

2D Heat Distribution Prediction in Parallel

Using Finite Difference

Brett D. Estrade

estrabd@mailcan.com

USM Fall 2004

The Finite Difference Method_[1]

- Approximates PDEs by a difference operator for some small but finite h
- h is the distance between points on the domain
- Doing this substitution for a large enough number of points in the domain gives a system of equations that can be solved algebraically
- The error between this approximate solution and the true solution is determined by the truncation error that is made by going from a differential operator to a difference operator

Problem Description

- Goal: predict the heat distribution in a 2D domain resulting from conduction
- Heat distribution can be described using the following partial differential equation (PDE):

$$u_{xx} + u_{yy} = f(x,y)$$

- $f(x,y) = 0$ since there are no internal heat sources in this problem
- There is only 1 heat source at a single boundary node, and all other nodes are to be held at 0 degrees. These are the boundary conditions
- Initial temperature of the entire domain is set to 0.0

Problem Description

T_0



$$u_{xx} + u_{yy} = f(x, y)$$

Numerical Methods

- The domain is evenly decomposed into a rectangular mesh where the nodes are evenly spaced h units apart in both directions: $\Delta x = \Delta y = h$
- The solution at each node is then computed in an iterative fashion until there is little change over the entire domain in the overall temperature distribution
- This process depends on the numerical method's ability to drive some test for convergence to 0: $\|u^{(k+1)} - u^{(k)}\|_2 < \varepsilon$
- Once the method has converged to zero or some acceptable tolerance, the solution is said to approximate the temperature distribution

Iterative Process

Applicable to both serial and parallel implementations

```
while (! converged (U_K,U_K_1)) {  
    /* update U_K which holds solution from previous iteration */  
    U_K = U_K_1;  
    solve(U_K,U_K_1);  
}
```

Where:

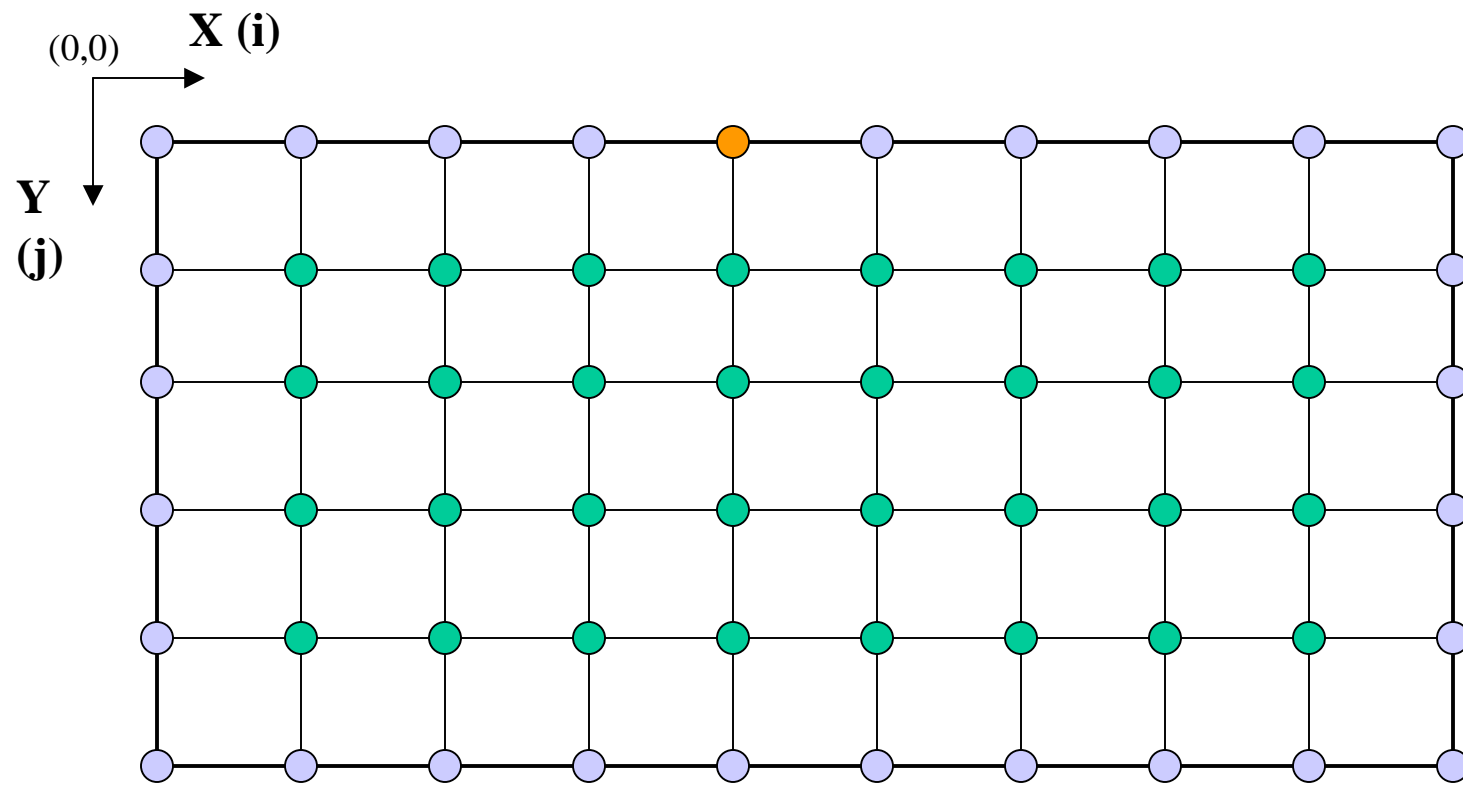
- U_K is a 2d array holding the solution for each node in the domain from the previous iteration
- U_K_1 is a 2d array that is storing the results for each node as it is being solved

Testing for Convergence

$$\|u^{(k+1)} - u^{(k)}\|_2 < \epsilon$$

```
sum = 0.0;
for (i=0;i<num_cols;i++) {
    for (j=0;j<num_rows;j++) {
        sum = sum + (U_K_1[I][j] - U_K[I][j])2;
    }
}
L2_Norm = sqrt(sum);
```

