Synchronous Shared Memory Parallel Examples

HPC Spring 2017

Prof. Robert van Engelen

Examples

- Data parallel prefix sum and OpenMP example
- Task parallel prefix sum and OpenMP example
- Simple heat distribution problem with OpenMP
- Iterative solver with OpenMP
- Simple heat distribution problem with HPF
- Iterative solver with HPF
- Gaussian Elimination with HPF

Dataparallel Prefix Sum

Serial

```
sum[0] = x[0];
for (i = 1; i<n; i++)
  sum[i] = sum[i-1] + x[i];
```

Dataparallel

```
for (j = 0; j < log2(n); j++)
  forall (i = 2^j; i < n; i++)
    x[i] = x[i] + x[i - 2^j];
```

Dataparallel = synchronous (lock-step)

Time complexity?

Parallel efficiency and speedup?

Dataparallel forall: concurrent write and read, but read always fetches old value: forall has "copy-in-copy-out" semantics (viz. CRCW PRAM model)

OpenMP Prefix Sum v1

```
for (j = 0; j < log2(n); j++)
{
  #pragma omp parallel private(i)
  {
    #pragma omp for
    for (i = 1<<j; i < n; i++)
      t[i] = x[i] + x[i - 1<<j];

    #pragma omp for
    for (i = 1<<j; i < n; i++)
      x[i] = t[i];
  }
}
```

"Copy out"

What about overhead: when/where would it be better to create/join threads?

Note: use bitshift to compute 2^j `= 1<<j`

Task Parallel Prefix Sum

```
for each processor 0 ≤ p < n
private j
{
  for (j = 1; j < n; j = 2*j)
  {
    if (p >= j)
      t[p] = x[p] + x[p-j];
    barrier;
    x[p] = t[p];
    barrier;
  }
}
```

Task/thread-parallel: parallelize outer loops, each processor takes an outer loop iteration to execute concurrently

OpenMP Prefix Sum v2

```
#pragma omp parallel shared(n,x,t) private(j,tid) num_threads(n)
{
  tid = omp_get_thread_num();

  for (j = 1; j < n; j = 2*j)
  {
    if (tid >= j)
      t[tid] = x[tid] + x[tid - j];
    #pragma omp barrier

    x[tid] = t[tid];
    #pragma omp barrier
  }
}
```

Alternative to `private(j,tid)`*:*
declare `j` *and* `tid` *locally in this block*

Uses n threads!
What if n is really large?

OpenMP Prefix Sum v3

```
#pragma omp parallel shared(n,nthr,x,z) private(i,j,tid,work,lo,hi)
{
  #pragma omp single
    nthr = omp_get_num_threads();
  tid = omp_get_thread_num();
  work = (n + nthr-1) / nthr;
  lo = work * tid;
  hi = lo + work;
  if (hi > n)
    hi = n;
  for (i = lo+1; i < hi; i++)
    x[i] = x[i] + x[i-1];
  z[tid] = x[hi-1];
  #pragma omp barrier
  for (j = 1; j < nthr; j = 2*j)
  {
    if (tid >= j)
      z[tid] = z[tid] + z[tid - j];
    #pragma omp barrier
  }
  for (i = lo; i < hi; i++)
    x[i] = x[i] + z[tid] - x[hi-1];
}
```

Note: assumes nthreads = 2^k

Local prefix sum over x

z = local prefix sum x[hi]

Global prefix sum over z

Update local prefix sum x

Dataparallel Heat Distribution Problem

$$h_{i,j} = \frac{h_{i-1,j} + h_{i+1,j} + h_{i,j-1} + h_{i,j+1}}{4}$$

```
for (iter = 0; iter < limit; iter++)
  forall (i = 0; i < n; i++)
    forall (j = 0; j < n; j++)
      h[i][j] = 0.25*(h[i-1][j]+h[i+1][j]+h[i][j-1]+h[i][j+1]);
```

Dataparallel = synchronous

Corresponds to Jacobi iteration

Dataparallel forall: concurrent write and read, but read always fetches old value: forall has "copy-in-copy-out" semantics (viz. CRCW PRAM model)

OpenMP Heat Distribution Problem

```
#pragma omp parallel shared(newh,h,limit,n) private(iter,i,j)
{
  for (iter = 0; iter < limit; iter++)
  {
    #pragma omp for
    for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
        newh[i][j] = 0.25*(h[i-1][j]+h[i+1][j]+h[i][j-1]+h[i][j+1]);
    #pragma omp for
    for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
        h[i][j] = newh[i][j];
  }
}
```

