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
- implemention
- 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

```
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

```
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

forall( i = 2:n-1, j = 2:n-1 )
  d(i, j) =  0.25*(c(i, j + 1) + c(i, j - 1) + c(i + 1, j) + c(i - 1, j)) - c(i,j)
  c(i, j) = c(i, j) + eps*d(i,j)
end forall
```


parallel operations on independent sets of data 2

loops over arrays of independent data

```
!$OMP PARALLEL PRIVATE(k), SHARED(a,b,c)
do k=1,kmax
   a(k)=b(k)+c(k)
enddo
```

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

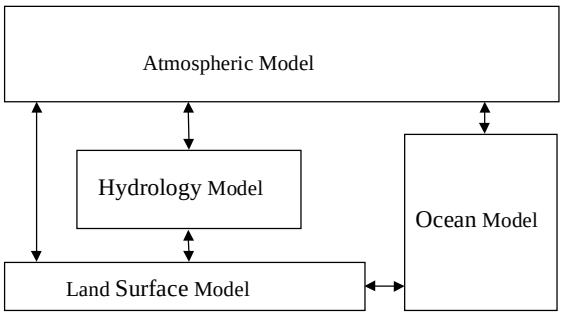
for shared memory machines

applicable to CRAY (TERA) MTA

today not applicable to other machines

nested parallelism in OpenMP

functional decomposition 1



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

functional decomposition 2

natural problem decomposition

how to define the interfaces (GRISSLY)

load balancing difficult

convergence problems for stiff problems

multiscale problems

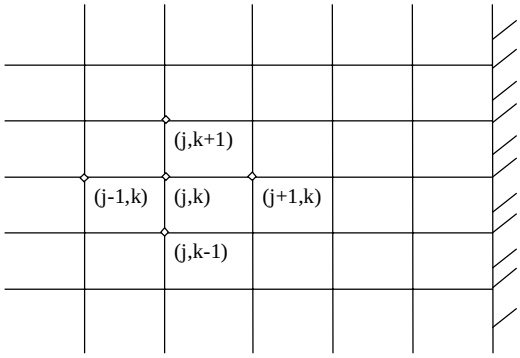
suitable for metacomputing

GRID project

neighbourhoods

neighbourhoods with small numbers of neighbours

Finite Differences
(Approximation of Differential Operators by Differences)



calculation points
on boundaries

Finite Differences 2

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

diskrete 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