Domain Decomposition



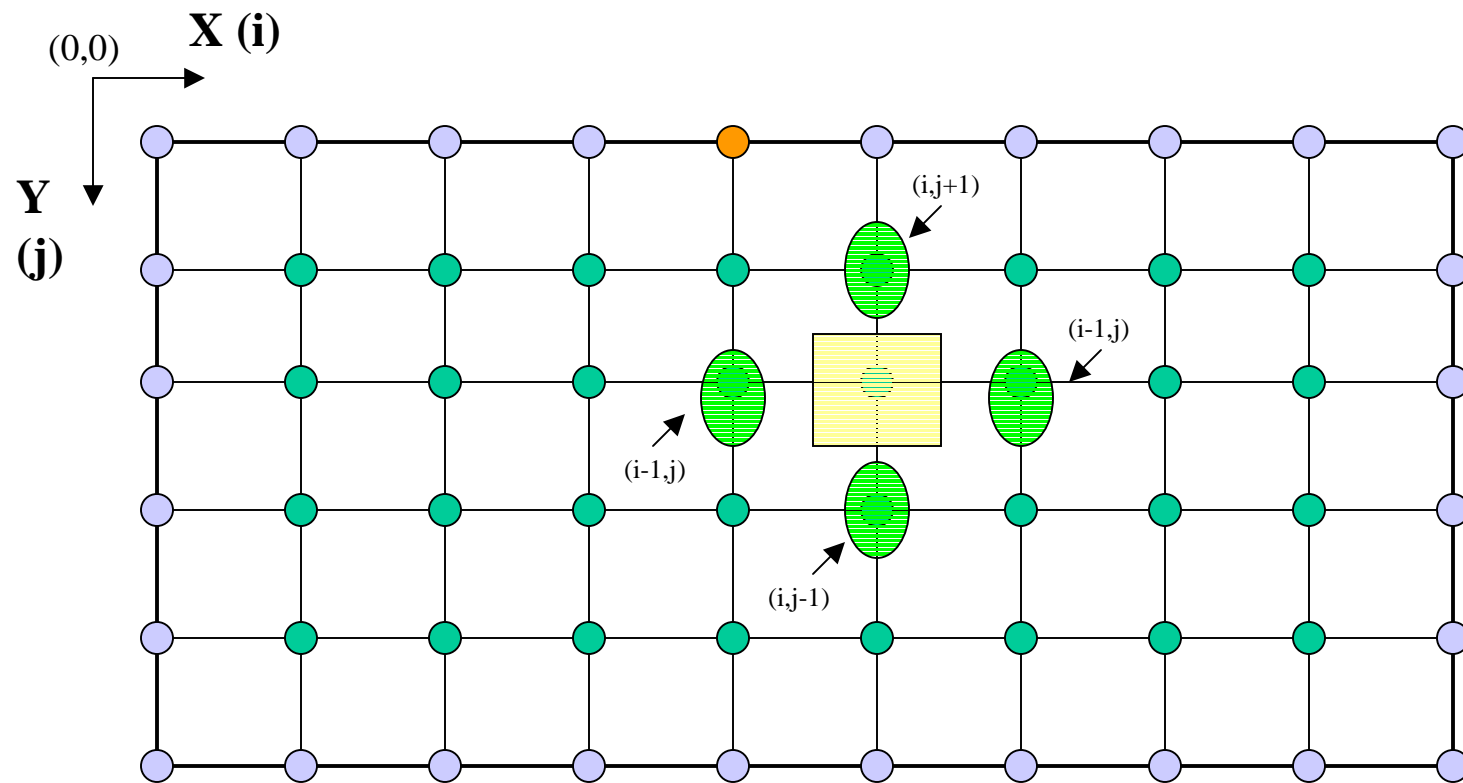
bc, T_0 - ● bc, fixed @ 0 - ● interior node - ●

Jacobi Method

- Takes what amounts to an average of the surrounding 4 nodes to figure out what the temperature should be for the node value being calculated
- Takes into account internal heat sources
- Relies strictly on values from the previous iteration
- Each node could theoretically be handled by its own process, computing the entire domain in one step
- If each node was calculated on its own processor, then the upper bound for communications would be $4 * \text{Num_Nodes}$, but in practice the number of communications should be lower if bcs are handled locally
- This also means that the maximum number of concurrent calculations is equal to the number of nodes.

Jacobi Method

$$u^{(k+1)}_{i,j} = (1/4) * (u^{(k)}_{i-1,j} + u^{(k)}_{i+1,j} + u^{(k)}_{i,j-1} + u^{(k)}_{i,j+1} - h^2 f_{i,j})$$



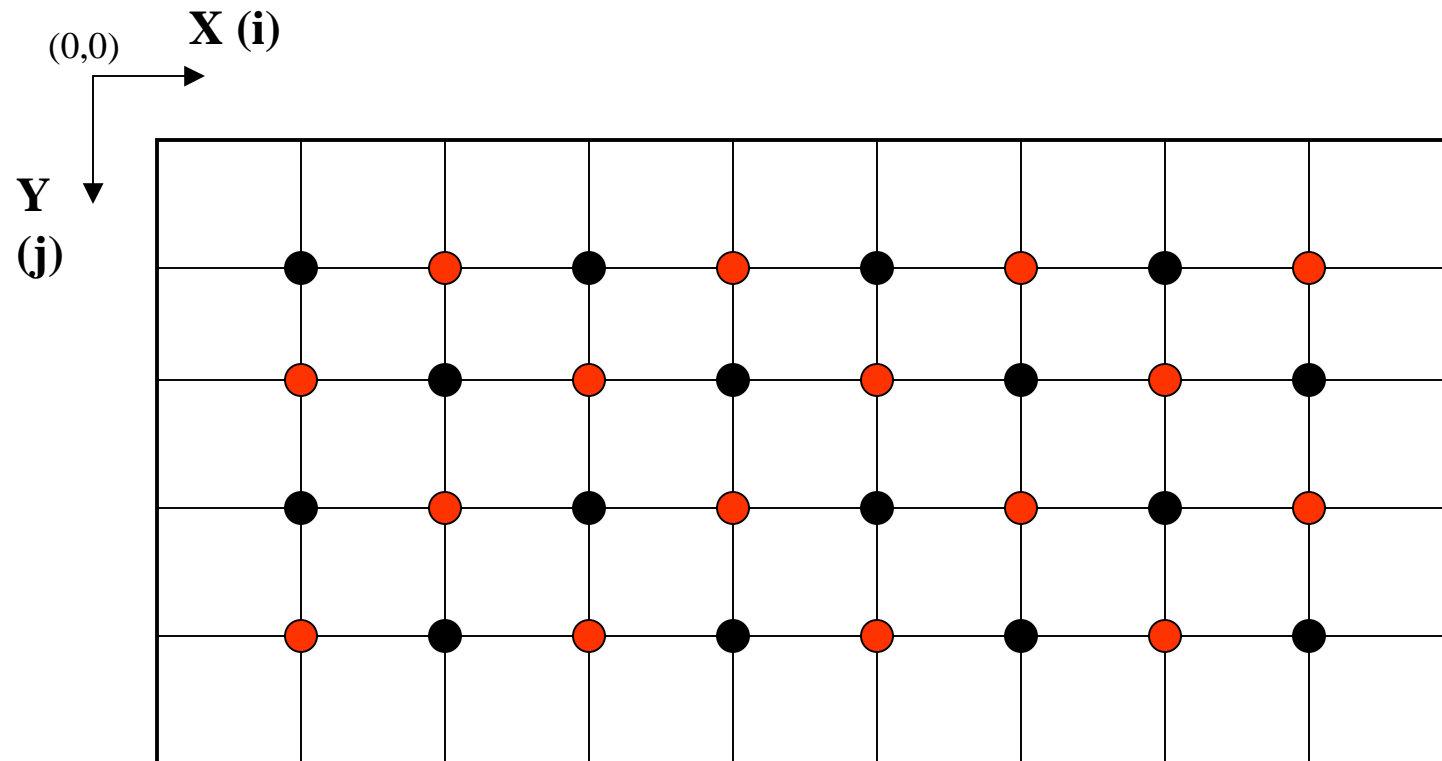
bc, T_0 - ● bc, fixed @ 0 - ● interior node - ●

Gauss-Seidel and SOR Methods

- Both methods are the same except for the constant “W” that is used to scale or “relax” the results.
- They are also both iterative processes, and converge faster than the Jacobi method.
- At each point (i,j) in the domains is donated a red or a black according to whether $i+j$ is odd or even.
- The “red” node calculations use values from the previous iteration.
- The “black” node calculations use values donated from the current iterations “red” nodes, so there is a task dependency here.