Corresponds to Jacobi iteration

Dataparallel Heat Distribution Red-black Ordering

```
for (iter = 0; iter < limit; iter++)
{
  forall (i = 0; i < n; i++)
    forall (j = 0; j < n; j++)
      if ((i+j) % 2 != 0)
        h[i][j] = 0.25*(h[i-1][j]+h[i+1][j]+h[i][j-1]+h[i][j+1]);
  forall (i = 0; i < n; i++)
    forall (j = 0; j < n; j++)
      if ((i+j) % 2 == 0)
        h[i][j] = 0.25*(h[i-1][j]+h[i+1][j]+h[i][j-1]+h[i][j+1]);
}
```

Dataparallel = synchronous

OpenMP Heat Distribution Red-black Ordering

```
#pragma omp parallel shared(h,limit,n) private(iter,i,j)
{
  for (iter = 0; iter < limit; iter++)
  {
    #pragma omp for
    for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
        if ((i+j) % 2 != 0)
          h[i][j] = 0.25*(h[i-1][j]+h[i+1][j]+h[i][j-1]+h[i][j+1]);
    #pragma omp for
    for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
        if ((i+j) % 2 == 0)
          h[i][j] = 0.25*(h[i-1][j]+h[i+1][j]+h[i][j-1]+h[i][j+1]);
  }
}
```

Iterative Solver

$$\mathbf{Ax} = \mathbf{b}$$

Jacobi iteration
$$x_i^k = \frac{1}{a_{i,i}} \left[b_i - \sum_{j \neq i} a_{i,j} x_j^{k-1} \right]$$

Stopping criteria:

$$\sqrt{\sum_{i=0}^{n-1} (x_i^k - x_i^{k-1})^2} < \epsilon$$

Pacheco

$$\left| \sum_{j=0}^{n-1} a_{i,j} x_j^k - b_i \right| < \epsilon \qquad \forall i = 0, \ldots, n-1$$

Bertsekas and Tsitsiklis

Iterative Solver: Jacobi Method

$$x_i^k = \frac{1}{a_{i,i}}\left[b_i - \sum_{j\neq i} a_{i,j}x_j^{k-1}\right]$$

```
for (i = 0; i < n; i++)
  x[i] = b[i];

for (iter = 0; iter < limit; iter++)
{
  for (i = 0; i < n; i++)
  {
    sum = -a[i][i] * x[i]; // correction to sum over j!=i
    for (j = 0; j < n; j++)
      sum = sum + a[i][j] * x[j];
    new_x[i] = (b[i] - sum) / a[i][i];
  }
  for (i = 0; i < n; i++)
    x[i] = new_x[i];
}
```

Note: stopping criterium omitted

Iterative Solver: Jacobi Method

$$x_i^k = \frac{1}{a_{i,i}}\left[b_i - \sum_{j\neq i} a_{i,j}x_j^{k-1}\right]$$

```
for (i = 0; i < n; i++)
  x[i] = b[i];

for (iter = 0; iter < limit; iter++)
{
  for (i = 0; i < n; i++)
    sum[i] = -a[i][i] * x[i]; // correction to sum over j!=i
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      sum[i] = sum[i] + a[i][j] * x[j];
  for (i = 0; i < n; i++)
    new_x[i] = (b[i] - sum[i]) / a[i][i];
  for (i = 0; i < n; i++)
    x[i] = new_x[i];
}
```

Note: after array expansion of scalar sum and loop fission

Note: stopping criterium omitted

Dataparallel Iterative Solver: Jacobi Method

$$x_i^k = \frac{1}{a_{i,i}}\left[b_i - \sum_{j\neq i} a_{i,j}x_j^{k-1}\right]$$

```
for (i = 0; i < n; i++)
  x[i] = b[i];

for (iter = 0; iter < limit; iter++)
{
  forall (i = 0; i < n; i++)
    sum[i] = -a[i][i] * x[i];
  for (j = 0; j < n; j++)
    forall (i = 0; i < n; i++)
      sum[i] = sum[i] + a[i][j] * x[j];
  forall (i = 0; i < n; i++)
    x[i] = (b[i] - sum[i]) / a[i][i];
}
```

