StreamFlo3D

Massimiliano Fatica

SSS meeting

January 21th 2003

Outline

- StreamFlo
- StreamFlo3D
- BROOKTRAN
- Streaming tridiagonal solver
- ASCI workshop on petaflop applications

StreamFlo

- StreamFlo is a complete application:
  - single block Euler solver on O-mesh.
  - single grid implemented and debugged.
  - multigrid implemented.
  - the code has 2600 lines.
  - the main code that controls all the multigrid strategy works on 1D arrays. The arrays are reshaped as 2D arrays in the functions.

The metacompiler only runs on Linux/x86

Remote compilation is now possible but handling multiple files is tricky

I am using m4 to produce a single file

```
ROOTDIR    = ../..
EXECUTABLE = streamflo
FILES      = streamflo
RUNTIME    = brt

include ../../common.mk

SUBROUTINE = init.br dflux.br eflux.br io.br metric.br \
             dfluxc.br  init.br   main.br  step.br flow.h collc.br \
             gmesh.br euler.br bcfar.br addw.br

streamflo.br: $(SUBROUTINE)
        m4 streamflo.m4 > streamflo.br
```

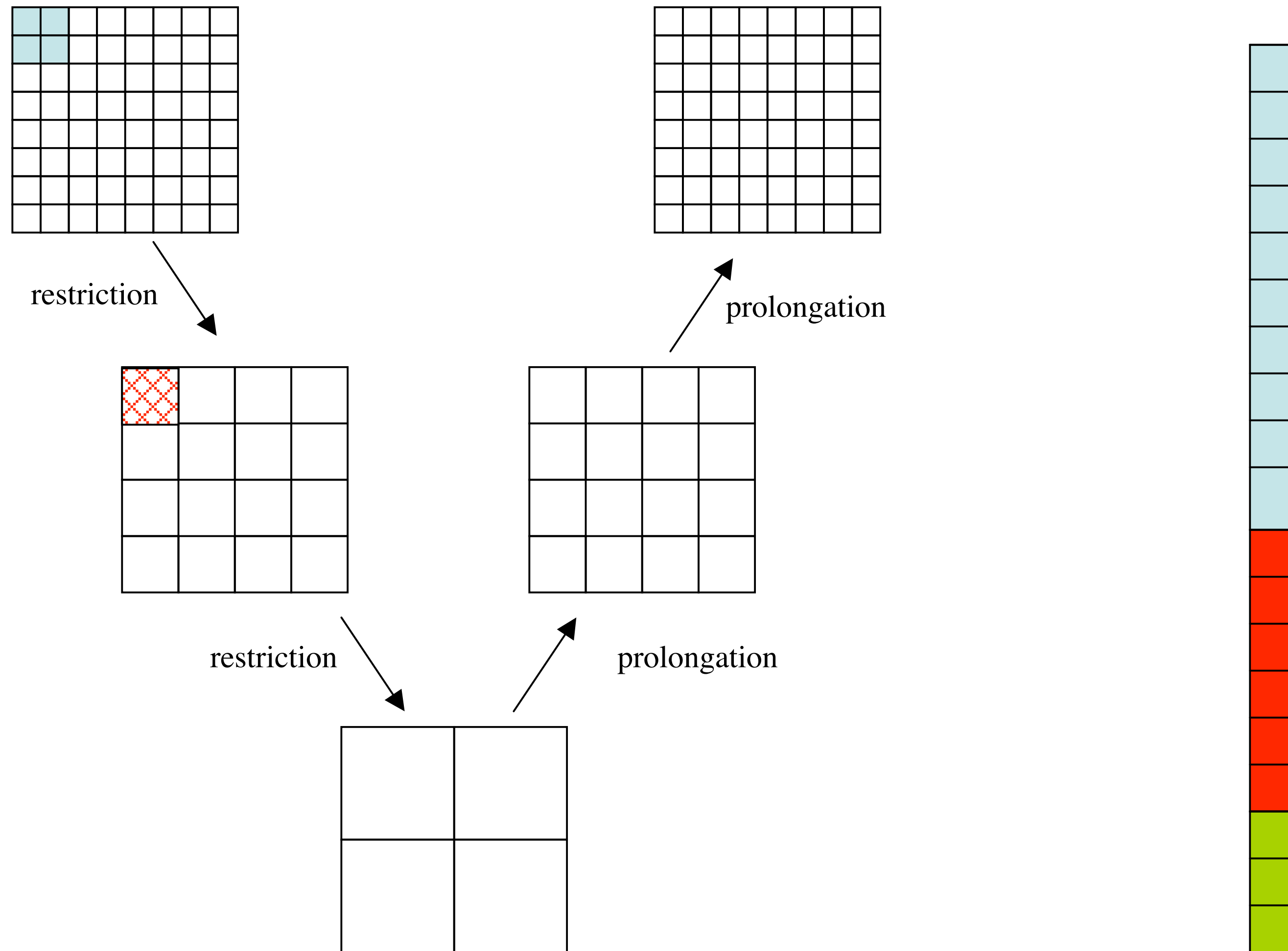
Moving to 3D

- Initial target was FLO87, the equivalent in 3D of FLO82, the starting point for StreamFLO
- After some discussions with Juan, we are now targeting FLO107-MB, that is equivalent to TFLO:
  - Navier-Stokes solver for external aerodynamics with turbulence model
  - multiblock
  - parallel
  - general geometry

