① Table of Contents

(2) (p. 14-19), Fick's 1st/2nd Law (p. 20-26), Diffusion on Infinite Bar (p. 27-29, see also theory), Diffusion Equation (p. 30) Finite Differences (p. 31-61): FD derivation (p. 31/32), Central/Backward/Forward Difference (p. 33), FD for Diffusion (p. 35-36), Tridiagonal systems/Thomas algorithm (p. 37-39), Von Neumann Stability Analysis (p. 40-43), Modified Equation/Consistency (p. 44-46), Crank-Nicholson Method (p. 42-54), 2D Diffusion (p. 55-57), Peaceman-Rachford Methods/ADI (p. 58-60), 3D Diffusion (p. 61), FFTs + PDEs (p. 63-68) Lecture 12: Particles/N-Body Solvers Particles Introduction (p. 2-52), Cell Lists (p. 53-64), Remeshing (p. 66-75) Lecture 12: Particle Strength Exchange (Notes) Approximation of a Continuous Function using Particles (p. 1), Smooth Particles (p. 1-5): See Theory for substitution Particle Strength Exchange (p. 5-7): See theory for derivation of (26) and (31) Lecture 13: Data-Level Parallelism/Vectorization ISA Extensions (p. 3-9): Types of Extensions (p. 9) Vector Registers, Alignment, Aliasing (p. 11-27): SIMD Lanes (p. 16), Alignment (p. 21), Aliasing (p. 22-27) Data Oriented Design (p. 29-34): Array of Structures/Structures of Arrays (p. 35-40) Compiler Vectorization/Intel Intrinsics (p. 42-67): Intel Intrinsics (p. 48-61): Intel SPMD Program Compiler (ISPC) (p. 63-89):

Tutorial 01: Scientific Computing Vectors (p. 5) Tutorial 02: BLAS Tutorial 03: Oja's Rule Taylor Series (p. 7) Tutorial 04: Name Mangling, Debugging, Memory Leaks, Profiling Name Mangling/Symbol Table (p. 3-11), Debugging (p. 12-16), Memory Leaks (p. 17-20), Profiling (p. 21-24) Tutorial 05: Parallel Data Storage Tutorial 06: Cell Lists Building Cell Lists (p. 9)

HW 01: Roofline Model/Performance Measures 3.) Operational Intensity: DAXPY/SGEMV/DGEMM, 1D Diffusion equation 4.) Roofline Model: Calculating Peak Performance, Diffusion Benchmark 5.) Amdahls Law! HW 02: Brownian Motion/Bug Hunting 1.) Brownian Motion: Reduction (histogram) 2.) OpenMP Bug Hunting HW 03: PCA/Oja's Rule 1.) Covariance Method: LAPACK 2.) PCA with Oja's Rule: Hebb's Rule, Oja's Rule, Sanger's Rule HW 04: Blocking MPI Routines/Processor Pipelines 1.) 2D Diffusion 2.) MPI Bug Hunt 3.) Processor Pipelining HW 05: Particle Methods/MPI 1.) Roll-up of a Vortex Line: Asynchronous MPI HW 06: 2D Diffusion with Structured Grids and Particles 1.) 2D Diffusion using ADI scheme: ADI discretization 2.) PSE

Exam Tips Searching in PDFs:

```
find . -name '*.pdf' -exec sh -c 'pdftotext "{}" - | grep --with-filename --label="{}" --color "Search term"' \;
```

(3) **Theory** **Fourier Transform: PDE Solution** **Fourier Transform:** Given $u(x,t)$, we have: $\hat{u}(k,t) = \int_{-\infty}^{\infty} u(x,t) e^{-ikx} dx$ **Inverse Fourier Transform:** Given $\hat{f}(k,t)$: $f(x,t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \hat{f}(k,t) e^{ikx} dk$ **Derivatives:** Taking the FT of a derivative of order n is the same as multiplication by $(ik)^n$, i.e. $\widehat{u_{xx}}(k,t) = (ik)^2 \hat{u}(k,t)$ **Diffusion on an Infinite Bar:** At $t=0$, $T = Q\delta(x)$ and at $x = \pm\infty$, $T=0$ $\forall t$. Solve $\frac{\partial T}{\partial t} = D \frac{\partial^2 T}{\partial x^2}$. We take the FT and get: $\frac{\partial \hat{T}(k,t)}{\partial t} = D(ik)^2 \hat{T}(k,t)$ [derivative property] $\rightarrow \frac{\partial \hat{T}(k,t)}{\partial t} = -Dk^2 \hat{T}(k,t)$. Integrating w.r.t. t leads to: $\hat{T}(k,t) = f(k) e^{-Dk^2 t}$. At $t=0$, we have $\hat{T}(k,0) = f(k)$ and from the initial condition $\hat{T}(k,0) = \int_{-\infty}^{+\infty} Q\delta(x) e^{-ix} dx = Q$. Combining both leads to $f(k) = Q$ and therefore $\hat{T}(k,t) = Qe^{-Dk^2 t}$. We now take the IFT: $T(x,t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \hat{T}(k,t) e^{ikx} dk = \frac{Q}{2\pi} \int_{-\infty}^{\infty} e^{-Dk^2 t} e^{ikx} dk = \frac{Q}{2\pi} e^{-\frac{x^2}{4Dt}} \int_{-\infty}^{\infty} e^{-Dt(k^2 - 2\frac{ikx}{2Dt} - \frac{x^2}{4D^2t^2})} dk = \frac{Q}{2\pi} e^{-\frac{x^2}{4Dt}} \int_{-\infty}^{\infty} e^{-Dt(k - \frac{ix}{2Dt})^2} dk$

