Lecture 5

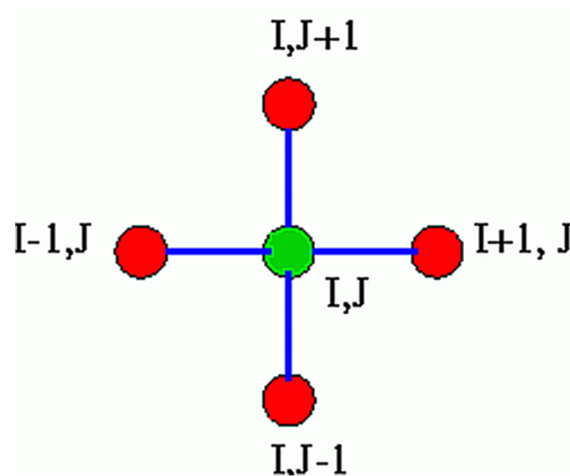
Programming with openmp

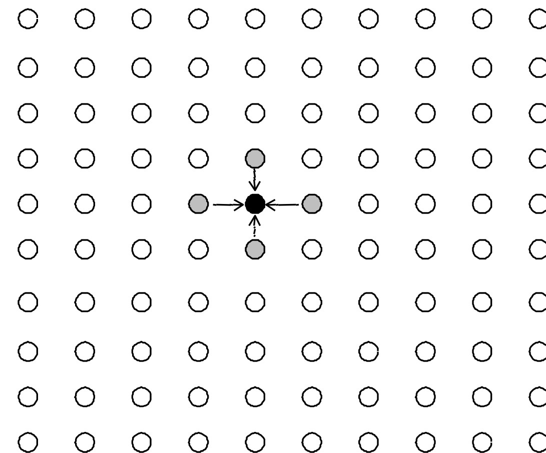
Jacobi's Method in 2D

- The update formula

```
for (i,j) in 0:N-1 x 0:N-1
  u'[i,j] = (u[i-1,j] + u[i+1,j] +
             u[i,j-1] + u[i, j+1] - h²f[i,j])/4
u =u'
```

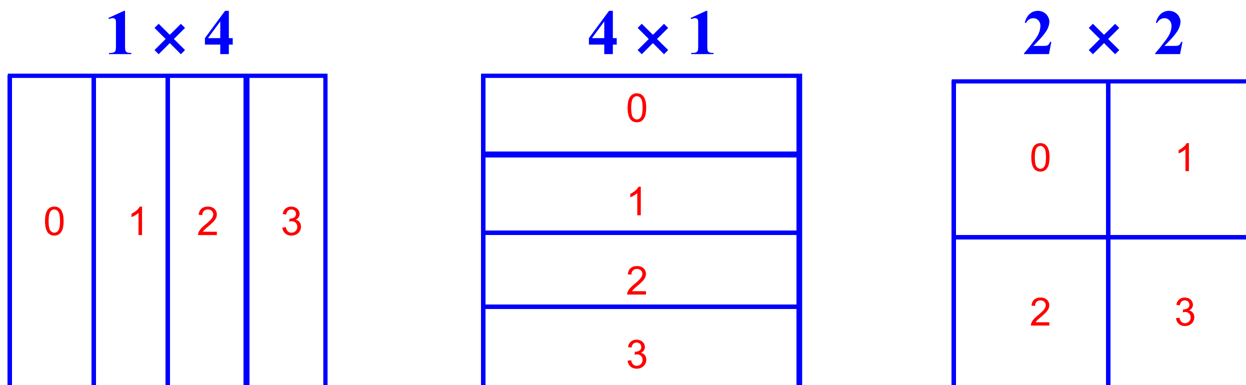
$$\begin{matrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{matrix}$$