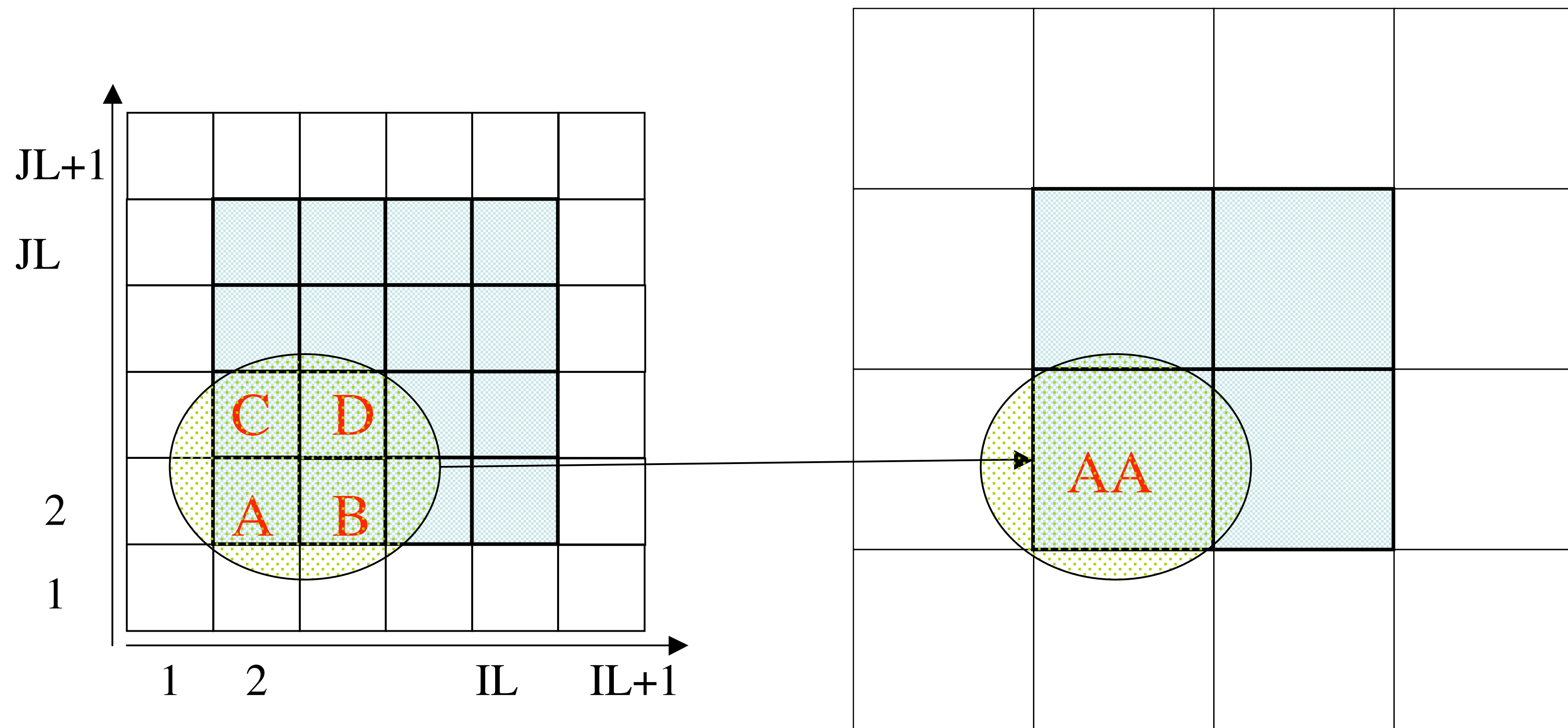
StreamFlo3D

- StreamFlo3D is work in progress:
  - The code reuses the main code of StreamFLO to control the multigrid strategy
  - The multidimensional syntax of the Brook operators allows for an easy translation of the data movement.
  - All the computational kernels have to be rewritten:
    - Possible splitting of some kernels:

Multigrid



Restriction operator



$$\phi(AA) = \frac{Area(A)\cdot\phi(A) + Area(B)\cdot\phi(B) + Area(C)\cdot\phi(C) + Area(D)\cdot\phi(D)}{Area(AA)}$$

Kernel code

```
kernel void TransferField2D(flow2d_s fine_flow, outfixed Flow coarse_flow)
{
   coarse_flow.rho=    fine_flow[0][0].rho +fine_flow[0][1].rho
                      +fine_flow[1][0].rho +fine_flow[1][1].rho;
…….
}
```

2D
(2x2)

```
kernel void TransferField3D(flow3d_s fine_flow, outfixed Flow coarse_flow)
{
   coarse_flow.rho=    fine_flow[0][0][0].rho +fine_flow[0][1][0].rho
                      +fine_flow[1][0][0].rho +fine_flow[1][1][0].rho
                      +fine_flow[0][0][1].rho +fine_flow[0][1][1].rho
                      +fine_flow[1][0][1].rho +fine_flow[1][1][1].rho;
…….
}
```

3D
(2x2x2)

The access pattern is defined in the following Brook code

2D code

```
// Fine mesh (nx+2,ny+2)
  streamSetLength(flow,(nx+2)*(ny+2));
  SetInitialField(flow,rho0,u0,v0,h0,p0);
  streamShape(flow,2,(nx+2),(ny+2));

// Coarse mesh (nx/2+2,ny/2+2)
