Remarks for parallel algorithms and implementation

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

overview

algorithms
parallelization (domain decomposition)
programming models
implementation
hardware properties
performance
Partial Differential Equations

- Poisson equation
- Heat exchange and Diffusion equation
- Elasticity equations
- Euler equations
- Shall water equations
- Navier Stokes equations (one or more phases)
- Maxwell equations
- Schrödinger equations

Solution Methods for Partial Differential Equations

- Finite Difference
- Finite Volume
- Finite Element
- Wavelets
- Fourier Transform
- Particle Methods
- Monte Carlo Simulation
- ...
different aspects of parallelism

algorithmic view
software view
hardware view
performance view

Parallelism in 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 in 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 in 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 in algorithmic view

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

Parallel operations on independent sets of data 1

- parallel jobs on independent data (cluster, internet)
- subprograms on independent data
Vectorization: multiple gather

subroutine multiple_gather_loop(a,b,ind1,ind2,n)
  real a(n),b(n)
  integer ind1(n),ind2(n)
  j1=0
  j2=0
  do 10 i=1,n
    if(a(i)*b(i).gt.0.) then
      j1=j1+1
      ind1(j1)=i
    elseif(a(i).lt.0.) then
      j2=j2+1
      ind2(j2)=i
    endif
  10    continue
  return
end

Vectorization: scattering data

subroutine scatter_loop(a,b,index,n)
  real a(n),b(n)
  integer index(n)
  *** vectorizable even if index not unique
  do 10 i=1,n
    a(index(i))=b(i)
  10    continue
  return
end

vectorizable even if index is not one to one
data parallel only if index is one to one
F90: array syntax

```fortran
integer, parameter :: max=10
real, dimension(max) :: a, b

a(1:max)=(/real(i), i=1,max)/ / real(max-1)
b=2.*a
b=sin(a)
b(1:max:2)=a(1:max:2)
```

F90: index vectors

```fortran
integer, dimension(n) :: population, index
integer, dimension(maxval(index)) :: pp
integer, dimension(n) :: permutation, inverse_permutation

population=pp(index)

! calculation of the inverse of a permutation
inverse_permutation(permutation) = (/ ( i , i = 1 , size(permutation) ) /)
```
**F95: forall ... end forall different from do ... enddo**

- no recursion!
- Jakobi, not Gauss-Seidel

\[
\begin{align*}
\text{forall} & \ (i = 2:n-1, j = 2:n-1) \\
& \quad d(i, j) = 0.25 \times (c(i, j + 1) + c(i, j - 1) + c(i + 1, j) + c(i - 1, j)) - c(i, j) \\
& \quad c(i, j) = c(i, j) + \varepsilon d(i, j) \\
& \text{end forall}
\end{align*}
\]

**parallel operations on independent sets of data 2**

- loops over arrays of independent data

\[
\begin{align*}
& \text{!$OMP PARALLEL PRIVATE(k), SHARED(a,b,c)} \\
& \text{do } k=1,k_{\text{max}} \\
& \quad a(k) = b(k) + c(k) \\
& \text{enddo}
\end{align*}
\]

needs shared memory

may be effective
recursive task generation 1

recursive function dotpro(a,b) result(value)
end=size(a)
  if(end == 1) then
    value=a(1)*b(1)
  else
    half=end/2
    left=dotpro(a(1:half),b(1:half)) ! new task
    right=dotpro(a(half+1,end),b(half+1,end)) ! new task
    value=left+right
  endif
end function dotpro

recursive task generation 2

needs short task creation times
for shared memory machines
applicable to CRAY (TERA) MTA
today not applicable to other machines
nested parallelism in OpenMP
functional decomposition 1