The integral is a Gaussian with $2\sigma^2 = \frac{1}{Dt} \rightarrow \sigma = \frac{1}{\sqrt{2Dt}}$. The general solution to a Gaussian: $\int_{-\infty}^{\infty} ae^{-\frac{x^2}{2\sigma^2}} dx = a\sigma\sqrt{2\pi}$ gives: $\frac{Q}{2\pi} e^{-\frac{x^2}{4Dt}} \frac{\sqrt{2\pi}}{\sqrt{2Dt}} = \frac{Q}{\sqrt{4\pi Dt}} e^{-\frac{x^2}{4Dt}}$ **Von Neumann Stability Analysis:** **Stability Analysis of Diffusion Equation:** We assume periodic BC, PDEs with linear, constant coefficients and a uniformly spaced grid. Then, we can write a solution in the form: $u_j^{(n)} = p^n e^{ikx_j}$. Applying the ansatz to the discretization $u_j^{(n+1)} = u_j^{(n)} + \frac{\nu\delta t}{\delta x^2}\left(u_{j+1}^{(n)} - 2u_j^{(n)} + u_{j-1}^{(n)}\right)$ gives: $p^{n+1} e^{ikx_j} = p^n e^{ikx_j} + \frac{\nu\delta t}{\delta x^2} p^n \left(e^{ikx_{j+1}} - 2e^{ikx_j} + e^{ikx_{j-1}}\right)$

We introduce $x_{j-1} = x_j - \delta x$, $x_{j+1} = x_j + \delta x$ and write: $p^{n+1} e^{ikx_j} = p^n \left(e^{ikx_j} + \frac{\nu\delta t}{\delta x^2}\left(e^{ik(x_j+\delta x)} - 2e^{ikx_j} + e^{ik(x_j-\delta x)}\right)\right)$, which leads to $p = 1 + \frac{\nu\delta t}{\delta x^2}\left(e^{ik\delta x} - 2 + e^{-ik\delta x}\right) = 1 + \frac{\nu\delta t}{\delta x^2}\left(\cos(k\delta x) + i\sin(k\delta x) - 2 + \cos(-k\delta x) + i\sin(-k\delta x)\right)$. Using $\sin(-x) = -\sin(x)$, $\cos(-x) = \cos(x)$: $p = 1 + \frac{\nu\delta t}{\delta x^2}(2\cos(k\delta x) - 2)$. For stability, we need to have $|p| \leq 1$, i.e. $-1 \leq 1 + \frac{2\nu\delta t}{\delta x^2}(2\cos(k\delta x) - 2) \leq 1$. $p \leq 1$ is always fulfilled, so we have: $p \geq -1 \Leftrightarrow \frac{\nu\delta t}{\delta x^2}(2\cos(k\delta x) - 2) \geq -2$. In the worst case, $\cos(k\delta x) = -1 \rightarrow -\frac{4\nu\delta t}{\delta x^2} \geq -2 \Leftrightarrow \frac{\nu\delta t}{\delta x^2} \leq \frac{1}{2} \Leftrightarrow \delta t \leq \frac{\delta x^2}{2\nu}$

FFT + PDEs For periodic boundary conditions. Idea: Represent using Fourier Series, form derivatives by multiplication, solve for next time step, take IFT.

