Recitation 5

MPI Programming

Topics

- **What is MPI**
- **MPI Basics**
- **Implementing a 2-D Grid Solver**
  - Structuring problem for message-passing parallelism
  - Message passing coding examples
  - Running on GHC and Latedays machines
  - Performance measurements and analysis
- **Suggestions for additional information**

- **All code in directory linked from schedule web page**

Background

- **Message Passing Interface**
  - Library + compiler support for message-passing parallel programs
    - Independent processes that communicate only by explicit sending and receiving of messages
  - Supports multiple styles of communication
    - Point-to-point
    - Broadcast
    - Reduction (e.g., global sum or minimum)
- **Multiple Implementations**
  - Runs on everything from data clusters to supercomputers
  - On GHC machines
    - Can utilize multiple cores within single machine
  - On Latedays machines
    - Multiple cores on one or more machines

MPI Can be Simple

- **Many parallel programs can be written using just these six functions:**
  - Setup/teardown
    - **`MPI_INIT`**
    - **`MPI_FINALIZE`**
  - Who am I?
    - **`MPI_COMM_SIZE`**
    - **`MPI_COMM_RANK`**
  - Message passing
    - **`MPI_SEND`**
    - **`MPI_RECV`**

… but Painful!

- **OpenMP**
  - Add pragmas to existing program
  - Compiler + runtime system arrange for parallel execution
  - Rely on shared memory for communication
- **MPI**
  - Must rewrite program to describe how single process should operate on its data and communicate with other processes
  - Explicit data movement: programmer must say exactly what data goes where and when
  - Advantage: Can operate on systems that don’t have shared memory

Process Identification

- **When running with *P* processes:**
  - Size: *P*
    - Total number of processes
  - Rank: Number between 0 and *P*–1
    - Identity of individual process
- **Library Functions**
  - `MPI_Comm_size`
  - `MPI_Comm_rank`

A Simple MPI Program

- **From hello.c**

```
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init(NULL, NULL);
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf("Hello!  I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

- `MPI_COMM_WORLD` indicates the set of all processes
- All MPI functions return error code
- Update values by passing pointers as arguments

A Simple MPI Program

- **From hello.c**

```
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init(NULL, NULL);
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf("Hello!  I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

- **Compile**
  - `mpicc -O2 -g -Wall hello.c -o hello`
- **Run**
  - `mpirun -np 2 ./hello`

```
Hello!  I am 0 of 2
Hello!  I am 1 of 2
```