  streamSetLength(coarse_flow,(nx/2+2)*(ny/2+2));
  streamShape(coarse_flow,2,(nx/2+2),(ny/2+2));

// Arrange the fine mesh in 2x2 group
 streamDomain(flow,flow,2,2,nx+1,2,ny+1);
  streamGroup(local_flow2d,flow,STREAM_GROUP_HALO,2,2,2);

//Check that the two streams have compatible length
  local_length=streamGetLength(local_flow2d);
  printf("Grouped initial flow field on fine mesh %d \n",local_lenght);
  local_lenght=streamGetLength(coarse_flow);
  printf("initial flow field on coarse mesh %d \n",local_lenght);

//Apply restriction operator
  TransferField2D(local_flow2d, coarse_flow)
```

3D code

```
// Fine mesh (nx+2,ny+2,nz+2)
  streamSetLength(flow,(nx+2)*(ny+2)*(nz+2));
  SetInitialField(flow,rho0,u0,v0,w0,h0,p0);
  streamShape(flow,3,(nx+2),(ny+2),(nz+2));

// Coarse mesh (nx/2+2,ny/2+2,nz/2+2)
  streamSetLength(coarse_flow,(nx/2+2)*(ny/2+2)*(nz/2+2));
  streamShape(coarse_flow,3,(nx/2+2),(ny/2+2),(nz/2+2));

// Arrange the fine mesh in 2x2x2 group
  streamDomain(flow,flow,3,2,nx+1,2,ny+1,2,nz+1);
streamGroup(local_flow3d,flow,STREAM_GROUP_HALO,3,2,2,2);

//Check that the two streams have compatible length
  local_length=streamGetLength(local_flow3d);
  printf("Grouped initial flow field on fine mesh %d \n",local_lenght);
  local_lenght=streamGetLength(coarse_flow);
  printf("initial flow field on coarse mesh %d \n",local_lenght);

//Apply restriction operator
  TransferField3D(local_flow3d, coarse_flow)
```