Note: after loop interchange and forall-parallelization

Dataparallel = synchronous (lock-step)

Note: stopping criterium omitted

Task Parallel Iterative Solver: Jacobi Method

$$x_i^k = \frac{1}{a_{i,i}}\left[b_i - \sum_{j\neq i} a_{i,j}x_j^{k-1}\right]$$

```
for each processor 0 ≤ p < n
private iter, sum, j
{
  x[p] = b[p];
  for (iter = 0; iter < limit; iter++)
  {
    sum = -a[p][p] * x[p];
    for (j = 0; j < n; j++)
      sum = sum + a[p][j] * x[j];
    barrier;
    x[p] = (b[p] - sum) / a[p][p];
    barrier;
  }
}
```

Note: each processor is assigned to an iteration i

Note: stopping criterium omitted

Iterative Solver: Jacobi Method in OpenMP

```
#pragma omp parallel shared(a,b,x,new_x,n) private(iter,i,j,sum)
{
  #pragma omp for
  for (i = 0; i < n; i++)
    x[i] = b[i];
  for (iter = 0; iter < limit; iter++)
  {
    #pragma omp for
    for (i = 0; i < n; i++)
    {
      sum = -a[i][i] * x[i];
      for (j = 0; j < n; j++)
        sum = sum + a[i][j] * x[j];
      new_x[i] = (b[i] - sum) / a[i][i];
    }
    #pragma omp for
    for (i = 0; i < n; i++)
      x[i] = new_x[i];
  }
}
```

$$x_i^k = \frac{1}{a_{i,i}} \left[b_i - \sum_{j \neq i} a_{i,j} x_j^{k-1} \right]$$

OpenMP Iterative Solver Checking for Convergence

```
#pragma omp parallel shared(a,b,x,new_x,n,notdone) …
{
  …
  for (iter = 0; iter < limit; iter++)
  {
    …
    #pragma omp for reduce(||:notdone) private(sum,i,j)
    for (i = 0; i < n; i++)
    { sum = 0;
      for (j = 0; j < n; j++)
        sum = sum + a[i][j] * x[j];
      if (fabs(sum - b[i]) >= tolerance)
        notdone = 1;
    }
    if (notdone == 0) break;
  }
}
```

Can we change this to not recomputing the sum? Is there another solution without reduce?

$$\left|\sum_{j=0}^{n-1} a_{i,j} x_j^k - b_i\right| < \epsilon \qquad \forall i = 0, \ldots, n-1$$

Bertsekas and Tsitsiklis

OpenMP Iterative Solver Gauss-Seidel Relaxation

```
#pragma omp parallel shared(a,b,x,n,nt) private(iter,i,j,sum,tid,work,lo,hi,loc_x)
{
  #pragma omp single
    nt = omp_get_num_threads();
  tid = omp_get_thread_num();
  work = (n + nt-1) / nt;
  lo = work * tid;
  hi = lo + work;
  if (hi > n)
    hi = n;
  for (i = lo; i < hi; i++)
    x[i] = b[i];
  #pragma omp flush(x) // we need this?
  for (iter = 0; iter < limit; iter++)
  {
    #pragma omp barrier
    for (i = lo; i < hi; i++)
    {
      sum = -a[i][i] * x[i];
      for (j = 0; j < n; j++)
      {
        if (j >= lo && j < i)
          sum = sum + a[i][j] * loc_x[j-lo];
        else
          sum = sum + a[i][j] * x[j];
      }
      loc_x[i-lo] = (b[i] - sum) / a[i][i];
    }
    #pragma omp barrier
    for (i = lo; i < hi; i++)
      x[i] = loc_x[i-lo];
    #pragma omp flush(x) // we need this?
  }
}
```

Departure from pure dataparallel model!