Atmospheric Model

Hydrology Model

Land Surface Model

Ocean Model

http://www.cs.reading.ac.uk/dbpphest

functional decomposition 2

natural problem decomposition
how to define the interfaces
(GRISSLY)
load balancing difficult
convergence problems for stiff problems
multiscale problems
suitable for metacomputing
GRID project
neighbourhoods

neighbourhoods with small numbers of neighbours

Finite Differences
(Approximation of Differential Operators by Differences)

calculation points on boundaries
Finite Differences 2

\[
(\Delta \phi)_{jk} = \frac{1}{\Delta x^2} \left( \Phi_{j-1k} - 2\Phi_{jk} + \Phi_{j+1k} \right) + \frac{1}{\Delta y^2} \left( \Phi_{jk-1} - 2\Phi_{jk} + \Phi_{jk+1} \right)
\]

discrete Laplace operator on an equidistant rectangular grid
can be represented as a product of a matrix with the vector of all states
the matrix depends on the numeration of the state vector
lexicographic ordering
(1,1), (2,1), ...(jmax,1), (1,2), (2,2), ... (jmax,2), ... (jmax, kmax)
the matrix can but need not appear in the program
different formulas at the boundary
also for unstructured grids

Matrix for regular grid with 5-points stencil

\begin{align*}
\begin{array}{cccc}
& k=1 & k=2 & k=3 & k=4 \\
\hline
j & & & & \\
\hline
1 & & & & \\
2 & & & & \\
3 & & & & \\
4 & & & & \\
\end{array}
\end{align*}

\text{j inner block index}
**Finite Volumes (Flux Balance over neighbouring sides)**

![Finite Volumes Diagram]

- Boundary on cell sides

---

**finite volume and finite differences**

Gauss theorem

\[
(\Delta \Phi)_j = \frac{1}{\text{Vol}_j} \int \text{div} \Phi \, d\lambda = \frac{1}{\text{Vol}_j} \oint \text{grad} \Phi \, df
\]

- Right face of a cell

\[
\int \text{grad} \Phi, df = \Delta y \frac{\Phi_{i+1,j} - \Phi_{i,j}}{\Delta x}
\]

- Complete integral

\[
(\Delta \Phi)_j = \frac{1}{\Delta x \Delta y} \left( \left\{ \Delta y \frac{\Phi_{j+1,i} - \Phi_{j,i}}{\Delta y} \right\} \Phi_{j+1,i+1} - \left\{ \Delta y \frac{\Phi_{j+1,i} - \Phi_{j,i}}{\Delta y} \right\} \Phi_{j+1,i-1} \right)
\]

It is identical to the finite difference operator for rectangular grids.
**Finite Volume**

finite volume discretization for
conservative partial differential equations for all kinds of cells
calculates weak solutions (shocks, slip faces, phase boundaries)
not suitable for all partial differential equations
reconstruction of the face integral by
the inner values (cell centered)
or
the node values (node centered)

---

**Triangular Grid with Relation Matrix**

position of matrix elements

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>Σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>
Finite Elements (local matrices form a global matrix)

Global matrices are sums of the element matrices:

\[
M'_{jk} = \int_{\epsilon} \varphi_j \varphi_k d\lambda
\]

\[
G^e_{jk} = \int_{\epsilon} \langle \nabla \varphi, \nabla \varphi \rangle d\lambda
\]

Local test functions result in sparse global matrices.

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: overlapping grids and communication

communications may have complicated pattern
domain decomposition: properties

- additional operations by overlapping grids
- small patches may fit into caches
- numerical problems for (semi-) implicit procedures
- enumeration problem:
  - map the numbers 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: \( 6 / n^{2/3} \)

communication/calculation ratio worse for 3D!
regular grid partitioning of regular grid

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

domain decomposition: irregular grid partitioning of regular grid

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

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

- *nodes* even number of nodes
- *E* edge relation of a graph
- Graph Laplacian is given by

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

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

determine Eigenvector (Fiedler vector) \( x \) for the first Eigenvalue \( \lambda > 0 \)

calculate

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

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

---

Recursive Spectral Bisection

CHACO:

METIS:
http://www-users.cs.umn.edu/~karypis/metis/

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

ZOLTAN:

CHACO:

METIS:
http://www-users.cs.umn.edu/~karypis/metis/

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

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: $10^4 - 10^9$)

the nonlinear systems are linearized:
replace
$L(q) = f$

by
$\partial L/\partial q \Delta q = f - L(q)$
$q_{\text{new}} = q + \Delta q$

two different approaches

explicit procedures like
$q^{n+1} - q^n = \Delta t Aq^n$

only need a single operator evaluation (several steps for Runge Kutta)
implicit problems
$q^{n+1} - \Delta t Aq^{n+1} = q^n$
require the solution of a large linear system
**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
    - Lapack-3,
    - Scalapack
  - direct solvers for some types of sparse matrices
simple iterative procedures: Gauss-Seidel

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

Gauss-Seidel Iteration for \( M=L+D \)
do element in all_elements
\[ \text{delta} = \text{operation}_\text{of}(\text{element}\%\text{value}, \text{neighbourhood}\_\text{elements}\%\text{value}) \]
\[ \text{element}\%\text{value} = \text{element}\%\text{value} + \omega \text{delta} \]
enddo
in general non parallelizable recursion

---

simple iterative procedures: Jacobi

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

Jacobi Iteration for \( M=D \)
do element in all_elements
\[ \text{delta}(\text{element}) = \text{operation}_\text{of}(\text{element}\%\text{value}, \text{neighbourhood}\_\text{elements}\%\text{value}) \]
enddo
do element in all_elements
\[ \text{element}\%\text{value} = \text{element}\%\text{value} + \omega \text{delta}(\text{element}) \]
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

Iteration

\( r_0 = L^{-1}A x_0 - b \)
\( z_0 = ? \)
\( q_0 = 0 \)
\( p_0 = 0 \)
\( \rho_0 = 1 \)

\( \alpha_k = - \rho_k / \sigma_k \)
\( q_{k+1} = u_k + \alpha_k v_k \)
\( w_k = \alpha_k U^{-1}(u_k + q_{k+1}) \)
\( r_{k+1} = r_k + L^{-1}A w_k \)
\( x_{k+1} = x_k + w_k \)

\( \beta_k = \rho_k / \rho_{k-1} \)
\( u_k = r_k + \beta_k q_k \)
\( p_k = u_k + \beta_k (u_k + \beta_k p_{k-1}) \)
\( v_k = L^{-1}A U^{-1} p_k \)
\( \sigma_k = \langle r_k , v_k \rangle \)
BICGSTAB(2)

---

**start values**

\[ x_0 \text{ is an initial guess; } \tau_0 = b - Ax_0; \]

\[ \tau_v \text{ is an arbitrary vector such that } (r, \tau_v) \neq 0, \]
\[ \text{e.g., } \tau_0 = r; \]
\[ \rho_0 = 1; \]
\[ u = 0; \]
\[ \alpha = 0; \]
\[ \omega_0 = 1; \]

---

**even Bi-CG step**

\[ \rho_i = (\tilde{r}_i, r_i); \quad \beta = \frac{\alpha}{\rho_i}; \quad \rho_i = \rho_i \]
\[ u = \tau_i - \beta u; \]
\[ v = Au; \]
\[ r = r_i - \alpha v; \]
\[ s = Av; \]
\[ x = x_i + \alpha u; \]

---

**odd Bi-CG step**

\[ \rho_i = (\tilde{r}_i, r_i); \quad \beta = \frac{\alpha}{\rho_i}; \quad \rho_i = \rho_i \]
\[ v = s - \beta v; \]
\[ w = Av; \]
\[ \gamma = (w, \tilde{r}_i); \quad \alpha = \frac{\rho_i}{\gamma}; \]
\[ u = r - \beta u; \]
\[ r = r_i - \alpha v; \]
\[ s = s - \alpha w; \]
\[ x = x_i + \alpha u; \]