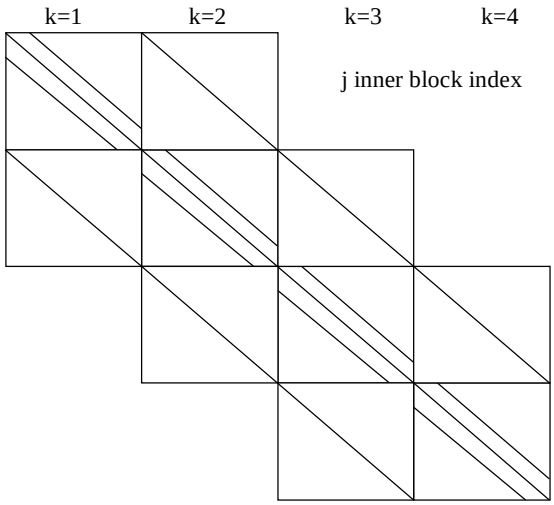
(1,1),(2,1),...,(jmax,1),(1,2),(2,2),...(jmax,2),.....,(jmax,kmax)

the matrix can but need not appear in the program

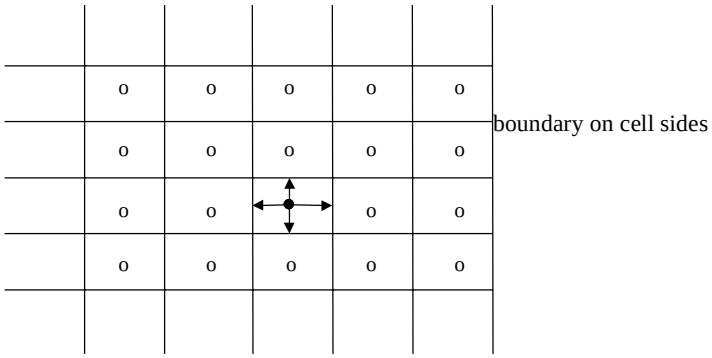
different formulas at the boundary

also for unstructured grids

Matrix for regular grid with 5-points stencil



Finite Volumes (Flux Balance over neighbouring sides)



finite volume and finite differences

Gauss theorem

$$(\Delta\Phi)_{jk} \approx \frac{1}{Vol_{jk}} \int_{Vol_{jk}} div\ grad\ \Phi\ d\lambda = \frac{1}{Vol_{jk}} \oint_{\partial Vol_{jk}} \langle grad\ \Phi, df \rangle$$

right face of a cell

$$\int \langle grad\ \Phi, df \rangle = \Delta y \frac{\Phi_{j+1k} - \Phi_{jk}}{\Delta x}$$

complete integral

$$(\Delta\Phi)_{jk} \approx \frac{1}{\Delta x \Delta y}\left(\left(\Delta y \frac{\Phi_{j+1k} - \Phi_{jk}}{\Delta x} - \Delta y \frac{\Phi_{jk} - \Phi_{j-1k}}{\Delta x}\right) + \left(\Delta x \frac{\Phi_{jk+1} - \Phi_{jk}}{\Delta y} - \Delta x \frac{\Phi_{jk} - \Phi_{jk-1}}{\Delta y}\right)\right)$$

is identical to 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

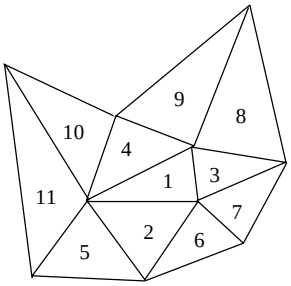
reconstruction of the face integral by

the inner values (cell centered)

or

the node values (node centered)

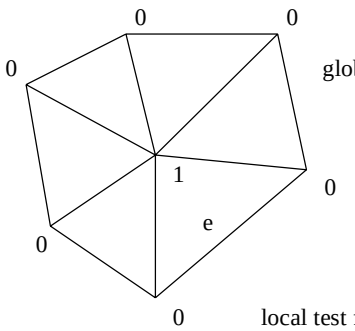
Triangular Grid with Relation Matrix



position of matrix elements

	1	2	3	4	5	6	7	8	9	10	11	Σ
1	●	●	●	●								4
2	●	●			●	●						4
3	●		●				●	●				4