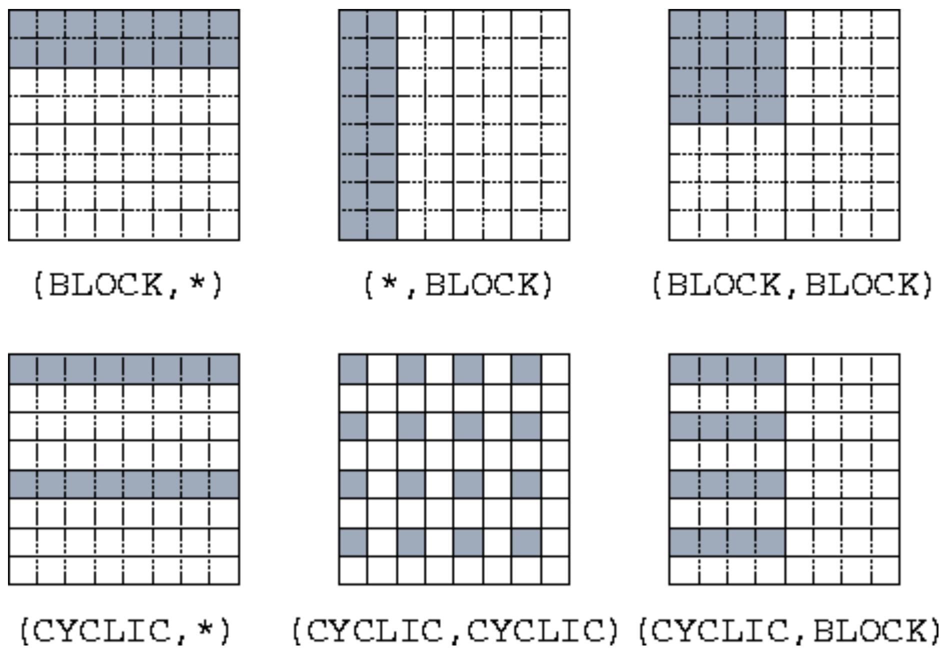
Synchronous Computing with High-Performance Fortran

- High Performance Fortran (HPF) is an extension of Fortran 90 with constructs for parallel computing
  - Dataparallel FORALL
  - PURE (side-effect free functions)
  - Directives for recommended data distributions over processors
  - Library routines for parallel sum, prefix (scan), scattering, sorting, …
- Uses the array syntax of Fortran 90 for as a dataparallel model of computation
  - Spreads the work of a single array computation over multiple processors
  - Allows efficient implementation on both SIMD and MIMD style architectures, shared memory and DSM
- But most users and vendors prefer OpenMP over HPF

HPF

```
!HPF$ PROCESSORS procname(dim1,…,dimN)
!HPF$ DISTRIBUTE array1(dist),…,arrayM(dist) ONTO procname
```