Stream definition

```
struct  Flow_struct {
            float rho;  /* density */
            float u;    /* momentum in x direction=density*velocity_x */
            float v;    /* momentum in y direction=density*velocity_y */
            float w;    /* momentum in z direction=density*velocity_z */
            float e;    /* Total energy= density*entalpy-pressure */
            float p;    /* pressure */ };

typedef stream struct Flow_struct Flow;
typedef stream float floats;
typedef stream Flow **flow2d_s;
typedef stream Flow ***flow3d_s;
typedef stream floats **float2d_s;

main(int argc, char** argv)
{
  Flow flow,local_flow,interior_flow,coarse_flow;
  flow2d_s local_flow2d "2,2";
  flow3d_s local_flow3d "2,2,2";
 ……..
}
```

Flux computations

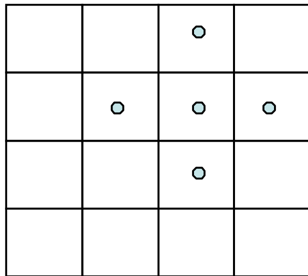
```
C
C  FLUX IN THE I DIRECTION
C
    DO J=2,JL
     DO  I=1,IL
     XY       = X(I,J,1)  -X(I,J-1,1)
     YY       = X(I,J,2)  -X(I,J-1,2)
     PA       = P(I+1,J)  +P(I,J)
     QSP      = (YY*W(I+1,J,2)  -XY*W(I+1,J,3))/W(I+1,J,1)
     QSM      = (YY*W(I,J,2)  -XY*W(I,J,3))/W(I,J,1)
     FS(I,J,1) = QSP*W(I+1,J,1)  +QSM*W(I,J,1)
     FS(I,J,2) = QSP*W(I+1,J,2)  +QSM*W(I,J,2)  +YY*PA
     FS(I,J,3) = QSP*W(I+1,J,3)  +QSM*W(I,J,3)  -XY*PA
     FS(I,J,4) = QSP*(W(I+1,J,4)  +P(I+1,J))  +QSM*(W(I,J,4)
         +P(I,J))
    END DO
  END DO

C
C    ACCUMULATE THE FLUX IN THE I DIRECTION
C
     DO N=1,4
        DO  J=2,JL
         DO  I=2,IL
                  DW(I,J,N) = FS(I,J,N)  -FS(I-1,J,N)
         END DO
        END DO
     END DO
```

```
C
C    FLUX IN THE J DIRECTION
C
     DO J=2,JL
     DO  I=2,IL
     XX       = X(I,J,1)  -X(I-1,J,1)
     YX       = X(I,J,2)  -X(I-1,J,2)
     PA       = P(I,J+1)  +P(I,J)
     QSP      = (XX*W(I,J+1,3)  -YX*W(I,J+1,2))/W(I,J+1,1)
     QSM      = (XX*W(I,J,3)  -YX*W(I,J,2))/W(I,J,1)
     FS(I,J,1) = QSP*W(I,J+1,1)  +QSM*W(I,J,1)
     FS(I,J,2) = QSP*W(I,J+1,2)  +QSM*W(I,J,2)  -YX*PA
     FS(I,J,3) = QSP*W(I,J+1,3)  +QSM*W(I,J,3)  +XX*PA
     FS(I,J,4) = QSP*(W(I,J+1,4)  +P(I,J+1))  +QSM*(W(I,J,4)
         +P(I,J))
    END DO
   END DO
C
C    ACCUMULATE THE FLUX IN THE J DIRECTION
C
    DO N=1,4
     DO J=2,JL
      DO I=2,IL
        DW(I,J,N) = DW(I,J,N)  +FS(I,J,N)  -FS(I,J-1,N)
        END DO
       END DO
     END DO
```