4	●			●					●	●		4
5		●			●						●	3
6		●				●	●					3
7			●			●	●					3
8			●					●	●			3
9				●				●	●			3
10				●						●	●	3

Finite Elements (local matrices form a global matrix)



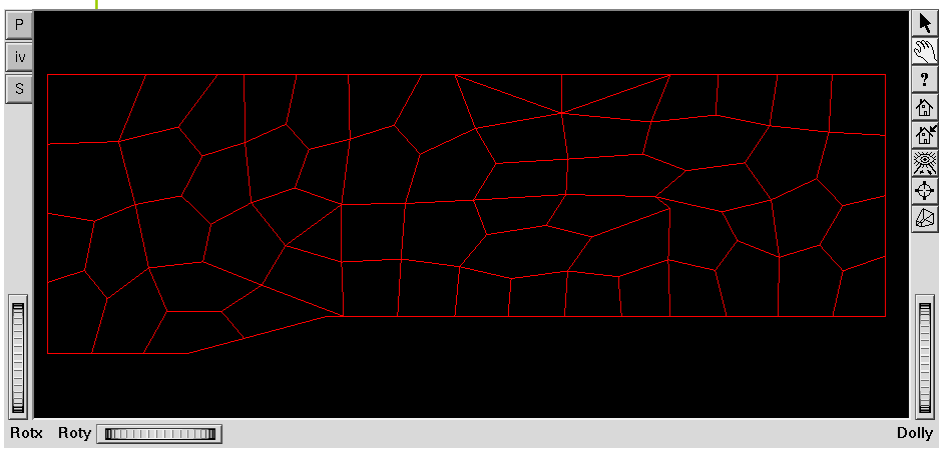
global matrices are sums of the element matrices:

$$M^{e}_{jk} = \int_{e} \varphi_j \varphi_k \, d\lambda$$

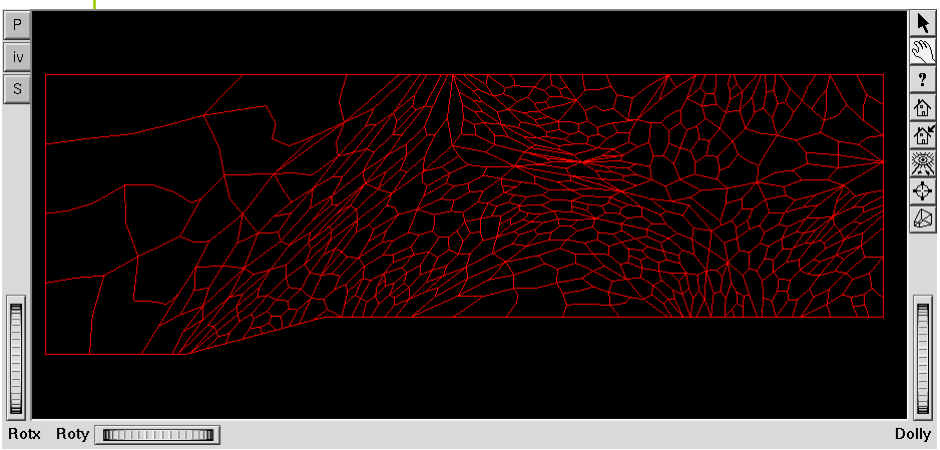
$$G^{e}_{jk} = \int_{e} \langle grad\,\varphi_j, grad\,\varphi_k \rangle \, d\lambda$$

local test functions result in sparse global matrices

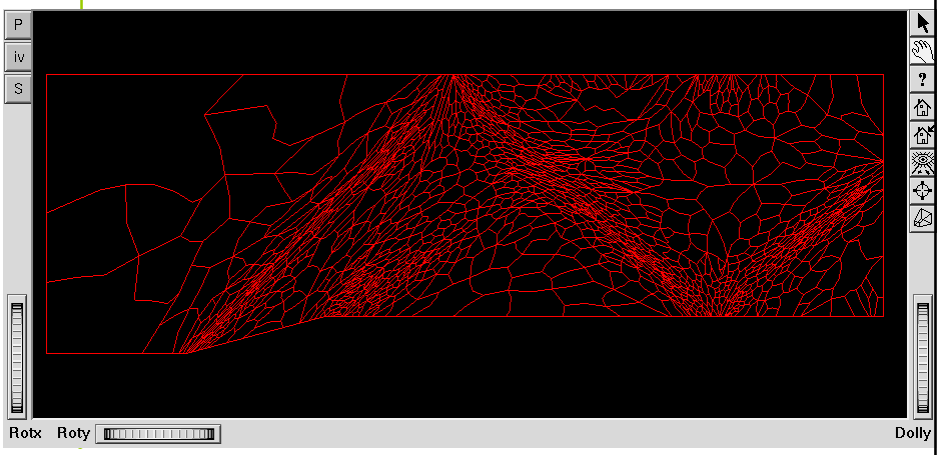
unstructured self adaptive grids: CEQ 1



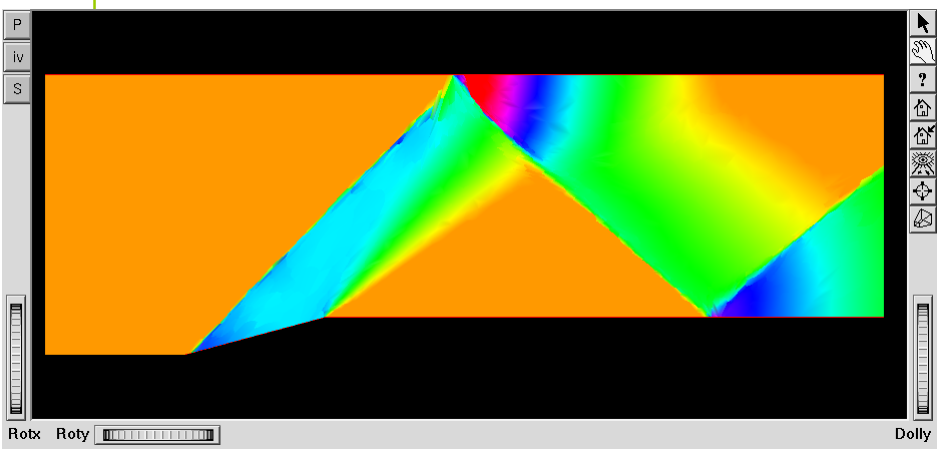
unstructured self adaptive grids: CEQ 2



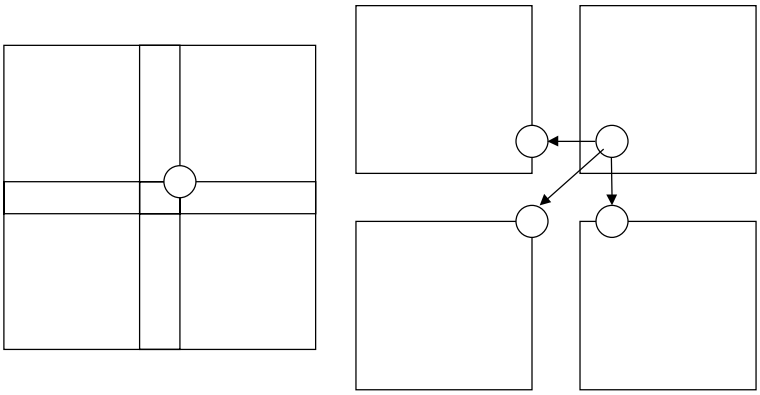
unstructured for self adaptive grids: CEQ 3



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: $m * m = n$

surface size: $4m = 4n^{1/2}$

relation surface/volume: $4 / n^{1/2}$

3D case

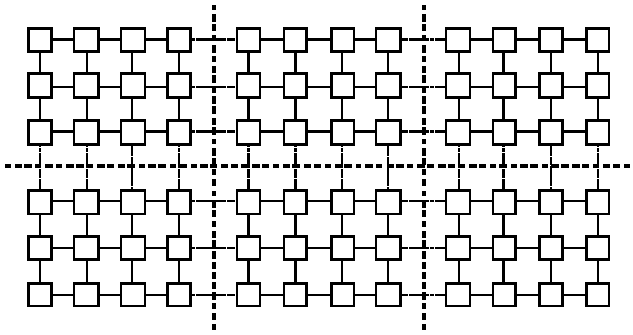
grid size: $m * m * m = n$

surface size: $6m * m = 6n^{2/3}$

relation surface/volume: $6 / n^{1/3}$

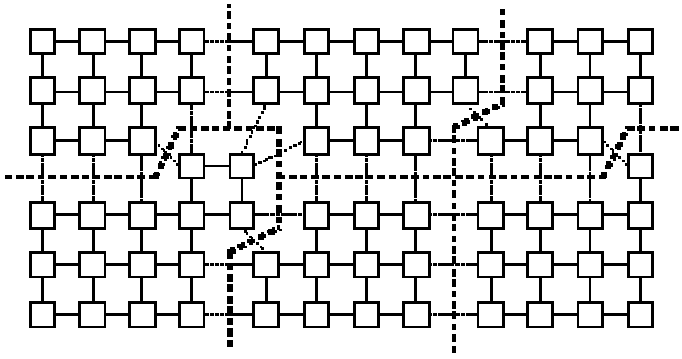
communication/calculation ratio worse for 3D!

regular grid partitioning of regular grid



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

domain decomposition: irregular grid partitioning of regular grid

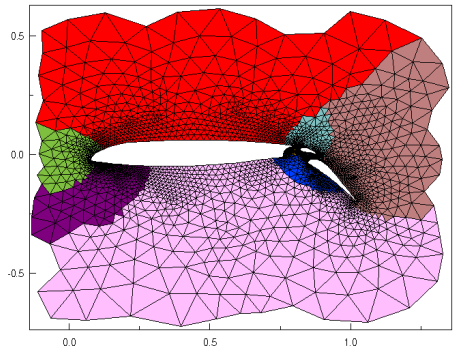


http://www.cs.reading.ac.uk/dbpp/text/node19.html

domain decomposition: recursive Spectral Bisection