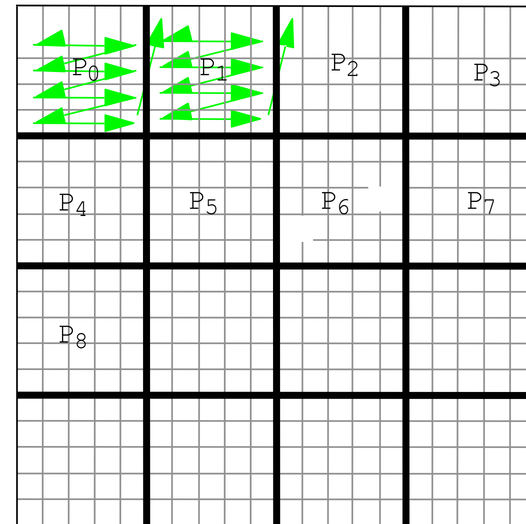
Partitioning

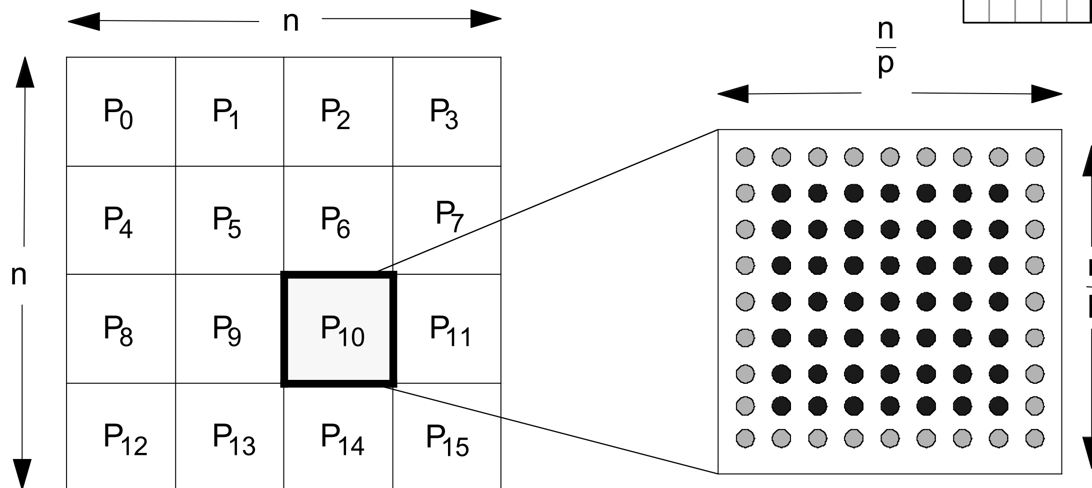
- Splits up the data over processors
- Different partitionings according to the ***processor geometry***
- For P processors geometries are of the form $p_0 \times p_1$, where $P = p_0\, p_1$
- For P=4, 3 possible geometries



Data access

- Off processor values surround each local subproblem
- Non-contiguous data
- Inefficient to access values on certain faces/edges





Multithreaded Solve()

```
Local mymin = 1 + ($TID * n/$nprocs),
      mymax = mymin + n/$nprocs -1;
Global resid, U[:,:], U^new[:,:]
Local  done = FALSE;
while (!done) do
    Local myResid = 0;
    resid = 0;
    update U^new and myResid
    resid += myResid;
    if (resid  < Tolerance) done = TRUE;
U[mymin:mymax,:] = U^new[mymin:mymax,:];
end while
```

```
for i = mymin to mymax do
        for j = 1 to n do
            U^new[i,j] = …
            myresid+= …
        end for
end for
```

0 1 2 3

0
1
2
3

Is this code correct?

Multithreaded Solve()

```
Local mymin = 1 + ($TID * n/$nprocs),
      mymax = mymin + n/$nprocs -1;
Global resid, U[:,:], Unew[:,:]
Local  done = FALSE;
while (!done) do
    Local myResid = 0;
    BARRIER
    Only on thread 0:  resid = 0;
    BARRIER
    update Unew and myResid
    CRITICAL SEC: resid += myResid
    BARRIER
    if (resid  < Tolerance) done = TRUE;
   Only on thread 0: U[mymin:mymax,:] = Unew[mymin:mymax,:];
end while
```

```
for i = mymin to mymax do
        for j = 1 to n do
            Unew[i,j] = …
            myresid+= …
        end for
 end for
```


Does this code use minimal synchronization?

