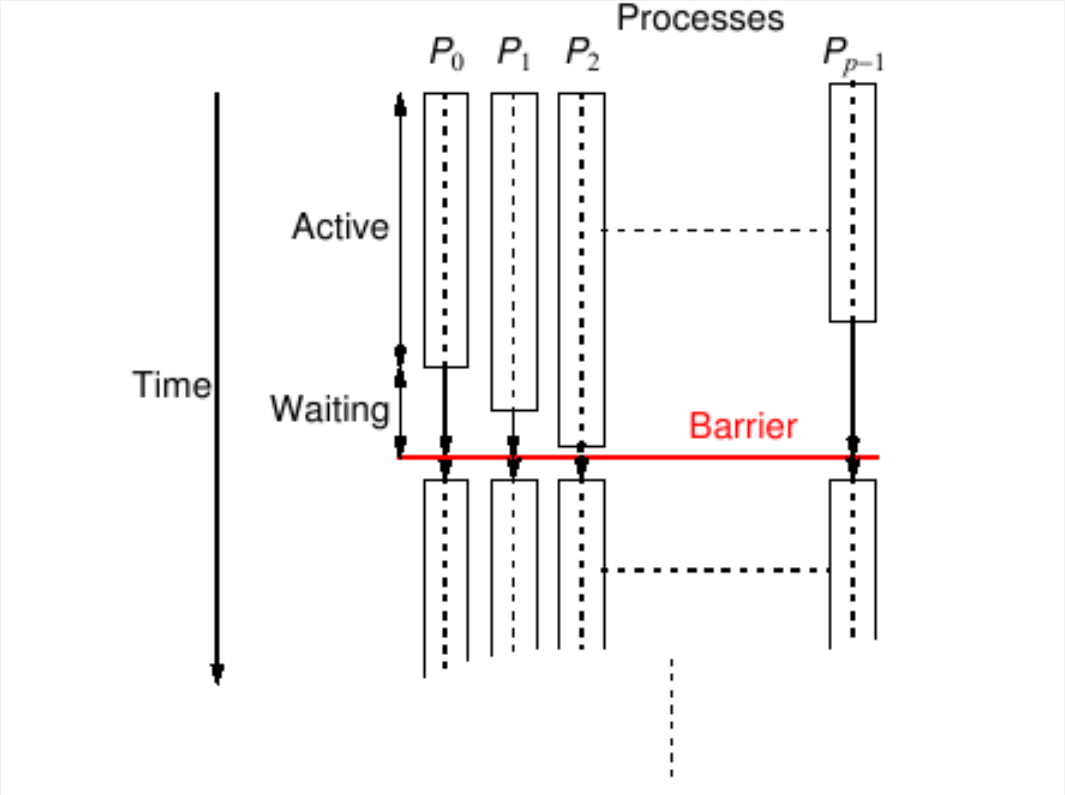
Synchronized Computations

Barrier

A barrier is a mechanism that allows multiple processes to synchronize at a point in their execution. All processes then continue execution after all processes have reached the synchronization point.
A variation allows processes to proceed from the barrier if a specified number of processes reach the barrier (less than the total number of processes in the barrier).

- MPI provides a `MPI_Barrier(MPI_Comm comm)` call.

Barrier



MPI Barrier Examples

- Example 1: lab/MPI/barrier/barrier_demo_mpi.c
- Example 2: lab/MPI/barrier/lb_demo_mpi.c

