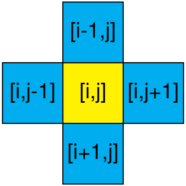
Easy Stencil Application

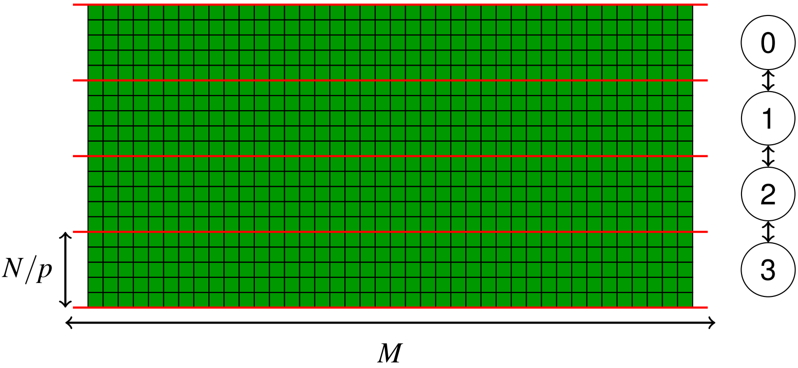
Stencil

```
int A[N][M], B[N][M];
int *tmp, *cur, int *new;
int t,i,j;

cur = &(A[0][0]); // cur points to A
new = &(B[0][0]); // new points to B

// Initialize A's content
...

// 1000 iterations
for (t=0; t < 1000; t++) {
  for (i=1; i < N-1; i++) {
    for (j=1; j < M-1; j++) {
      new[i*N+j] = update(cur[i*N+j],cur[i*N+j-1],cur[i*N+j+1],
                          cur[(i-1)*N+j],cur[(i+1)*N+j]);
    }
  }
  // Swap array pointers
  tmp = cur; cur = new; new = tmp;
}
```

Update based on current value and **old** values of neighbors

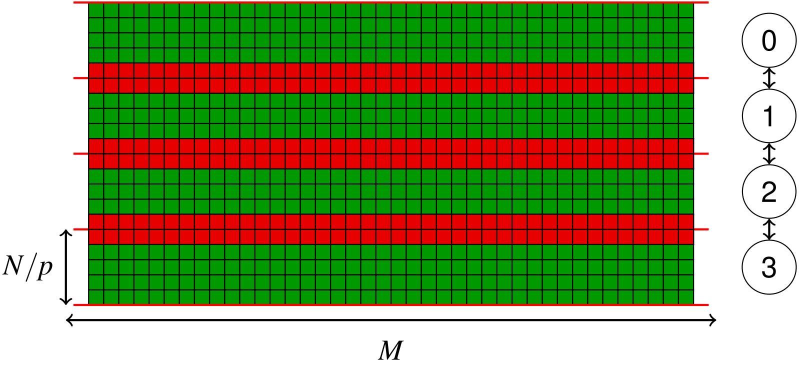


Data Distribution



- Each of the p processes allocates a $N/p \times M$ array

Data Distribution



- These red cells require values from neighbors

Distributed Memory Code

Parallel stencil sketch

```
int A[N*M/p], B[N*M/p];
...

for (t=0; t < 1000; t++) {
  [ send row 0 to rank-1]
  [ recv row from rank-1 ]
  [ send row N/p-1 to rank+1 ]
  [ recv red row from rank+1 ]
  < update my green row(s) >
  < update my red row(s) >
  < swap buffers as in sequential version >
}
```

- Note that the real code would be more complex because some processes have only one neighbor
- We assume a bi-directional ring, so if links are not full-duplex we will have contention, which may be ok

Using Non-Blocking Communications

Parallel stencil sketch

```
int A[N*M/p], B[N*M/p];
...

for (t=0; t < 1000; t++) {
  [ send row 0 to rank-1, asynchronously ]
  [ send row N/p-1 to rank+1, asynchronously ]
  < update my green row(s) >
  [ wait to receive red row from rank-1 ]
  [ wait to receive red row from rank+1 ]
  < update my red row(s) >
  < swap buffers as in sequential version >
}
```

- One can use asynchronous communication (`MPI_Isend`, `MPI_Irecv`) for these communications
- If the time to send/receive rows is shorter than the time to update green cells at a processor, then *communication is fully hidden*
- Depends on network speed, computing speed, size of the domain, and number of processors
- With fully hidden communication one can hope for 100% parallel efficiency