Shaded area: P0 has **X(1:2,1:8)**

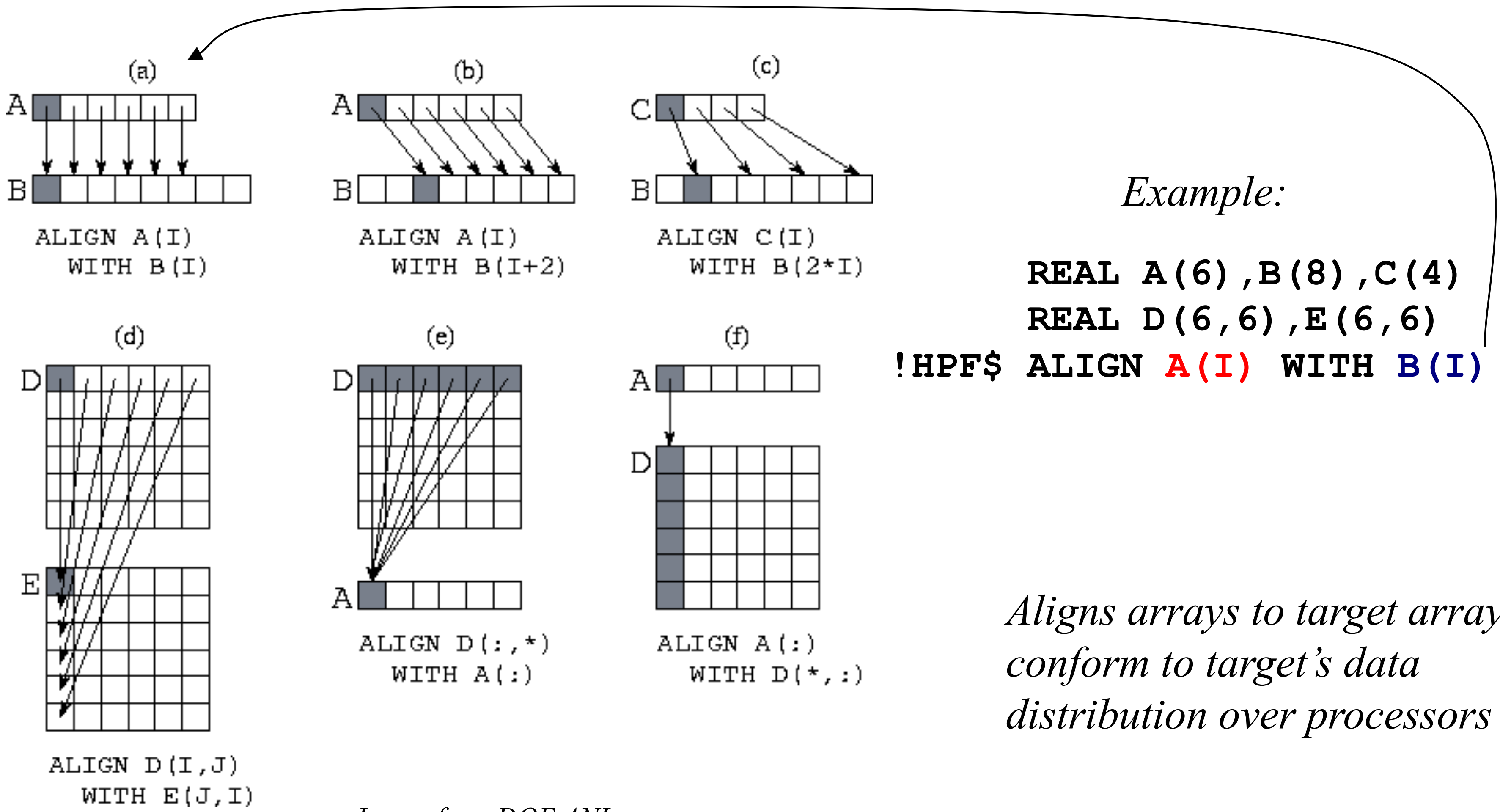


Example:

```
!HPF$ PROCESSORS pr(2,2)
      REAL X(8,8)
!HPF$ DISTRIBUTE X(BLOCK,*) ONTO pr
```

HPF

`!HPF$ ALIGN` ***`array`*** **`WITH`** ***`target`***



Example:

```
      REAL A(6),B(8),C(4)
      REAL D(6,6),E(6,6)
!HPF$ ALIGN A(I) WITH B(I)
```

Aligns arrays to target arrays to conform to target's data distribution over processors

Image from DOE ANL

HPF Heat Distribution Problem

$$h_{i,j} = \frac{h_{i-1,j} + h_{i+1,j} + h_{i,j-1} + h_{i,j+1}}{4}$$

```
!HPF$ PROCESSORS pr(4)
      REAL h(100,100)
!HPF$ DISTRIBUTE h(BLOCK,*) ONTO pr
      …
      h(2:99,2:99) = 0.25*( h(1:98,2:99)+h(3:100,2:99)
                          +h(2:99,1:98)+h(2:99,3:100))
```

Alternatively, with **FORALL**

```
FORALL (i=2:99,j=2:99) h(i,j) = 0.25*( h(i-1,j)+h(i+1,j)
                                      +h(i,j-1)+h(i,j+1))
```

Remember: forall has "copy-in-copy-out" semantics, also HPF array assignments

HPF Heat Distribution Problem Red-black Ordering

```
!HPF$ PROCESSORS pr(4)
      REAL h(100,100)
!HPF$ DISTRIBUTE h(BLOCK,*) ONTO pr
      …
      FORALL (i=2:99, j=2:99, MOD(i+j,2).EQ.0)
        h(i,j) = 0.25*(h(i-1,j)+h(i+1,j)+h(i,j-1)+h(i,j+1))
      FORALL (i=2:99, j=2:99, MOD(i+j,2).EQ.1)
        h(i,j) = 0.25*(h(i-1,j)+h(i+1,j)+h(i,j-1)+h(i,j+1))
```