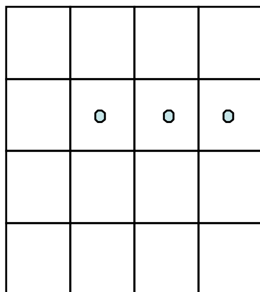
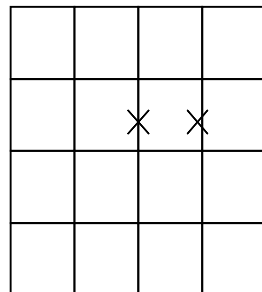
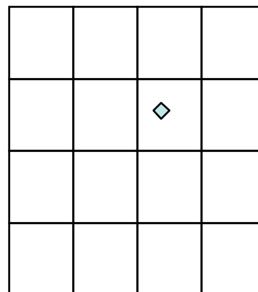
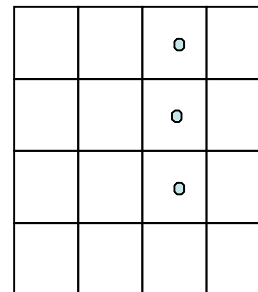
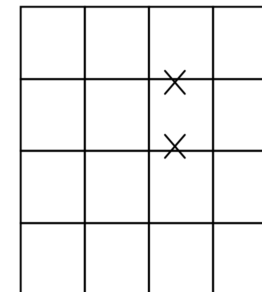
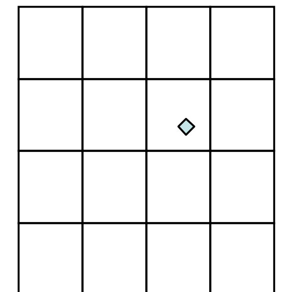
Flux computations in Brook

- We use StreamStencil to compute the flux but we recompute twice the flux on each face



This approach generates long kernels and use a lot of registers.

We can break down the original subroutine in the following kernels

In the x direction

In the y direction

This approach generates short kernels but moves a lot of data

FORTRAN syntax

- Why we need FORTRAN support?
  - most of the scientific and high performance codes are in FORTRAN
  - FORTRAN is still the language of choice for scientific applications
  - National Labs, NASA, aerospace companies have a huge investment in FORTRAN codes.
  - The codes have been thoroughly tested and validated
  - Rewrite a code in a different language may not be a big deal but the validation process is.
  - A code not fully validated can be ok for academia but not for real missions.

Porting codes to SSS

- The path from C to Brook is much easier that the one from FORTRAN.

  1) C to Brook: similar to OpenMP parallelization

  Start from original code and start to "streamify" functions. You can start working on the time consuming part of the code.
  Very easy to check the results.
  All the I/O , utilities functions are working

  2) Fortran to Brook: similar to MPI parallelization.

  The code has to be written from scratch.
  You need to spend a lot of time on I/O and utilities function
  Checking the results and tracking down the bugs is time consuming.

Possible paths from FORTRAN to Brook

FORTRAN + Brook: Mixed language programming.

We can keep the original structure of the FORTRAN code

The subroutines that are floating-point intensive are replaced by Brook kernels.

The streams are a view of memory, we just need to pass the proper memory location to Brook.

It will require a FORTRAN compiler on SSS (not a big deal once we find a commercial partner, the scalar processor has already a FORTRAN compiler available)

```
// FORTRAN main
program sample
real, allocatable, dimension(:):: a,b,c
integer:: n
n=1000
allocate(a(n),b(n),c(n))
call brook_sum(a,b,c,n)
end program sample

// Brook kernel
kernel add_array( floats a, floats b, floats)
{ c=a+b; }

// Brook function
void brook_sum(a,b,c,n)
float a[],b[],c[];
int *n;
{
 floats stream_a, stream_b, stream_c;
 streamLoad(stream_a,a,n);
 streamLoad(stream_b,b,n);
 add_array(stream_a,stream_b, stream_c);
streamStore(stream_c,c,n);
}
```

Possible paths from FORTRAN to Brook

BROOKTRAN: Streaming language that use the FORTRAN syntax.

