatrix\left(\frac{1-d}{N}\right) + dW^T$$

$$\rightarrow \vec{p} = constantVector\left(\frac{1-d}{N}\right) + dW^T\vec{p}$$

All those problems need: 1. Sparse Matrix/Vector multiplication, 2. Dense Vector operations
Storage of sparse Matrices: Best : not store at all and just code the operation. Or: store packed.
Compressed Sparse Rows (CSR) format: 3 arrays (column indices, values and row starts)
Matrix/vector multiplication with CSR: (if multiplication with transposed matrix: prefer CSC)

```
#pragma omp parallel for // loop over all rows
for(size_type row = 0; row < dimension() ; ++ row) // loop over nonzeros
        for(size_type i = row_starts[row] ; i != row_starts[row+1] ; ++i)
                y[row]+= data[i] * x[col_indices[i]];
return y;
```

MPI (Message Passing Interface)

Network interconnects (connection structure between nodes): Cray Gemini (3D Torus), IBM Blue Gene (5D Torus), K computer (6D Torus)
Beowulf clusters (Brutus) have similar nodes as high end comp. but cheaper networks.
Distributed Memory Programming

- #processes usually static (1 per core): numbered by integer "ranks" [0,1, ... , p-1]
- All data local to 1 some processor (in protected memory space): no race conditions!
- Access to data of other processors must be explicitly managed by **message passing**

```
module load open_mpi // on Euler
module load mpi/openmpi-x86_64  //on slab
module load mpi/mpich-x86_64  //on slab
```

Compile with:

```
mpic++ ... evt. mpicc, mpicxx
```

Run the program with:

```
bsub -n 48 < script //Euler
mpiexec.gforker -n number_of_processes ./a.out //slab
or: mpiexec –mca btl self,sm -n number_of_processes ./a.out //slab
```

script:

```
mpirun -n number_of_processes1 ./a.out >output.txt
mpirun -n number_of_processes2 ./a.out >output.txt
```

Include header:

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

Initialize MPI

```
int main(int argc, char **argv){
        MPI_Init(&arc, &argv);//initialize environment

        int rank, size;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);

        MPI_Finalize();                 //clean up environment
        return 0;}
```

More initialization/termination functions

```
MPI_Abort( MPI_Comm comm, int errorcode);    //terminates MPI
MPI_finalized(int *flag);                    //sets flag to TRUE if finalized
MPI_Initialized(int *flag);      //sets flag to TRUE if initialized
```

```
int MPI_Barrier(MPI_Comm comm) //waits for all ranks in comm(sync)
```

Measure Time

```
double T1 = MPI_Wtime();      //barrier which syncs
...do stuff
double T2 = MPI_Wtime();
double elapsed = T2-T1;
```

Messages:

envelope				body		
source	destination	communicator	tag	buffer	count	datatype

```
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);
```

! Buffer size on receiving side is the allocated memory, and thus the max size that can be received (not the actual size!)

Wildcards: MPI_ANY_TAG, MPI_ANY_SOURCE (receives messages from any rank with any tag)

Example: Send/Receive

```
int main(int argc, char** argv) {
   MPI_Init(&argc, &argv);
   int rank;
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   if(rank==0) {      // "master"
        MPI_Status status1 status2;
        char txt[100];
        MPI_Recv(txt, 100, MPI_CHAR, 1,42,MPI_COMM_WORLD,&status1);
        std::cout << txt << "\n";
        int y;
        MPI_Recv(&y, 1, MPI_INT, 1,123,MPI_COMM_WORLD,&status2);}
   else{  // "worker"
        std::string text="Hello world!";
        int x = 130;
        MPI_Send(const_cast<char*>(text.c_str()),text.size()+1,
               MPI_CHAR,0, 42, MPI_COMM_WORLD);
        MPI_Send(&x,1, MPI_INT,0, 123, MPI_COMM_WORLD)}
MPI_Finalize();
return 0; }
```

Probing for Messages

```
MPI_Status status;
int count;  // wait for a message
MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, &status);
std::cout << "A message is waiting from "<< status->MPI_SOURCE<<  "with
tag "<< status->MPI_TAG;
MPI_Get_count(&status, MPI_INT, &count)  // get the element count
std::cout << "and assuming it contains ints there are "
       << count << "elements";
```

Different versions of Send/Receive

- MPI_Ssend // synchronous send: returns when the destination has started to receive the message
- MPI_Bsend // buffered send: returns after making a copy of the buffer. The destination might not yet have started to receive the message
- MPI_Send // standard send: can be synchronous or buffered, depending on message size
- MPI_Rsend // ready send: an optimized send if the user can guarantee that the destination has already posted the matching receive
- MPI_Recv // blocking receive: returns once the message has been received. The status object can be queried for more information about the message