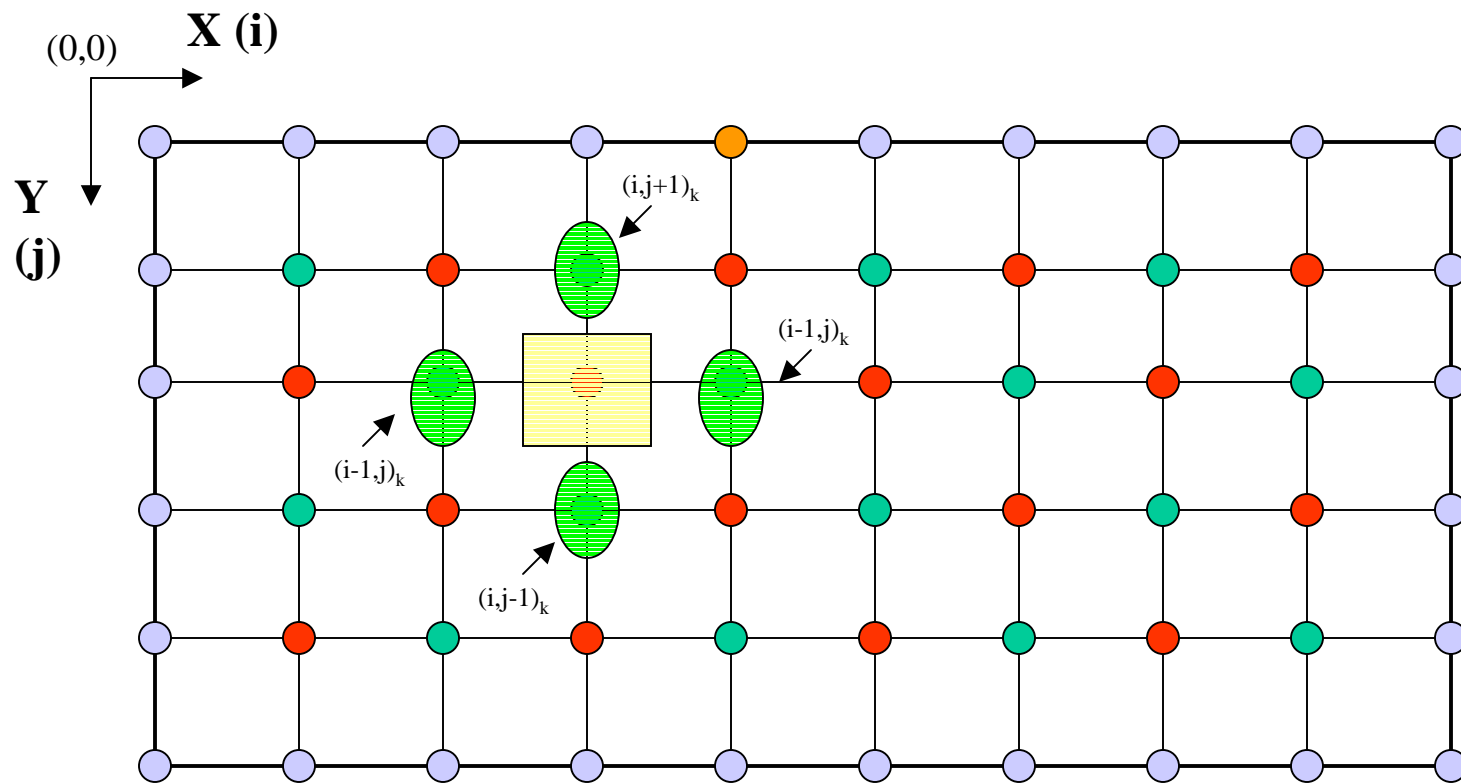
Gauss-Seidel and SOR Methods

Red/black interior nodal distribution



Gauss-Seidel and SOR Methods

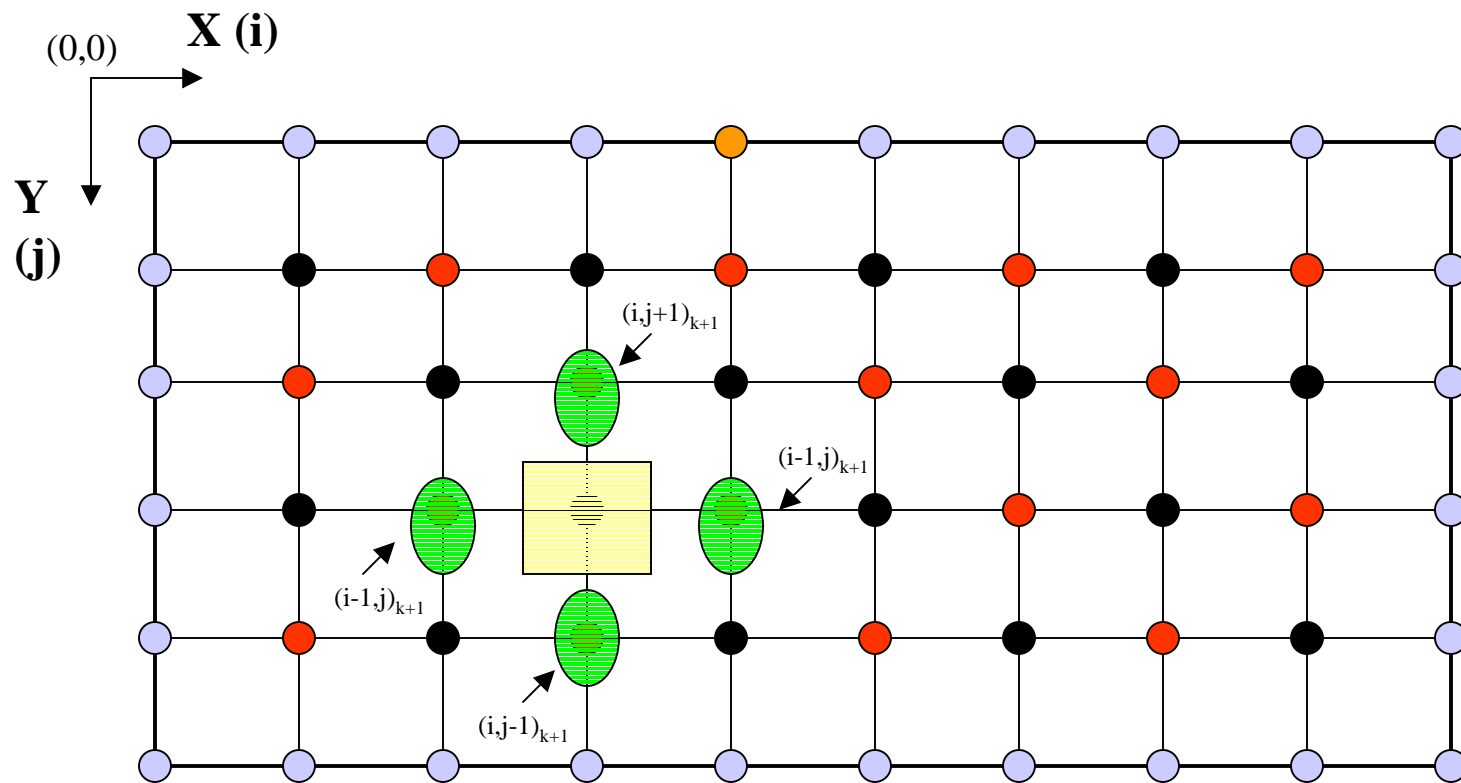
$$u^{(k+1)}_{i,j} = u^{(k)}_{i,j} + (1/W) * (u^{(k)}_{i-1,j} + u^{(k)}_{i+1,j} + u^{(k)}_{i,j-1} + u^{(k)}_{i,j+1} - 4u^{(k)}_{i,j} - h^2 f_{i,j})$$



