HPC – Algorithms and Applications

Dwarf #5 – Structured Grids

Michael Bader
Winter 2015/2016
Dwarf #5 – Structured Grids

1. dense linear algebra
2. sparse linear algebra
3. spectral methods
4. N-body methods
5. structured grids
6. unstructured grids
7. Monte Carlo
Part I

Modelling on Structured Grids
Motivation: Heat Transfer

- **objective:** compute the temperature distribution of some object
- **under certain prerequisites:**
  - temperature at object boundaries given
  - heat sources
  - material parameters
- **observation from physical experiments:**

\[ q \approx k \cdot \delta T \]

heat flow proportional to temperature differences
A Finite Volume Model

- object: a rectangular metal plate (again)
- model as a collection of small connected rectangular cells

- examine the heat flow across the cell edges
Heat Flow Across the Cell Boundaries

- Heat flow across a given edge is proportional to
  - temperature difference \((T_1 - T_0)\) between the adjacent cells
  - length \(h\) of the edge
- e.g.: heat flow across the left edge:

  \[
  q_{ij}^{(\text{left})} = k_x (T_{ij} - T_{i-1,j}) \, h_y
  \]

- heat flow across all edges determines change of heat energy:

  \[
  q_{ij} = k_x (T_{ij} - T_{i-1,j}) \, h_y + k_x (T_{ij} - T_{i+1,j}) \, h_y \\
  + k_y (T_{ij} - T_{i,j-1}) \, h_x + k_y (T_{ij} - T_{i,j+1}) \, h_x
  \]
Temperature change due to heat flow

- in equilibrium: total heat flow equal to 0
- but: consider additional source term $F_{ij}$ due to
  - external heating
  - radiation
- $F_{ij} = f_{ij} h_x h_y$ ($f_{ij}$ heat flow per area)
- equilibrium with source term requires $q_{ij} + F_{ij} = 0$:

$$f_{ij} h_x h_y = -k_x h_y (2T_{ij} - T_{i-1,j} - T_{i+1,j})$$
$$-k_y h_x (2T_{ij} - T_{i,j-1} - T_{i,j+1})$$
Finite Volume Model

- divide by $h_x h_y$:

$$f_{ij} = -\frac{k_x}{h_x} (2T_{ij} - T_{i-1,j} - T_{i+1,j}) - \frac{k_y}{h_y} (2T_{ij} - T_{i,j-1} - T_{i,j+1})$$

- again, system of linear equations
- how to treat boundaries?
  - prescribe temperature in a cell
    (e.g. boundary layer of cells)
  - prescribe heat flow across an edge;
    for example insulation at left edge: $q_{ij}^{(\text{left})} = 0$
- $h_x, h_y \to 0$, towards partial differential equation:

$$\frac{\partial^2}{\partial x^2} T(x, y) + \frac{\partial^2}{\partial y^2} T(x, y) = f(x, y)$$
Towards a Time Dependent Model

- idea: set up ODE for each cell
- simplification: no external heat sources or drains, i.e. $f_{ij} = 0$
- change of temperature per time is proportional to heat flow into the cell (no longer 0):

$$\dot{T}_{ij}(t) = \frac{\kappa_x}{h_x} \left(2T_{ij}(t) - T_{i-1,j}(t) - T_{i+1,j}(t)\right)$$

$$+ \frac{\kappa_y}{h_y} \left(2T_{ij}(t) - T_{i,j-1}(t) - T_{i,j+1}(t)\right)$$

- solve system of ODE
  → using Euler time stepping, e.g.:

$$T_{ij}^{(n+1)} = T_{ij}^{(n)} + \tau \frac{\kappa_x}{h_x} \left(2T_{ij}^{(n)} - T_{i-1,j}^{(n)} - T_{i+1,j}^{(n)}\right)$$

$$+ \tau \frac{\kappa_y}{h_y} \left(2T_{ij}^{(n)} - T_{i,j-1}^{(n)} - T_{i,j+1}^{(n)}\right)$$
Finite Difference Methods – Wiremesh Model