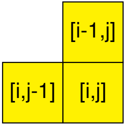
Outline

Less Easy Stencil Application

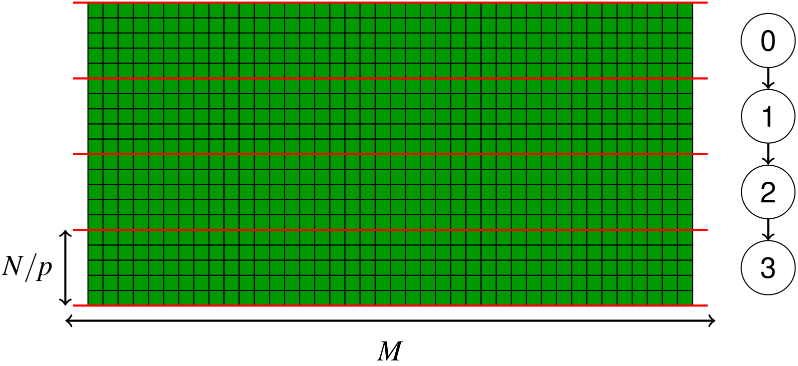
Stencil

```
int A[N*M];
int t,i,j;

// Initialize A's content
...

// 1000 iterations
for (t=0; t < 1000; t++) {
  for (i=1; i < N; i++) {
    for (j=1; j < M; j++) {
      A[i*N+j] = update(A[i*N+j],A[i*N+j-1], A[(i-1)*N+j]);
    }
  }
}
```

Update based on current value and current values of West and North neighbors

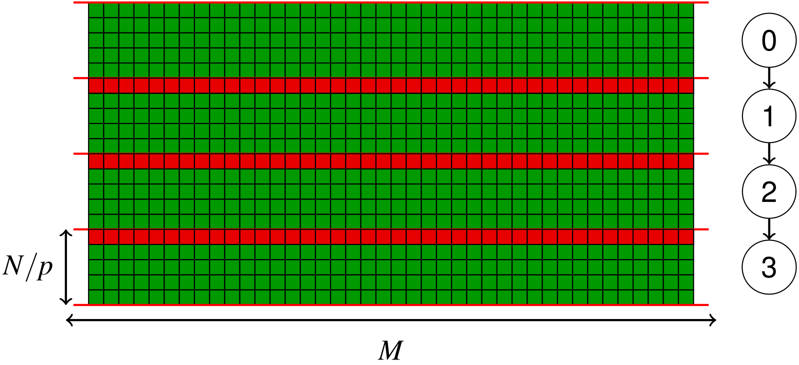


Same Data Distribution as Before



- Each of the p processes allocates a $N/p \times M$ array

Same Data Distribution as Before



- These red cells require values from neighbors

Naïve Algorithm (one iteration)

Stencil

```
p = num_procs();
rank = my_rank();
int A[N/p+1][M]; // One extra row at each process to hold
                 // the received row from neighbor

if (rank != 0) {
  // Receive my predecessor's last row
  receive(&(A[0][0]),N)
}
// Update all my green cells
for (i=1; i < N/p; i++) {
  for (j=0; j < M; j++) {
    update(i,j)
  }
}
if (rank != p-1) {
  // Send my last row to rank r+1
  send(&(A[N/p-1][0]),N)
}
```

It this code good?

Naïve Algorithm (one iteration)

Stencil

```
p = num_procs();
rank = my_rank();
int A[N/p+1][M]; // One extra row at each process to hold
                 // the received row from neighbor

if (rank != 0) {
  // Receive my predecessor's last row
  receive(&(A[0][0]),N)
}
// Update all my green cells
for (i=1; i < N/p; i++) {
  for (j=0; j < M; j++) {
    update(i,j)
  }
}
if (rank != p-1) {
  // Send my last row to rank r+1
  send(&(A[N/p-1][0]),N)
}
```

It this code good?

No!!! It's **sequential**

Making it parallel

- This code is sequential because process $r+1$ has to wait for process r to finish computing all its rows
- What we need:
  - Process r should compute the elements of its last row as early as possible
  - Each element should be sent to process $r+1$ *at once*, without waiting for the whole row to be computed
- One option is to have each process go down columns first rather than rows
- Let's try this....