---

**GCR(2) - part**

\[ a_0 = (s, s); \quad \mu = (s, s); \quad v = (s, r_i); \quad r = (r_i, r_i) \]
\[ a_0 = (r, r_i); \quad w = r_i - v_i; \]
\[ a_0 = (a_0 - \mu a_0) / \mu; \]
\[ x_{i+1} = x + a_0 r_i + a_0 s_i + a_0 w \]

if \( x_{i+1} \) accurate enough then quit
\[ u = u - a_0 v - a_0 w \]
**end**

---

**operations of Krylov space algorithms**

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

- all these operations are vectorizable and parallelizable
- all operations allow domain decomposition
- arrays as data structure
- 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

preconditioning

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

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

- preconditioning is a problem for vectorization and parallelization
- Jakobi (diagonal) preconditioning simple
- Block Jakobi may be efficient
- 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 for the blocks of a domain decomposition
**MILU for general matrix**

\[
\begin{align*}
&\text{do } l = 1, n-1 \\
&\quad \text{do } j = l+1, n \\
&\quad \quad \text{if } ((j, l) \in S) \text{ then} \\
&\quad \quad \quad a_{a} = a_{j:l}/a_{l:l} \\
&\quad \quad \quad a_{a} = a_{j:l} = a_{a} \\
&\quad \quad \quad \text{do } k = l+1, n \\
&\quad \quad \quad \quad \text{if } ((l, k) \in S) \text{ then} \\
&\quad \quad \quad \quad \quad \text{if } ((j, k) \in S) \text{ then} \\
&\quad \quad \quad \quad \quad \quad a_{n} = a_{n:k} - a_{a}a_{l:k} \\
&\quad \quad \quad \quad \quad \text{else} \\
&\quad \quad \quad \quad \quad \quad a_{n} = a_{n} - a_{a}a_{l:k} \\
&\quad \quad \quad \quad \text{endif} \\
&\quad \quad \quad \text{endif} \\
&\quad \quad \text{enddo} \\
&\quad \text{endif} \\
&\text{enddo} \\
&\text{enddo}
\end{align*}
\]

**overlapping domain decomposition**

[Diagram showing overlapping domain decomposition]
For overlapping domain decomposition (DD) there are dependencies across a boundary. With nonoverlapping DD the boundary points are separated and calculated in a different phase.
nonoverlapping DD: several 2D domains

\[ I_1 \quad I_2 \quad I_3 \quad I_n \]

independent edges

\( \circ \) cross points; coarse grid points

Equation to be analysed:

\[-\partial_x a \partial_y \phi - \partial_y b \partial_y \phi = F \]

+ boundary conditions

FEM or Finite Difference Discretization
nonoverlapping DD: separation of inner and boundary nodes

the numbers of the interior of all domains come first
then the node numbers of the boundaries

\[
\begin{bmatrix}
A_{II} & A_{IB} \\
A_{BI} & A_{BB}
\end{bmatrix}
\begin{bmatrix}
\phi_I \\
\phi_B
\end{bmatrix}
= 
\begin{bmatrix}
f_I \\
f_B
\end{bmatrix}
\]
nonoverlapping DD: Schur complement

is the equivalent equation on the B-points
I points are eliminated

\[ S = A_{BB} - A_{BI} A^{-1}_I A_{IB} \]
\[ S \phi_B = f_B - A_{BI} A^{-1}_I f_I \]
detailed

\[ S = A_{BB} - \sum_{i=1}^{n} A_{BI_i} A^{-1}_{I_i} A_{I_i B} \quad \text{parallelizable} \]
\[ S \phi_B = f_B - \sum_{i=1}^{n} A_{BI_i} A^{-1}_{I_i} f_{I_i} \quad \text{to be solved} \]

nonoverlapping DD:

The Schur complement equation is much smaller
and much denser than the original equation.
The calculations of the operator and the solution of the equation
are parallelizable
But this procedure is not scalable because the number of B points increases
if the number of domains is increased
nonoverlapping DD: dividing boundary nodes

\[ B \text{ is the union of the edges and the domain cross points} \]

\[ B = V \cup \bigcup E_k \]

the edges appear as separated independent operators; only the cross points \( V \) are connecting the edges this will be reflected in the construction of the preconditioning matrix

nonoverlapping DD: Preconditioner 1

\[ R_k \text{ injection operators for the edge } k \]

\[ S_k = R_k S R_k^T \text{ restriction of the Schur complement to the edge} \]

\[ R_0 \text{ some specific mapping to the coarse grid} \]

\[ S_V = R_0 S R_0^T \text{ restriction of the Schur complement to the coarse grid} \]
nonoverlapping DD: Preconditioner 2

\[ M = \sum_{E_k} R_k^T S^{-1} R_k + R_0^T S^{-1} R_0 \quad \text{local and global part} \]

Preconditioner for the solution of

\[ S \phi = f - A \phi^{-1} f \]

The equation can be solved by a preconditioned CG procedure

\[ \kappa(\text{MS}) = O(1 + (\log \frac{H}{h})^2) \quad \text{estimation of number of iterations} \]

h is the size of the elements; H is the size of the domains

nonoverlapping DD: reference

Carvalho, Giraud, Le Tallec:
Algebraic two-level preconditioner for the Schur complement method
coloring: defining independent sets

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

coloring of a graph: any neighbour has different color
vectorization or parallelization in a single color
a color may be a complete patch in a decomposed domain
only a small number of colors
finding the smallest number of colors is \textit{np}-complete
calculation of colors is expensive
pays out only if used several times

\textbf{Multigrid 1}

\begin{align*}
\text{to be solved} & \quad L_f(x_f) = f_f \\
\text{presmoothing} & \quad x_f := S_f^\nu(x_f) \\
\text{defect} & \quad d_f := -(L_f(x_f) - f_f) \\
\text{restriction} & \quad r_g := I_{f \rightarrow g}(d_f) \\
\text{coarse grid correction} & \quad L_g(x_g + \varepsilon \Delta x_g) = L_g(x_g) + \varepsilon r_g \\
\text{prolongation} & \quad x_f := x_f + I_{g \rightarrow f}(\Delta x_g) \\
\text{postsmoothing} & \quad x_f := S_f^\nu(x_f)
\end{align*}

solve coarse grid correction with the next coarser grid
Multigrid 2

- smoothing by a usual relaxation technique
- restriction is a problem dependent interpolation
- coarse grid correction do be done by the same mechanism
- prolongation is a problem dependent interpolation
- postsmoothing by the usual relaxation technique
- coarse grids should reside on the same processor as the fine grids
- very coarse grids should be calculate on only one processor
- distribution of load difficult
- poor cache reuse

Hierarchy of triangles

- some triangles have only 2 children!
High Performance for Fast Methods

numerical efficiency has to be complemented by hardware efficiency

the efficiency of modern processors is in the 1% - 90 % range

how to avoid performance loss?
parameters of performance limits

floating point computing capability = frequency * floating point pipes

bandwidth = transfer path width * frequency
   (limit by number of open transactions)

latency = time to get data after releasing the fetch/load command

latency is the hardest problem

bandwidth

is a severe problem
• all modern systems suffer from small bandwidth
• reuse cached data very intensively (but how!)
• blocking code
• use of dense matrix algebra (fast even for small problem sizes!)
• use of all data in a cacheline
but bandwidth is ‘only’ a technical problem
(DDR, RAMBUS, bidirectional use of lines)
latency

important for the interconnect of distributed systems
more important for the processor-memory system
reasons:
• (signal speed: 30m/100 nsec)
• cache and memory distance
• acknowledgement times
• operation system overhead
• program overhead

overcoming latency

block transfer reduces latency per unit (cache lines)
early prefetching enables data access just in time
hardware prefetch for strided data
software prefetch by the compiler
Latency from processor to various locations