bc, T₀ - ● bc, fixed @ 0 - ● interior node - ● node being solved - ●

Gauss-Seidel and SOR Methods

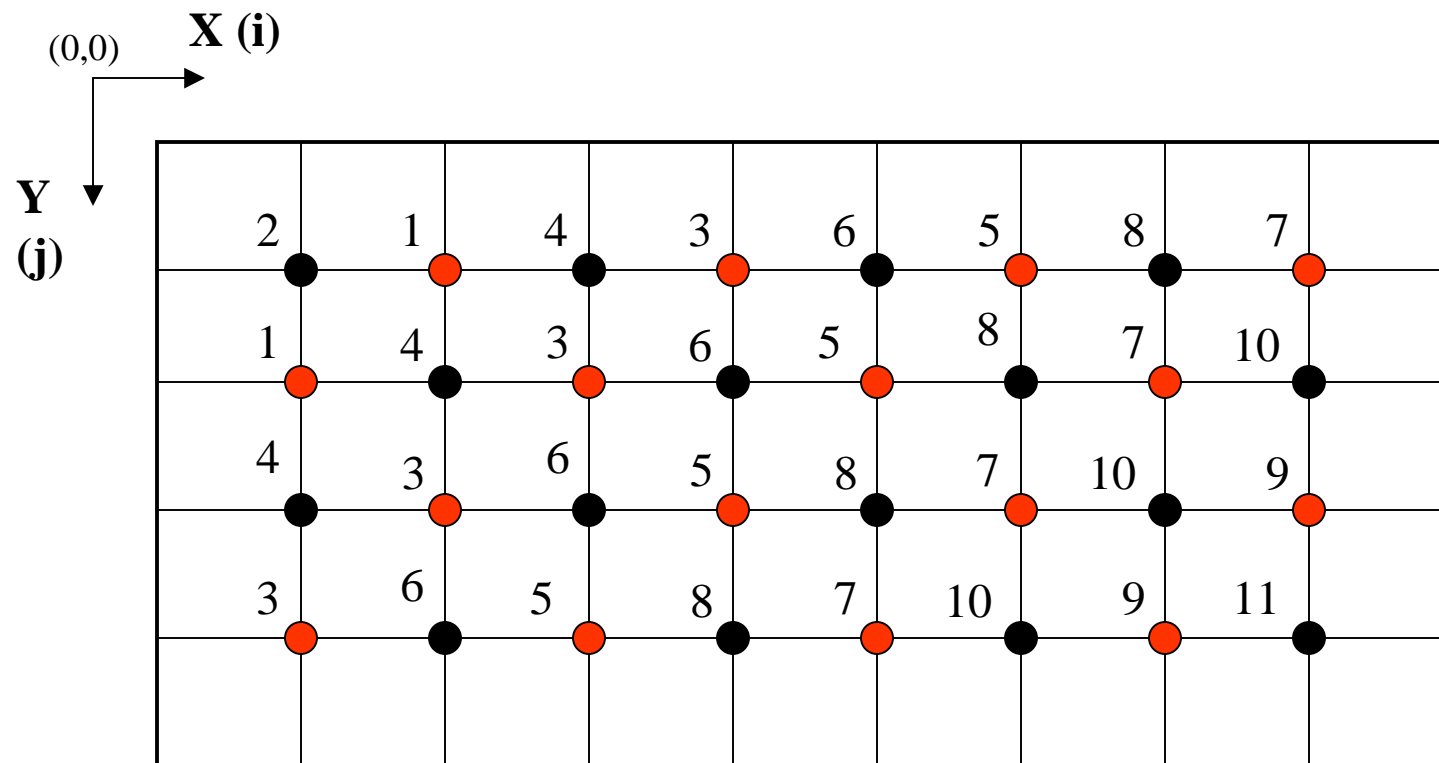
$$u^{(k+1)}_{i,j} = u^{(k+1)}_{i,j} + (1/W) * (u^{(k+1)}_{i-1,j} + u^{(k+1)}_{i+1,j} + u^{(k+1)}_{i,j-1} + u^{(k+1)}_{i,j+1} - 4u^{(k)}_{i,j} - h^2 f_{i,j})$$



bc, T_0 - ● bc, fixed @ 0 - ● interior node - ● node being solved - ●

Gauss-Seidel and SOR Methods

Task Dependency



Iterative Process For Red/Black

A non-ideal method was used for red/black calculations

```
while (! converged (U_K,U_K_1)) {  
    /* update U_K which holds solution from previous iteration */  
    U_K = U_K_1;  
    /* solve for red values using solution from previous iteration */  
    solve(U_K,U_K_1);  
    /* solve for black values using red values for this iteration */  
    solve(U_K_1,U_K_1);  
}
```

Where:

- U_K is a 2d array holding the solution for each node in the domain from the previous iteration
- U_K_1 is a 2d array that is storing the results for each node as it is being solved

Parallel Implementation

- Single Program Multiple Data
- Rows are evenly distributed to each processor
- No “global” matrix
- Communications are minimized by using “ghost rows”
- Convergence calculated using an MPI_Reduce, “MPI_SUM” reduction operation
- Boundary conditions enforced locally
- All serial results were identical to the parallel results of their respective methods

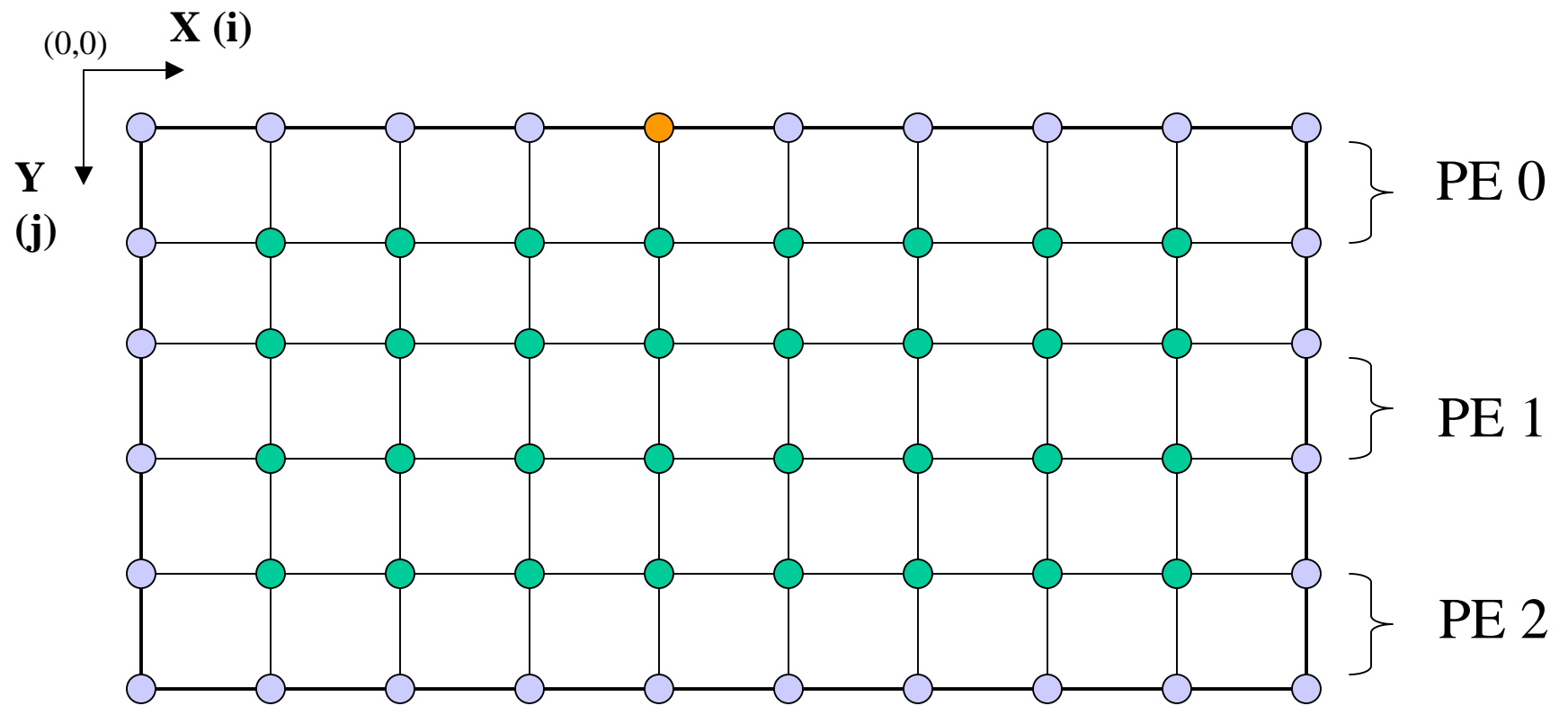
Single Program Multiple Data

- A single program that manages data distribution and number of processes is executed
- Rows are distributed to each processor evenly to improve load balancing
- Each process owns its rows
- For each process, communications are restricted to the processes operating on rows above and below it.
- Root node shares in the work load