The setup of stream is done through library calls

The kernels are written using a FORTRAN syntax and have the same constraints of the Brook kernels (all the memory accesses through the explicit interface,.....)

```
// FORTRAN main
program sample
real, allocatable, dimension(:):: a,b,c
integer:: n
n=1000
allocate(a(n),b(n),c(n))
call streamLoad(stream_a, a,n)
call streamLoad(stream_b, b,n)
call add_array(stream_a,stream_b,stream_c)
call StreamStore(stream_c,c,n)
end program sample

// Brooktran kernel
kernel add_array( floats a, floats b, floats)
{
c=a+b
 }
```

BROOKTRAN

Two possible paths:

a) Once we have the software infrastructure in place to handle Brook to SVM code, it can be implemented with a parser that translates FORTRAN to Brook (similar to the current Vectoral compiler that translate Vectoral to C).

b) a real compiler that from Brooktran will generate SVM or SSS code

Streaming tridiagonal solver

- Tridiagonal solvers are encountered in several numerical methods
- The standard algorithm is basically impossible to parallelize (some pipelining can be done for multiple right hand sides)
- Modification of a parallel version for the streaming architecture

Parallel tridiagonal solver

Several methods available. An interesting class is described as *parallel factorization*

•Factor the original matrix into a product of block matrix and a reduced matrix, which couples the block problem
•Solve each block problem on one processor
•Solve the reduced matrix problem

Algorithm proposed by Mattor, Williams and Hewett (Parallel Computing 1995, Vol. 21 Number 11,pp 1769-1782) avoid the first step

Parallel tridiagonal algorithm (Mattor et el.)

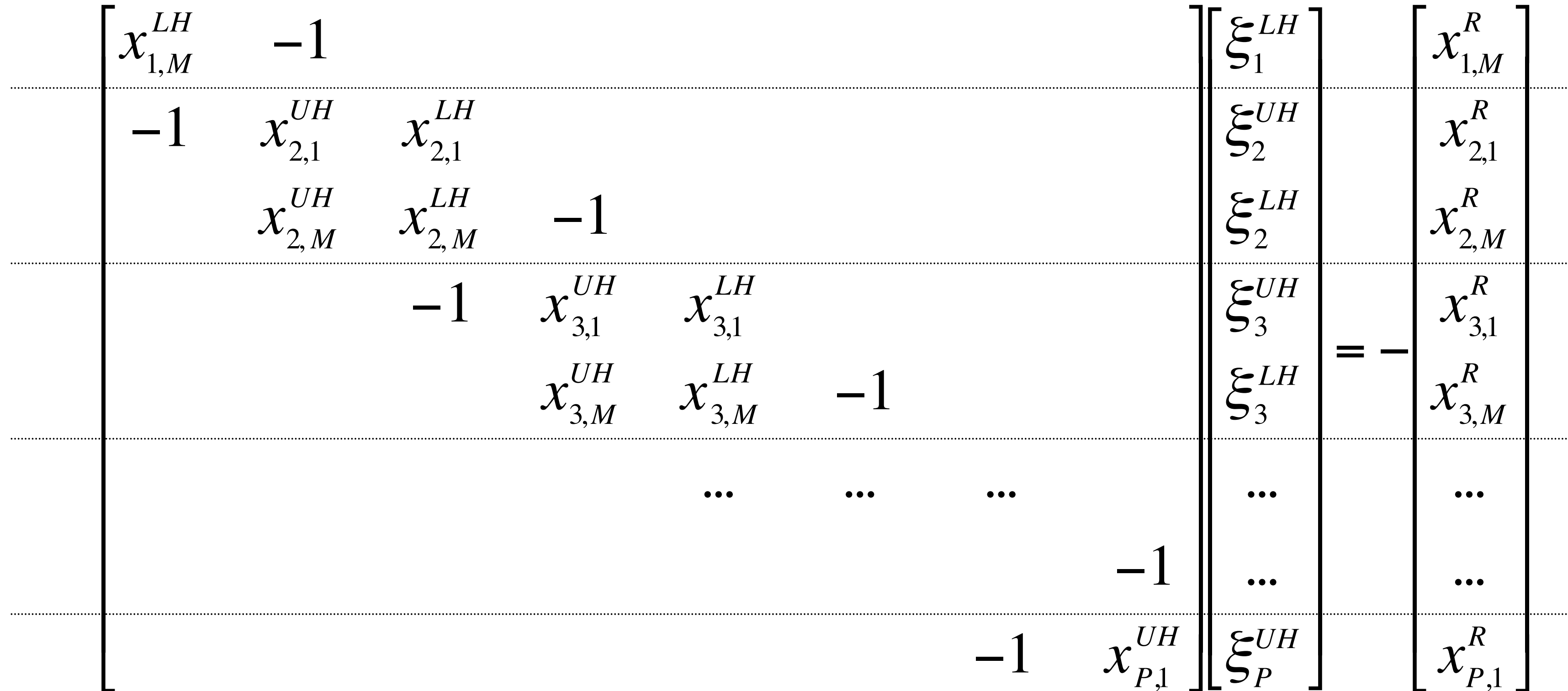
$$\underbrace{\left[\begin{array}{cccc|ccc} b_1 & c_1 & & & & & \\ a_2 & b_2 & c_2 & & & & \\ & a_3 & b_3 & c_3 & & & \\ & & a_4 & b_4 & c_4 & & \\ \hline & & & a_5 & b_5 & c_5 & \\ & & & & a_6 & b_6 & c_6 \\ & & & & & a_7 & b_7 \end{array}\right]}_{A} \cdot \underbrace{\left[\begin{array}{c} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \hline x_5 \\ x_6 \\ x_7 \end{array}\right]}_{x} = \underbrace{\left[\begin{array}{c} r_1 \\ r_2 \\ r_3 \\ r_4 \\ \hline r_5 \\ r_6 \\ r_7 \end{array}\right]}_{r}$$