OpenMP programming

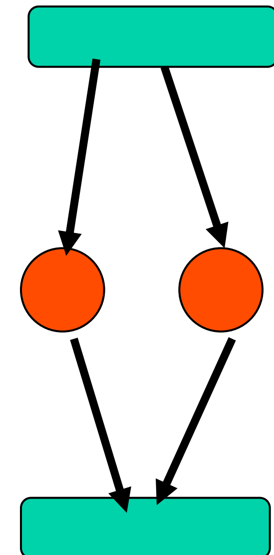
- Simpler interface than explicit threads
- Parallelization handled via annotations
- See http://www.openmp.org
- Parallel loop:

```
#pragma omp parallel private(i) shared(n)
{
#pragma omp for
for(i=0; i < n; i++)
    work(i);
}
```

```
i0 = $TID*n/$nthreads;
i1 = i0 + n/$nthreads;
for (i=i0; i< i1; i++)
   work(i);
```

Parallel Sections

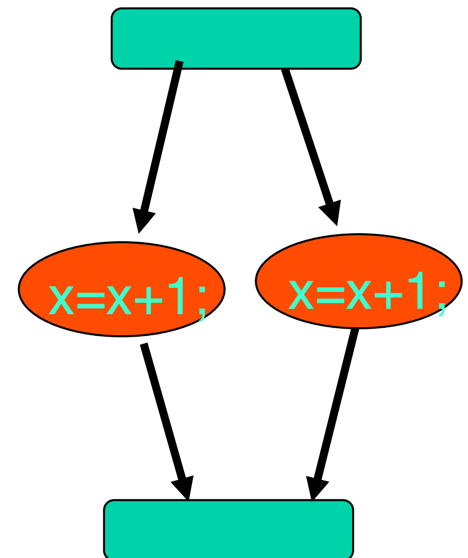
```
#pragma omp parallel // Begin a parallel construct
{ // form a team
  // Each team member executes the same code
  #pragma omp sections    // Begin work sharing
  {
    #pragma omp section    // A unit of work
    {x = x + 1;}

    #pragma omp section    // Another unit
    {x = x + 1;}

  } // Wait until both units complete

} // End of Parallel Construct; disband team

// continue serial execution
```



Critical Sections

- Only one thread at a time may run the code in a critical section
- Uses mutual exclusion to implement critical sections

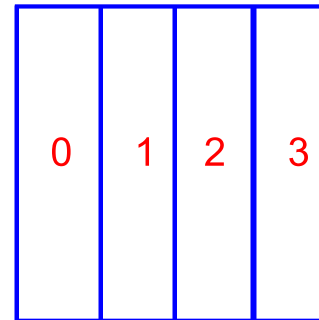
```
#pragma omp parallel // Begin a parallel construct
{
 #pragma omp sections    // Begin worksharing
 { //
   #pragma omp critical    // Critical section
   {x = x + 1}
   #pragma omp critical    // Another critical section
   {x = x + 1}
 ... // More Replicated Code
  #pragma omp barrier // Wait for all members to arrive
   } // Wait until both units of work complete
}
```

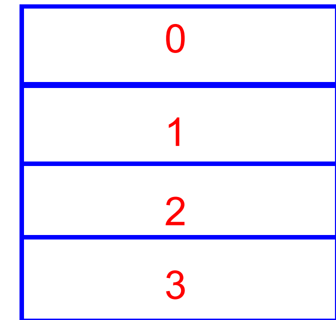


Workload decomposition in OpenMP

- We use static assignment here, since n is known
- Dynamic assignment for irregular problems (later on)
- Translator automatically generates appropriate local loop bounds

```
#pragma omp parallel private(i) shared(n)
{
#pragma omp for
for(i=0; i < n; i++)
    work(i);
}
```





- Here we parallelize the outer loop index

```
#pragma omp parallel private(i) shared(n)
  #pragma omp for
  for(i=0; i < n; i++)
     for(j=0; j < n; j++) {
       unew[i,j] = (u[i-1,j] + u[i+1,j]+ u[i,j-1]+ u[i, j+1] - h2f[i,j])/4
     }
```

$u^{new}[i,j] = (u[i-1,j] + u[i+1,j] + u[i,j-1] + u[i,j+1] - h^2 f[i,j])/4$

Parallelization via OpenMP

- We parallelize the outer loop index

```
#pragma omp parallel private(i) shared(n)
   #pragma omp for
   for(i=0; i < n; i++)
      for(j=0; j < n; j++) {
        unew[i,j] = (u[i-1,j] + u[i+1,j]+ u[i,j-1]+ u[i, j+1] - h2f[i,j])/4
   }
```

- Generated code

```
   mymin = 1 + ($TID * n/nprocs),     mymax = mymin + n/nprocs -1
   for(i=mymin; i < mymax; i++)
      for(j=0; j < n; j++) {
        unew[i,j] = (u[i-1,j] + u[i+1,j]+ u[i,j-1]+ u[i, j+1] - h2f[i,j])/4
   }
```