Particle Strength Exchange **Smooth Particles Derivation:** From (12) to (15): Let $z = \frac{x-r}{\varepsilon}$, $\frac{dz}{dx} = -\frac{1}{\varepsilon} \rightarrow dz = \frac{-dr}{\varepsilon}$. Substituting z into (12) gives: $\sum_{k=1}^{\infty} \frac{\delta^{(k)}(x)}{k!} \frac{\varepsilon^k}{\varepsilon^k} \int_{+\infty}^{-\infty} -(-z\varepsilon)^k \frac{1}{\varepsilon} \eta(z)\varepsilon dz = \sum_{k=1}^{\infty} \frac{\delta^{(k)}(x)}{k!} \int_{-\infty}^{+\infty} (-1)^k \varepsilon^k z^k \eta(z) dz = \sum_{k=1}^{\infty} \frac{\delta^{(k)}(x)}{k!} (-1)^k \varepsilon^k M_k$ with $M_k = \int_{-\infty}^{\infty} z^k \eta(z) dz$

PSE Derivation: To get (26): Apply (27) to (24): $\frac{\partial f(x_i,t)}{\partial t} = \frac{D}{\varepsilon^2} \sum_{p=1}^{N} [f(x_p,t) - f(x_i,t)] V_p \eta_\varepsilon(x_i - x_p)$. Apply (25): $\frac{d\frac{Q_i}{V_i}}{dt} = \frac{D}{\varepsilon^2} \sum_{p=1}^{N} \left[\frac{Q_p}{V_p} - \frac{Q_i}{V_i}\right] V_p \eta_\varepsilon(x_i - x_p) \rightarrow \frac{dQ_i}{dt} = \frac{D}{\varepsilon^2} \sum_{p=1}^{N} [Q_p V_i - Q_i V_p] \eta_\varepsilon(x_i - x_p)$ (multiplication of both sides with V_i, term on RHS with V_p). **Regular Lattice:** The moments for the given kernel are: $\hat{M}_1 = (-1)\cdot 1 + 0\cdot 2 + 1\cdot 1 = 0$, $\hat{M}_2 = (-1)^2\cdot 1 + 0\cdot 2 + 1^2\cdot 1 = 2$, $\hat{M}_3 = (-1)^3\cdot 1 + 0\cdot 2 + 1^3\cdot 1 = 0$. We get eq. (31) by: Use (26) with (7) and $d=1$: $\frac{dQ_i}{dt} = \frac{D}{\varepsilon^2} \sum_{p=1}^{N} [Q_p V_i - Q_i V_p] \frac{1}{\varepsilon} \eta\left(\frac{x_i - x_p}{\varepsilon}\right)$. We have $V_p = V_i = h$, $\varepsilon = h$, $x_i - x_p = (i-p)h$ and therefore: $\frac{D}{h^2} \sum_{p=1}^{N} [Q_p - Q_i] \eta\left(\frac{(i-p)h}{h}\right) = \frac{D}{h^2} \sum_{p=1}^{N} [Q_p - Q_i] \eta(i-p)$. $\eta(i-p)$ is 0, unless $p = i-1, i, i+1$ i.e.: $\frac{dQ_i}{dt} = \frac{D}{h^2}[(Q_{i-1} - Q_i)\cdot 1 + (Q_i - Q_i)\cdot 2 + (Q_{i+1} - Q_i)\cdot 1] = \frac{D}{h^2}[Q_{i-1} - 2Q_i + Q_{i+1}]$

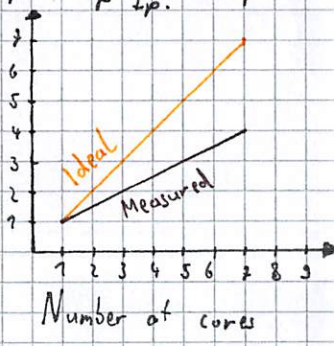
Advantages / Disadvantages Particle Methods Advantages: 1.) Conservative 2.) Can solve some advection problems exactly (e.g. linear advection) 3.) Lower / No numerical diffusion 4.) Easier Refinement Disadvantages: 1.) More complex algorithms for finding neighbor particles 2.) Fewer choices of methods to solve linear systems 3.) Harder to impose boundary conditions

PDE Notation $\nabla = \left(\frac{\partial}{\partial x_1}, \dots, \frac{\partial}{\partial x_n}\right)^T$, $u_{xx} = \frac{\partial^2}{\partial x^2} u(x,\dots)$, $u_{xy} = \frac{\partial^2}{\partial x \partial y} u(x,y,\dots)$, $\Delta = \nabla^T \cdot \nabla = \sum_{i=1}^{d} \frac{\partial^2}{\partial x_i^2}$

Trigonometric Identities $e^{ix} = \cos(x) + i\sin(x)$, $e^{i\pi} + 1 = 0$, $\cos(x) = \frac{e^{ix} + e^{-ix}}{2}$, $\sin(x) = \frac{e^{ix} + e^{-ix}}{2i}$, $\sin(-\theta) = -\sin(\theta)$, $\cos(-\theta) = \cos(\theta)$, $\sin(\theta \pm \frac{\pi}{2}) = \pm\cos(\theta)$, $\cos(\theta \pm \frac{\pi}{2}) = \mp\sin(\theta)$, $\sin(\theta + \pi) = -\sin\theta$, $\cos(\theta + \pi) = -\cos\theta$, $\pi = 180°$, $e^{-ix} = \cos(x) - i\sin(x)$, $\cos^2(x) + \sin^2(x) = 1$