- compute temperature $T_{ij}$ at nodes $x_{ij}$ of a mesh
- Finite Diff.: approximate derivatives in PDE
- wiremesh model: rectangular plate as fine mesh of wires
Finite Difference Methods

• consider simple PDE (Poisson equation):

\[ \frac{\partial^2}{\partial x^2} T(x, y) + \frac{\partial^2}{\partial y^2} T(x, y) = f(x, y) \]

• approximate derivatives at all grid points \( x_{ij} \):

\[ \left. \frac{\partial^2}{\partial x^2} T(x, y) \right|_{i,j} = \frac{T_{i-1,j} - 2T_{ij} + T_{i+1,j}}{h_x^2} \]

\[ \left. \frac{\partial^2}{\partial y^2} T(x, y) \right|_{i,j} = \frac{T_{i,j-1} - 2T_{ij} + T_{i,j+1}}{h_y^2} \]

• temperature known at (part of) the boundary, e.g. \( T_{0,j} = 0 \)

• task: solve system of linear equations
General Pattern: Stencil Computation

Characterisation of stencil codes:

- update of unknowns, elements, etc., according to a fixed pattern
- pattern usually defined by neighbours in a structured grid/lattice
- task: “update all unknowns/elements” → traversal
- multiple traversals for iterative solvers (in case of systems of equations) or time stepping (in case of time-dependent problems)

Additional example in the tutorials:
shallow water equation on Cartesian grid (Finite Volume Model)
Stencil Notation

- illustrate structure of system of equations or unknown/element-local update as a *discretisation stencil*
- represents one line of the system matrix (in matrix-vector notation)
- matrix elements (in general: update weights) placed according to their corresponding geometrical position
- stencils for the Poisson equation ($h^2$ factors ignored):

$$1D: \begin{bmatrix} 1 & -2 & 1 \end{bmatrix}$$

$$2D: \begin{bmatrix} 1 & -4 & 1 \\ 1 \end{bmatrix}$$
Part II

Structured Grids – Classification and Overview
Structured Grids – Characterisation

- construction of points or elements follows regular process
- geometric (coordinates) and topological information (neighbour relations) can be derived (i.e. are not stored)
- memory addresses can be easily computed
Regular Structured Grids

- **rectangular/cartesian grids:** rectangles (2D) or cuboids (3D)
- **triangular meshes:** triangles (2D) or tetrahedra (3D)
- **often:** row-major or column-major traversal and storage
Transformed Structured Grids

- transformation of the unit square to the computational domain
- regular grid is transformed likewise

Variants:
- **algebraic**: interpolation-based
- **PDE-based**: solve system of PDEs to obtain $\xi(x, y)$ and $\eta(x, y)$
Composite Structured Grids

- subdivide (complicated) domain into subdomains of simpler form
- and use regular meshes on each subdomain
- at interfaces:
  - conforming at interface (“glue” required?)
  - overlapping grids (chimera grids)
Block Structured Grids

Special case of composite grids:

- subdivision into *logically* rectangular subdomains (with logically rectangular local grids)
- subdomains fit together in an unstructured way, but continuity is ensured (coinciding grid points)
- popular in computational fluid dynamics
Adaptive Grids

Characterization of adaptive grids:

- size of grid cells varies considerably
- to locally improve accuracy
- sometimes requirement from numerics

Challenge for structured grids:

- efficient storage/traversal
- retrieve structural information (neighbours, etc.)
Block Adaptive Grids

- retain regular structure
- refinement of entire blocks
- similar to block structured grids

- efficient storage and processing
- but limited w.r.t. adaptivity
Recursively Structured Adaptive Grids

- based on recursive subdivision of parent cell(s)
- leads to tree structures
- quadtree/octree or substructuring of triangles:

- efficient storage; flexible adaptivity
- but complicated processing (recursive algorithms)
Quadtrees and Octree Grids

Recursive construction and corresponding quadtree:
Quadtree and Octree Grids (2)

Example: geometry representation of a car body
Part III

Stencil Codes – Parallelization
Stencil Codes – Parallelization

Finding Concurrency:

- update of all unknowns/elements in parallel?
  → Jacobi iteration: yes
  → Gauss-Seidel iteration: no (at least limited)
  → time stepping: yes (“old” vs. “new” values)

- parallel access to shared data:
  → limited to direct neighbours

Efficiency Considerations:

- low computational intensity:
  → typically $O(2D)$ or $O(3^D)$ operations per stencil update
  → low potential for cache usage
  → challenge for computation/communication ratio

- performance typically \textit{memory-bound}
Stencil Codes – Parallelization (Overview)

Domain Decomposition:
- geometry-oriented decomposition: 1D, 2D, or 3D decomposition?
- “patch” concepts

Communication Patterns:
- communication only to edge-/face-connected neighbours (or all neighbours)?
- ghost cells or non-overlapping domain decomposition?
- multiple ghost cell layers (↔ overlapping domain decomposition)
1D Domain Decomposition – Slice-Oriented
2D Domain Decomposition – Block-Oriented

+ length of domain boundaries (communication volume)
- fit number of processes to layout of boxes
3D Domain Decomposition – Cuboid-Oriented
“Patches” Concept for Domain Decomposition

- more fine-grain load distribution
- “empty patches” allow flexible representation of complicated domains
- overhead for additional, interior boundaries
- requires scheme to assign patches to processes
Ghost Cells

- replicate data from neighbouring partitions
  → requires exchange after each time step or iteration
- multiple layers to reduce communication operations
  or to allow more complicated data access stencils
- overhead can be large for patches concept (esp. in 3D)
Direct-Neighbour vs. “Diagonal” Communication

2-step scheme to exchange data of “diagonal” ghost cells:

- several “hops” replace diagonal communication
- slight increase of volume of communication (bandwidth), but reduces number of messages (at the cost of latency)
- similar in 3D (26 neighbours → 6 neighbours!)
Cartesian Grids in the BSP Model

Compute & Communicate:

- assume partition with \( n^d \) grid points on each process
  \( \Rightarrow c \cdot n^d \) floating-point operations (\( c \) dep. on stencil)
- send and receive ghost layer data on \( 2d \) faces
  \( \Rightarrow \) approx. \( 2d \cdot n^{d-1} \) floating-point operations

“Diagonal” exchange of data:

- each partition receives data from \( 3^d - 1 \) neighbours (incl. “diagonal” neighbours); thus receives \( 3^d - 1 \) messages
- total volume received per partition is equal to size of ghost layer: in total \( 2d \cdot (n + 1)^{d-1} \) cells
  \( \Rightarrow \) Note: BSP only considers total communication volume!
- total cost therefore: \( c \cdot n^d + 2d(n + 1)^{d-1} g + l \)
Cartesian Grids in the BSP Model (2)

Exchange data according to $d$-step scheme:

- each partition receives data from $2d$ neighbours (excl. “diagonal” neighbours)
- organised as $d$ successive steps
  $\Rightarrow$ technically $d$ supersteps in the BSP model
- in each of the $d$ supersteps: communicate $2(n + 2)^{d-1}$ points (incl. “diagonal” ghost cells) and synchronize
- total cost therefore: $c \cdot n^d + d(2(n + 2)^{d-1}g + l)$

Compare “diagonal” and $d$-step scheme:

- “diagonal scheme”: $c \cdot n^d + 2d(n + 1)^{d-1}g + l$
- $d$-step scheme: $c \cdot n^d + 2d(n + 2)^{d-1}g + d \cdot l$
- Note: BSP does not penalise multiple short messages
Ghost Cells for Quadtree Grids

- here: ghost cells only for direct (non-diagonal) neighbours
- more complicated than for Cartesian grids, e.g.:
  → how to identify neighbours in a quadtree?
  → data structure for ghost layer?
Scalability of Structured-Grid Approaches

Typically:

- excellent weak scalability
  - computation time dominates communication time
  - as long as partitions are big enough
    (then: volume $\gg$ boundary, as $n^d \gg n^{d-1}$)
- excellent sequential performance (but memory-bound)
  - simple data structures (arrays, etc.)
  - low memory footprint
  - supports optimisation (vectorisation, etc.)
  - but: low arithmetic intensity ($\Rightarrow$ memory-bound)