<table>
<thead>
<tr>
<th>Location</th>
<th>Seconds</th>
<th>Clocks</th>
<th>Future Development</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock rate</td>
<td>0.5-4 nsec</td>
<td></td>
<td>decreasing</td>
</tr>
<tr>
<td>Register</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1 cache</td>
<td>1-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2 cache</td>
<td>4-20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L3 cache</td>
<td>10-100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td>60-1300 nsec</td>
<td>60-1300</td>
<td>slowly decreasing</td>
</tr>
<tr>
<td>Remote memory</td>
<td>500-1400</td>
<td></td>
<td>slowly decreasing</td>
</tr>
<tr>
<td>SHMEM</td>
<td>~3 μsec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI</td>
<td>5-20 μsec</td>
<td></td>
<td>slowly decreasing</td>
</tr>
<tr>
<td>Disk</td>
<td>~8 msec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tape</td>
<td>~30 sec</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dependency of bandwidth and latency

- $T_{block}$
- $T_{path}$
- $W_{width}$

Single block transfer!
### Dependency of Bandwidth and Latency 2

\[
\begin{align*}
T_{\text{block}} &= \frac{\text{Blocksize}}{\text{Width} \times \text{velo} \times \text{freq}} \\
T_{\text{complete}} &= 2 \times T_{\text{path}} + T_{\text{block}} \quad \text{(Acknowledgement!)} \\
\text{Latency} &= 2 \times T_{\text{path}} \\
\text{Bandwidth}(\text{Blocksize}) &= \frac{\text{Blocksize}}{T_{\text{complete}}} \\
&= \frac{\text{Blocksize}}{(\text{Latency} + \text{Blocksize}/(\text{Width} \times \text{velo} \times \text{freq}))} \\
\text{BW}_{\text{max}} &= \text{Width} \times \text{velo} \times \text{freq} \quad (\text{Blocksize} \rightarrow \infty) \\
\text{Bandwidth}(\text{Blocksize}) &= \frac{\text{BW}_{\text{max}}}{(\text{BW}_{\text{max}} \times \text{Latency}/\text{Blocksize} + 1)} \\
\text{Hyperbola!}
\end{align*}
\]

### Bandwidth-Performance Relation for Multi-Block Grid with Cubes

- number of cells in node: \( n_{\text{cell}} \)
- operations per cell: \( op \)
- latency: \( lat \)
- number of variables per exchanged cell: \( v_{\text{each}} \)
- performance: \( \text{perf} \)
- bandwidth: \( BW \)

Exchange time less 1/10 of computing time:

\[
\frac{op \times n_{\text{cell}}}{\text{perf}} \geq 10 \left( \frac{v_{\text{each}} \times 6 \times n_{\text{cell}}^{2/3} \times 8B}{BW} + 6 \times lat \right)
\]
Example multiblock technique (optimal case)

- number of cells in node: 1 Mill
- operations per cell: 2000
- latency: 10 μsec
- number of variables per exchanged cell: 5
- effective performance: 10 GFLOPs
- effective bandwidth: 1GB/sec

\[
\frac{op \times n_{cell}}{perf} \geq 10 \times \left( \frac{v_{exch} \times 6 \times n_{cell}^{2/3} \times 8B}{BW} + 6 \times lat \right)
\]

0.2 sec > 10*(0.0024 sec + 0.000006 sec)

BW is important

distributions in quadratic patches

10*4=40 connections
distribution in line patches

7*8=56 connections

High Performance Systems

**System:**
- 512 nodes
- 8192 CPUs
- 10.9 TFLOP/s
- 6.2 MW
- 85 Mill $ 

**Node:**
- 16 CPUs
- 32 GB Memory
- 21.3 GFLOP/s
- 16 GB/sec Memory b/w

**CPU:**
- 1.3 GFLOP/s @ 333 MHz
- Power 5
- 1 Proc per Chip
- 1.6 GB/s Memory b/w

**Earth-Simulator Project: end 2001**

**System:**
- 640 nodes
- 41 TFLOP/s
- 10.2 TB
- 10 MW
- 7.8 Bill Yen = 56 Mill $

**Node:**
- 8 CPUs
- 16 GB Shared-Memory
- 64 GFLOP/s
- 32 MMU
- 1 RCU (to ext. Xbar b/w: 2*16 GB/s, special access pattern)