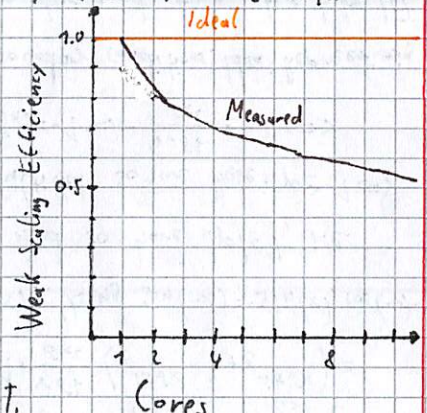
Amdahl's Law: Speedup $S_n = \frac{1}{1 - p + \frac{p}{n}}$, where p = parallel fraction. If $n \rightarrow \infty$, $S_\infty = \frac{1}{1-p}$

Strong Scaling Analysis We keep the problem size fixed and report $S_p = \frac{t_1}{t_p}$. We plot the number of cores / threads (x-axis) against the speedup: Strong scaling efficiency is defined as $\eta_s(p) = \frac{S_p}{p}$. Would be 1 in the ideal case. **Weak Scaling Analysis** We increase the number of nodes and the problem size s.t. the work per node is constant. If amount of work is e.g. linear with problem size (e.g. 1D diffusion with forward FD),



④ multiplying nodes by c requires multiplying problem size by c. If amount of work is quadratic (e.g. naive N body interactions), multiplication by $\sqrt{c}$ is required.

Weak scaling efficiency: $\eta_w(p) = \frac{T(1)}{T(p)}$. We plot the number of cores/threads against the weak scaling efficiency. Roofline Model Operational Intensity: $\frac{\text{FLOPs}}{\text{Byte}}$



Given nominal peak bandwidth B (in GB/s) and nominal peak performance $\bar{\pi}$ (in GFlop/s), the performance of code with operational intensity I is: $P = \min(BI, \bar{\pi})$. The ridge point is $I_b = \frac{\bar{\pi}}{B}$, if $I < I_b$ memory bound and if $I > I_b$ compute bound. Roofline graph: Horizontal line at $\bar{\pi}$ until ridge point, then ~~down~~ line that goes through B at 1/10° (lecture 02, p. 85).

Data Types Single Precision / float: 4 bytes, Double Precision / double: 8 bytes. (unsigned) short int: 2 bytes, (unsigned) int / (unsigned) long int: 4 bytes, (unsigned) long long int = 8 bytes 1D Diffusion: Finite Differences Explicit: $u_i^{n+1} = u_i^n + r\frac{\delta t}{\delta x^2}(u_{i+1}^n + u_{i-1}^n - 2u_i^n)$ Implicit: $u_i^{n+1} = u_i^n + r\frac{\delta t}{\delta x^2}(u_{i+1}^{n+1} + u_{i-1}^{n+1} - 2u_i^{n+1})$ 2D Diffusion: Finite Differences

$$p_{i,j}^{n+1} = p_{i,j}^n + \Delta t D\left(\frac{p_{i,j-1}^n - 2p_{i,j}^n + p_{i,j+1}^n}{\Delta x^2} + \frac{p_{i-1,j}^n - 2p_{i,j}^n + p_{i+1,j}^n}{\Delta y^2}\right)$$ With $\Delta x = \Delta y = \Delta r$ becomes:

$$p_{i,j}^{n+1} = p_{i,j}^n + \frac{\Delta t D}{\Delta r^2}\left(p_{i,j-1}^n - 2p_{i,j}^n + p_{i,j+1}^n + p_{i-1,j}^n - 2p_{i,j}^n + p_{i+1,j}^n\right)$$

⑤ **Programming** **C++** **Arrays:** Static creation: double data[70]; Dynamic: double* data = new double[70]; Initialization: int x[] = {1,2,3}; Iteration: for(int i=0; i<len; i++){ arr[i]} **Vectors:** Required header: #include <vector>, Creation: std::vector<int> vec; or with length n, initialized to 0: std::vector(n,0); Appending: vec.push_back(i); Length: vec.size(); Iterating: for(int i=0; i<vec.size(); i++){ vec[i]}, Deleting by index i: vec.erase(vec.begin()+i); Accessing as Array: int* a = &vec[0]; Copying: When initializing new vector: std::vector<int> vec2(vec); otherwise: vec2 = vec; (assignment copies content), Swapping (constant complexity): vec.swap(vec1); Swapping for arrays: std::swap(a,b); **Emitting Assembly:** g++ -S test.cpp **printf:** Required header: #include <stdio.h>, Syntax: printf("format string", arg1, arg2,...); Format specifiers: %d: Signed int, %u: Unsigned int, %c: Character, %s: String, %p: Pointer address **Makefiles:** E.g.: exec_omp:↵ \t g++ -O3 -std=c++11 -Wall -fopenmp main.cpp -o $@ **Debugging Shared Memory:** g++ -g -fopenmp -fsanitize=thread will report data races. **cout:** Required header: #include <iostream>, Syntax: std::cout << "Output" << std::endl; **Structs:** E.g.: struct Student {char name[20]; int id; int age;}, Creating an instance: Student s; Accessing fields: s.id=4;

