Parallel Numerics
Remarks for Parallel Algorithms and Implementation

Uwe Küster
kuester@hlrs.de

University of Stuttgart
High-Performance Computing Center Stuttgart (HLRS)
www.hlrs.de

overview

- parallel programming models
- algorithms
- parallelization by domain decomposition
- algorithms for large equation systems
- (implementation)
- (hardware properties)
different aspects of parallelism

- algorithmic view
- software view
- hardware view
- performance view

parallelism: hardware view

- pipelining in a pipeline of concatenated pipelined units (load -> add -> store)
- different units execute different instructions (load, store, multiply-add, integer add, branch, ...)
- parallel threads (pthread, OpenMP,...) on shared memory systems
- parallel threads on ccNUMA systems
- HPF on shared memory and distributed memory systems
- parallel message passing on distributed memory systems

- future model: dynamic generation and destruction of threads
parallelism: software view

- vectorization and software pipelining
  (automatic by Hardware and compiler, F90 array syntax)
- parallel execution of outer or splitted loops (pthread, OpenMP)
- parallel execution of independent program parts
  (pthread, OpenMP)
- parallel execution in distributed arrays
  (HPF, automatic by compiler)
- parallel execution of different domains
  (OpenMP, Message Passing, MLP, one sided message passing, active messages)

parallelism: performance view

- vectorization and software pipelining
  - factor of 1 ->50 due to programming techniques
  - caching, arrays, no calls
- parallel execution of outer or splitted loops
  - for small number of processors ( < 16 )
  - penalties by frequent synchronization at loop ends
- parallel execution of independent program parts
  - limited number
- parallel execution of different domains
  - message passing: large domains because of latency overhead
  - OpenMP: avoid cacheline sharing, suppress cache coherency mechanism
parallelism: algorithmic view

- parallel operations on independent sets of data (SIMD)
- functional decomposition (MPMD)
- recursively generated subtasks, tightly connected threads
- domain decomposition of calculation areas (SPMD)

former Flynn notation: SISD, SIMD, (MISD), MIMD

parallel operations on independent sets of data:

- parallel jobs on independent data (cluster, internet)
  - most simple and most effective approach
  - very loosely coupling needed
  - Great Internet Mersenne Prime Search (GIMPS)
    - $2^{13466917} - 1$ 39-th prime number
    - 205000 computers
    - http://www.mersenne.org
- subprograms on independent data
  - spawn processes and collect the data
functional decomposition: example

Atmospheric Model

Hydrology Model

Land Surface Model

Ocean Model

http://www.cs.reading.ac.uk/dbpp/text

recursive task generation, tightly connected threads:

- needs short task creation times
- for shared memory machines
- applicable to CRAY (TERA) MTA
- today not applicable to other machines
- if nested parallelism in OpenMP
- multithreaded processors:
  - Intel Pentium 5+
  - Intel-Compaq-DEC EV8
  - Intel-Madison+
- tightly connected processors:
  - IBM Power 4
domain decomposition of calculation areas:

- neighbourhoods with a small numbers of neighbours form the base of domain decomposition
neighbourhoods: Finite Differences

- approximation of differential operators by differences

<table>
<thead>
<tr>
<th>j-1,k</th>
<th>(j,k)</th>
<th>(j+1,k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(j,k-1)</td>
<td>(j,k)</td>
<td>(j+1,k)</td>
</tr>
<tr>
<td>(j,k+1)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

calculation points on boundaries

neighbourhoods: Matrix for Regular Grid with 5-Points Stencil

<table>
<thead>
<tr>
<th>k=1</th>
<th>k=2</th>
<th>k=3</th>
<th>k=4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j inner block index</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
neighbourhoods: Finite Volumes