$$\tilde{A} = \left[\begin{array}{cccc|ccc} b_1 & c_1 & & & & & \\ a_2 & b_2 & c_2 & & & & \\ & a_3 & b_3 & c_3 & & & \\ & & a_4 & b_4 & & & \\ \hline & & & & b_5 & c_5 & \\ & & & & a_6 & b_6 & c_6 \\ & & & & & a_7 & b_7 \end{array}\right], \quad r^{LH} = \left[\begin{array}{c} 0 \\ 0 \\ 0 \\ -c_4 \\ \hline 0 \\ 0 \\ 0 \end{array}\right], \quad r^{UH} = \left[\begin{array}{c} 0 \\ 0 \\ 0 \\ 0 \\ \hline -a_5 \\ 0 \\ 0 \end{array}\right]$$

i) $$Ax = r \Longrightarrow \begin{array}{l} \tilde{A}x^{P} = r \\ \tilde{A}x^{UH} = r^{UH} \\ \tilde{A}x^{LH} = r^{LH} \end{array} \Longrightarrow x = x^{P} + \gamma^{UH} x^{UH} + \gamma^{LH} x^{LH}$$

ii) The coefficient γ^{UH} and γ^{LH} are obtained from a tridiagonal system of size 2*(Np-1) where the matrix is function of x^{UH}, x^{LH} and the right hand side of x^{P}