Reductions in OpenMP

- OpenMP uses a local accumulator, which it then accumulates globally when the loop is over

```
#pragma omp parallel reduction(+:sum)
  for (int i=i0; i< i1; i++)
      sum += x[i];
```

Jac3D in OpenMP

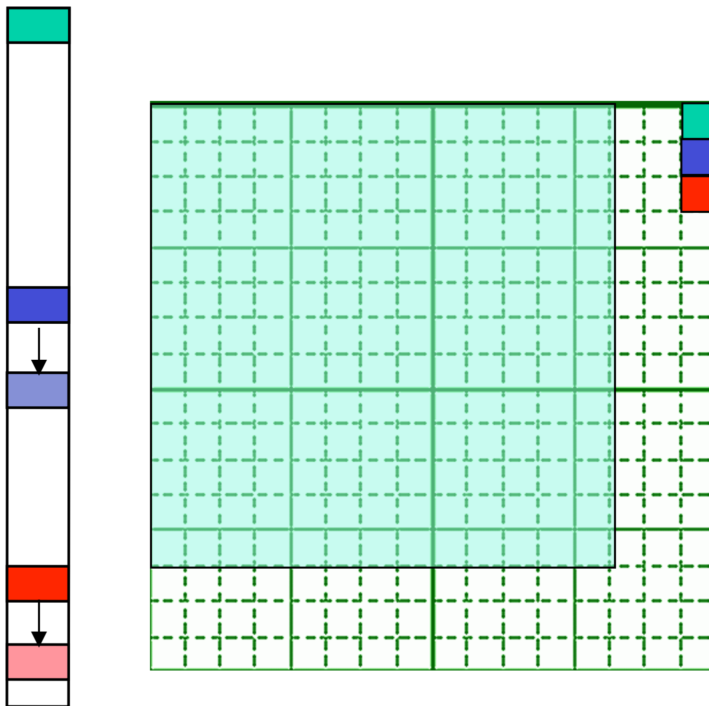
```
#ifdef _OPENMP
#include <omp.h>

int nthreads = 1;
#pragma omp parallel
{
   int tid = omp_get_thread_num();
   if (tid == 0) {
      nthreads = omp_get_num_threads();
      printf("Number of openMP threads: %d\n", nthreads);
   }
}
#endif
```

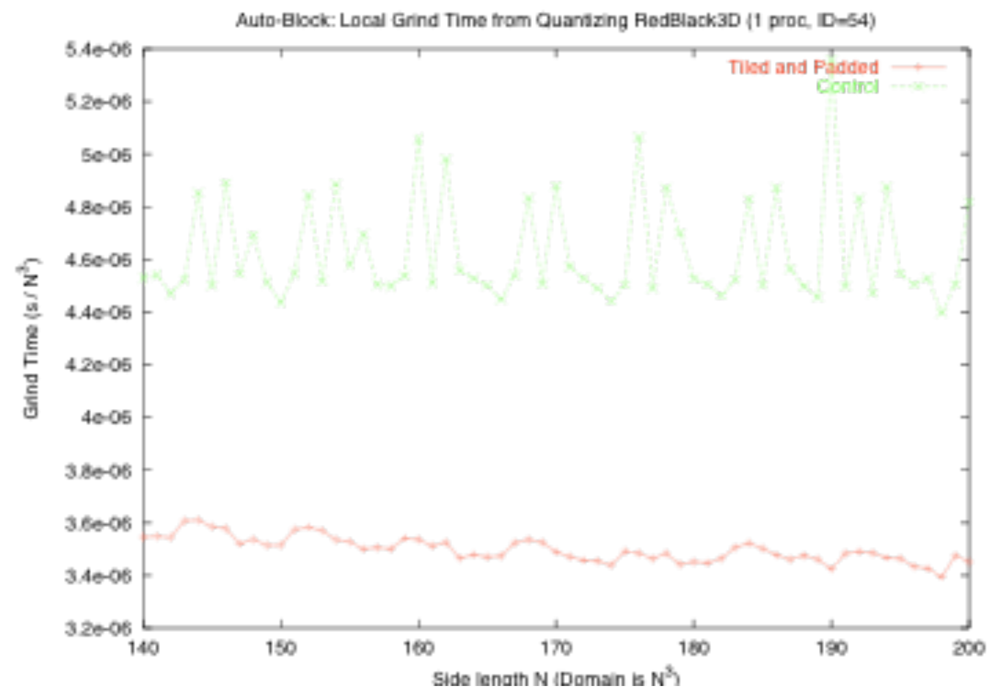
Jac3D in OpenMP

```
#pragma omp for schedule(static,bi)
for (i=1; i<nx1; i++)
     for (j=1; j<ny1; j++)
          for (k=1; k<nz1; k++)
               Un[i][j][k] = (U[i-1][j][k] + U[i+1][j][k] + U[i][j-1][k] + U[i][j+1][k] +
                                   U[i][j][k-1] + U[i][j][k+1]  - h*h*b[i-1][j-1][k-1]) / 6.0;
#pragma omp parallel shared(U,B,c)
#pragma omp for reduction(+:err) for (int i=1; i<=nx; i++)
for (int j=1; j<=ny; j++)
    for (int k=1; k<=nz; k++){
         double du = c * (U[i-1][j][k] + U[i+1][j][k] + U[i][j-1][k] +
                                   U[i][j+1][k] + U[i][j][k-1] + U[i][j][k+1] - 6.0*b[i-1][j-1][k-1]);
         double  r = b[i-1][j-1][k-1] - du;
         err = err +  r*r;
    }
return sqrt(err)/(float)((nx+1)*(ny+1)*(nz+1));
}
```