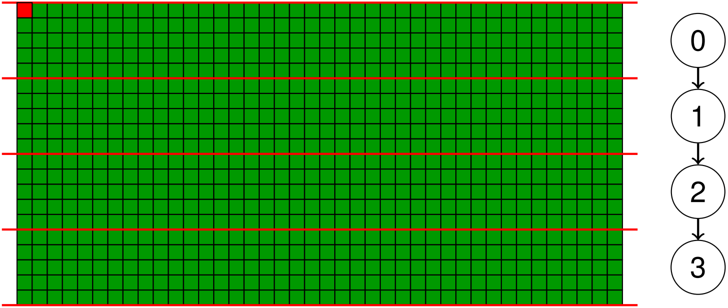
Less Naïve Algorithm (one iteration)

Stencil

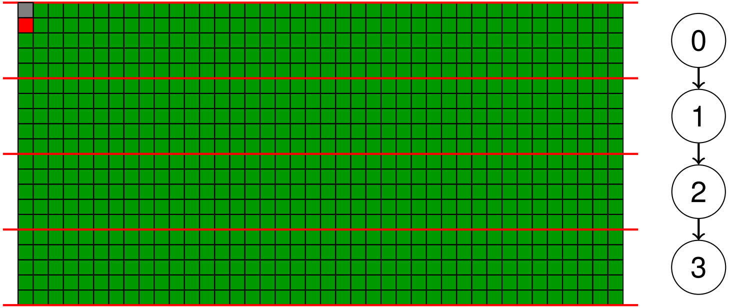
```
p = num_procs();
rank = my_rank();
int A[N/p+1][M]; // One extra row at each process to hold
                 // the received row from neighbor

for (j=0; j < M; j++) {
  for (i=0; i < N/p; i++) {
    if (rank != 0) {
      // Receive my predecessor's last element in column j
      receive(&(A[0][j]))
    }
    // Update all my green cells in column j
    for (i=1; i < N/p; i++) {
      update(i,j)
    }
    if (rank != p-1) {
      // Send my last element in column j to rank r+1
      send(A[N/p-1][j])
    }
  }
}
```

Let's visualize the order of computation step by step...