! Deadlocks possible (Ssend, Recv, for both ranks -> both ranks wait for receive) change order, or:

```
MPI_Sendrecv(&ds,1,MPI_DOUBLE,1,tag,&dr,1,MPI_DOUBLE,1,tag,MPI_COMM_WORLD,
&status); // send double ds, get double dr
```

Asynchronos functions return immedeatly: (fills in an MPI_Request object, which can be tested) Exists also as Issend, Ibsend,Irecv. send communication request ->do calculations which can be done without info ->wait for info -> do rest of calculations. Irecv before Isend!

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

int MPI_Wait (MPI_Request *request, MPI_Status *status)MPI1 slide36
// waits for the communication to finish and fills in the status
```

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype
datatype, MPI_Op op, int root, MPI_Comm comm);
E.g:    MPI_Reduce(&localsum, &sum,1, MPI_LONG_DOUBLE, MPI_SUM, 0,
       MPI_COMM_WORLD);//collect all to master rank(0),exec.by all
int MPI_Allreduce(...) //reduc. & bcast, result avail. on all rank
```

MPI_IN_PLACE can be used in send buffer to avoid local sums:

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

Gather operations

```
int MPI_Gather(void*sendbuf,int sendcnt,MPI_Datatype sendtype,void
*recvbuf,int recvcnt,MPI_Datatype recvtype,int root,MPI_Comm comm)
//sendcnt needs to be the same for all ranks other fun.MPI1 slide45
Example parallel sum:
int nlocal = N / num_processes;
std::vector<float> x(nlocal,-1.2), y(nlocal,3.4), z(nlocal);
for( inti = 0; i < nlocal; i++ ) z[i] = x[i] + y[i];
std::vector<float> fullz(N);
MPI_Gather(&z[0],nlocal,MPI_FLOAT,&fullz[0],nlocal,MPI_FLOAT, 0,
       MPI_COMM_WORLD);
if (rank == 0) std::cout << std::accumulate( fullz.begin(),
              fullz.end(), 0. ) << std::endl;
```

Scatter operations:

```
int MPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype sendtype,
void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm
comm)  // recvcnt needs to be the same on all ranks
```

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.

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
```

Packing and unpacking data MPI1 slide 52, Distributed vector operations MPI1 slide 55

Scan & Exscan (cumulative sum) MPI2 slide51

File I/O see MPI2 slide 56-72