Barrier Implementations

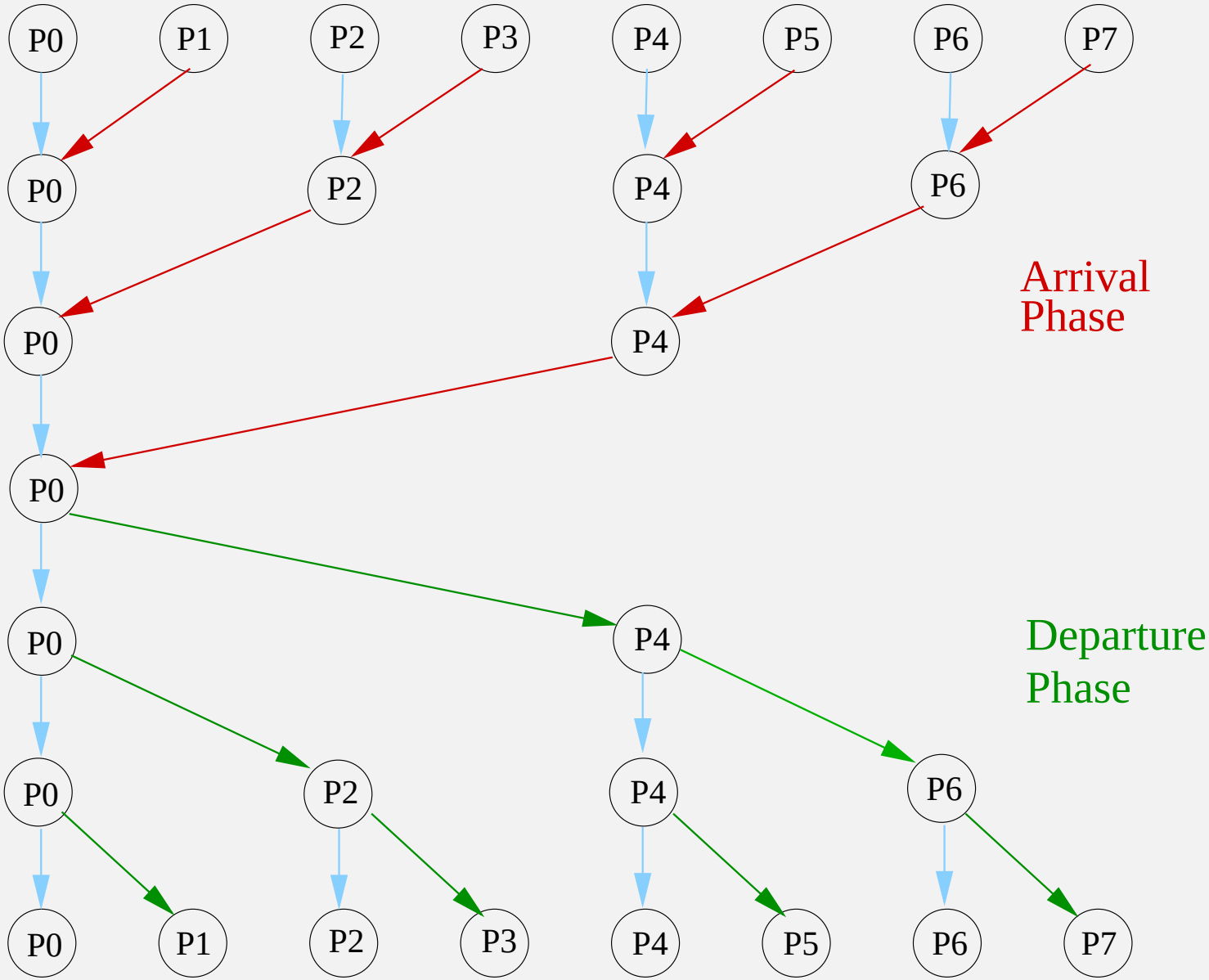
- Single counter in the coordinator process
- Tree synchronization
- Butterfly synchronization

Barrier Implemented Using a Counter

```
barrier(pid)
//process pid, 0 ≤ i ≤ p − 1
//value is a dummy variable
if (pid = 0) //coordinator
    //arrival phase
    for (i=1; i<p; i++)
        recv(value, P_ANY)
    //departure phase
    for (i=1; i<p; i++)
        send(value, P_i)
else //other processes
    send(value, P_0)
    recv(value, P_0)
```

$2(p-1)$ communication steps.