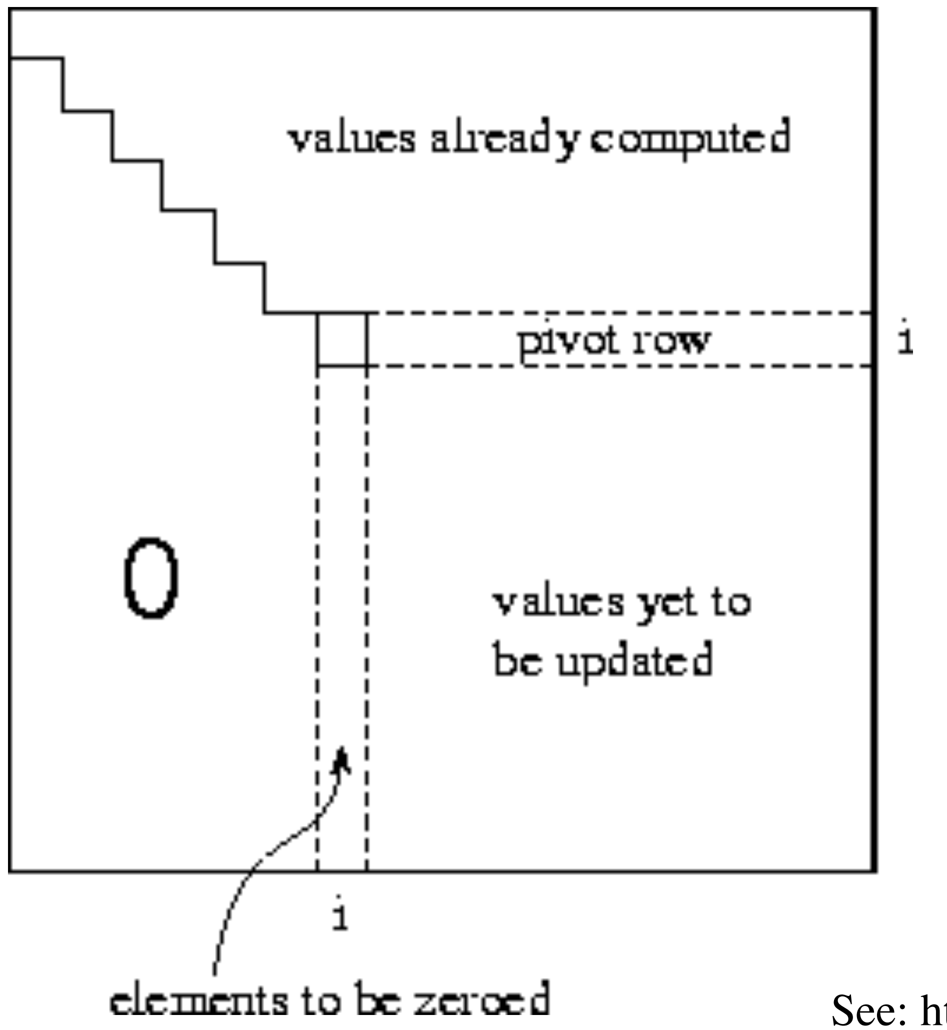
HPF Iterative Solver: Jacobi Method

```
!HPF$ PROCESSORS pr(4)
      REAL a(100,100),b(100),x(100)
!HPF$ ALIGN x(:) WITH a(:,*)
!HPF$ ALIGN b(:) WITH x(:)
!HPF$ ALIGN s(:) WITH x(:)
!HPF$ DISTRIBUTE a(BLOCK,*) ONTO pr
      x = b
      FORALL (i = 1:n) s(i) = SUM(a(i,:)*x(:))
      DO iter = 0,limit
        FORALL (i = 1:n) x(i) = (b(i) - s(i) + a(i,i)*x(i)) / a(i,i)
        FORALL (i = 1:n) s(i) = SUM(a(i,:)*x(:))
        IF (MAXVAL(ABS(s - b)) < tolerance) EXIT
      ENDDO
```

$$x_i^k = \frac{1}{a_{i,i}}\left[b_i - \sum_{j \neq i} a_{i,j} x_j^{k-1}\right]$$

$$\left|\sum_{j=0}^{n-1} a_{i,j} x_j^k - b_i\right| < \epsilon \qquad \forall i = 0, \ldots, n-1$$