Row Distribution

- Rows are distributed as evenly as possible
- The maximum number of rows owned by any process will be at most 1 more than the number of rows owned by a process with the fewest number of rows.
- This ensures proper load balancing and a minimization in idle processes

Row Distribution



bc, T_0 - ● bc, fixed @ 0 - ● interior node - ●

No Global Matrices

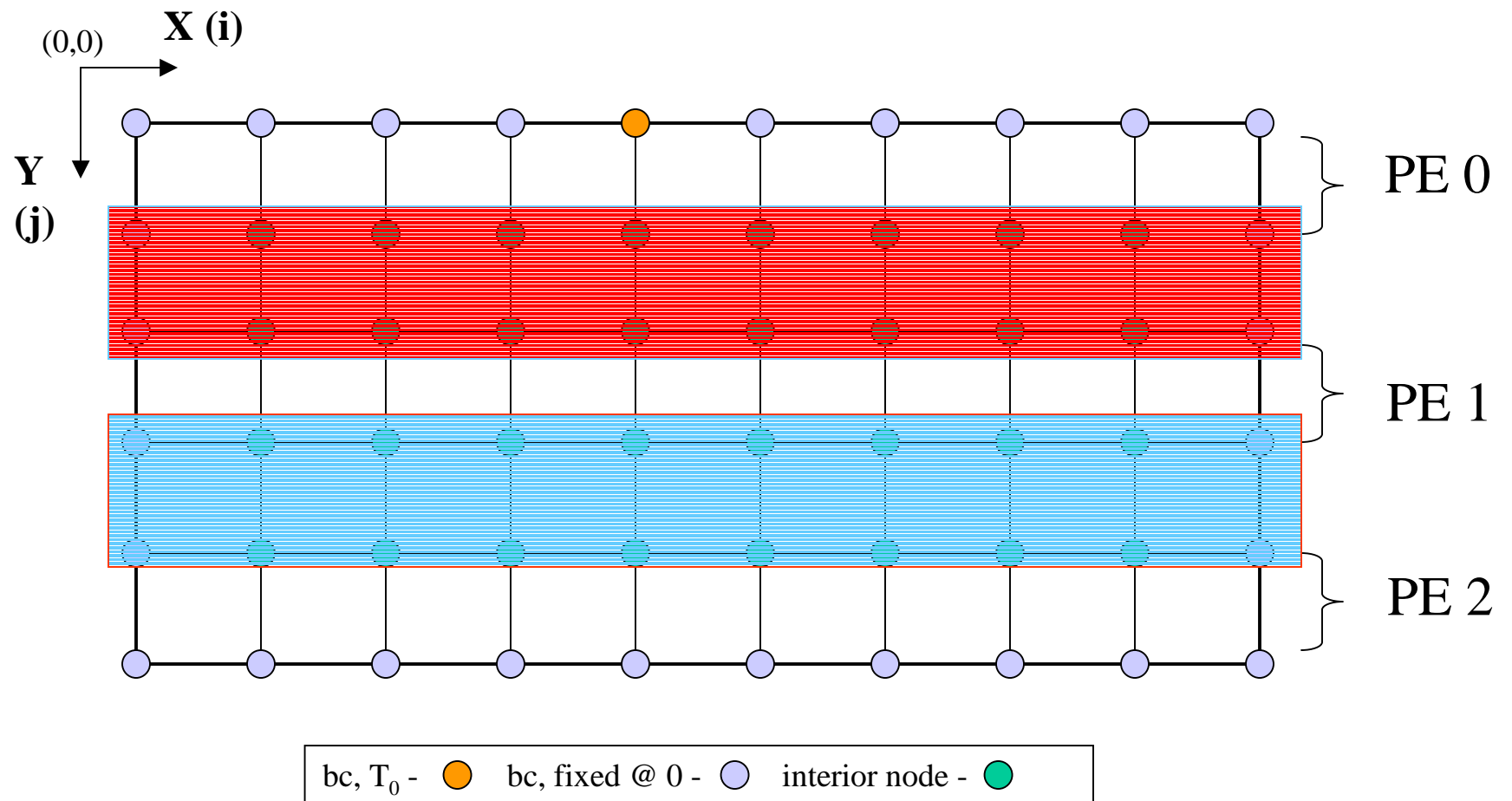
- Each process knows about its assigned rows and at most 2 more 1D “ghost rows” of a size corresponding to the number of nodes in the x-direction
- There is no globalization of the matrix at any point during the solving of the iterative method
- This greatly reduces the memory overhead associated with each processor containing a copy of the global matrix
- **Because of this approach, the serial and parallel algorithms are exactly the same.**

Ghost Rows

- Ghost rows are the rows that are either immediately above or immediately below a PE's own rows
- Rows are communicated between adjacent processors at the beginning of each iteration
- The root PE only receives one ghost row from below
- The last PE only sends one ghost row to the PE above it
- This results in a communication overhead of $2*(P-1)$ for each iteration
- Sends are non-blocking
- Receives are blocking

Ghost Rows

Color bands indicate ghost row communication seams



Ghost Row Communications

```
/* send/receive bottom rows */
if (my_rank < (p-1)) {
    /* populate send buffer with bottom row */
    for (i=0;i<(int)floor(WIDTH/H);i++) {
        U_Send_Buffer[i] = current_ptr[my_num_rows-1][i];
    }
    /* non blocking send */
    MPI_Isend(U_Send_Buffer,(int)floor(WIDTH/H),MPI_FLOAT,my_rank+1,0,MPI_COMM_WORLD,&request);
}
if (my_rank > ROOT) {
    /* blocking receive */
    MPI_Recv(U_Curr_Above,(int)floor(WIDTH/H),MPI_FLOAT,my_rank-1,0,MPI_COMM_WORLD,&status);
}
MPI_Barrier(MPI_COMM_WORLD);
```

Ghost Row Communications

```
/* send/receive top rows */
if (my_rank > ROOT) {
    /* populate send buffer with top row */
    for (i=0;i<(int)floor(WIDTH/H);i++) {
        U_Send_Buffer[i] = current_ptr[0][i];
    }
    /* non blocking send */
    MPI_Isend(U_Send_Buffer,(int)floor(WIDTH/H),MPI_FLOAT,my_rank-1,0,MPI_COMM_WORLD,&request);
}
if (my_rank < (p-1)) {
    /* blocking receive */
    MPI_Recv(U_Curr_Below,(int)floor(WIDTH/H),MPI_FLOAT,my_rank+1,0,MPI_COMM_WORLD,&status);
}
MPI_Barrier(MPI_COMM_WORLD);
```