Barrier Implemented Using a Tree



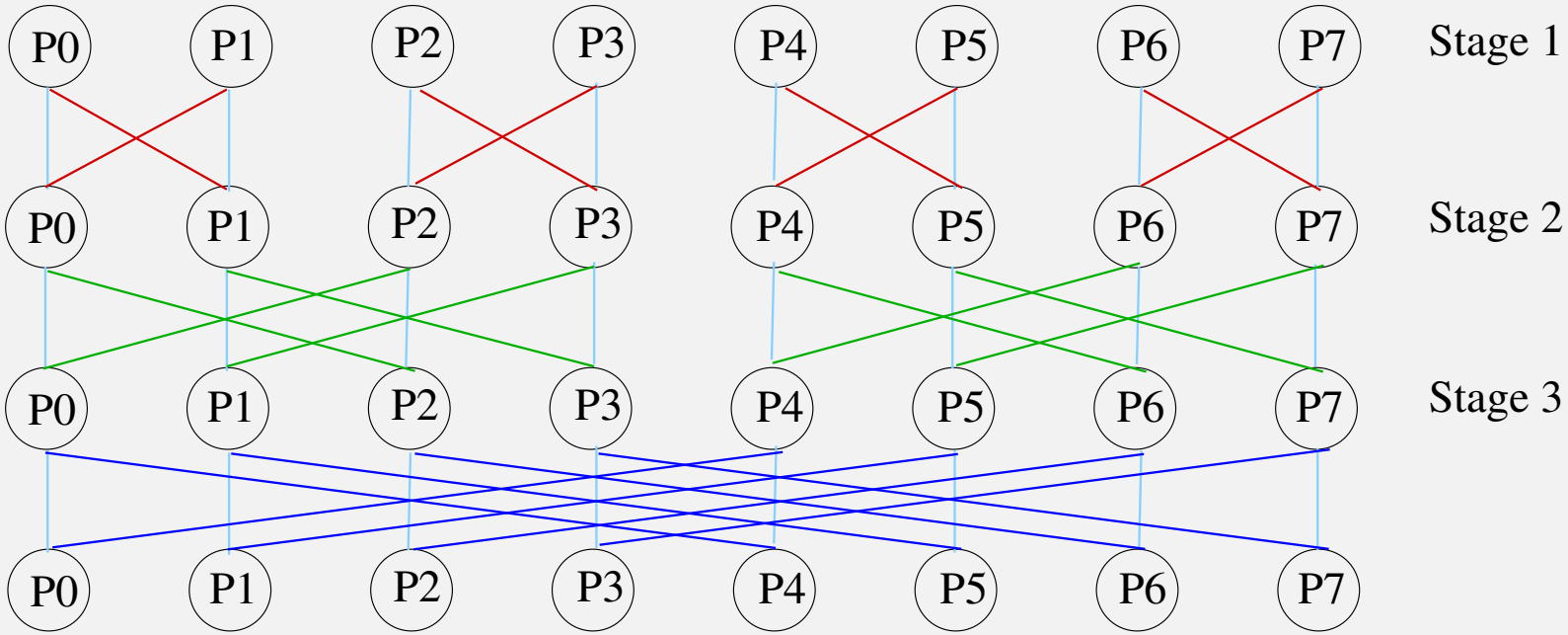
$2 \lg p$ communication steps.

Tree Barrier Pseudo-code

```
tree-barrier(pid)
//process pid, 0 ≤ i ≤ p − 1, p = 2^k
//value is a dummy variable
//arrival phase
for (h=1; h≤ lg p; h++)
    if (pid mod 2^h) = 0
        recv(value, P_{pid+2^{h−1}})
    else
        send(value, P_{pid−2^{h−1}})
        break
//departure phase
if (pid ≠ 0) recv(value, P_ANY)
for (h =lg p; h≥ 1; h--)
    if (pid mod 2^h) = 0
        send(value, P_{pid+2^{h−1}})
```

How to modify if the number of processes isn't a power of 2?

Barrier Implemented Using Butterfly Synchronization



Butterfly Synchronization

$\lg p$ communication steps.