Reducing conflict misses

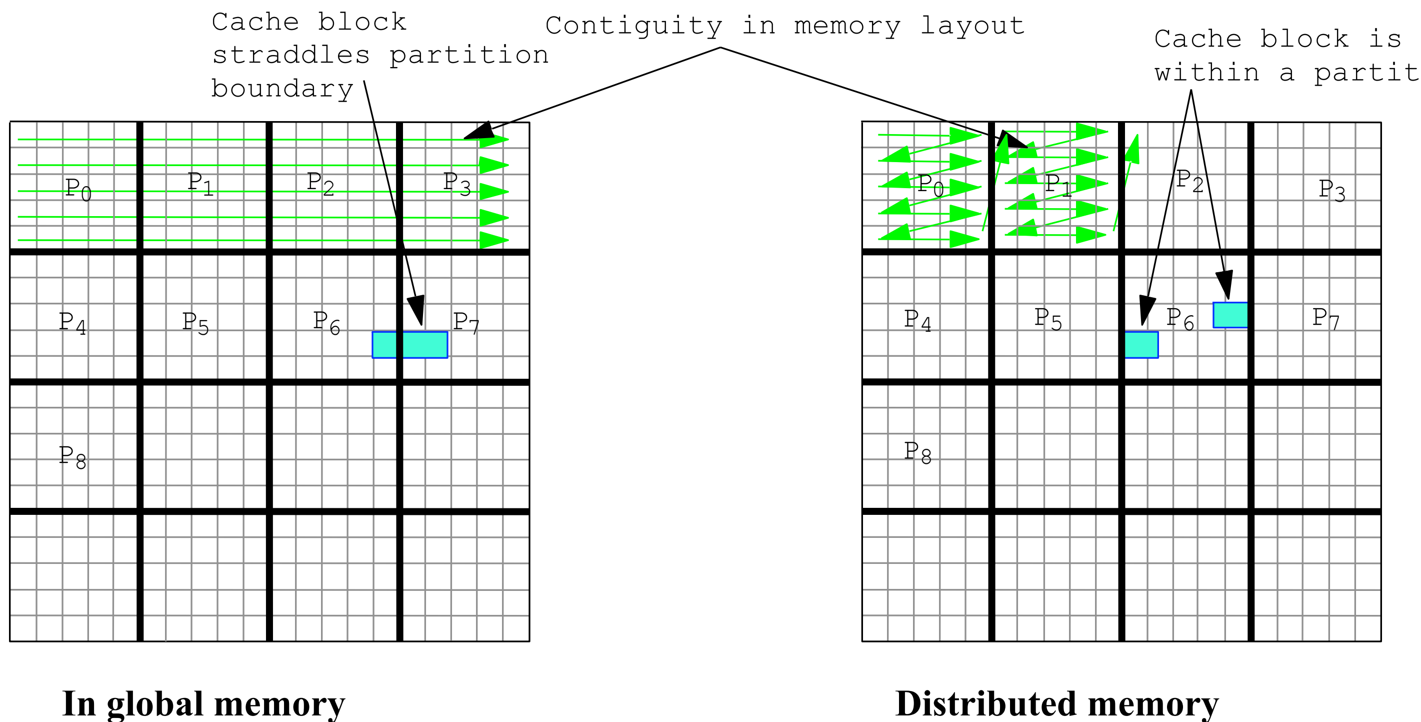
- Pad the array with unused cells to change the memory access patterns
- Rivera & Tseng [Sigplan, 1998]
- Any other ways?