Communication Costs

- The sending and receiving of the ghost rows is done at the beginning of each iterative method that amounts to a total Send/Receive cost of:

$$\text{Comm}_{\text{sendrecv}} = I * (2(P-1))$$

where I is the total number of iterations and P is the total number of PEs

- Each, MPI_Reduce is called once for the convergence calculations:

$$\text{Comm}_{\text{reduce}} = I$$

where I is the total number of iterations

- So, total the number of communications is:

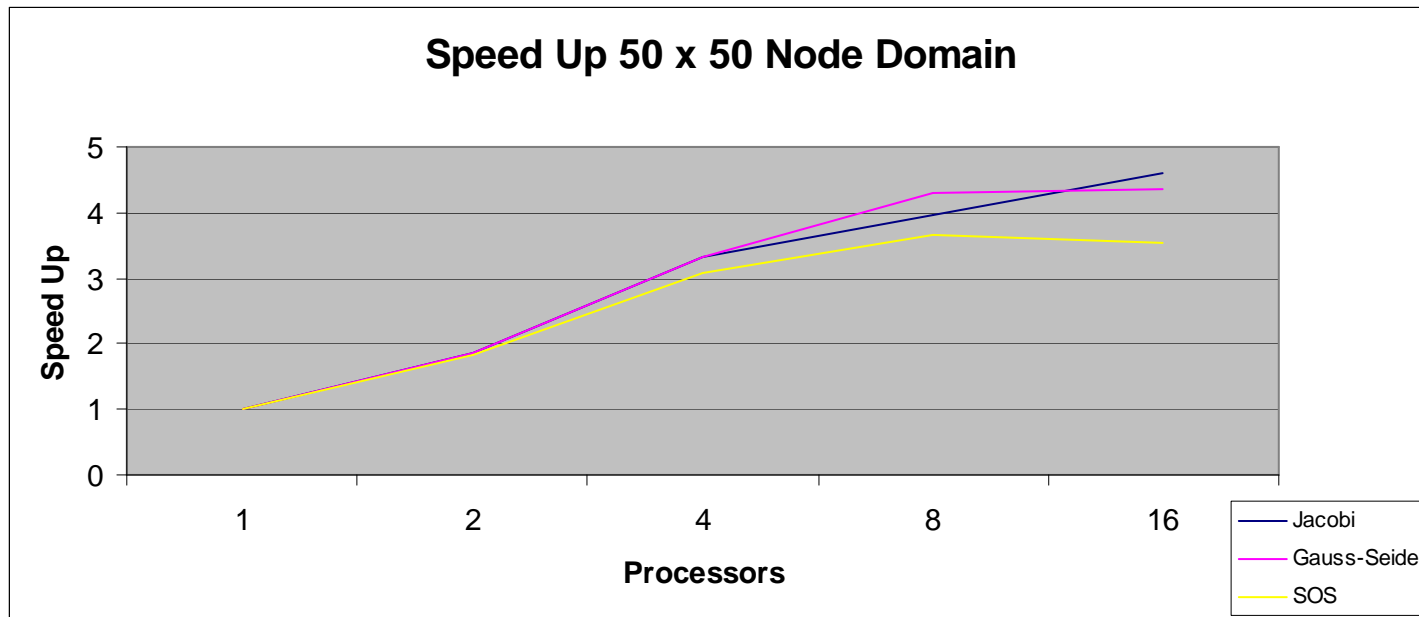
$$\text{Comm}_{\text{tot}} = I * (2(P-1) + 1)$$

where I is the total number of iterations and P is the total number of PEs

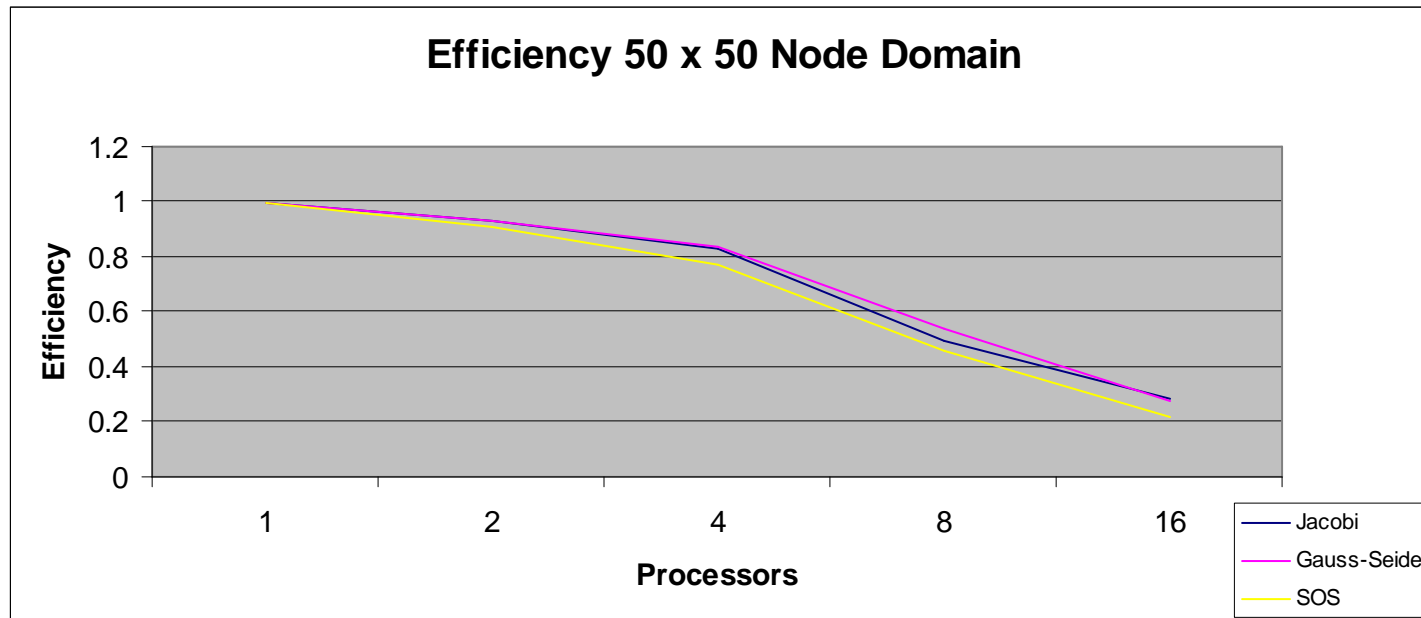
Development and Benchmarks

- Development was done on both Linux (Fedora Core 2) and FreeBSD (4.10) using the MPICH implementation of the MPI standard
- Benchmarks were done on 8, dual Xeon 2.6 Ghz, pcs running Fedora Core 2 (Linux) (16 processors total)
- Program **should** run on any machine with MPICH installed, especially any of the Unix variants
- This virtual “cluster” is on a 100mbs Ethernet “bus” network topology
- MPICH utilized rsh (remote shell) to facilitate all communications
- Program **should** run on any number of processors as long as each processor can have at least one row of its own