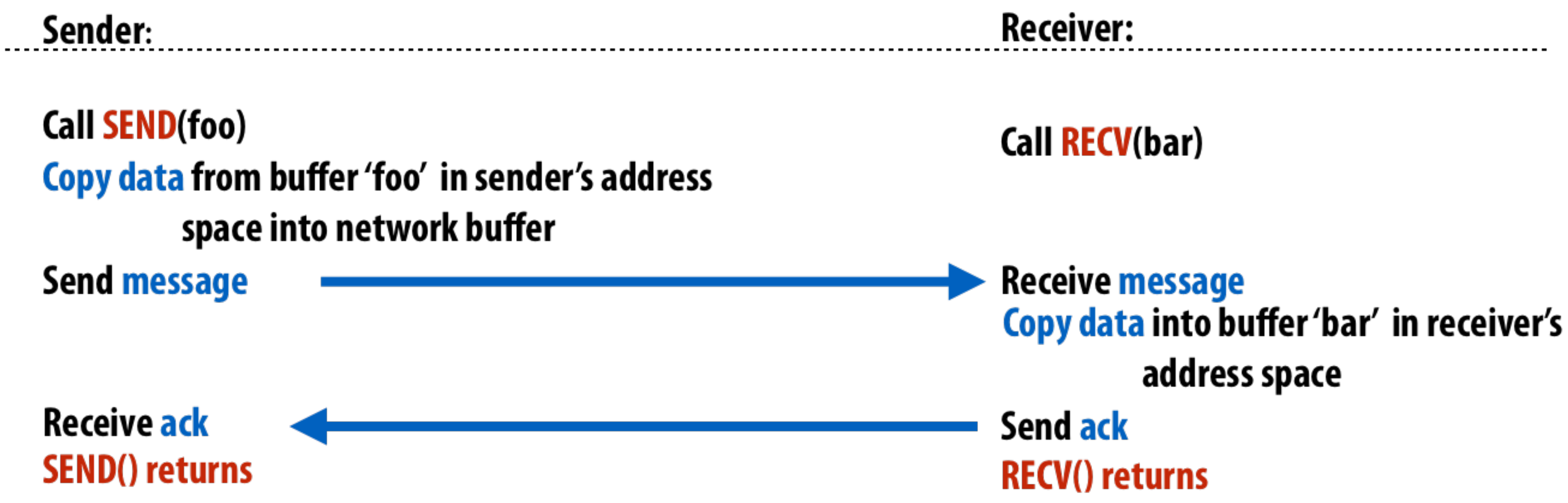
Comparing Frameworks

- **Multiple Processes with MPI**
  - Fixed number of processes created as program starts
  - All execute code starting with **`main`**
  - Isolated address spaces
- **Multiple Processes with `fork`**
  - Processes created during program execution
  - Replicate address space upon creation, but then isolated
- **Multiple Threads with `pthread_create`**
  - Threads created during program execution
  - Shared address space
- **Multiple Threads with OpenMP**
  - Set of threads created at beginning of program (conceptually)
  - Recruited to execute tasks spawned by **`#pragma omp parallel`**
  - Shared address space

Synchronous Sending and Receiving (From Lecture #5)

- send(): call returns when sender receives acknowledgement that message data resides in address space of receiver
- recv(): call returns when data from received message is copied into address space of receiver and acknowledgement sent back to sender
- Potential for deadlock if all processes attempt to send and then receive

Synchronous Sending and Receiving



Sender:

Call SEND(foo)

Copy data from buffer 'foo' in sender's address space into network buffer

Send message

Receive ack

SEND() returns

Receiver:

Call RECV(bar)

Receive message

Copy data into buffer 'bar' in receiver's address space

Send ack

RECV() returns

Asynchronous Sending and Receiving

- Low-level communication handled by additional threads
- send(): call returns immediately
  - Buffer provided to send() cannot be modified by calling thread since message processing occurs concurrently with process execution
  - Calling thread can perform other work while waiting for message to be sent
- recv(): posts intent to receive in the future, returns immediately
  - Use checksend(), checkrecv() to determine actual status of send/receipt
  - Calling thread can perform other work while waiting for message to be received

Asynchronous Sending and Receiving

Sender:	Receiver:
Call SEND(foo)	Call RECV(bar)
SEND returns handle h1	RECV(bar) returns handle h2
Copy data from 'foo' into network buffer	
Send message $\longrightarrow$	Receive message
	Messaging library copies data into 'bar'
Call CHECKSEND(h1) // if message sent, now safe for thread to modify 'foo'	Call CHECKRECV(h2)
	// if received, now safe for thread
	// to access 'bar'

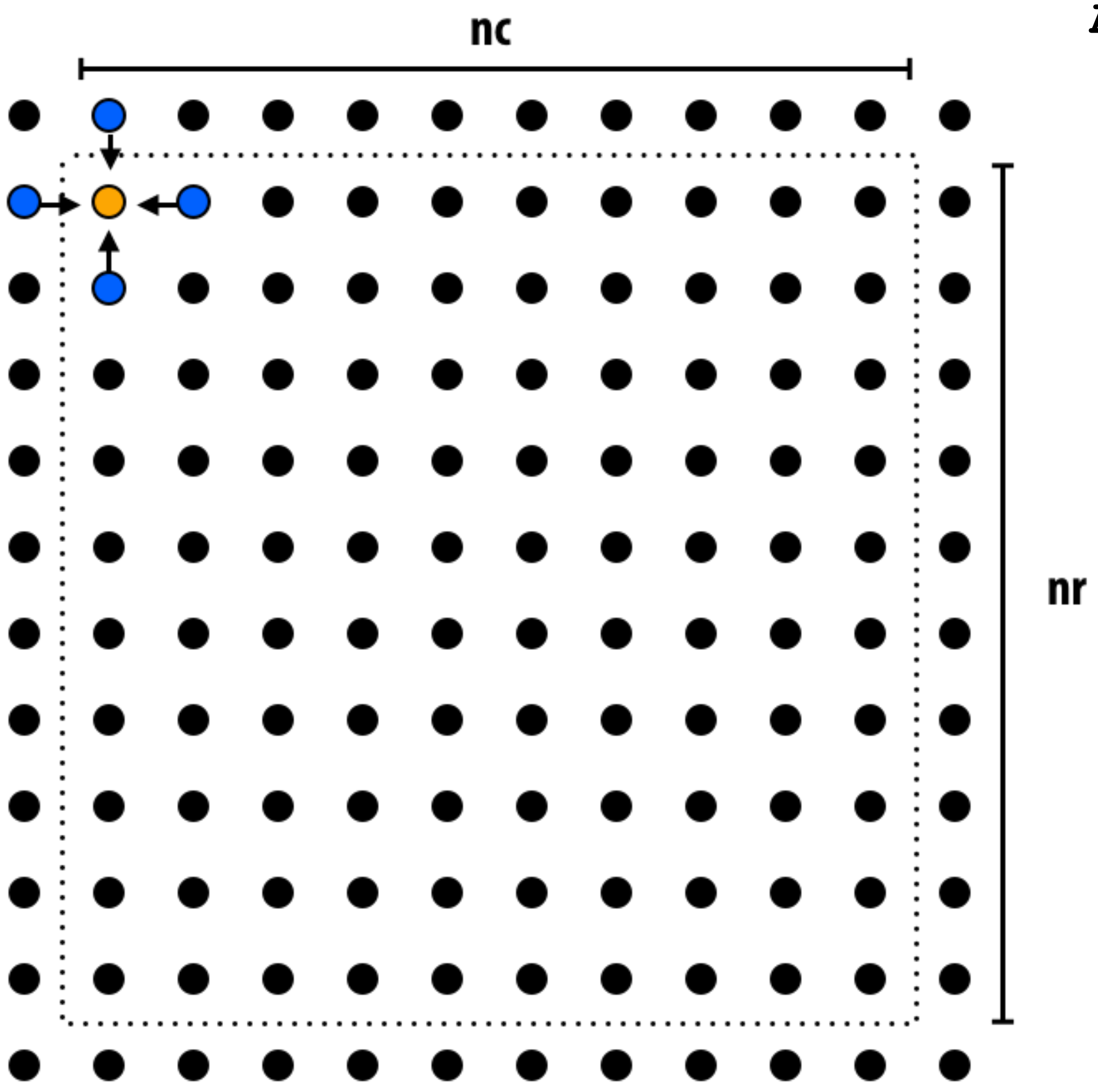
RED TEXT = executes concurrently with application thread