SSE: Example (SSE):

```
#include <stdlib.h>
#include <x86intrin.h>
:
float *x, *y;
posix_memalign((void**)&x, 16, N*sizeof(float));
posix_memalign((void**)&y, 16, N*sizeof(float));
const int simd_width = 16/sizeof(float);
for(int i=0; i<N; i += simd_width) {
  const __m128 x4 = _mm_load_ps(x+i);
  const __m128 y4 = _mm_load_ps(y+i);
  _mm_store_ps(y+i, _mm_add_ps(x4, y4));
}
free(x); free(y);
```

AVX with doubles:

```
const __m256d x4 = _mm256_load_pd(x+i);
```

ISPC: Example:

```
export void vector_add(
  uniform float* const uniform x,
  uniform float* const uniform y,
  const uniform int N) {
  for(int i=programIndex; i<N; i+=programCount) {
    y[i] = x[i] + y[i];
  }
```

OR

```
  foreach (i=0 ... N)
    y[i] = x[i] + y[i]
}
```

HW code (cont.) periodic_dist: double dist = x1-x2; if(dist < -.5*DOMAIN_SIZE){ return dist + DOMAIN_SIZE; } if(dist < .5*DOMAIN_SIZE){ return dist; } return dist - DOMAIN_SIZE;

6 OpenMP Compilation: g++ -fopenmp Running: Programs can be run normally, can use OMP_NUM_THREADS to control number of threads. I.e. OMP_NUM_THREADS=4 ./prog for one run, export OMP_NUM_THREADS=4 for persistence. Header: #include <omp.h> Only required for library calls. Parallel Regions: #pragma omp parallel [clause [[,] clause ...] with clause: 1.) if([parallel:] scalar-expression) -> Execute only if true 2.) num_threads(n) -> Number of threads in team 3.) default(shared|none) -> default data-sharing 4.) private(list) -> Uninitialized private copy 5.) firstprivate(list) -> Initialized (to master value) private copy 6.) shared(list) 7.) proc_bind(master|close|spread) -> Binding, see lecture 04, p. 18 Loop Construct: With implicit barrier: #pragma omp for [clause [[,] clause ...] with clause: 1.) private(list) 2.) firstprivate(list) 3.) lastprivate(list) -> Uninitialized private copy, in the end updated according to last lexical section 4.) schedule(static [, chunk] | dynamic [, chunk] | guided [, chunk] | runtime | auto) -> lecture 03, p. 31 5.) collapse(n) -> Collapse n (perfectly nested) levels 6.) reduction(red-id: list) -> red-id $\in$ {+, -, *, &, |, ^, &&, ||, min, max} 7.) nowait Other Pragmas: *= with barrier. #pragma omp ... 1.) master -> Only performed by master thread 2.) single*: By one thread 3.) critical [(name)] -> Critical section 4.) barrier* 5.) atomic -> Replaced by atomic H/W operations 6.) sections*: Each section assigned to (one) different thread, works also with more sections than threads Library Routines: 1.) omp_get_thread_num() -> Get thread ID 2.) omp_get_num_threads() -> Threads in team 3.) omp_set_num_threads(n) -> Set number of threads 4.) omp_get_wtime() -> Per thread wall time (delta=t2-t1;) Environment Variables: 1.) OMP_DYNAMIC="TRUE" or "FALSE" -> Dynamic mode, i.e. adjustment of number of threads 2.) OMP_NESTED="TRUE" or "FALSE" 3.) OMP_PROC_BIND="TRUE" or "FALSE" -> Whether threads are allowed to be moved ("FALSE") between processors.