Butterfly Synchronization

- Assume that the number of processes is a power of 2.
- Similar to a tree except we don't need separate arrival/departure phases.
- At stage s, process i synchronizes with process $i \oplus 2^{s-1}$, where $\oplus$ is the bit-wise exclusive or operation.
- How to handle the case when the number of processes isn't a power of 2?

Recommended exercise: Write pseudo-code for barrier using butterfly synchronization.

Near Neighbor Synchronization

A common pattern is when process P_i needs to synchronize with neighboring processes P_{i-1} and P_{i+1}, where $0 \leq i \leq p-1$ with or without wrap-around.

Process P_{i-1}	Process P_i	Process P_{i+1}
send(P_i)	send(P_{i-1})	send(P_i)
send(P_{i-2})	send(P_{i+1})	send(P_{i+2})
recv(P_i)	recv(P_{i-1})	recv(P_i)
recv(P_{i-2})	recv(P_{i+1})	recv(P_{i+2})

If each process is executing send's first, then there is potential for deadlock using synchronous sends or blocking send without sufficient buffering. We can resolve this problem by interleaving send's and recv's.

Examples using Synchronized Computations

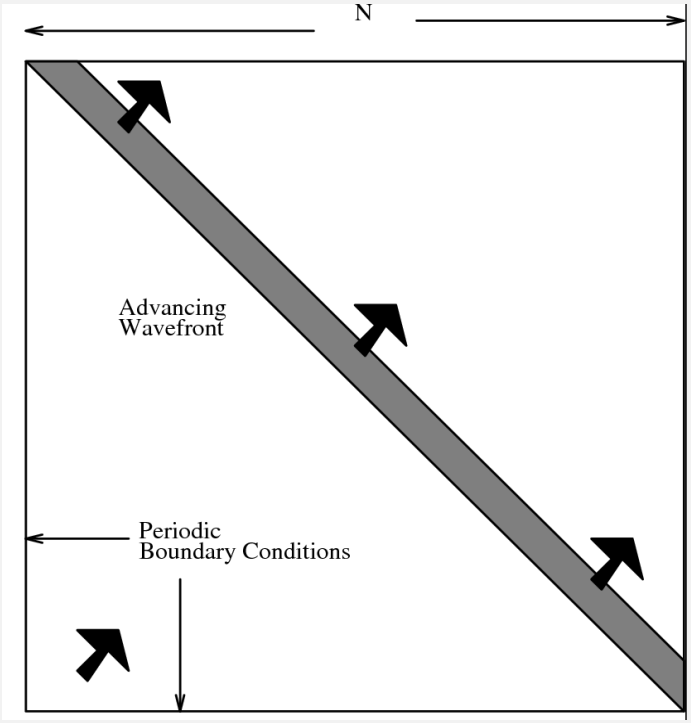
- Wave Simulation. Simulate a wave through 2-d or 3-d space or materials.
- Heat Distribution. Given a material and an ambient temperature, how does the heat distribute across the material.
- Cellular Automata. E.g. Conway's Game of Life.

Wave Simulation

The *Wave Equation* is given by

$$u_{tt} = c(u_{xx} + u_{yy})$$

where c is a constant and $u = u(t, x, y)$ is a function of three variables. The variable t represents time and x and y are spatial variables.



We want to solve it using the finite difference method.

Discretization of the Wave Simulation

A discretization of the wave equation on a two-dimensional grid on $[0,1] \times [0,1]$ using finite differences gives:

$$u_{t+1,x,y} = 1(1-2\rho^2)u_{t,x,y} + \rho^2(u_{t,x+1,y} + u_{t,x-1,y} + u_{t,x,y+1} + u_{t,x,y-1}) - u_{t-1,x,y}$$

where $\rho = \Delta t/\Delta h$, Δt is the increment in time, h is the step size in the x and y directions.
Calculating new values of u at a time $t+1$ only requires the knowledge of values of u at time t and $t-1$. Hence only two time steps need to be kept in memory simultaneously.
We assume that $c = 1$ and for stability reasons choose $(\Delta t)^2 = (h/c)^2/2$. We want to study the case when we introduce a diagonal wave at time $t = 0$. Assume periodic boundary conditions (that is, as if the square is wrapped around like a toroid).