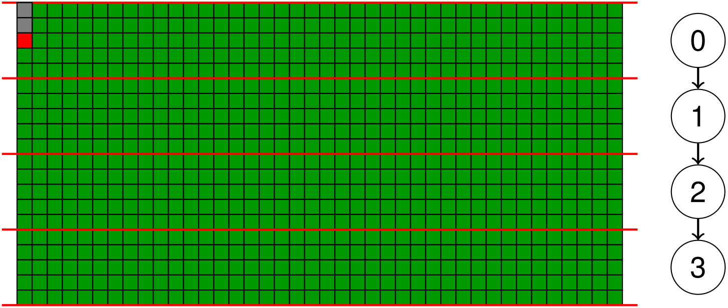
Parallel Execution



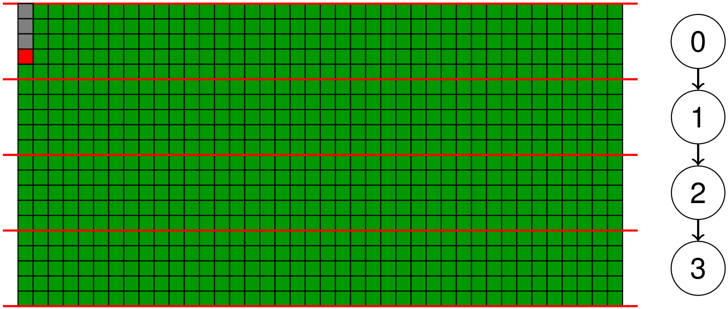
Parallel Execution



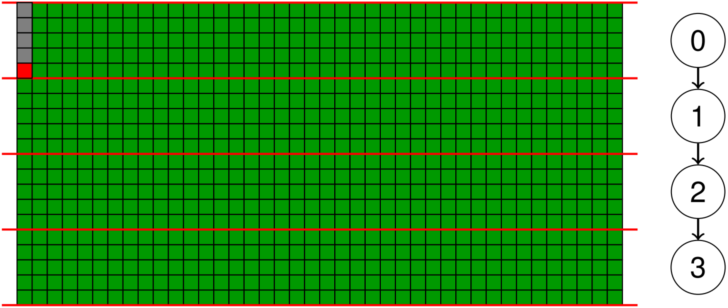
Parallel Execution



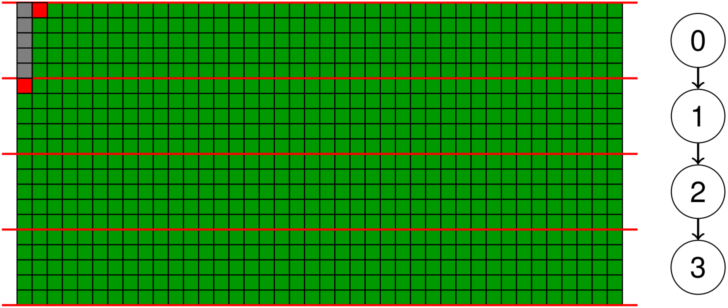
Parallel Execution



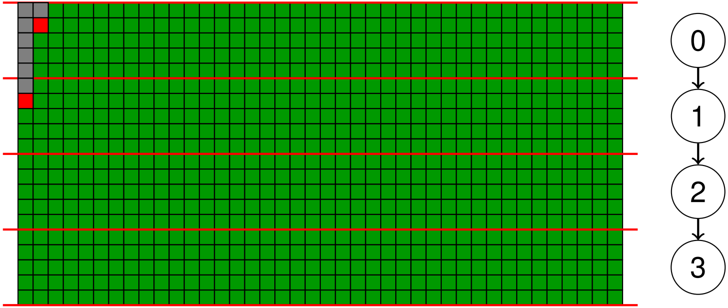
Parallel Execution



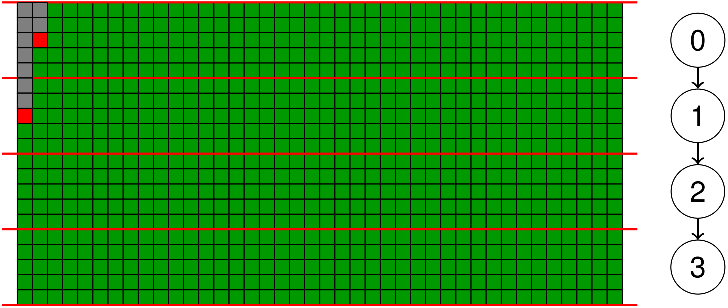
Parallel Execution



Parallel Execution



Parallel Execution



Parallel Speedup

- Let's assume an infinitely fast network so that communicating cells takes zero time
- Let c be the time to update a cell
- Let's figure out the parallel execution time... any ideas?

Parallel Speedup

- Let's assume an infinitely fast network so that communicating cells takes zero time
- Let c be the time to update a cell
- The last process begins computing at time: $(p-1) \times (N/p) \times c$
- It then computes for time $(N/p) \times M \times c$ time units
- It is the last one to finish computing, so the overall parallel execution time if $(p-1) \times (N/p) \times c + (N/p) \times M \times c$
- The sequential execution time is $N \times M \times c$
- So the parallel speedup is: $pM/(p-1+M)$
- If $M \rightarrow +\infty$, then speedup $\rightarrow p$

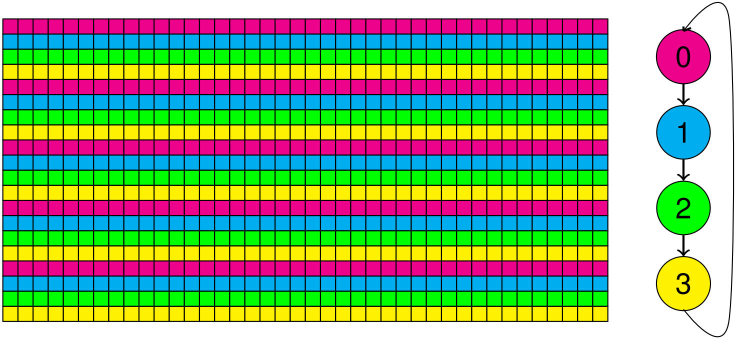
Can we do better?