Slotted Airfoil (8034 elements)

Spectral Bisection



http://www.cs.sandia.gov/CRF/gif/domain_mapping_fig2.gif

domain decomposition:

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

$$Q = (q_{vw})_{v,w \in nodes}$$

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

determine Eigenvector (Fiedlervector) x for the first Eigenvalue $\lambda > 0$

calculate

$$median = \frac{1}{|nodes|} \sum_{v \in nodes} x_v$$

define sets by

$$p_v^+ = \begin{cases} 1 & x_v > median \\ 0 & otherwise \end{cases}$$

$$p_v^- = \begin{cases} 1 & x_v < median \\ 0 & otherwise \end{cases}$$

if $p^+ + p^- = 1$

then $p^+ and\ p^-$ are connected

and best in some sense

Recursive Spectral Bisection

CHACO:
http://www.cs.sandia.gov/CRF/chac.html
METIS:
http://www-users.cs.umn.edu/~karypis/metis/
JOSTLE:
http://www.gre.ac.uk/~jjg01/

Partitioning Software

ZOLTAN:
http://www.cs.sandia.gov/~kddevin/Zoltan_html/

CHACO:
http://www.cs.sandia.gov/CRF/chac.html

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

$$\partial L/\partial q \;\; \Delta q = f - L(q)$$

$$q\_new = q + \Delta q$$

two different approaches

explicit procedures like

$$q^{n+1} - q^n = \Delta t A q^n$$

only need a single operator evaluation

(several steps for Runge Kutta)

implicit problems

$$q^{n+1} - \Delta t A q^{n+1} = q^n$$

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

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

Gauss-Seidel Iteration for M=L+D