Wave Simulation Details

- Set the number of points on the grid to be $N = 64, 128, 256, \ldots$ and the number of time steps.
- Note that $N = 1/h$ and the initial value of u is 1 on the diagonal and 0 otherwise. Take the value of u at time step 1 to be a diagonal wave of magnitude 1, but shifted right by one step including wrap-around.
- The goal is to calculate the wave function at the end of the equation.

Sequential Wave Simulation

```
/* lab/misc/wave/slow_seq_wave.c */

#define M   (MAX+2)
float u[MAX+2][MAX+2], uold[MAX+2][MAX+2], unew[MAX+2][MAX+2];
char data[MAX][MAX];

    h = 1/(float) N;
    dt =  h/(float) sqrt((double)2.0);
    rho = dt/h;

    for (i=0; i<N-1; i++) {
        uold[i][i]=u[i][i]=1.0;
        u[i][i+1]=1.0;
    }
    uold[N-1][N-1]=u[N-1][N-1]=1.0;
    u[N-1][0]=1.0;
    ...
```

Sequential Wave Simulation (contd.)

```
/* lab/misc/wave/slow_seq_wave.c */
    ...
    for (t=0; t<steps; t++) {
        for (i=0; i<N; i++) {
            iS = (i<N-1 ? (i+1):(0));
            iN = (i>0 ? (i-1):(N-1));
            for (j=0; j<N; j++) {
                jE = (j<N-1 ? (j+1):(0));
                jW = (j>0 ? (j-1):(N-1));
                unew[i][j] = 2.0*(1-2.0*rho*rho)*u[i][j] + rho*rho*(u[iS][j]
                                  u[iN][j] + u[i][jW] + u[i][jE]) - uold[i][j];
            }
        }
        for (i=0; i<N; i++) {
            for (j=0; j<N; j++) {
                uold[i][j]=u[i][j];
                u[i][j]=unew[i][j];
            }
        }
    }
```

Compiler Optimization

Use the optimizer that comes with your compiler! The following table shows the gain with optimization with the gcc compiler. The option -O turns on the optimizer. It solves a wave equation of size 1024 and 1000 iterations.

Optimizer level	time (seconds)
default (-O0)	25.5
-O	22.4
-O2	21.5
-O3	20.0
-O5	19.7

Over 20% improvement.

Improvements to Sequential Wave Simulation

- Instead of copying arrays, we can cyclically swap pointers.
- Eliminate *if* statements in the body of the loop.
- Use pointer arithmetic. (CAUTION: be very careful here, not needed for most cases. This makes the code less readable and maintainable.)

After the improvements, the time dropped to 7.5s, over 340% improvement!

Faster Sequential Wave Simulation

```
/* lab/misc/wave/fast_seq_wave.c */

    /* initialize pointers */
    p_u   = &u[0][0];
    p_uold = &uold[0][0];
    p_unew = &unew[0][0];

    h = 1/(float) N;
    dt =  h/(float) sqrt((double)2.0);
    rho = dt/h;
    rho_2 = rho * rho;

    for (i=1; i<N; i++) {
        uold[i][i]=u[i][i]=1.0;
        u[i][i+1]=1.0;
    }
    uold[N][N]=u[N][N]=1.0;
    /*u[N][1]=1.0;*/

    ...
}
```

Faster Sequential Wave Simulation (contd.)

```
/* lab/misc/wave/fast_seq_wave.c */
    ...
    for (t=0; t<steps; t++) {
        i1=N*M; i2=i1+M; i3=M; i4=M+N;
        for (i=1; i<=N; i++) {
            i1++; i2++;
            *(p_u+i) = *(p_u+i1);
            *(p_u+i2) = *(p_u+M+i);
            *(p_u+i3)= *(p_u+i4);
            *(p_u+i4+1)= *(p_u+i3+1);
            i3 += M; i4 += M;
        }
        for (i=1; i<=N; i++) {
            ij = i*M;
            ij1 = ij+M;
            ij2 = ij-M;
            for (j=1; j<=N; j++) {
                ij++;ij1++;ij2++;
                *(p_unew+ij) = 2.0*(1-2.0*rho_2)* ( *(p_u+ij)) +
                    rho_2*(*(p_u+ij1) + *(p_u+ij2) +
                    *(p_u+ij-1) + *(p_u+ij+1)) - *(p_uold+ij);
            }
        }
        tmp1 = p_u; p_u = p_unew; p_unew = p_uold; p_uold = tmp1;
    }
```