```
#include <mpio.h>
```

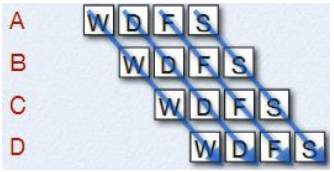
Example

```
double data[nlocal];
for(inti = 0; i < nlocal; i++) data[i] = 100*rank + i;
int step = 0;
char filename[256];
sprintf(filename,"mydata_%05d.bin", step);
MPI_File f;
MPI_File_open(MPI_COMM_WORLD, filename , MPI_MODE_WRONLY |
MPI_MODE_CREATE, MPI_INFO_NULL, &f);
MPI_File_set_size (f, 0);
MPI_Offset base;
MPI_File_get_position(f, &base);
MPI_Offset len = nlocal*sizeof(double);
MPI_Offset offset = rank*len;
MPI_Status status;
MPI_File_write_at_all(f, base + offset, data, nlocal, MPI_DOUBLE,
       &status);
MPI_File_close(&f);
MPI_Finalize();
Read:
MPI_File_open(MPI_COMM_WORLD, filename , MPI_MODE_RDONLY,
       MPI_INFO_NULL, &f);
MPI_File_read_at_all(f, base + offset, data, nlocal,
       MPI_DOUBLE, &status);
```

Pipelining MPI2 slide 73-81

For large #instructions: speedup~#pipe stages

Piplining improver throughput. Timeloss at start & end



Particles (N-Body Problem)

Equations of motion of the particles:

$$m_i \frac{d^2 \vec{x}_i}{dt^2} = \vec{F}_i = -\nabla_{\vec{x}_i} V(\vec{x}_1, \dots, \vec{x}_N)$$

Total energy: $E_{tot} = E_{kin} + V$ with $E_{kin} = \sum_i \frac{m_i}{2} |\vec{v}_i|^2$

$V(\vec{x}_1, \dots, \vec{x}_N) = \sum_{i<j} V_{ij}(r_{ij}) = \frac{1}{2}\sum_{i \neq j} V_{ij}(r_{ij})$ *where*: $r_{ij} = |x_i - x_j|$

Potential: Coulomb: $V_{ij}(r) = {}^{q_i q_j}/_{r}$, Gravity: $V_{ij}(r) = -g * {}^{m_i m_j}/_{r}$, LennardJones ParticlesS3

Verlet integrator:

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

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

Initial conditions with fixed energy E_{tot}: Place particles in some way, calculate potential energy V, choose random velocities, calculate kinetic energy E_{kin}. Rescale velocities: $\lambda = \sqrt{\frac{E_{tot} - E_{pot}}{E_{kin}}}$ and $\vec{v}_{i,new} = \lambda\, \vec{v}_i$.

Boundary types: Open Boundaries, Hard Walls, Periodic Boundaries

Truncation: Long range forces have extremely high comp. effort. -> set a cutoff distance r_c from which on the interaction is zero. But pay attention to offset (discontinuity!) shift whole Potential upwards/downwards by $V(r_c)$. -> $V_{trunc}(r) = V(r) - V(r_c)$

Cells: Subdivide the domain into cells. Cutoff distance = cell size -> only consider adjacent cells

Cell Lists: A: For each cell, create container with particles in this cell
B: For each particle, save in which cell it belongs, then sort

Particle Mesh Methods slides 1-16

Moment conserving interpolations: $\int W(x)dx = 0$ *and* $\int x^{\alpha} W(x)dx = 0$

m-th order: Vanishing M.conditions for 1<= α <= m-1 require at least m interpolation points

Higher dimensions: Tensor product of 1D kernels $W(\bar{x}) = W_1(x_1) * W_2(x_2) * \dots * W_n(x_n)$

Finite Differences

Derivative approximations:

Central difference: $f_i' \cong \frac{y_{i+1} - y_{i-1}}{2\delta x}$

Backward difference: $f_i' \cong \frac{y_i - y_{i-1}}{\delta x}$

Forward difference: $f_i' \cong \frac{y_{i+1} - y_i}{\delta x}$

Second derivative: $f_i'' \cong \frac{f_{i+1} + f_{i-1} - 2f_i}{\delta x^2}$

Applied to diffusion 1D:

$$f_{i,j+1} = f_{i,j} + \frac{D\delta t}{\delta x^2}\left(f_{i+1,j} + f_{i-1,j} - 2f_{i,j}\right)$$

Von-Neumann stability analysis: Assume solution of the form: $f = \rho^n e^{ikx_j}$, insert into equation, then $\rho \leq 1$ has to hold. In this case it results in $\leq \frac{\delta x^2}{2D}$. D has to be independent of space, if it is dependent take $\max(D)$ for the stability analysis.

Diffusion

Can be described as random walks of multiple particles:

$x_i(n) = x_i(n-1) \pm \delta$, wether it is + oder – has the same probability.

Mean displacement if particles start at same point: $\langle x(n) \rangle = \frac{1}{N}\sum_{i=0}^{N} x_i(n) = \frac{1}{N}\sum_{i=0}^{N}[x_i(n-1) \pm \delta] = \frac{1}{N}\sum_{i=0}^{N} x_i(n-1) = \langle x(n-1) \rangle = \dots = \langle x(0) \rangle$

Mean square displacement is: $\langle x^2(n) \rangle = n\delta^2$

Fick's 1st Law: $J_x = -D\frac{\delta f}{\delta x}$, J_x is flux.

Fick's 2nd Law: $\frac{\delta f}{\delta t} = D\frac{\delta^2 f}{\delta x^2}$, in 3D: $\frac{\delta f}{\delta t} = D\left(\frac{\delta^2 f}{\delta x^2} + \frac{\delta^2 f}{\delta y^2} + \frac{\delta^2 f}{\delta z^2}\right)$

Monte Carlo Integration

Integration in higher order dimensions (> 8) with numerical methods inefficient (N points, d dimensions, n integrator order, error is of order $O(N^{-n/d})$).

Monte Carlo: do the integration at random points.

Want to compute an integral: $\langle f \rangle = \int_\Omega f(\vec{x})d\vec{x} / \int_\Omega d\vec{x}$

Sum up the values of the function at M random points x_i in Ω:

$\langle f \rangle = \frac{1}{M}\sum_{i=1}^{M} f(\vec{x}_i)$ *the error is*: $\sqrt{\frac{\mathrm{Var}(f)}{M}}$ *with*: $\mathrm{Var}(f) = \langle f^2 \rangle - \langle f \rangle^2$

Recipe:

1. Draw M random points x and evaluate the function $X = f(x)$
2. Store the number (M), sum $(\sum_{i=1}^{M} X_i)$, sum of squares $(\sum_{i=1}^{M} X_i^2)$
3. Calculate mean as approximation of expectation value $E[X] \approx \bar{X} = \frac{1}{M}\sum_{i=1}^{M} X_i$
4. Uncorrelated measurements: Error $= \sqrt{\frac{1}{N-1}\left(\overline{X^2} - \bar{X}^2\right)}$

Importance Sampling: By considering more points in the region with larger values, error is improved. Distribute points according to a probability function $p(x)$:

$\langle f \rangle = \langle {}^{f}/_{p} \rangle_p = \int_\Omega \frac{f(\vec{x})}{p(\vec{x})} p(\vec{x})d\vec{x} / \int_\Omega d\vec{x}$ Error $= \sqrt{\frac{\mathrm{Var}\left({}^{f}/_{p}\right)}{M}}$ -> find a function p similar to f.

Markov Chain Monte Carlo (Metropolis)

Markov Chain: Samples from a probability distribution. Where the probability of a new sample only depends on the previous, and not on the whole history

Metropolis Algorithm: Let f(x) be a PDF proportional to the desired PDF P(x)

1) Initialization: -Choose an arbitrary point x0 to be the first sample.
-Choose an arbitrary probability density Q(x|x0), that suggests a candidate for the next sample value x given x0. A usual choice for Q is a Gaussian distr.
! Q has nothing to do with P! but Q symmetric Q(x|x0)= Q(x0|x)

2) Iterations: -Generate a candidate x* by drawing it from Q(x*| x(n))
-Calculate acceptance ratio α = f(x*)/f(x(n))
→ because f is proportional to P: α = f(x*)/f(x(n)) = P(x*)/P(x(n))
-If α >=1: x* is more likely than x(n) and is automatically accepted → x(n+1) = x*. If α < 1: x* is accepted with probability α. If x* is rejected, then x(n+1) = x(n). Can be written as : Prob(keep x*) = max(1,α)

Metropolis -Hastings: Q not symmetric, α = f(x*) Q(x(n)|x*)/[f(x(n)) Q(x*|x(n))

Detailed Balance condition: f(x) Q(x|x*) = f(x*) Q(x*|x) necessary for Markov chain to converge

Disadvantages compared to direct sampling:

-Samples from Metropolis are correlated. If we want a set of independent samples, we have to throw away the majority, e.g. take only every n-th sample. (Value of n determined by autocorrelation between adjacent samples)

-Although the Markov Chain eventually converges to the desired distribution, the initial distribution may be quite different (especially if x0 is chosen in a low density area). Therefore, a equilibration period is needed, where the first samples are dumped (Amdahl: parallelizing 1 chain maybe better)

General C++ Stuff

Large numbers: 1e9

argc: #strings argv points to (1+#arguments) argv: argument vector. argv[0] = ./a.out