```
do element in all_elements
   delta = operation_of( element%value , neighbourhood_elements%value )
   element%value= element%value + omega*delta
enddo
```

in general non parallelizable recursion

simple iterative procedures: Jakobi

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

Jakobi Iteration for M=D

```
do element in all_elements
  delta(element)=operation_of(element%value,neighbourhood_elements%value)
enddo
do element in all_elements
  element%value= element%value+omega*delta(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 buildung blocks

Conjugate Gradient Squared (CGS)

start values

$$r_0 = L^{-1} A x_0 - b$$
$$\bar{r}_0 = ?$$
$$q_0 = 0$$
$$p_{-1} = 0$$
$$\rho_{-1} = 1$$

Iteration

$$do\ k = 1, k\max$$
$$\rho_k = \langle \bar{r}_0, r_k \rangle$$
$$\beta_k = \rho_k / \rho_{k-1}$$
$$u_k = r_k + \beta_k q_k$$
$$p_k = u_k + \beta_k (q_k + \beta_k p_{k-1})$$
$$v_k = L^{-1} A U^{-1} p_k$$
$$\sigma_k = \langle \bar{r}_0, v_k \rangle$$
$$\alpha_k = -\rho_k / \sigma_k$$
$$q_{k+1} = u_k + \alpha_k v_k$$
$$w_k = \alpha_k U^{-1} (u_k + q_{k+1})$$
$$r_{k+1} = r_k + L^{-1} A w_k$$
$$x_{k+1} = x_k + w_k$$
$$enddo$$

BiCGSTAB (2)

start values

x_0 *is an initial guess;* $r_0 = b - Ax_0$;
$\tilde{r}_0$ *is an arbitrary vector, such that* $(r, \tilde{r}_0) \neq 0$,
e.g., $\tilde{r}_0 = r$;
$\rho_0 = 1; u = 0; \alpha = 0; \omega_2 = 1;$

even Bi-CG step

for $i = 0,2,4,6,\cdots$
$\rho_0 = -\omega_2 \rho_0$
$\rho_1 = (\tilde{r}_0, r_i); \beta = \alpha\, \rho_1 / \rho_0; \rho_0 = \rho_1$
$u = r_i - \beta u;$
$v = Au$
$\gamma = (v, \tilde{r}_0); \alpha = \rho_0 / \gamma;$
$r = r_i - \alpha v;$
$s = Ar$
$x = x_i + \alpha\, u;$

odd Bi-CG step

$\rho_1 = (\tilde{r}_0, s); \beta = \alpha\, \rho_1 / \rho_0; \rho_0 = \rho_1$
$v = s - \beta\, v;$
$w = Av$
$\gamma = (w, \tilde{r}_0); \alpha = \rho_0 / \gamma;$
$u = r - \beta\, u$
$r = r - \alpha\, v$
$s = s - \alpha\, w$
$t = As$

GCR(2) - part

$\omega_1 = (r, s); \mu = (s, s); v = (s, t); r = (t, t)$
$\omega_2 = (r, t); r = r - v^2 / \mu;$
$\omega_2 = (\omega_2 - v\omega_1 / \mu) / r;$
$\omega_{1=} (\omega_1 - v\omega_2) / \mu$
$x_{i+2} = x + \omega_1 r + \omega_2 s + \alpha\, u$
$r_{i+2} = r - \omega_1 s - \omega_2 t$
if x_{i+2} *accurate enough then quit*
$u = u - \omega_1 v - \omega_2 w$
end

operations of Krylov space algorithms

- some scalar operations
- scalarproduct 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 datastructure

work can be done in loops

all essential loops are be 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

$$Ax = b \quad \rightarrow \quad (L^{-1}AU^{-1})\,Ux = L^{-1}b$$

preconditioning is a problem for vectorization and parallelization

Jakobi (diagonal) preconditioning simple

Block Jakobi 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

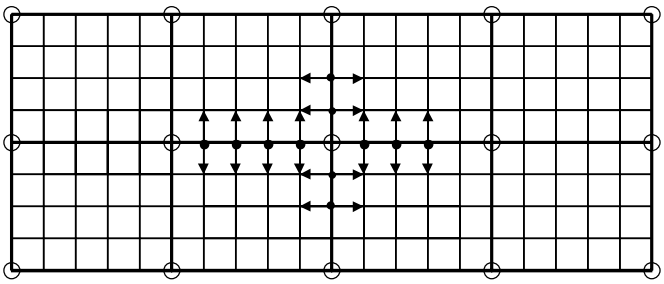
not applicable to vector computers

ILU for the blocks of a domain decomposition

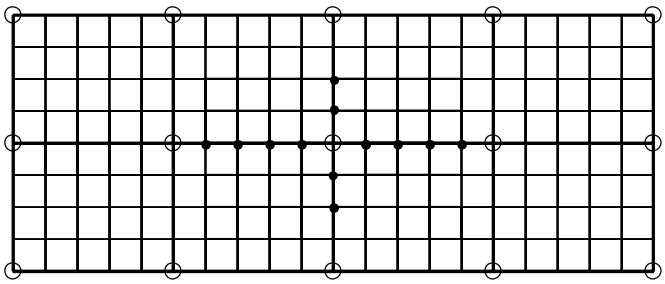
MILU for general matrix

```
do   l = 1, n-1
        do  j = l+1, n
         if ((j, l) ∈ S ) then
            aa = a_{jl}/a_{ll}
            aa_{jl} = aa
            do k = l+1, n
               if ((l, k) ∈ S ) then
                  if ((j, k) ∈  S ) then
                     a_{jk} = a_{jk} - aa*a_{lk}
                  else
                     a_{jj}  = a_{jj}  - aa*a_{lk}
                  endif
               endif
            enddo
         endif
      enddo
   enddo
```