- Our algorithm is *asymptotically optimal*
  - It has asymptotically optimal parallelism
- But if M isn't very large compared to p, then we're not in great shape
  - Parallelism is not great because the last processor starts computation “late"
- How can we do better?
- How can we have each process start computing as early as possible? Any idea?

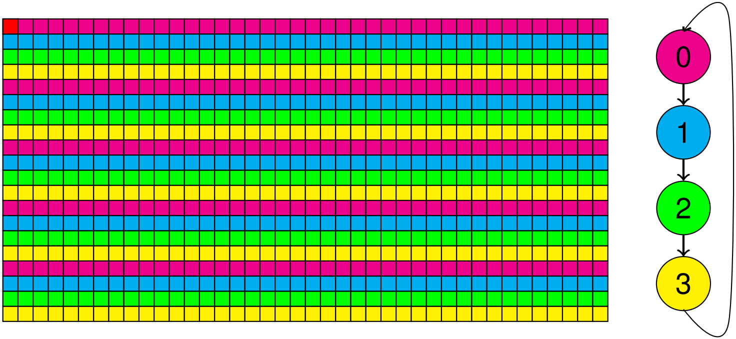
Can we do better?

- Our algorithm is *asymptotically optimal*
  - It has asymptotically optimal parallelism
- But if M isn't very large compared to p, then we're not in great shape
  - Parallelism is not great because the last processor starts computation “late"
- How can we do better?
- We can use a cyclic data distribution
- Processor r is assigned row i if $i \mod p = r$