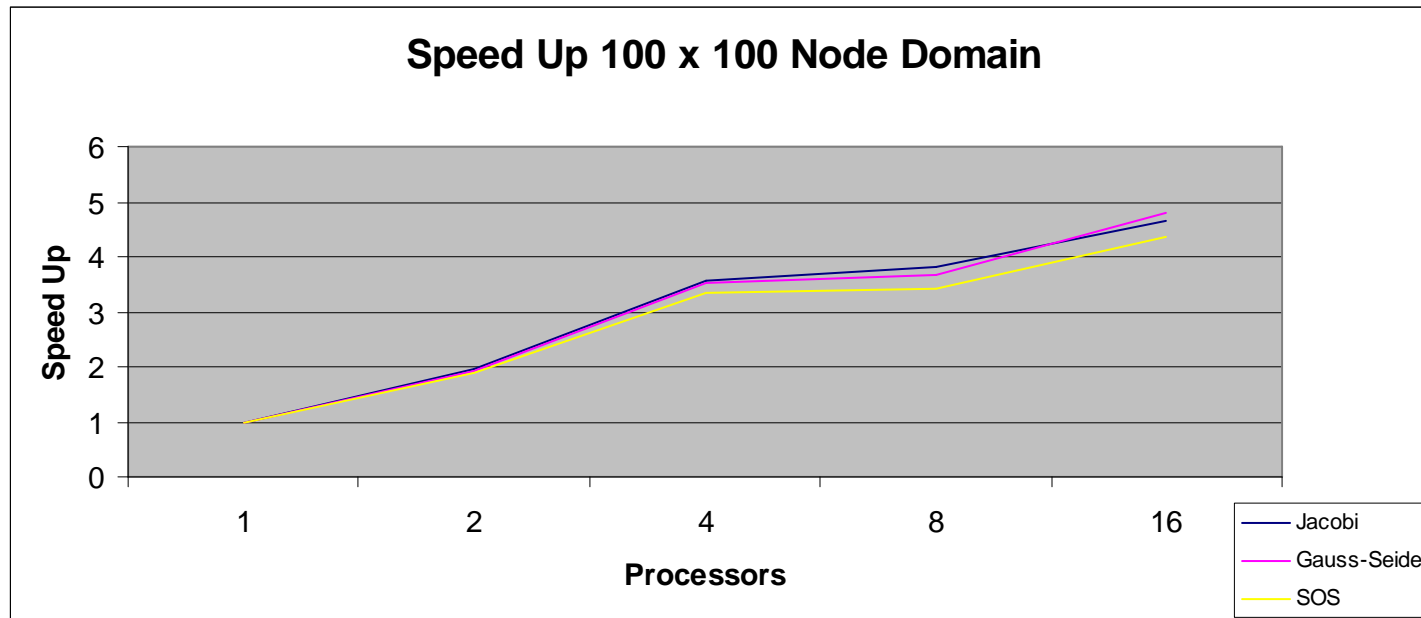
Speed Up 50x50 Node Domain



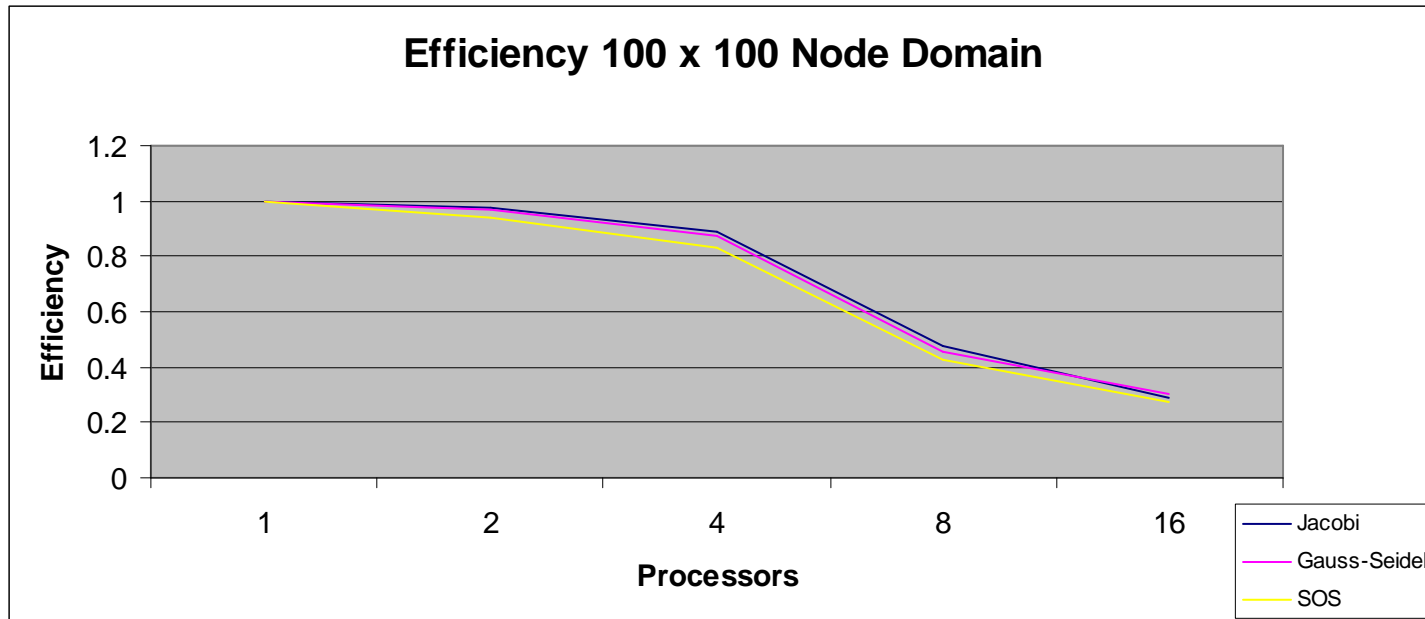
Efficiency 50x50 Node Domain



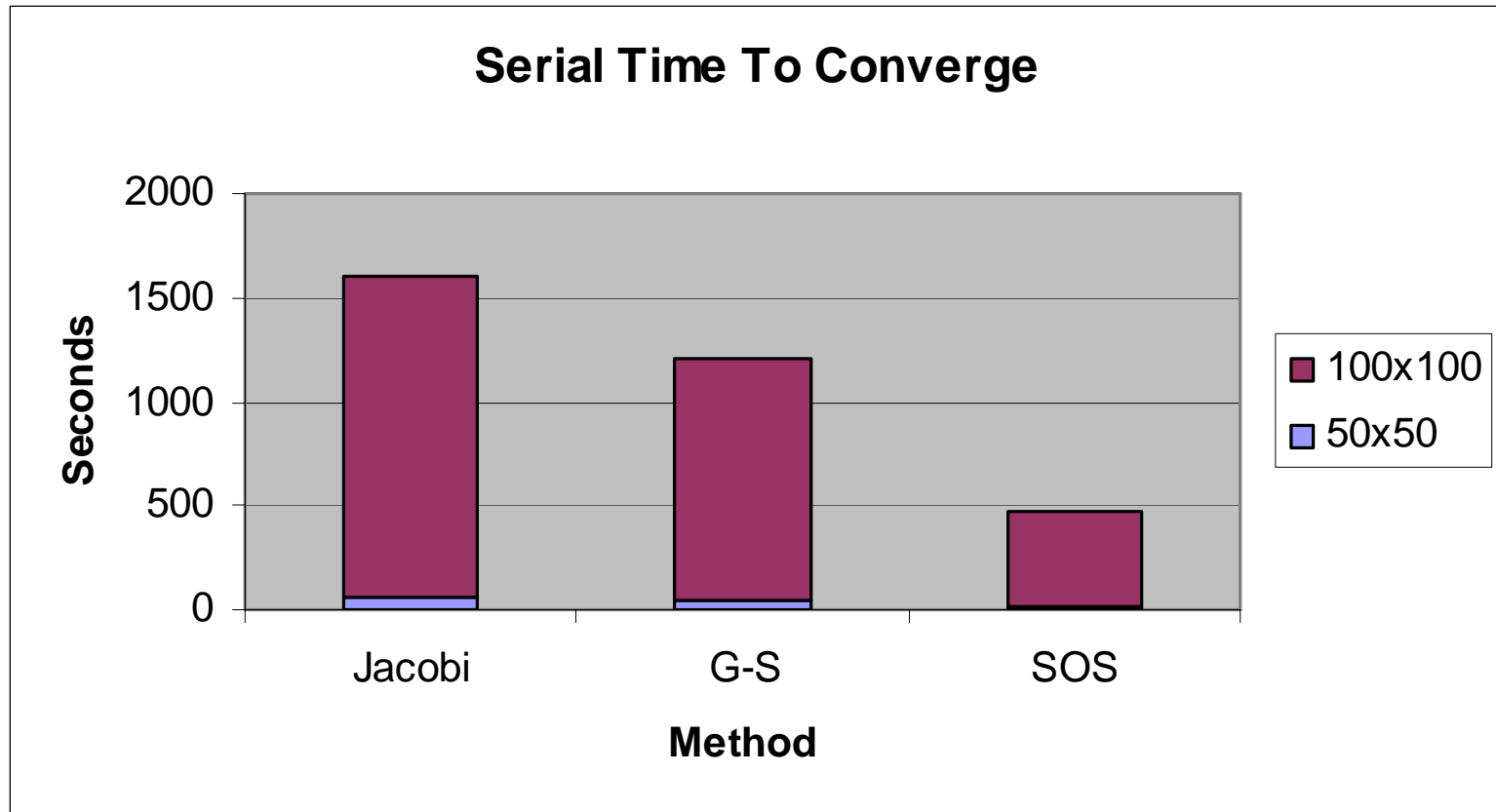
Speed Up 100x100 Node Domain



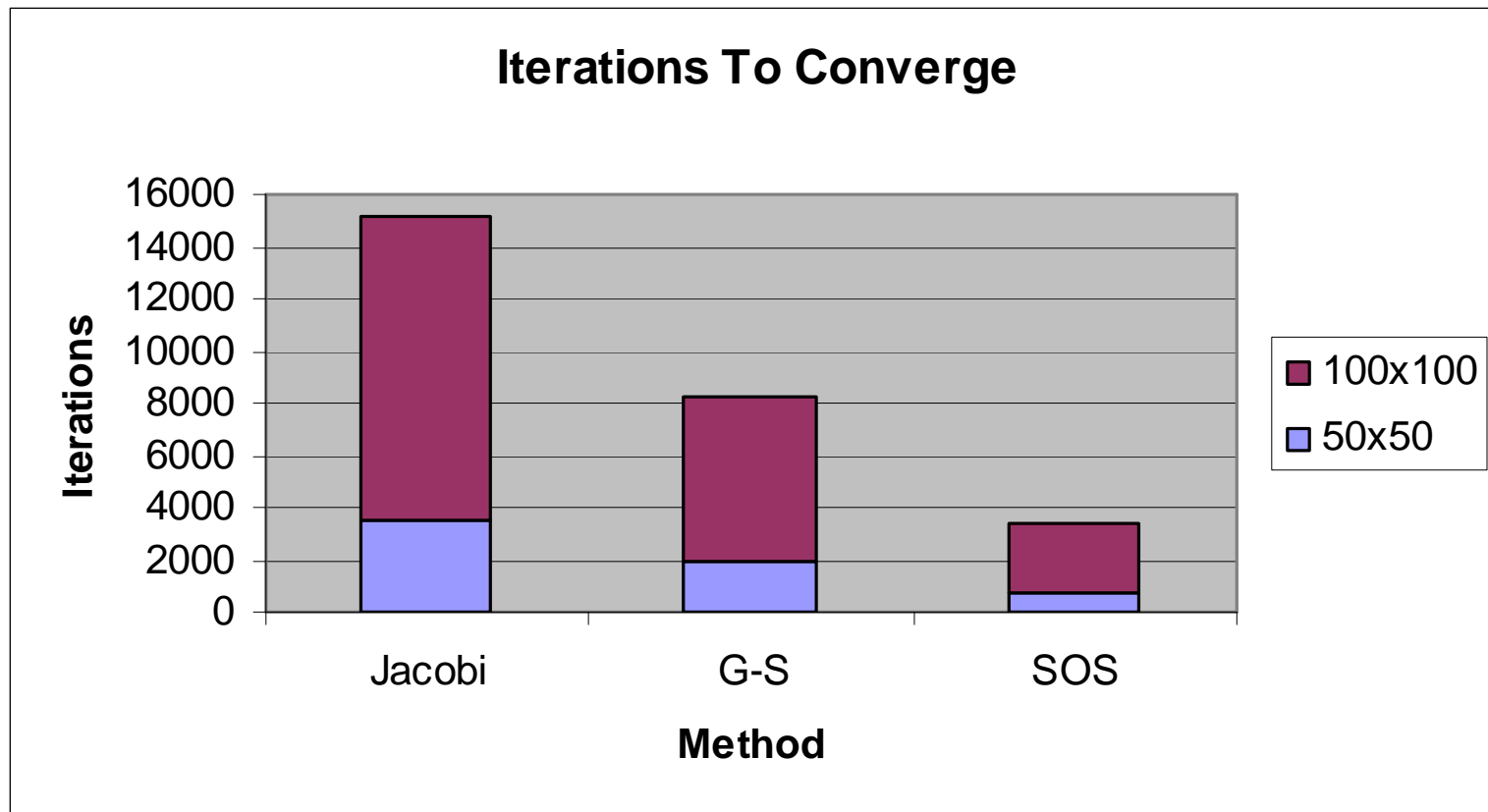
Efficiency 100x100 Node Domain



Method Comparisons



Method Comparisons



Note: both serial and parallel implementations of each method converged in the same number of iterations

Note: Convergence criteria, ϵ , was 0.0001

Method Comparisons

Results for rows 1 & 2 on a 10x10 domain (both serial and parallel)

x	y	Jacobi	Gauss-Seide	SOR
0	1	0.000000	0.000000	0.000000
1	1	2.857111	2.857120	2.857145
2	1	7.705517	7.705528	7.705578
3	1	19.306114	19.306137	19.306194
4	1	53.455193	53.455212	53.455284
5	1	19.488251	19.488277	19.488346
6	1	8.120687	8.120704	8.120773
7	1	3.637105	3.637123	3.637166
8	1	1.448818	1.448824	1.448848
9	1	0.000000	0.000000	0.000000
0	2	0.000000	0.000000	0.000000
1	2	3.722941	3.722953	3.722986
2	2	8.658842	8.658875	8.658949
3	2	16.063782	16.063810	16.063929
4	2	25.026405	25.026455	25.026577
5	2	16.377161	16.377195	16.377331
6	2	9.357396	9.357441	9.357549
7	2	4.978940	4.978962	4.979038
8	2	2.158168	2.158185	2.158218
9	2	0.000000	0.000000	0.000000

Lessons Learned

- Arrays in C are row-major, and are referenced [row],[col]
- Dynamically allocating array in C is tedious, but the flexibility is worth it
- Ghost rows allow one to minimize communications, but taking advantage of the low task dependence could add to the efficiency of a parallel finite difference method
- Implementing the red/black schemes by taking advantage of their task dependencies would speed these parallel computations up significantly

Conclusion

- The SOR Method converged the fastest, followed by the Gauss-Seidel Method; The Jacobi Method was considerably slower
- The results of all three methods were consistent, though different after the 100th decimal place
- Greater speed up and a less severe drop in efficiency was seen when the number of nodes in the domain were increase
- Greater speed up and efficiency would be achieved by optimizing both task management and communications
- Parallelization must yield results identical to the serial implementation, yet must also provide an increase in performance

Credits and Resources

- [1] <http://www.wikipedia.com>
- Personal communications
- MPICH Homepage:
<http://www-unix.mcs.anl.gov/mpi/mpich/>
- FreeBSD Homepage
<http://www.freebsd.org>
- Fedora Homepage:
<http://fedora.redhat.com/>