overlapping domain decomposition



non overlapping domain decompostion

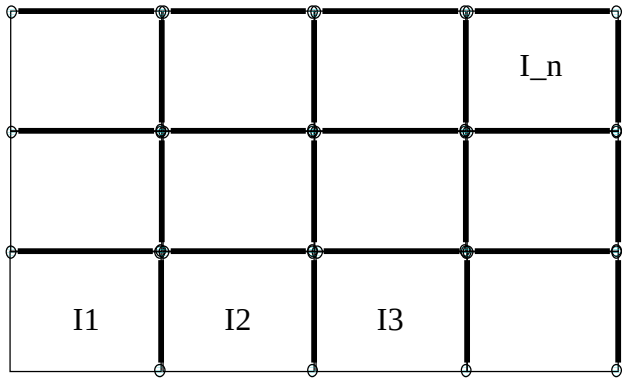


nonoverlapping Domain Decomposition

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



independent edges

cross points; coarse grid points

nonoverlapping DD:

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

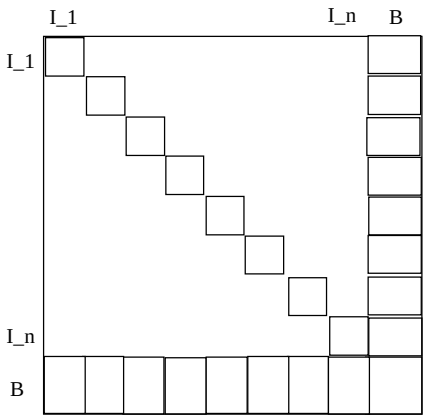
the numbers of the interior of all domains come first

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 Domain Decomposition

Matrix of decomposed nonoverlapped domain



nonoverlapping DD: Schur complement

is the equivalent equation on the B-points

I points are eliminated

$$S = A_{BB} - A_{BI} A_{II}^{-1} A_{IB}$$

$$S \phi_B = f_B - A_{BI} A_{II}^{-1} f_I$$

detailed

$$S = A_{BB} - \sum_{i=1}^{n} A_{BI_i} A_{I_i I_i}^{-1} A_{I_i B} \qquad \text{parallelizable}$$

$$S \phi_B = f_B - \sum_{i=1}^{n} A_{BI_i} A_{I_i I_i}^{-1} f_{I_i} \qquad \text{to be solved}$$

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 is the union of the edges and the domain cross points

$$B = V \cup \bigcup_k E_k$$

the edges appear as separated independent operators;

only the cross points V are connecting the edges

this will be reflcted in the construction of the preconditioning matrix

nonoverlapping DD: Preconditioner 1

R_k	injection operators for the edge k
$S_k = R_k S R_k^T$	restriction of the Schur complement to the edge
R_0	some specific mapping to the coarse grid
$S_V = R_0 S R_0^T$	restriction of the Schur complement to the coarse grid

nonoverlapping DD: Preconditioner 2

$$M = \sum_{E_k} R_k^T S_{kk}^{-1} R_k + R_0^T S_V^{-1} R_0$$ local and global part

Preconditioner for the solution of

$$S\phi_B = f_B - A_{BI} A_{II}^{-1} f_I$$

The equation can be solved by a preconditioned CG procedure

$$\kappa(MS) = O(1 + (\log(H/h))^2)$$ 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

http://www.cerfacs.fr/algor/reports/algo_reports_1997.html

coloring: defining independent sets

helps preconditioning

histogramm loop

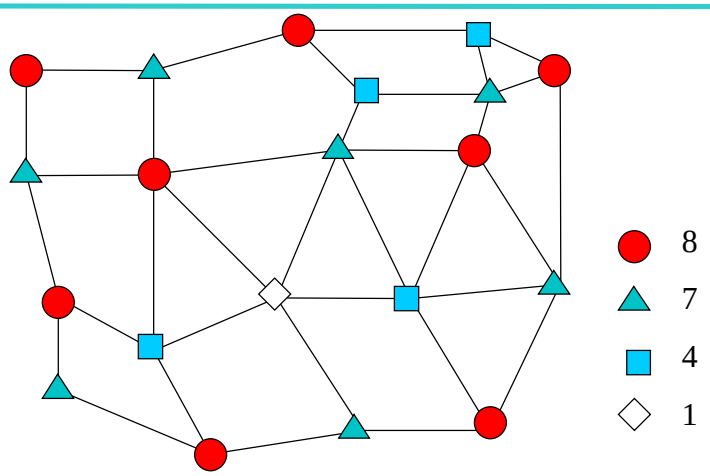
assembling stiffness matrices

assembling forces on nodes

distribution of states on the neighbourhood

(collection is no problem)