- flux balance over neighbouring sides

\[
\begin{array}{ccccccc}
  & 0 & 0 & 0 & 0 & 0 & \\
 0 & 0 & 0 & 0 & 0 & 0 & \\
 0 & 0 & 0 & 0 & 0 & 0 & \\
 0 & 0 & 0 & 0 & 0 & 0 & \\
 0 & 0 & 0 & 0 & 0 & 0 & \\
\end{array}
\]

boundary on cell sides

neighbourhoods: Triangular Grid with Relation Matrix

position of matrix elements

\[
\begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
1 & 0 & 0 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\
2 & 0 & 0 & 0 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\
3 & 0 & 0 & 0 & 0 & 3 & 3 & 3 & 3 & 3 & 3 \\
4 & 0 & 0 & 0 & 0 & 0 & 3 & 3 & 3 & 3 & 3 \\
5 & 0 & 0 & 0 & 0 & 0 & 0 & 3 & 3 & 3 & 3 \\
6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 & 3 & 3 \\
7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 & 3 \\
8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\
9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\
10 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\
11 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\
\end{array}
\]
neighbourhoods: Finite Elements

- Global matrix defined by local matrices
- Global matrices are sums of the element matrices:
  \[ M^{e}_{jk} = \int_{e} \phi_j \phi_k d\lambda \]
  \[ G^{e}_{jk} = \int_{e} \langle \nabla \phi_j, \nabla \phi_k \rangle d\lambda \]
- Local test functions result in sparse global matrices

domain decomposition

- The natural approach for distribution of work
- The distribution of connected subgrids (subgraphs) to different processors
domain decomposition: regular partitioning of regular grid

http://www.cs.reading.ac.uk/dbpp/text/node19.html

domain decomposition: irregular partitioning of regular grid

http://www.cs.reading.ac.uk/dbpp/text/node19.html
domain decomposition: recursive Spectral Bisection

unstructured self adaptive grids: CEQ 1
unstructured self adaptive grids: CEQ 2

unstructured for self adaptive grids: CEQ 3
unstructured self adaptive grids: CEQ 4

domain decomposition: unstructured self adaptive grids CEQ

- Finite volume mechanism:
  - each cell is defined by cell_to_face relation and momentums
  - each face is defined by face_to_node relation and momentums
  - each cell contains additionally states in a separate data structure
  - each edge contains additionally states in a separate data structure

- states have to be communicated at each time step
- relations and momentums have to be communicated at each refinement and recoarsening step
domain decomposition: the difficult part of implementation

- the grids/graphs are not independent
- communication of overlapping boundaries has to ensure consistent behaviour of algorithmic steps
- the implementation has to be correct and efficient
- possible on all kinds of interconnected computers
- simpler on shared memory machines

communications may have complicated pattern for special nodes
domain decomposition: data exchange including the corner

- inner
- boundary
- corner

communication in 2 serial steps (2D)

domain decomposition: data exchange including the corners / edges

- useful if corner points are part of the discretization stencil and if latencies are large
- reduces number of messages
  - for 2D regular patches 4 messages per patch instead of 4+4 messages per patch
  - but 2 serial message groups
- important for 3D problems
  - 6 messages per patch instead of 6+12+8 messages
  - but 3 serial message groups
domain decomposition: working with extended halo

exchange

+++

domain decomposition: general properties

- additional operations by overlapping grids
- small patches may fit into caches
- numerical and communication problems for (semi-) implicit procedures
- enumeration problem:
  - mapping numbers/pointers of the neighbouring grids
  - global numbering would be simpler, but no support on pure distributed memory machines
- essential approach for parallelization
- may help in cache reuse for larger caches
domain decomposition: communicated surface for cubus

2D case
- grid size: \( m \times m = n \)
- surface size: \( 4m = 4n^{1/2} \)
- relation surface/volume: \( 4 / n^{1/2} \)

3D case
- grid size: \( m \times m \times m = n \)
- surface size: \( 6m \times m = 6n^{2/3} \)
- relation surface/volume: \( 6 / n^{1/3} \)

communication/calculation ratio worse for 3D!