Challenges: “science per flop”

- adaptive refinement required?
- complicated domains and domain boundaries?
Part IV

(Cache-)Efficient (Parallel) Algorithms for Structured Grids
Analysis of Cache-Usage for 2D/3D Stencil Computation

We will assume:

- 2D or 3D Cartesian mesh with $N = n^d$ grid points
- stencil only accesses nearest neighbours
  $\rightarrow$ typically $c_M := 2d$ or $c_M := 3d$ accesses per stencil
- $c_F$ floating-point operations per stencil, $c_F \in \mathcal{O}(c_M)$

We will examine:

- number of memory transfers in the Parallel External Memory model (equiv. to cache misses)
- for different implementations and algorithms
- similar for ratio of communication to computation
Straight-Forward, Loop-Based Implementation

Example:

```plaintext
for i from 1 to n do
    for j from 1 to n do {
        x[i, j] = 0.25*(x[i-1,j]+x[i+1,j]+x[i, j-1]+x[i, j+1])
    }
```

Algorithm in the (Parallel) External Memory Model:

- x[i, j-1], x[i, j], and x[i, j+1] shall be stored contiguously in memory
- strategy: keep 3 rows x[i-1,:), x[i,:], and x[i+1,:) in cache ⇒ requires cache size $M > 3n$ floats
- load new row x[i+1,:] before start of j-loop ⇒ $n/L$ transfers
- in total: $n^2/L = N/L$ transfers
  (subject to a minimum cache size $M > 3n$)
Straight-Forward, Loop-Based Implementation (2)

Example:

\[
\text{for } i \text{ from } 1 \text{ to } n \text{ do } \\
\quad \text{for } j \text{ from } 1 \text{ to } n \text{ do } \\
\quad \quad x[i,j] = 0.25 \times (x[i-1,j] + x[i+1,j] + x[i,j-1] + x[i,j+1]) \\
\quad \}
\]

Idealized analysis of cache misses:

- \( x[i, j-1] \) and \( x[i, j] \) have been used in previous \( j \)-iteration: therefore accessed from cache!
- \( x[i, j+1] \) stored contiguously in memory after \( x[i, j] \) \( \Rightarrow \) cache miss every \( L \)-iteration (new cache line starts)
- \( x[i-1,j] \) has been accessed before, but can only be in cache, if cache holds more than one row of \( x \): \( M > n \)
- \( x[i+1,j] \) has not been accessed before
Straight-Forward, Loop-Based Implementation (2)

Example:

```plaintext
for i from 1 to n do
  for j from 1 to n do {
    x[i, j] = 0.25*(x[i-1, j]+x[i+1, j]+x[i, j-1]+x[i, j+1])
  }
```

Idealized analysis of cache misses: (cont.)

- consider case $n > M$: cache misses expected for $x[i-1, j]$ and $x[i+1, j]$
- but: might be in the same cache line as $x[i-1, j-1]$ and $x[i+1, j-1]$ (access in previous j-iteration)
- on average: three cache misses on every $L$-th iteration
- in total: $3N/L = 3n^2/L$ cache misses (in case of small cache size $M < n$)
Loop-Based Implementation with Blocking

Example:

```python
for ii from 1 to n by b do
    for jj from 1 to n by b do
        for i from ii to ii + b - 1 do
            for j from jj to jj + b - 1 do {
                x[i, j] = 0.25 * (x[i - 1, j] + x[i + 1, j] + x[i, j - 1] + x[i, j + 1])
            }
```

Number of cache line transfers:

- choose \( b \) such that the cache can hold 3 rows of \( x \): \( M > 3b \)
- then: \( N/L \) transfers
- again no further influence of cache size \( M \)
  (besides condition for \( b \); i.e., larger caches do not help)
Loop-Based Implementation with Blocking (2)