MPI Compilation: mpic++ main.cpp -o main Running: mpiexec -n 4 main Header: #include <mpi.h> Basic Program: int rank, size; MPI_Init(&argc, &argv); MPI_Comm_rank(MPI_COMM_WORLD, &rank); MPI_Comm_size(MPI_COMM_WORLD, &size); MPI_Finalize(); Blocking MPI Point-to-Point: MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm); / MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status); Tag can be MPI_ANY_TAG, source MPI_ANY_SOURCE and status MPI_STATUS_IGNORE MPI_Datatype: 1.) char: MPI_CHAR 2.) int: MPI_INT 3.) float: MPI_FLOAT 4.) double: MPI_DOUBLE (lecture 08, p. 20) Reduction: MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm); with MPI_OP $\in$ {MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, ...} (lecture 08, p. 34) Non-Blocking Point-to-Point: MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request); / MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request); Operations on request: MPI_Wait(MPI_Request *request, MPI_Status *status); -> Wait for completion, MPI_Test(MPI_Request *request, int *flag, MPI_Status *status); -> flag=1 if completed. MPI I/O: MPI_File_open(MPI_Comm comm, const char *filename, int amode, MPI_Info info, MPI_File *fh); amode -> lecture 10, p. 10, info can be MPI_INFO_NULL. MPI_File_close(MPI_File *fh); for closing. I/O: Sequential Writing: MPI_File_write_ordered(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status); -> Writes are ordered by rank. Hybrid MPI: Initialization: MPI_Init_thread(int *argc, char ***argv, int required, int *provided); with (levels: 1.) MPI_THREAD_SINGLE 2.) MPI_THREAD_FUNNELED (only „master" makes MPI call(s)) 3.) MPI_THREAD_SERIALIZED (no overlapping MPI call(s)) 4.) MPI_THREAD_MULTIPLE. Need to check if provided < requested, abort in this case (lecture 10, p. 35)

⑦ HW Code HW 01 Diffusion Benchmark: Includes: #include <chrono>, Time measurement: for(int i=0;i<Ntimesteps;i++){ const auto t0=std::chrono::steady_clock::now(); sweep(uold, unew, N, c); const auto t1=std::chrono::steady_clock::now(); tsum+=t1-t0; std::swap(uold, unew); } (tsum declared as std::chrono::duration<double> tsum(0)). Performance calculation: const double flop=5; const double Nflop=flop*N*Ntimesteps; const double fperf=Nflop/tsum.count()*1.0e-9; HW 02 Brownian Motion: Local Generators: struct Gen { std::default_random_engine gen; char p[64]; }; std::vector<Gen> gg(omp_get_max_threads()); for(size_t p=0;p<gg.size();p++){ gg[p].gen.seed(p); } Particle initialization: std::vector<double> xx(N); std::uniform_real_distribution<double> dis(-0.5,0.5); for(size_t i;i<N;i++){ xx[i]=dis(gg[0].gen); } M random walk steps: #pragma omp parallel { size_t p=omp_get_thread_num(); auto& g=gg[p].gen; std::normal_distribution dis(0., std::sqrt(dt)); #pragma omp for nowait for(size_t i=0;i<N;i++){ for(size_t m=0;m<M;m++){ xx[i]+=dis(g); } } } Histogram: std::vector<double> hh(nb,0); #pragma omp parallel { std::vector<double> hh(nb,0); #pragma omp for nowait for(size_t i=i;i<xx.size();i++){ int j=(xx[i]-xmin)/(xmax-xmin)*nb; j=std::max(0, std::min(int(nb)-1,j)); hh[j]+=1; } #pragma omp critical(for(size_t j=0;j<hh.size();j++){ hh[j]+=hh[j]; } } HW 03 Covariance: Transpose: for(int d=0;d<D;d++){ for(int n=0;n<N;n++){ data_T[d*N+n]=data[n*D+d]; } } Mean: #pragma omp parallel for for(int d=0;d<D;d++){ mean[d]=0; for(int n=0;n<N;n++){ mean[d]+=data_T[d*N+n]; } mean[d]/=N; } Std: for(int d=0;d<D;d++){ std[d]=0; for(int n=0;n<N;n++){ double temp=data_T[d*N+n]-mean[d]; std[d]+=temp*temp; } std[d]=std::sqrt(std[d]/(N-1)); } Standardize Col-Major: #pragma omp parallel for for(int d=0;d<D;d++){ for(int n=0;n<N;n++){ data_T[d*N+n]=(data_T[d*N+n]-mean[d])/std[d]; } } Construct Covariance: #pragma omp parallel for schedule(dynamic,4) for(int j=0;j<D;j++){ for(int k=0;k<=j;k++){ double cov=0.0; for(int i=0;i<N;i++){ cov+=data_T[j*N+i]*data_T[k*N+i]; } C[j*D+k]=cov/(N-1.0); } } getEigenvectors: #pragma omp parallel for for(int k=0;k<NC;k++){ for(int d=0;d<D;d++){ V[k*D+d]=C[(D-1-k)*D+d]; } } reduceDimensionality: #pragma omp parallel for for(int n=0;n<N;n++){ for(int k=0;k<NC;k++){ double sum=0.0; for(int d=0;d<D;d++){ sum+=V[k*D+d]*data_T[d*N+n]; } data_red[n*NC+k]=sum; } } reconstruct-Dataset Row Major: #pragma omp parallel for for(int n=0;n<N;n++){ for(int d=0;d<D;d++){ double sum=0.0; for(int c=0;c<NC;c++){ sum+=V[c*D+d]*data_red[n*NC+c]; } data_rec[n*D+d]=sum; } } Computing Eigenvectors/Eigenvalues: char jobz='V'; char uplo='U'; int info, lwork; double* W=new double[D]; double* work=new double[2]; lwork=-1; dsyev_(&jobz,&uplo,&D,data_cov,&D,W,work,&lwork,&info); lwork=(int)work[0]; delete[] work; work=new double[lwork]; dsyev_(&jobz,&uplo,&D,data_cov,&D,W,work,&lwork,&info); Compression Ratio: dim_old=N*d; dim_comp=num_comp*(D+N)+2*D; ratio=dim_old/dim_comp; Oja's Rule: forward: for(int k=0;k<batch_size;k++){ for(int i=0;i<nInputs;i++){ for(int o=0;o<nOutputs;o++){ output[o+k*nOutputs]+=weights[o+i*nOutputs]*input[i+k*nInputs]; }}} ojasRuleGradient: for(int k=0;k<batch_size;k++){ for(int i=0;i<nInputs;i++){ for(int o=0;o<nOutputs;o++){ gradient[o+i*nOutputs]+=output[o+k*nOutputs]*(input[i+k*nInputs]-output[o+k*nOutputs]*weights[o+i*nOutputs]); } } } SangersRuleGradient: memset(gradient, 0.0, sizeof(double)*nOutputs*nInputs); for(int k=0;k<batch_size;k++){ for(int i=0;i<nInputs;i++){ for(int o=0;o<nOutputs;o++){ double sum_=0.0; for(int m=0;m<=o;m++){ sum_+=output[m+k*nOutputs]*weights[m+i*nOutputs]; } gradient[o+i*nOutputs]+=output[o+k*nOutputs]*(input[i+k*nInputs]-sum_); } } } compute Eigenvalues: for(int k=0;k<batch_size;k++){ for(int o=0;o<nOutputs;o++){ mean[o]+=output[o+k*nOutputs]; } } for(int o=0;o<nOutputs;o++){ mean[o]/=batch_size; } for(int o=0;o<nOutputs;o++){ double sum=0.0; for(int k=0;k<batch_size;k++){ double temp=output[o+k*nOutputs]-mean[o]; sum+=temp*temp; } eigenvalues[o]=sum/(batch_size-1); } HW 04 2D Diffusion: advance: MPI_Status status[2]; int prev_rank=rank_-1; int next_rank=rank_+1; if(prev_rank<0){ prev_rank=MPI_PROC_NULL; } if(next_rank>=procs_){ next_rank=MPI_PROC_NULL; } MPI_Sendrecv(&rho_[