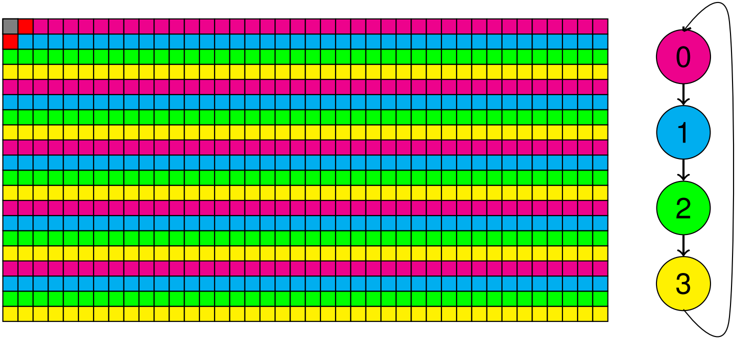
Cyclic Data Distribution



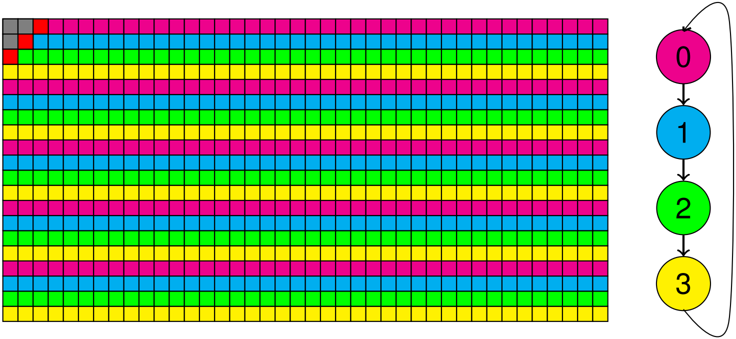
Cyclic Data Distribution



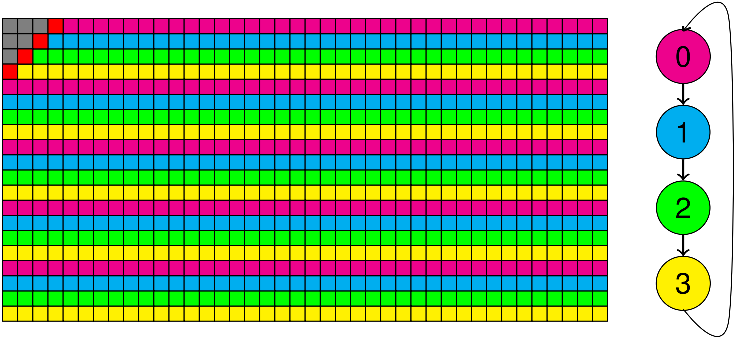
Cyclic Data Distribution



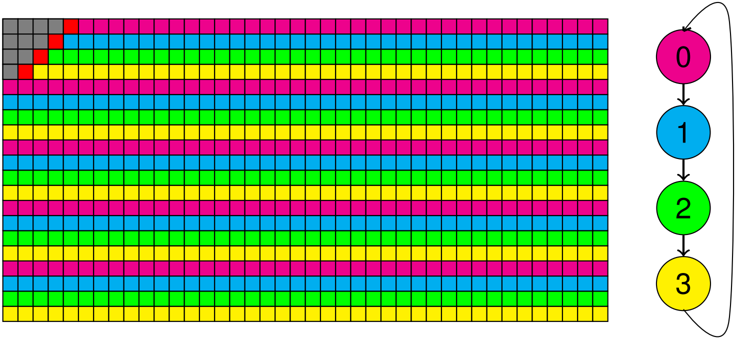
Cyclic Data Distribution



Cyclic Data Distribution



Cyclic Data Distribution



Parallel Speedup

- Let's again assume an infinitely fast network so that communicating cells takes zero time
- The last process begins computing at time: $(p-1) \times c$
- It then computes for time $(N/p) \times M \times c$ time units (one cell computed each time unit)
- It is the last one to finish computing, so the overall parallel execution time if $(p-1) \times c + (N/p) \times M \times c$
- The sequential execution time is $N \times M \times c$
- So the parallel speedup is: $pM/(p(p-1)/N + M)$
  - Was $pM/(p-1+M)$
- We have made the denominator smaller (because $N > p$). We've improved parallelism as much as possible

Cyclic Algorithm (one iteration)

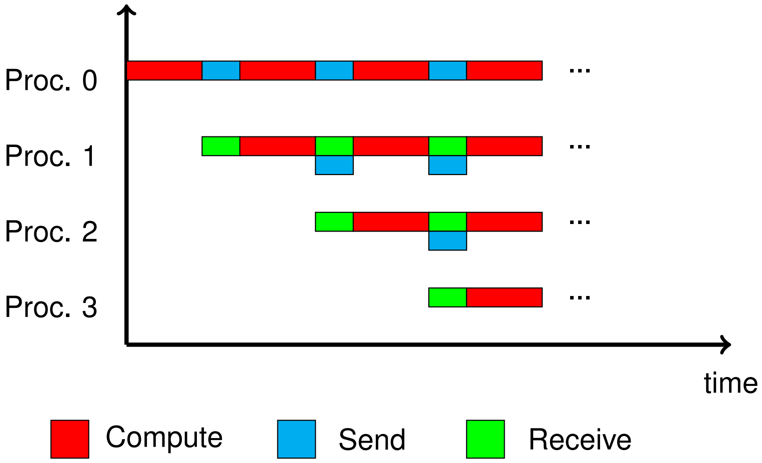
Stencil

```
p = num_procs();
rank = my_rank();
int A[N/p][M];
int cell_above;

for (i=0; i < N/p; i++) {
  for (j=0; j < M; j++) {
    if ((i > 0) && (rank > 0)) {
      // Receive my predecessor's last element in column j
      receive(&cell_above)
    }
    // Update my current cell
    update(i,j,cell_above)
    if ((i < N/p-1) && (rank < p-1)) {
      // Send my current cell to my successor
      send(A[i][j])
    }
  }
}
```

Network Overhead

- Network communication isn't zero-overhead
- Typical model of time to send x bytes: $\alpha + \beta x$
  - α: latency
  - β: inverse of the data rate
- Let s be the size of a cell value, in bytes
- The last process begins computing at time: $(p - 1) \times (c + \alpha + \beta s)$
- It then computes for time $(N/p) \times M \times (c + \alpha + \beta s)$ time units (one cell computed and communicated each time unit)
- This assumes that a process can send and receive at the same time
- This is called the "two-port model"
- Let's see this on a Gantt chart..

The Two-Port Model



Parallel Speedup

- Parallel execution time:
  $(p-1) \times (c + \alpha + \beta s) + (N/p) \times M \times (c + \alpha + \beta s)$
- Sequential time: $N \times M \times c$ (no communication!)
- So the parallel speedup is the ratio of the two
- When $NM \rightarrow +\infty$, speedup $\rightarrow pc/(c + \alpha + \beta s)$
- This could be bad if communications are expensive
- If $c = \alpha + \beta s$, then speedup $\rightarrow p/2$ (50% parallel efficiency)
- In practice α could be huge compared to c
  - CPU Clock rate is high, network latencies can be high
- Our parallelism is great
- But our overhead is terrible!