domain decomposition: load balancing

- load for all processes should be the same
- time defect of only one processor affects hundreds of other processors
- different techniques of load balancing
- problem of load shifting for dynamically generated load
Recursive Spectral Bisection as example

- Even number of nodes
- Edge relation of a graph
- Graph Laplacian given by

\[ Q = \left( q_{vw} \right)_{v,w \in \text{nodes}} \]

\[ q_{vw} = \begin{cases} 
-1 & (v, w) \in E \\
\text{degree}(v) & v = w \\
0 & \text{otherwise}
\end{cases} \]

- Calculate median

\[ \text{median} = \frac{1}{\text{nodes}} \sum_{v \in \text{nodes}} x_v \]

- Define sets by

\[ p^*_v = \begin{cases} 
1 & x_v > \text{median} \\
0 & \text{otherwise}
\end{cases} \]

\[ p^-_v = \begin{cases} 
1 & x_v < \text{median} \\
0 & \text{otherwise}
\end{cases} \]

- If \( p^* + p^- = 1 \), then \( p^* \) and \( p^- \) are connected and best in some sense

---

Large systems

Solution of Large Systems
large systems

- the discrete solution of partial differential equations results in the problem of solving a large (non)linear system (dimension of the system: $10^4 - 10^9$)
- nonlinear systems are linearized:
  replace
  \[ L(q) = f \]
  by
  \[ \frac{\partial L}{\partial q} \Delta q = f - L(q) \]
  \[ q_{\text{new}} = q + \Delta q \]

large systems: two different approaches

- explicit procedures like
  \[ q^{n+1} - q^n = \Delta t A q^n \]
  only need a single operator evaluation (or a few for Runge Kutta)
- implicit problems
  \[ q^{n+1} - \Delta t A q^{n+1} = q^n \]
  require the solution of a large linear system
large systems: main directions of linear solvers

- direct solvers for 'small' matrices
- simple iterative procedures
- Krylov space procedures
- Multi level procedures $O(n)$, $O(n \log n)$, $O(n (\log n)^2)$

direct solvers

- good public domain software for dense matrices:
  - direct solvers for dense matrices and eigenvalue solvers for dense matrices in
    - Lapack-3,
    - Scalapack
- direct solvers for some types of sparse matrices (SPD)
  - MUMPS
simple iterative procedures: Gauss-Seidel

\[ M\Delta q = -(Aq - f) \quad \leftrightarrow \]
\[ q_{\text{new}} = q + \omega \Delta q \]

Gauss-Seidel Iteration for \( M=L+D \)

do element in all_elements
\[ \delta = \text{operation\_of}(\text{element}\%\text{value}, \text{neighbourhood\_elements}\%\text{value}) \]
\[ \text{element}\%\text{value} = \text{element}\%\text{value} + \omega \delta \]
enddo

in general non parallelizable recursion

parallelizable by coloring

simple iterative procedures: Jakobi

\[ M\Delta q = -(Aq - f) \quad \leftrightarrow \]
\[ q_{\text{new}} = q + \omega \Delta q \]

Jakobi Iteration for \( M=D \)

do element in all_elements
\[ \delta(\text{element}) = \text{operation\_of}(\text{element}\%\text{value}, \text{neighbourhood\_elements}\%\text{value}) \]
enddo

do element in all_elements
\[ \text{element}\%\text{value} = \text{element}\%\text{value} + \omega \delta(\text{element}) \]
enddo

all operations can be done in parallel
Krylov space algorithms

- CG
- Lanczos
- BiCO
- CGS
- BiCGSTAB(l)
- TFQMR
- ORTHOMIN
- GMRES
- GMRESR

all have the same building blocks

Conjugate Gradient Squared (CGS)