⑧ real_N_+1], N_, MPI_DOUBLE, prev_rank, 0, &rho_[(local_N_+1)*real_N_+1], N_, MPI_DOUBLE, next_rank, 0, MPI_COMM_WORLD, &status[0]); MPI_Sendrecv(&rho_[local_N_*real_N_+1], N_, MPI_DOUBLE, next_rank, 1, &rho_[1], N_, MPI_DOUBLE, prev_rank, 1, MPI_COMM_WORLD, &status[1]); for(int i=1; i<=local_N_; i++){ for(int j=0; j<=N_; j++){ rho_tmp_[i*real_N_+j]=rho_[i*real_N_+j]+fac_*(rho_[i*real_N_+(j+1)]+rho_[i*real_N_+(j-1)]+rho_[(i+1)*real_N_+j]+rho_[(i-1)*real_N_+j]-4.*rho_[i*real_N_+j]); swap(rho_tmp_, rho_); compute_diagnostics: MPI_Reduce(rank_==0?MPI_IN_PLACE:&heat, &heat, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD); compute_histogram: double (min_rho=min_rho; double (max_rho=max_rho; MPI_Allreduce(&lmin_rho, &min_rho, 1, MPI_DOUBLE, MPI_MIN, MPI_COMM_WORLD); MPI_Allreduce(&lmax_rho, &max_rho, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD); double epsilon=1e-8; double dh=(max_rho-min_rho+epsilon)/M; for(int i=0; i<=local_N_; i++){ for(int j=1; j<=N_; j++){ unsigned int bin=(rho_[i*real_N_+j]-min_rho)/dh; hist[bin]++; } int g_hist[M]={0}; MPI_Reduce(hist, g_hist, M, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD); HW 05 Vortex-Line Roll-up: compute_interaction: Code in PDF,