Reducing Overhead

- Each time we send one message, we incur an α overhead!
- This is the typical “parallel application that sends tons of tiny messages" problems
- Idea: send groups of cell together
- Initially we sent a whole row, that was too many cells
- But sending one cell is too few
- So let’s send m cells, where we choose m
  - We assume m divides M, for simplicity
- Let’s look at the code...

Cyclic Algorithm, m cells (one iteration)

Stencil

```
p = num_procs();
rank = my_rank();
int A[N/p][M];
int cells_above[m];

for (i=0; i < N/p; i++) {
  for (j=0; j < M; j+=m) {
    if ((i > 0) && (rank != 0)) {
      // Receive my predecessor's last m cells
      receive(&cells_above,m)
    }
    // Update my current @m@ cells
    for (k=0; k < m; k++)
      update(i,j+k,cell_above)

    if ((i < N/p-1) && (rank != p-1)) {
      // Send my current m cells to my successor
      send(&(A[i][j]),m)
    }
  }
}
```

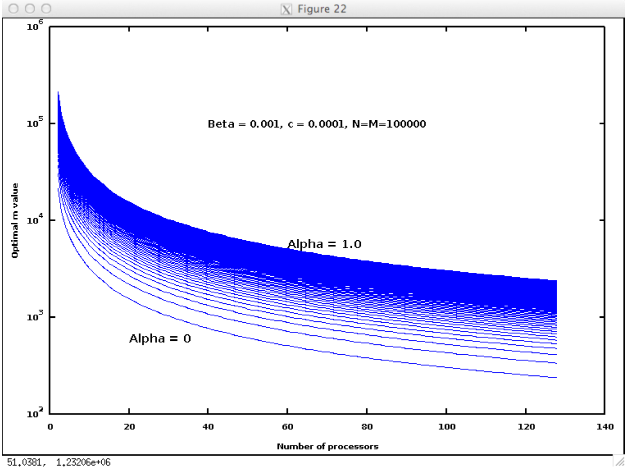
Parallel Speedup

- The last processor begins computing at time: $(p-1) \times (mc + \alpha + \beta m)$
- Then it computes for: $(NM/mp)(mc + \alpha + \beta m)$
- Parallel time is the sum of the two
- Sequential time: $N \times M \times c$
- Parallel speedup: the ratio of the two
  - For $m = 1$ we get our previous speedup
- When $NM \to +\infty$, speedup $\to pc/(c + \alpha/m + \beta)$
- Compared to before we've divided α my m
- We've decreased parallelism
- But we've also decreased overhead
- What's a good value of m?
- Let's find out...

Best m value

- Parallel time:
  $T = (p-1) \times (mc + \alpha + \beta m) + (NM/mp)(mc + \alpha + \beta m)$
  $\frac{\partial T}{\partial m} = (p-1)(c+\beta) - \frac{NM\alpha}{pm^2}$
  $\frac{\partial T}{\partial m} = 0 \implies m = \sqrt{\frac{NM\alpha}{p(p-1)(c+\beta)}}$
- We should select the divisor of M that's the closest to the above (real) value
- Let's plot the (not rounded off) optimal value above...

Best m vs. p and α



Stencil Application

- If we plug in the best m into the *asymptotic* parallel speedup we get:

$$p \times \frac{c}{c+\sqrt{\frac{\alpha p(p-1)(c+\beta)}{NM}}+\beta}$$

- But this formula is for $NM \to +\infty$, so we get $p \times \frac{c}{c+\beta}$
- So we're not asymptotically optimal because of β
  - Makes sense: for each c you have to do a β
- And if p^2 is large or comparable to NM, then the speedup gets really poor
- In the end, this is just a difficult application to parallelize, and one shouldn't expect great parallel efficiency
  - Unless c is large, which could happen for a complicated stencil, but then that stencil may involve more neighbors...
- Side note: there is a yearly "HPC Stencil" conference, there are "stencil" research groups, etc.