Parallelization of the Wave Simulation

```
for (t = 0; t < iter; t++){
    /* send info to adjacent processes */

    /* compute internal positions (those that don't require info *
    /* from adjacent processors).                                 *

    /* get the info from adjacent processes */

    /* compute edge positions, also compute min and max values
       for this worker */
    ...
    barrier()
    /* This section handles the rotation of the matrices */
    tmp = uold; uold = u; u = unew; unew = tmp;
}
```

Partitioning of the wave problem

- Checkerboard
- Strips (by rows or columns)
- How to decide which one will be better?

Comparing Strip versus Block partitioning

Assume that the wave size is $n \times n$ with p processes.

- For partitoning by *strips*, each strip needs to send and receive one column (or row) of size n with processes on either side for total communication per process to be:

$$T_{comm} = 4(t_{startup} + nt_{data})$$

- For partitioning by blocks, each block is of size $n/\sqrt{p}$. We need to communicate with four neighboring processes (north, south, east and west). In each case one edge is sent and one edge is received. So the communication time per process is given by:

$$T_{comm} = 8(t_{startup} + \frac{n}{\sqrt{p}} t_{data})$$

Comparing Strip versus Block partitioning (contd.)

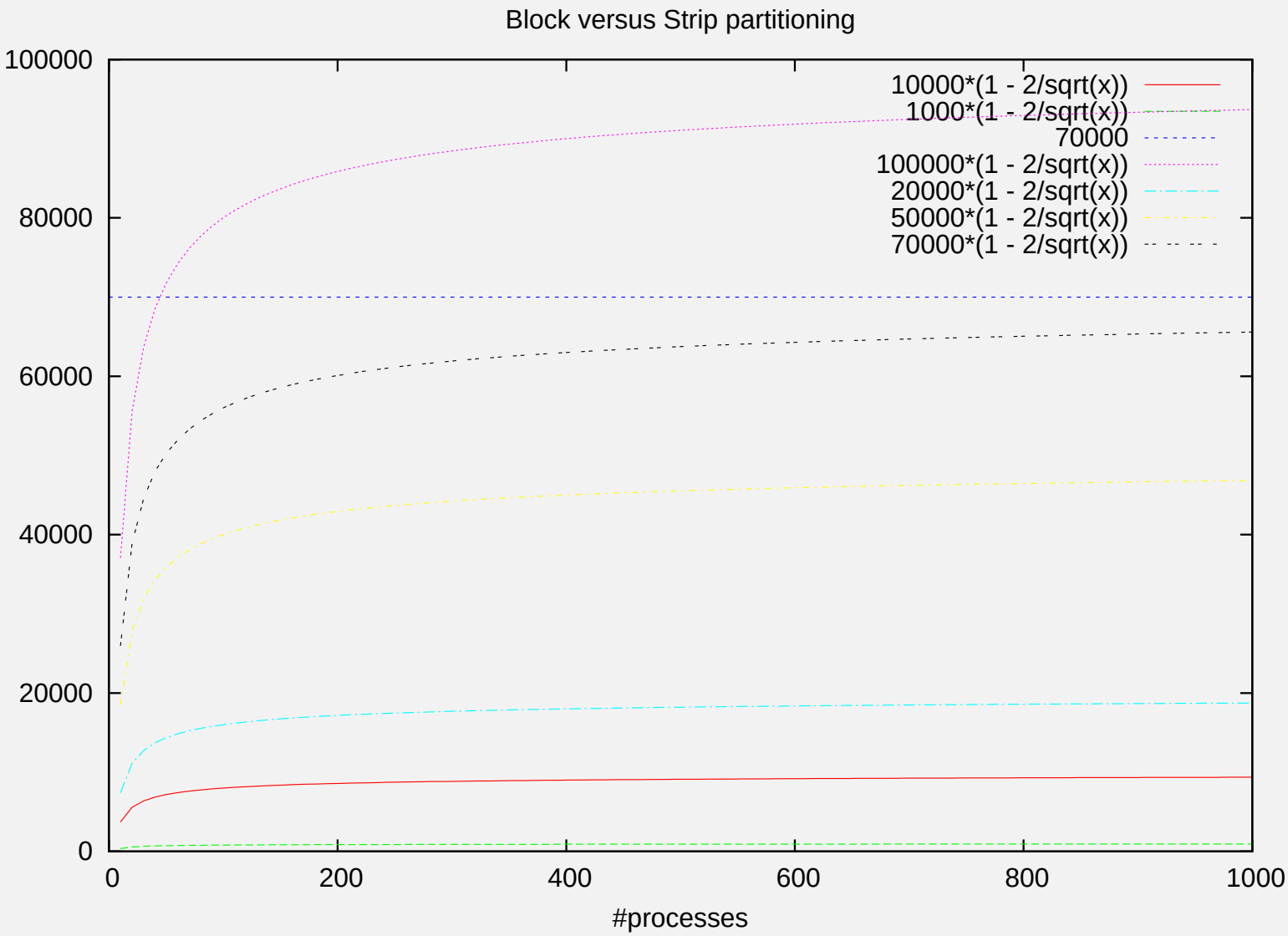
Assume that the wave size is $n \times n$ with p processes.