```
int main(int argc, char* argv[])
int N = std::stoul(argv[1]);
double K = std::stod(argv[2]);
```

New

```
double*s = new double;  or: double* s = new double[N];
```

Function

```
void my_func(int* a, ...){ ... return res;};
call: my_func(a);
```

Define functions

```
define MULADD(b,x,a)  (b*(x+a))
```

Vectors

```
std::vector<double> x(n,100) //n=#elements, 100=init. val.
Perf.push_back(a)              //add value a at last position
sort(x.begin(),x.end(),std::greater<double>());
std::accumulate(x.begin(), x.end(),0.);    //#include<numeric>
x.insert(x.end(),val,n);
```

Templates

```
template<Typename T>
T max(T x, T y){
T temp ;
if(x>y){temp = x} ;
else{temp = y} ;
return value;}
```

Auto Keyword: tells compiler to deduce type of variable by initializer argument:

```
auto value = 0.1; //compiler recognizes it as double
```

Makefiles

Automatic variables:

$@	target name
$^	dependency list
$<	first dependency

Example:

```
#Define Tools & Parameters
CXX = g++
CXXFLAGS = -std=c++14 -Wall -Wextra
SYSFLAGS = -openmp -pthread

.PHONY all          //no file with name "all" therefore PHONY
all: main_serial main_parallel
main_serial: main_serial.cpp
        $(CXX) $(CXXFLAGS) -o $@ $<
main_parallel: main_parallel.cpp
        $(CXX) $(CXXFLAGS) $(SYSFLAGS) -o $@ $<
.PHONY clean
make clean:
        rm -f main_serial main_parallel
```