**CPU:**
- 8 GFLOP/s @ 500 MHz
- Vector (8 pipe) Single-Chip
- 32 GB/s Memory b/w
- 1 load/store
programming techniques

to do and to avoid

getting processor performance by loops with arrays

nearly all work has to be done in

- explicit loops
  do n=1,nmax
  a(n)=??
  enddo
- no (hidden) call
- no small objects
- no IO
- many computations
- few data accesses
- data access by arrays
- few decisions
- enable compiler prefetching

appropriate for all types of modern processors!
**flexible but expensive: linked list 1**

```fortran
module linked_list

  type edge_pointer_type
    real(kind=real_kind) :: var1
    real(kind=real_kind) :: var2
    type(edge_pointer_type), pointer :: edge_1
    type(edge_pointer_type), pointer :: edge_2
  end type edge_pointer_type

  edge => edge%edge_1
  do
    edge%var1 = edge%edge_1%var2 + edge%edge_2%var2
    list_length = list_length + 1
    if (.not. associated(edge)) exit
  enddo
end module
```

**flexible but expensive: linked list 2**

- the processor cannot foresee the data to load in the near future
- prefetch is not possible
- the loop end has to be calculated by analysing data
- not by comparing to a fixed register value
- replace by explicit loops with indirect addressing

```fortran
module linked_list

  do
    edge%var1 = edge%edge_1%var2 + edge%edge_2%var2
    list_length = list_length + 1
    if (.not. associated(edge)) exit
  enddo
end module
```
nice but expensive: recursive calls

recursive calls have nice functionality for a lot of applications:
W cycle in multigrid instead of V cycle
simple handling of any kind of trees
• quadtrees, octrees for refinement
• linked lists
• hierarchy of triangles

recursive calls 2

type element_type
  real,dimension(10) :: data
  integer :: max_child
  type(element_type),pointer,dimension:: child ! allows an arbitrary
  ! number of children
end type element_type

recursive subroutine sub(element)
type(element_type) :: element
integer :: nn,no
do nn=1,size(element%data)
  element%data(nn)=real(nn)
enddo
do no=1,element%max_child
call sub(element%child(no))
enddo
end subroutine sub
modifying a hierarchical grid

running over or changing all variables of an hierarchical grid can be done by

• recursion over the different level of parentship (elegant)

or

• nested loops (fast)

inhbiting processor performance

• frequent use of dynamic data
• frequent copy of data
• data aliasing inhibiting data dependence analysis
  pointer usage
  usage of indirection on the left side of assignments
• large number of calls
• large number of small objects in object oriented techniques
• frequent calls of virtual functions
• small granular recursions
• linked lists of small objects
• large number of divides
good for performance

- arrays as data structure
- explicit loops
- explicit constant bounds
- the length of short loops must be visible to the compiler
- vectorizable loops (also for modern microprocessors)
- non short loops (size depends on the cache size)
- much computation with small amount of data
- small number of divides (frequently part of intrinsics!)
- neighbourhoods by sparse matrices
- cache reuse
- prefetching capabilities

Cache reuse for heat transfer: different implementations on SGI R10000

[mflops vs. \log_2(\text{loop_length}) plot]

normal
blocked
index
object oriented programming

hides complexity from the programmer
reusable code by data abstraction

but
many (invisible) procedure calls
compiler optimization is theoretically possible
high compilation times

compromise
use large objects with array based methods!

formulation of flexible neighbourhoods
discrete operators and sparse matrices

need to solve a large linear system
differential operators are replaced by locally operating discrete operators
these operators are represented by sparse matrices
(nearly all matrix elements are vanishing)

handling sparse matrices

avoid storage of zero elements
key data structure in using Krylov space algorithms
fast under special circumstances
formulate flexible neighbourhoods
may replace recursive data structures
### Sparse Matrix Example

<table>
<thead>
<tr>
<th>current_number</th>
<th>element</th>
<th>indices</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>11 8 7</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>4 5</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>5 16 8</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>5 7 2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>8 12</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>11 16 9</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
<td>4 12 7</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>4 12</td>
</tr>
<tr>
<td>8</td>
<td>12</td>
<td>2 16 9</td>
</tr>
</tbody>
</table>

Matrix values are omitted, real array analogous to integer indices array.