**start values**

\[ r_0 = L^{-1}(A\mathbf{x}_0 - \mathbf{b}) \leftrightarrow \]

\[ r_0 = \mathbf{b} \leftrightarrow \]

\[ q_0 = 0 \]

\[ p_0 = 0 \]

\[ \rho_0 = 1 \]

**Iteration**

\[ \alpha_k = -\frac{\rho_k}{\rho_{k-1}} \]

\[ q_{k+1} = \mathbf{u}_k + \alpha_k \mathbf{v}_k \]

\[ w_k = \alpha_k \mathbf{U}^{-1}(\mathbf{u}_k + \mathbf{q}_{k+1}) \leftrightarrow \]

\[ r_{k+1} = r_k + L^{-1}A w_k \leftrightarrow \]

\[ x_{k+1} = x_k + w_k \]

**endo**

\[ \mathbf{v}_k = L^T A U^{-1} p_k \leftrightarrow \]

\[ \sigma_k = \langle \mathbf{r}_k, \mathbf{v}_k \rangle \leftrightarrow \]

\[ \text{data exchange between processors} \]
**BICGSTAB(2)**

**start values**
\[ x_0 \text{ is an initial guess; } r_0 = b - Ax_0; \]
\[ r_0 \text{ is an arbitrary vector, such that } (r_0, r_0) \neq 0, \]
\[ e.g., r_0 = r; \]
\[ \rho_0 = 1; u_0 = 0; \alpha_0 = 0; \omega_0 = 1; \]

**even Bi-CG step**
\[
\text{for } i = 0, 2, 4, 6, \ldots \\
\rho_i = -\omega_i \rho_{i-1} \\
\beta = \frac{r_i^T r_i}{\omega_{i-1}} \\
u = r_i - \beta u_{i-1} \\
v = A v \\
\gamma = (v, r_i) \\
\alpha = \sigma_0 / \gamma; \\
r = r_i - \alpha v; \\
s = A r \\
x = x + \alpha u; \\
\]

**odd Bi-CG step**
\[
\beta = \frac{r_i^T r_i}{\omega_{i-1}} \\
v = s - \beta v; \\
w = A v \\
\gamma = (w, r_i) \\
\alpha = \sigma_0 / \gamma; \\
r = r_i - \alpha w; \\
s = A r \\
x = x + \alpha u; \\
\]

**GCR(2) - part**
\[
a_0 = (r, r); \mu = (s, s); v = (s, r); r = (s, s) \\
a_0 = (r, r); \\
r = r - \beta v; \\
\omega = (s, s); \mu = (s, s); \\
x_{i+1} = x + \alpha r + \omega_0 s + \alpha u \\
r_{i+1} = r - \alpha x - \omega_0 f; \\
\]

**data exchange between processors**

---

**operations of Krylov space algorithms**

- some scalar operations
- scalar product of vectors
  \[ \text{a} = \text{b} + \text{alpha*c} \]
  (daxpy)
- vector = matrix * vector
  (time critical)
- preconditioning
operations of Krylov space algorithms

- all these operations are vectorizable and parallelizable
- all operations allow domain decomposition
- arrays as datastructure
- work can be done in loops
- all essential loops are large
- limited cache reuse
- matrix X vector multiply may be formulated in general way
  - if possible, use your specific formulation

Intel Xeon 2GHz, performance of CG specific operations

\[ a(i) = b(i) + c(i) \]
\[ a(i) = b(i) + c(i) \cdot d(index(i)) \]
\[ \alpha = \text{dot product}(b(:), c(:)) \]
\[ a(i) = b(i) + \alpha \cdot c(i) \]
preconditioning

- Krylov space are fast for matrices with small condition
- preconditioning decreases condition number

\[ Ax = b \quad \rightarrow \quad (L^{-1}AU^{-1})UX = L^{-1}b \]

- preconditioning is a problem for vectorization and parallelization
- Jakobi (diagonal) preconditioning simple
- Block Jakobi (diagonal blocks) may be efficient
ILU preconditioning