Parallel External Memory – Algorithm:

\[
\text{for } ii \text{ from } 1 \text{ to } n \text{ by } b \text{ do in parallel}
\]
\[
\text{for } jj \text{ from } 1 \text{ to } n \text{ by } b \text{ do in parallel}
\]
\[
\text{for } i \text{ from } ii \text{ to } ii + b - 1 \text{ do}
\]
\[
\text{for } j \text{ from } jj \text{ to } jj + b - 1 \text{ do } \{
\]
\[
x[i, j] = 0.25 \times (x[i - 1, j] + x[i + 1, j] + x[i, j - 1] + x[i, j + 1])
\]
\[
\}
\]

Execution of cache line transfers:

- assume \( \frac{n}{b} \times \frac{n}{b} \) processors
- each processor executes i- and j-loop sequentially
- then: each processor reads \((b + 2) \times (b + 2)\) array elements (incl. “ghost layer”)
- more exact formula: \( (\frac{n}{b})^2 \times (b + 2)^2 / L \) cache line transfers
Extension to 3D stencils

Simple loops:
- if cache holds 3 planes of $x$, $M > 3n^2$, then $N/L$ transfers
- if cache holds less than 1 plane, $M < n^2$, then $3N/L$ transfers
- if cache holds less than 1 row, $M < n$, then $5N/L$ transfers (if $c_M = 6$) or $9N/L$ transfers (if $c_M = 3^3 = 27$)

With blocking:
- cache needs to hold 3 planes of a $b^3$ block: $M > 3b^2$
- then: $N/L$ transfers; again no further dependence on cache size $M$ (besides condition for $b$)
Further Increase of Cache Reuse

Requires multiple stencil evaluations:

\[
\text{\textbf{for}} \ t \ \text{\textbf{from}} \ 1 \ \text{\textbf{to}} \ m \ \text{\textbf{do}}
\]
\[
\text{\textbf{for}} \ i \ \text{\textbf{from}} \ 1 \ \text{\textbf{to}} \ n \ \text{\textbf{do}}
\]
\[
\text{\textbf{for}} \ j \ \text{\textbf{from}} \ 1 \ \text{\textbf{to}} \ n \ \text{\textbf{do}} \ 
\{
\text{x}[i, j] = 0.25 \times (x[i-1, j] + x[i+1, j] + x[i, j-1] + x[i, j+1])
\}
\]

→ for multiple iterations or time steps, e.g.

Possible approaches:

- blocking in space and time?
- what about precedence conditions of stencil updates?
Region of Influence for Stencil Updates

1D Example:

- area of “valid” points narrows by stencil size in each step
- leads to trapezoidal update regions
- similar, but more complicated, in 2D and 3D
Example for Implementation in 1D: Time Skewing

- choose extend of trapezoids in time and space: \( S \times T \)
- then: \( S + T \) grid points need to fit into cache
- goal: reduce number of cache line transfers by factor \( T - 1 \)
- ideal case: 
  \[
  \frac{N}{S} \times \frac{S}{B} \times \frac{1}{T - 1} = O\left(\frac{N}{BT}\right)
  \]
  transfers per iteration
Divide & Conquer Algorithm: Space Split

1D Example:

- applied, if spatial domain is at least “twice as large” as number of time steps
- note precedence condition for left vs. right subdomain
Divide & Conquer Algorithm: Time Split

1D Example:

- applied, if spatial domain is less than “twice as large” as number of time steps
- space split likely as the next split for the lower domain
Cache Oblivious Algorithms for Structured Grids

Algorithm by Frigo & Strumpen:

- divide & conquer approach alternates between time and space splits
- time split, if region is narrow (less than two grid points per time step)
- space split, if region is wide (otherwise)

Number of cache line transfers:

- in 1D: $O(M)$ grid points need to fit into cache
- in 2D: $O(\sqrt{M}) \times O(\sqrt{M})$ grid points need to fit into cache
- roughly half of $O(\sqrt{M})$ time steps executed in cache
- thus: $O(N/d\sqrt{M})$ cache misses in “cache oblivious” model ("Parallel External Memory" with only 1 CPU and “ideal cache")
References/Literature:

• Matteo Frigo and Volker Strumpen: *Cache Oblivious Stencil Computations*, Int. Conf. on Supercomput., ACM, 2005.


→ check links on lecture website!