- In general, strip partition is better for large startup time whereas block partition is better for small startup time. Block partition has a larger communication time than strip partition if

$$t_{startup} > n(1 - \frac{2}{\sqrt{p}})t_{data}$$

  The above requires $p \geq 9$ for the equation to be valid.
- For our current lab setup, $t_{startup} = 78$ microseconds, while the $t_{data} = 1.12$ nanoseconds. So we can plot the value of p for various fixed values of n

Comparing Strip versus Block partitioning (contd.)



Program Safety

- If all processes send simultaneously, we may get deadlock. To solve this, use the usual MPI techniques like:
  - Alternate the order of sends/recvs in adjacent processes.
  - Use combined send/recvs with MPI_Sendrecv()
  - Buffered sends.
  - Nonblocking sends.

Heat Distribution

A two-dimensional grid is given with $m \times m$ points on it. Temperature is known around the edges. Find the temperature distribution inside the area using simulation.
The value at a point is given by:

$$x_{i,j} = (x_{i-1,j} + x_{i+1,j} + x_{i,j-1} + x_{i,j+1})/4, \ 0 \leq i,j \leq m-1$$

Repeat for a fixed number of iterations or until the maximum difference in temperature between iterations for all points is less than some pre-specified precision.

Partially Asynchronous Computations

- Synchronizing parallel processes is an expensive operation that significantly slows the computation.
- Computations in which individual processes operate without needing to synchronize with other processes on every iteration. For example, in the Heat distribution problem, it is possible to get convergence without synchronizing on every iteration.

Conway's Game of Life

Board game - theoretically infinite two-dimensional array of cells. Each cell can hold one "organism" and has eight neighboring cells, including those diagonally adjacent. Initially, some cells occupied. The following rules apply:

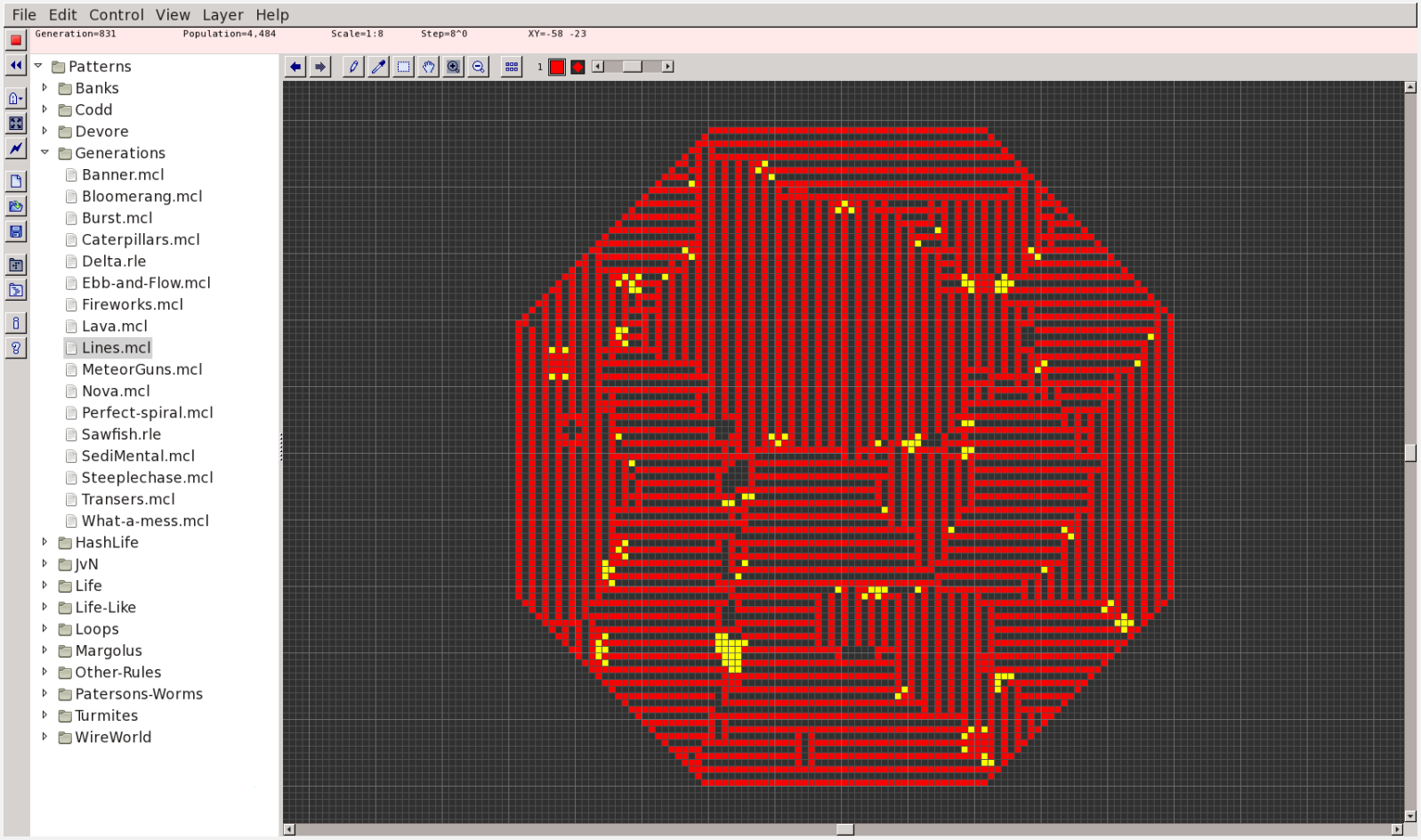
- Every organism with two or three neighboring organisms survives for the next generation.
- Every organism with four or more neighbors dies from overpopulation.
- Every organism with one neighbor or none dies from isolation.
- Each empty cell adjacent to exactly three occupied neighbors will give birth to an organism.

On a Fedora system, try

```
yum install golly
```

to try a Game of Life program.

Game of Life Example



Parallel Game of Life

Sketch out how you would parallelize the $n \times n$ Game of Life on a p processor cluster, where $p << n$. For the simulation, you may assume that the two-dimensional array is a torus so each cell always has eight neighbors.

- Does your implementation need to use a barrier. If so, where?
- Are there load balancing issues?
- Analyze the computation time and the speedup that your approach would yield.