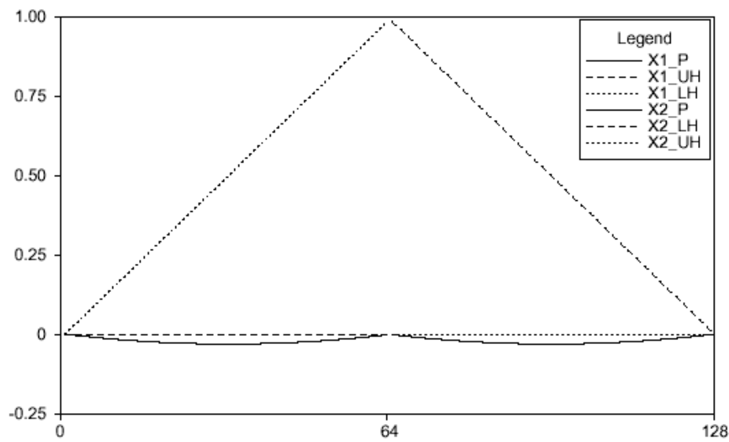
Reduced system



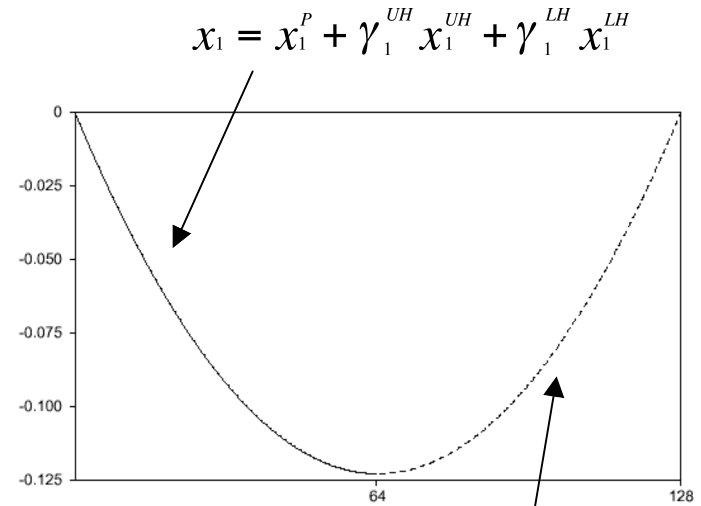
$$
\begin{bmatrix}
x_{1,M}^{LH} & -1 & & & & & & \\
-1 & x_{2,1}^{UH} & x_{2,1}^{LH} & & & & & \\
 & x_{2,M}^{UH} & x_{2,M}^{LH} & -1 & & & & \\
 & & -1 & x_{3,1}^{UH} & x_{3,1}^{LH} & & & \\
 & & & x_{3,M}^{UH} & x_{3,M}^{LH} & -1 & & \\
 & & & & \dots & \dots & \dots & \\
 & & & & & & & -1 \\
 & & & & & & -1 & x_{P,1}^{UH}
\end{bmatrix}
\begin{bmatrix}
\xi_1^{LH} \\ \xi_2^{UH} \\ \xi_2^{LH} \\ \xi_3^{UH} \\ \xi_3^{LH} \\ \dots \\ \dots \\ \xi_P^{UH}
\end{bmatrix}
= -
\begin{bmatrix}
x_{1,M}^{R} \\ x_{2,1}^{R} \\ x_{2,M}^{R} \\ x_{3,1}^{R} \\ x_{3,M}^{R} \\ \dots \\ \dots \\ x_{P,1}^{R}
\end{bmatrix}
$$

$$A \cdot x = b$$

$$\left[\begin{array}{ccc|ccc} 1 & 0 & & & & \\ 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ \hline & & 1 & -2 & 1 & \\ & & & 1 & -2 & 1 \\ & & & & 0 & 1 \end{array}\right] \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \hline x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ \hline 1 \\ 1 \\ 0 \end{bmatrix}$$



$$x_1 = x_1^P + \gamma_1^{UH} x_1^{UH} + \gamma_1^{LH} x_1^{LH}$$



$$x_2 = x_2^P + \gamma_2^{UH} x_2^{UH} + \gamma_2^{LH} x_2^{LH}$$

$$\gamma_1^{UH} = 0 \qquad \gamma_2^{UH} = -0.12304....$$

$$\gamma_1^{LH} = -0.12304.... \qquad \gamma_2^{LH} = 0$$

- It is very useful when the matrix A changes during the computation

- If the matrix A is constant, the homogeneous solutions can be stored

- The method can be extended to pentadiagonal or higher band matrix

Streaming tridiagonal solver

- Send each block problem to a stream processor
- Solve the coupling system on the scalar processor
- Find the solution on the stream processor

For writing numerical libraries, it will be nice to have:

- syncronization directive in Brook for scalar processor / stream processor interaction? Wait in kernel for additional input from scalar processor
- Hints on stream cache load/store/staging

ASCI meeting: Petaflop applications

– A strategic view on whether apps can evolve to petaflops, or will a revolution be needed i.e. different algorithms,different languages?
– What are the issues that need to be addressed to get to petaflop applications? Bottlenecks? Ie. random memory access.
– What are the requirements for getting to petaflop applications?
– What are the limits to scaling up to 100K processors?

ASCI meeting: Petaflop architectures

– BlueGene/L
– DARPA HPSS
– SNL advanced architectures
– LANL advanced architectures

Virtues of their approaches, explore novel architectures