### Sparse Matrices: Row Ordered Example

<table>
<thead>
<tr>
<th>row_number</th>
<th>begin</th>
<th>index</th>
<th>end</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>0 11 17 8 2 8 0 2</td>
<td>1 4</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>3 4 5</td>
<td>1 7</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>2 5 16 2 8</td>
<td>3 10</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>3 8 11 13 7 10 2</td>
<td>4 12</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>11 8 12 5 4</td>
<td>5 15</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>11 14 16 9</td>
<td>6 18</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
<td>16 14 12 18 7</td>
<td>7 20</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>19 4 20 12</td>
<td>8 23</td>
</tr>
<tr>
<td>8</td>
<td>12</td>
<td>21 2 16 23 9</td>
<td></td>
</tr>
</tbody>
</table>

Index arrays beginning with 0.
**row ordered and jagged diagonal matrices: data structure**

```fortran
type sparse_matrix_type
  logical :: fleeting ! .true. if function result
  integer :: type_of_relation !
  integer :: number_of_rows ! dim of row_number
  integer :: number_of_indices ! dim of index, values
  integer :: maximal_neighbourhood ! max number cols
  integer,pointer,dimension(:) :: begin ! offset of row, pseudo col
  integer,pointer,dimension(:) :: length ! length row, pseudo col
  integer,pointer,dimension(:) :: index ! index array
  integer,pointer,dimension(:) :: row_number ! row_number
  integer,pointer,dimension(:) :: value ! matrix values
end type sparse_matrix_type

value can also be an array of fixed sized matrices!
```

**row ordered matrix: matrix vector multiplication**

```fortran
subroutine matrix_mult_vector_row_ord(matrix,vector,result_vector)
  type(sparse_matrix_type) :: matrix
  type(vector_type) :: vector, result_vector
  real(kind=real_kind),dimension(matrix%number_of_rows) :: temp
  integer :: cn, no, pseudo_col, index
  temp=0.
  do cn=1,matrix%number_of_rows
    pseudo_col=matrix%begin(cn)
    do no=1,matrix%length(cn)
      index=matrix%index(no + pseudo_col)
      temp(cn)=temp(cn)+matrix%value(no + pseudo_col)*vector%value(index)
    enddo
  enddo
  result_vector%value(matrix%row_number)=temp
end subroutine matrix_mult_vector_row_ord
```
**sparse matrices: jagged diagonal example**

<table>
<thead>
<tr>
<th>row_number</th>
<th>begin</th>
<th>index</th>
<th>end</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

Index arrays beginning with 0

---

**jagged diagonal matrix: matrix vector multiplication**

```
subroutine matrix_mult_vector_jaggdiag(matrix,vector,result_vector)
type(sparse_matrix_type) :: matrix
type(vector_type) :: vector, result_vector
real(kind=real_kind),dimension(matrix%number_of_rows) :: temp
integer :: cn, no, pseudo_col, index

temp=0.
do no=1,matrix%maximal_neighbourhood
   pseudo_col=matrix%begin(no)
do cn=1,matrix%length(no)
   index=matrix%index(cn+ pseudo_col)
   temp(cn)=temp(cn)+matrix%value(cn+ pseudo_col)*vector%value(index)
endo
dendo

result_vector%value(matrix%row_number)=temp
end subroutine matrix_mult_vector_jaggdiag
```
row ordered and jagged diagonal matrix formulation: performance

• row ordered
  short loops
  better on cache machines
  simpler to use
  redistribution not necessary in many cases

• jagged diagonal
  long loops
  better on vector machines
  redistribution necessary

• combination of both
  very flexible
  sophisticated

  good performance by using block sparse matrices

Conclusion

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


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/~jjg91/


end