Gaussian Elimination



- The original system of equations is reduced to an upper triangular form

  $$\boldsymbol{Ux} = \boldsymbol{y}$$

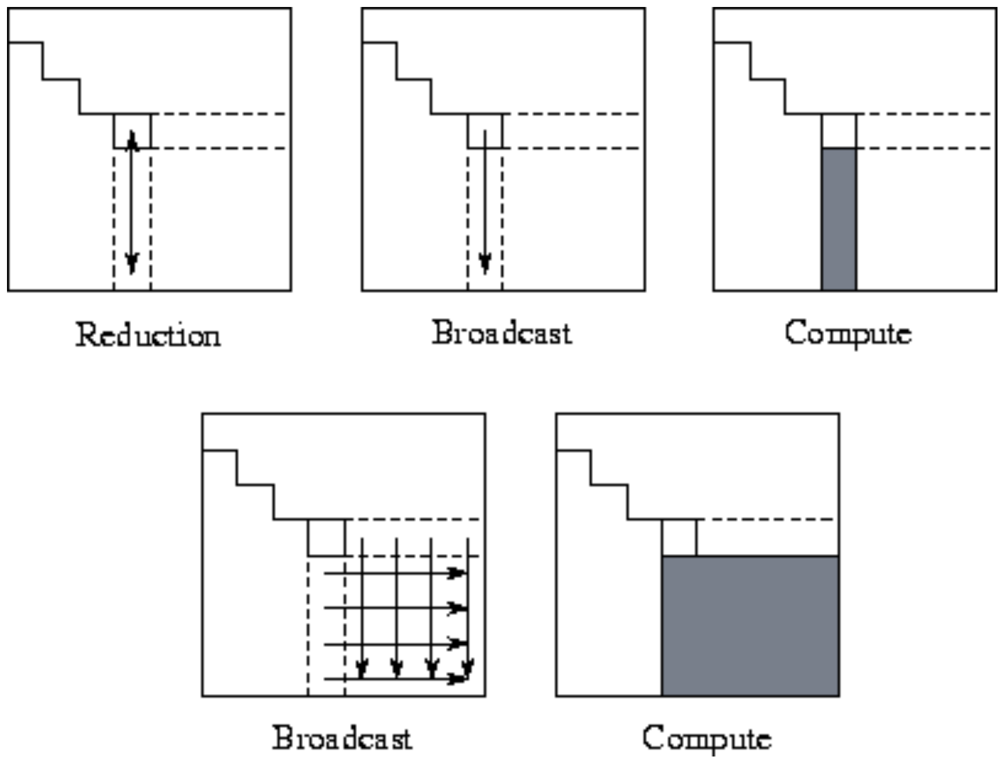
  where U is a matrix of size $N \times N$ in which all elements below the diagonal are zero, and diagonal elements have the value 1
- Back substitution (Gauss-Jordan elimination): the new system of equations is solved to obtain the values of $\boldsymbol{x}$

See: http://www-unix.mcs.anl.gov/dbpp/text/node82.html

HPF Gaussian Elimination 1

```
      REAL A(n,n+1), X(n), Fac(n), Row(n+1)
      INTEGER indx(n), itmp(1), max_indx, i, j, k
!HPF$ ALIGN Row(j) WITH A(1,j)
!HPF$ ALIGN X(i) WITH A(i,N+1)
!HPF$ DISTRIBUTE A(*,CYCLIC)
      indx = 0
      DO i = 1,n
        itmp = MAXLOC(ABS(A(:,i)), MASK=indx.EQ.0)  ! Stage 1
        max_indx = itmp(1)                          ! Stage 2
        indx(max_indx) = i
        Fac = A(:,i) / A(max_indx,i)                ! Stage 3+4
        Row = A(max_indx,:)
        FORALL (j=1:n, k=i:n+1, indx(j).EQ.0)       ! Stage 5
          A(j,k) = A(j,k) - Fac(j)*Row(k)
      ENDDO
!     Row exchange
      FORALL (j=1:n) A(indx(j),:) = A(j,:)
!     Backsubstitution, uses B(:) stored in A(1:n,n+1)
      DO j = n,1,-1
        X(j) = A(j,n+1) / A(j,j)
        A(1:j-1,n+1) = A(1:j-1,n+1) - A(1:j-1,j)*X(j)
      ENDDO
```

HPF Gaussian Elimination 2



- Computing the upper triangular form takes five stages:
  1. Reduction with **`MAXLOC`**
  2. Broadcast (copy) **`max_indx`**
  3. Compute scale factors **`Fac`**
  4. Broadcast scale factor **`Fac`** and pivot row value **`Row(k)`**
  5. Row update with **`FORALL`**