coloring of a graph: any neighbour has different color



coloring

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 np-complete

calculation of colors is expensive

pays out only if used several times

Multigrid 1

to be solved $L_f(x_f) = f_f$

presmoothing $x_f := S_f^{\nu_1}(x_f)$

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

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

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

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

postsmoothing $x_f := S_f^{\nu_2}(x_f)$

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

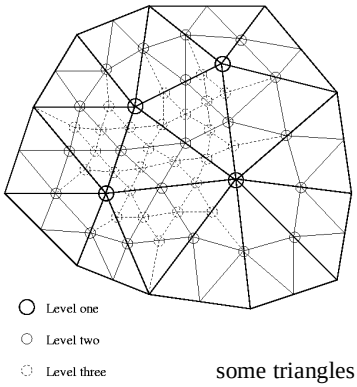
coarse grids should reside on the same processor as the fine grids

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!

hardware and implementation

hardware conditions for implementation

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

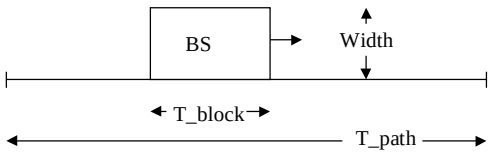
hardware prefetch for strided data

software prefetch by the compiler

Latency from processor to various locations

	seconds	clocks	future development
clock rate	0.5-4 nsec		decreasing
register		1	
L1 cache		1-2	
L2 cache		4-20	
L3 cache		10-100	
memory	60-1300 nsec	60-1300	slowly decreasing
remote memory		500-1400	slowly decreasing
shmem	~3 µsec		
MPI	5-20 µsec		slowly decreasing
disk	~8 msec		
tape	~30 sec		

dependency of bandwidth and latency



single block transfer!

dependency of bandwidth and latency 2

T_block = Blocksize/(Width*velo*freq)

T_complete = 2*T_path+T_block (Acknowledgement!)

Latency = 2*T_path

Bandwidth(Blocksize) = Blocksize/T_complete

= Blocksize/(Latency+Blocksize/(Width*velo*freq))

BW_max = Width*velo*freq (Blocksize → ∞)

Bandwidth(Blocksize) = BW_max/(BW_max*Latency/Blocksize + 1)

Hyperbola!

Bandwidth-Performance relation for multi-block grid with cubes

number of cells in node	n_{cell}
operations per cell	op
latency	lat
number of variables per exchanged cell	v_{exch}
performance	$perf$
bandwidth	BW

Exchange time less 1/10 of computing time:

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

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

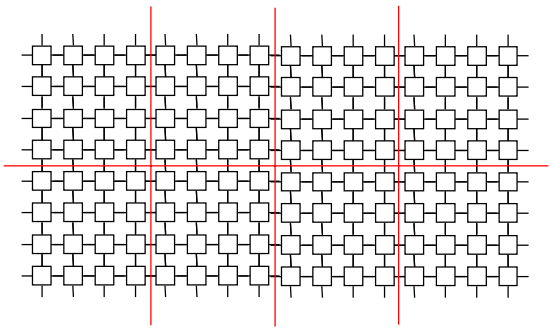
$$\frac{op * n_{cell}}{perf} \geq 10 * \left(\frac{v_{exch} * 6 * n_{cell}^{2/3} * 8B}{BW} + 6 * lat \right)$$

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

BW is important

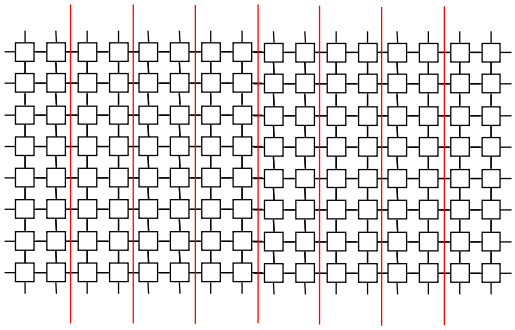
distributions in quadratic patches

10*4=40 connections



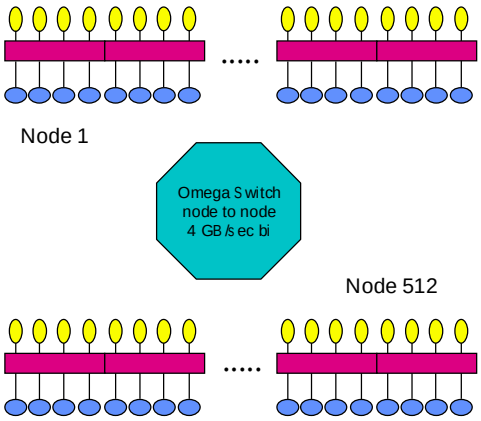
distribution in line patches

7*8=56 connections



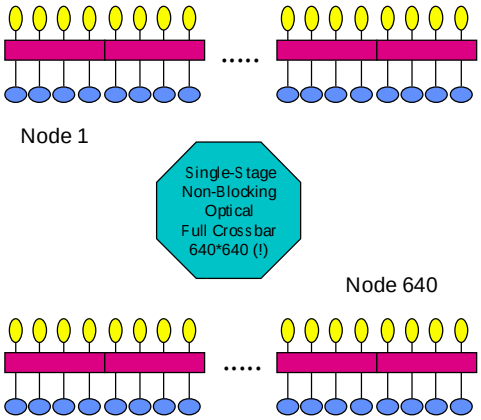
High Performance Systems

ASCI White IBM (2000)