False sharing and conflict misses

- Boundary values, false sharing
- Large memory access strides, conflict misses
- Compare with distributed memory solution



Parallel Computer Architecture, Culler, Singh, & Gupta

Eliminating false sharing

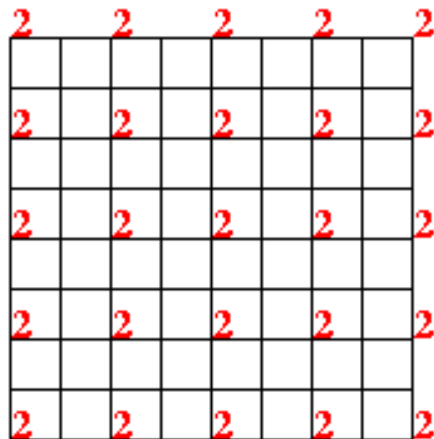
- Cleanly separate locations updated by different processors
  - Manually assign scalars to a pre-allocated region of memory using pointers
  - With a block partitioned array, we want partition boundaries to coincide with a cache line boundary
- Compilers can perform some of these optimizations

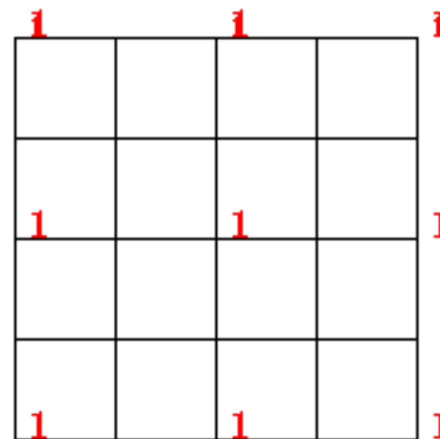
Assignment #3

- Threaded implementation of a 3D Poisson Solver
- Serial and Matlab code are provided
- If you work in a team of 3, use the convergence rate accelerator

Accelerating the solve

- We initialize the solver with a guess of the solution
- The better the guess, the faster we converge
- Idea: let’s down sample the solution and solve on a coarser mesh
- Reduces computation time significantly





The ingredients

- The solution operator S(i) takes $U^{(i)}$ and computes an improved solution $U_{improved}(i)$ on same grid

  $u_{improved}(i) = S(i)(f(i), U(i))$

- The restriction operator R(i) maps $U^{(i)}$ to $U^{(i-1)}$

  Restricts problem on fine grid $U^{(i)}$ to coarse grid $U^{(i-1)}$ by subsampling or averaging grids of size 2^i-1

  $f(i-1) = R(i)(f(i))$

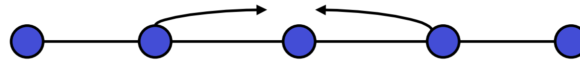
- The prolongation (interpolation) operator P(i-1) maps an approximate solution $U^{(i-1)}$ to $U^{(i)}$

  Interpolates (upsamples) solution on coarse grid $U^{(i-1)}$ to fine grid $U^{(i)}$

  $U(i) = P(i-1)\, U^{(i-1)}$

The Solution Operator S(i)

- The solution operator, S(i), is a weighted Jacobi
- Consider the 1D problem



- Pure Jacobi update:

  $x^{new}(j) := 1/2\ (u(j-1) + u(j+1) + b(j)\ )$
- Weighted Jacobi update:

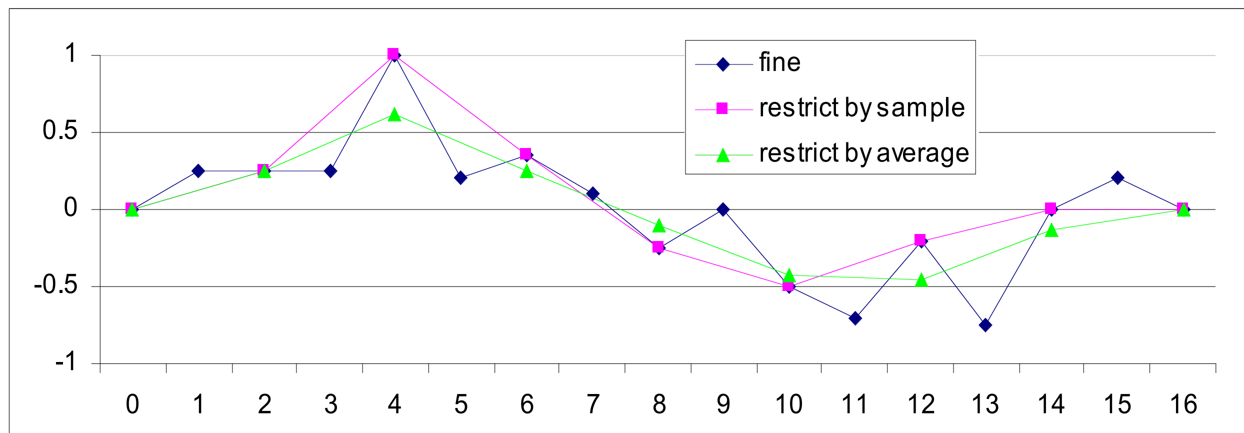
  $x^{new}(j) := 1/3\ (u(j-1) + u(j) + u(j+1) + b(j)\ )$

- In 2D, similar average of nearest neighbors

The Restriction Operator R(i)

- The restriction operator, R(i), takes
  - a problem $U^{(i)}$ with RHS $f^{(i)}$ and
  - maps it to a coarser problem $U^{(i-1)}$ with RHS $f^{(i-1)}$
  - Averaging or sampling
- Average values of neighbors

$$u_{coarse}(i) = \frac{1}{4}u_{fine}(i-1) + \frac{1}{2}u_{fine}(i) + \frac{1}{4}u_{fine}(i+1)$$



Courtesy Jim Demmel and Katherine Yelick

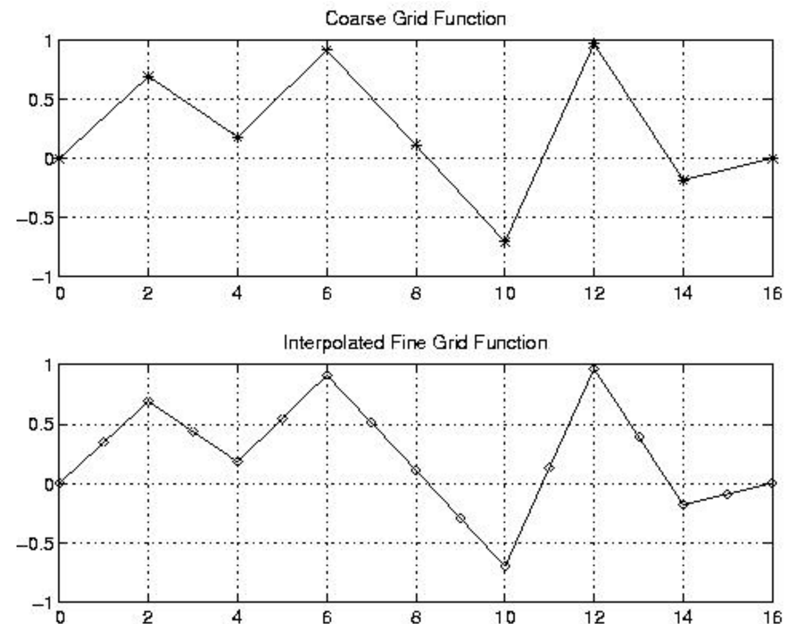
Prolongation Operator P(i)

- The prolongation operator P(i-1) converts a coarse grid solution $U^{(i-1)}$ to a fine grid $U^{(i)}$
- In 1D: linearly interpolate nearest coarse neighbors

  $u_{fine}(i) = u_{coarse}(i)$ if the fine grid point i is also a coarse one,

  else

  $u_{fine}(i) = ½(u_{coarse}(\text{left of } i) + u_{coarse}(\text{right of } i))$



Courtesy Jim Demmel