- ILU is a LU decomposition with no or limited fill in with respect to the original sparse matrix. Neglected elements may be added to the diagonal.
- ILU preconditioning is highly recursive as well as for the decomposition step as well as for the calculation step
  - applicable to microprocessors
  - not applicable to vector computers
- ILU shows the same parallelization problems as the sparse matrix vector multiplication. Additional synchronization points are necessary as well for the decomposing step as for the preconditioning step
- ILU for the blocks of a domain decomposition

MILU for general matrix

```markdown
\[
\text{do} \quad i = 1, n-1 \\
\text{do} \quad j = i+1, n \\
\text{if} \quad ((j, l) \in S) \text{ then} \\
\quad a_{jl} = a_{jl}/a_{ll} \\
\quad a_{jl} = a_{jl} \\
\text{do} \quad k = i+1, n \\
\text{if} \quad ((i, k) \in S) \text{ then} \\
\quad a_{ik} = a_{ik} - a_{il}a_{lk} \\
\text{else} \\
\quad a_{ij} = a_{ij} - aa*a_{lk} \\
\text{endif} \\
\text{endif} \\
\text{enddo} \\
\text{endif} \\
\text{enddo}
\]
```
graph coloring: defining independent sets for parallelization

- helps preconditioning
- histogram loop
- assembling stiffness matrices
- assembling forces on nodes
- distribution of states on the neighbourhood
- (collection is no problem)

graph coloring: my neighbour has not my color
graph coloring:

- vectorization or parallelization for all elements in a single color
- a color may be a complete patch in a decomposed domain
- only a small number of colors for large independent sets
- finding the smallest number of colors is NP-complete!
  some approximation sufficient
- calculation of colors is expensive
- pays out only if used several times

Multigrid 1

to be solved \( L_f(x_f) = f_f \)

presmoothing \( x_f := S_f^0(x_f) \)

defect \( d_f := -(L_f(x_f) - f_f) \)

restriction \( r_g := I_{f \rightarrow g}(d_f) \)

coarse grid correction \( L_g(x_g + \varepsilon \Delta x_g) = L_g(x_g) + \varepsilon r_g \)

prolongation \( x_f := x_f + I_{g \rightarrow f}(\Delta x_g) \)

postsmoothing \( x_f := S_f^\infty(x_f) \)

solve coarse grid correction with the next coarser grid
Multigrid 2

- smoothing and postsmoothing by a usual relaxation technique
- restriction and prolongation should be a problem dependent interpolation
- coarse grid correction do be done by the same mechanism
- coarse grids should reside on the same processor as the fine grids
- communication necessary in all steps
- very coarse grids should be calculate on only one processor
- distribution of load difficult
- poor cache reuse but fast mixing of information

Hierarchy of triangles

some triangles have only 2 children!
Concluding remarks

- programs with high performance today are a combination of pipelined (vectorized) code and parallel execution with a domain decomposition technique
- decrease the bandwidth needs of the code!
- try to increase the size of the patches and to decrease their surface
- load balancing and load distributing can be difficult
- use arrays and loops for the computing intensive parts
- there is a trade off between 'good' programming style and processor performance
- flexibility and dynamic features decrease performance in the time dominant parts

Books and URLs 1

Andreas Meister:
Numerik linearer Gleichungssysteme - Eine Einführung in moderne Verfahren.
Vieweg 1999.

Ian Foster
Designing and Building Parallel Programs
Addison-Wesley, 1995
ISBN 0-201-57594-9
http://www.cs.reading.ac.uk/dbpp/text
Books and URLs 2

JOSTLE
http://www.gre.ac.uk/~jjg01/

DRAMA

Patrick Amestoy, Iain Duff, Jean Yves L’Excellent, and Petr Plecháč.
PARASOL An integrated programming environment for parallel sparse matrix solvers.
http://www.genias.de/projects/parasol/

end