Async version: for(int pass=0; pass<mpi_size; pass++){ MPI_Request reqs[6]; if(pass+1<mpi_size){ const int rank_to=(mpi_rank+(pass+1))%mpi_size; const int rank_from=(mpi_rank-(pass+1)+mpi_size)%mpi_size; MPI_Isend(particles.pos_x(), n_particles, MPI_VALUE_T, rank_to, 0, MPI_COMM_WORLD, &reqs[0]); MPI_Isend(particles.pos_y(), n_particles, MPI_VALUE_T, rank_to, 1, MPI_COMM_WORLD, &reqs[2]); MPI_Isend(particles.gamma(), n_particles, MPI_VALUE_T, rank_to, 2, MPI_COMM_WORLD, &reqs[4]); MPI_Irecv(exchange.pos_x(), n_particle, MPI_VALUE_T, rank_from, 0, MPI_COMM_WORLD, &reqs[1]); MPI_Irecv(exchange.pos_y(), n_particle, MPI_VALUE_T, rank_from, 1, MPI_COMM_WORLD, &reqs[3]); MPI_Irecv(exchange.gamma(), n_particles, MPI_VALUE_T, rank_from, 2, MPI_COMM_WORLD, &reqs[5]); } compute_interaction(pass==0?particles:sources, particles); if(pass+1<mpi_size){ MPI_Waitall(6, reqs, MPI_STATUSES_IGNORE); std::swap(exchange, sources); } HW 06 ADI:

advance: #pragma omp parallel for for(int iy=1; iy<real_N_-1; iy++){ for(int ix=1; ix<real_N_-1; ix++){ int k=iy*real_N_+ix; int k1=(iy-1)*real_N_+ix; int k2=(iy+1)*real_N_+ix; rhs_[k]=phi_[k]+R_*(phi_[k1]-2.*phi_[k]+phi_[k2]); } #pragma omp parallel for for(int iy=1; iy<real_N_-1; iy++){ thomas(0,iy); } #pragma omp parallel for for(int iy=1; iy<real_N_-1; iy++){ for(int ix=1; ix<real_N_-1; ix++){ int k=iy*real_N_+ix; int k1=iy*real_N_+(ix-1); int k2=iy*real_N_+(ix+1); rhs_[k]=phi_[k]+R_*(phi_[k1]-2.*phi_[k]+phi_[k2]); #pragma omp parallel for for(int ix=1; ix<real_N_-1; ix++){ thomas(1, ix); } thomas(const int dir, const int off){ std::vector<double> L_(N_), U_(N_), z_(N_); int k; k=(0==dir)?(off+real_N_+1):(real_N_+off); L_[0]=a_; U_[0]=b_/a_; z_[0]=rhs_[k]/L_[0]; for(int i=1; i<N_-1; i++){ k=(0==dir)?(off+real_N_+(i+1)):((i+1)*real_N_+off); L_[i]=a_-c_*U_[i-1]; const double Linv=1.0/L_[i]; U_[i]=b_*Linv; z_[i]=(rhs_[k]-c_*z_[i-1])*Linv; } k=(0==dir)?(off+real_N_+N_):((N_)*real_N_+off); L_[N_-1]=a_-c_*U_[N_-2]; phi_[k]=(rhs_[k]-c_*z_[N_-2])/L_[N_-1]; int kp1=k; for(int i=N_-2; i>=0; --i){ k=(0==dir)?(off+real_N_+(i+1)):((i+1)*real_N_+off); phi_[k]=z_[i]-U_[i]*phi_[kp1]; kp1=k; }}

initialize_phi: double bound=0.25*L_; #pragma omp parallel for for(int i=1; i<real_N_-1; i++){ for(int j=1; j<real_N_-1; j++){ int k=i*real_N_+j; if(std::abs((i-1)*dx_-L_/2.)<bound && std::abs((j-1)*dx_-L_/2.)<bound){ phi_[k]=1.; } else { phi_[k]=0.; }}

compute_diagnostics: double heat=0.0; #pragma omp parallel for reduction(+:heat) for(int i=1; i<real_N_-1; i++){ for(int j=1; j<real_N_-1; j++){ heat+=dx_*dx_*phi_[i*real_N_+j]; }} diag_.push_back(Diagnostics(t, heat)); PSE: timestep: for(int i=0; i<N; i++){ double sum=0.0; for(int j=0; j<N; j++){ sum+=(phi[j]-phi[i])*gaussian_eps(periodic_dist(x[i], x[j]), periodic_dist(y[i], y[j])); } dphi[i]=nu*volume/(eps*eps)*sum; } for(int i=0; i<N; i++){ phi[i]+=dt*dphi[i]; } gaussian_eps: constexpr double inv=1.0/eps; return inv*inv*gaussian(inv*x, inv*y); gaussian: double dd=x*x+y*y; return 1/M_PI*std::exp(-dd);