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 3
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

```
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                                          ! jump back label
   edge%var1 =  edge%edge_1%var2+edge%edge_2%var2
   list_length=list_length+1
   if( .not. associated(edge) ) exit        ! controls the do loop
   edge => edge%edge_1
enddo
```


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

```
do                                          ! jump back label
   edge%var1 =  edge%edge_1%var2+edge%edge_2%var2
   list_length=list_length+1
   if( .not. associated(edge) ) exit        ! controls the do loop
   edge => edge%edge_1
enddo
```

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(ele_type),pointer,dimension(:) :: child        ! allows an arbitray
                                                    ! 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)

inhibiting processor performance

- frequent use of dynamic data
- frequent copy of data
- data aliasing inhibiting data dependence analysis

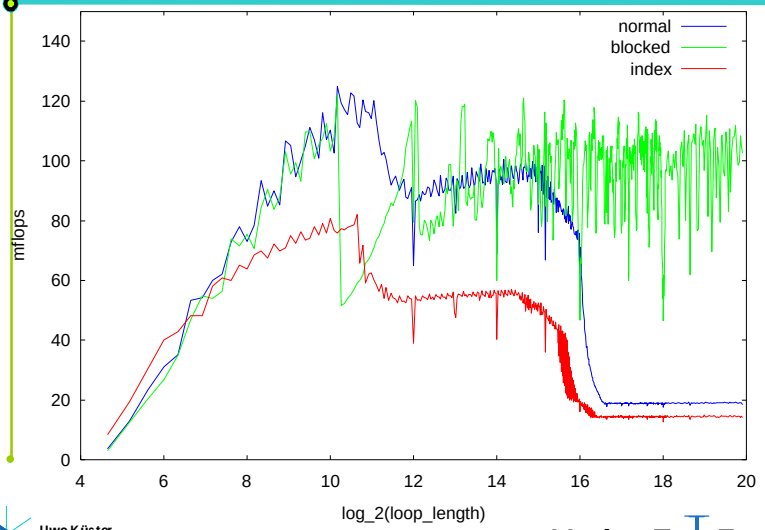
  pointer usage

  usage of indirection on the left side of assignments
- large number of calls
  large number of small objects in object oriented techiques
- 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



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

current_number	element	indices		
0	5	11	8	7
1	11	4	5	
2	7	5	16	8
3	8	5	7	2
4	2	8	12	
5	4	11	16	9
6	16	4	12	7
7	9	4	12	
8	12	2	16	9

matrix values are omitted, real array analogous to integer indices array

sparse matrices: row ordered example

row_number

0	5
1	11
2	7
3	8
4	2
5	4
6	16
7	9
8	12

begin

0	0
1	3
2	5
3	8
4	11
5	13
6	16
7	19
8	21

index

0	11	1	7	2	8
3	4	4	5		
5	5	6	16	7	8
8	5	9	7	10	2
11	8	12	12		
13	11	14	16	15	9
16	4	17	12	18	7
19	4	20	12		
21	2	22	16	23	9

end

0	2
1	4
2	7
3	10
4	12
5	15
6	18
7	20
8	23

index arrays beginning with 0

row ordered and jagged diagonal matrices: data structure

```
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 matrizes!

row ordered matrix: matrix vector multiplication

```
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

row_number

0	5
1	7
2	8
3	4
4	16
5	12
6	11
7	2
8	9

begin

0	0
1	9
2	18

index

0	11	9	7	18	8
1	5	10	16	19	8
2	5	11	7	20	2
3	11	12	16	21	9
4	4	13	12	22	7
5	2	14	16	23	9
6	4	15	5		
7	8	16	12		
8	4	17	12		

end

0	8
1	17
2	23

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)
  enddo
enddo

  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 flexibel
  - 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

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
http://www.cs.kuleuven.ac.be/cwis/research/natw/DRAMA/index.html

Patrick Amestoy, Iain Duff, Jean Yves L'Excellent, and Petr Plechác.
PARASOL An integrated programming environment for parallel sparse matrix solvers.
Technical report, Department of Computation and Information, 1998.
http://www.genias.de/projects/parasol/

end