X

MPI Send/Receive Operations

- Synchronous
  - `MPI_Send`
  - `MPI_Recv`
- Asynchronous
  - `MPI_Isend`
  - `MPI_Irecv`
  - `MPI_Wait`

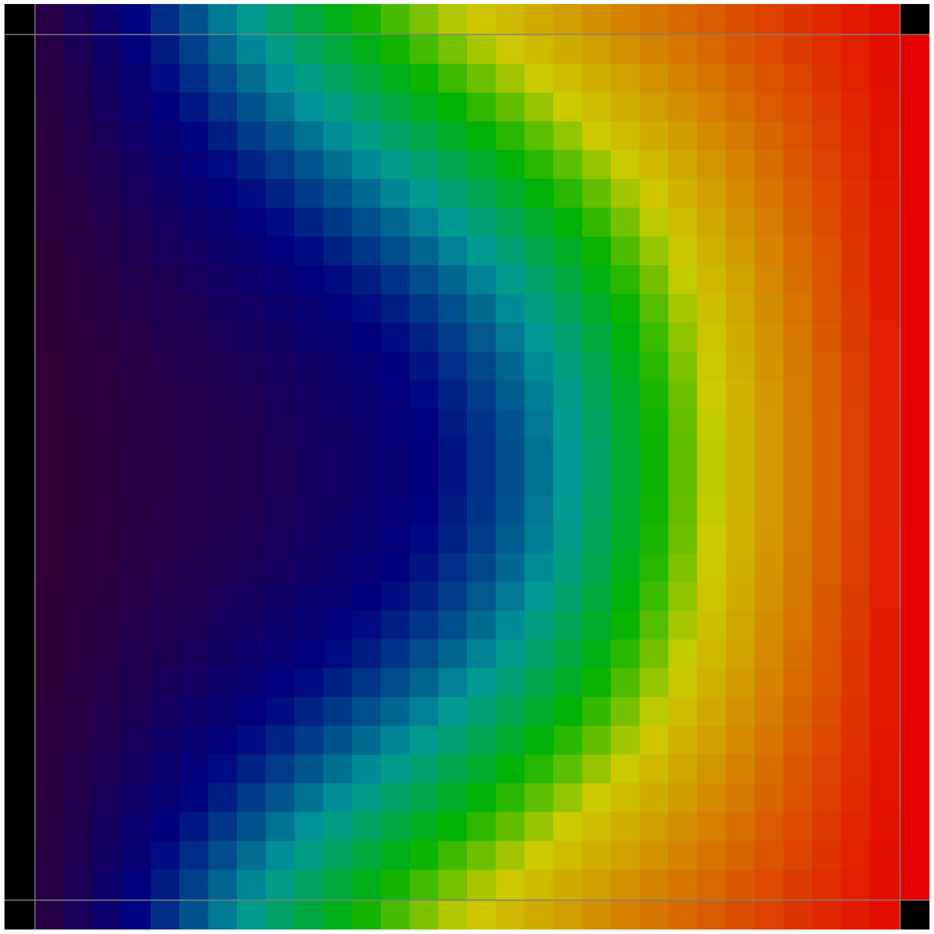
Example Application

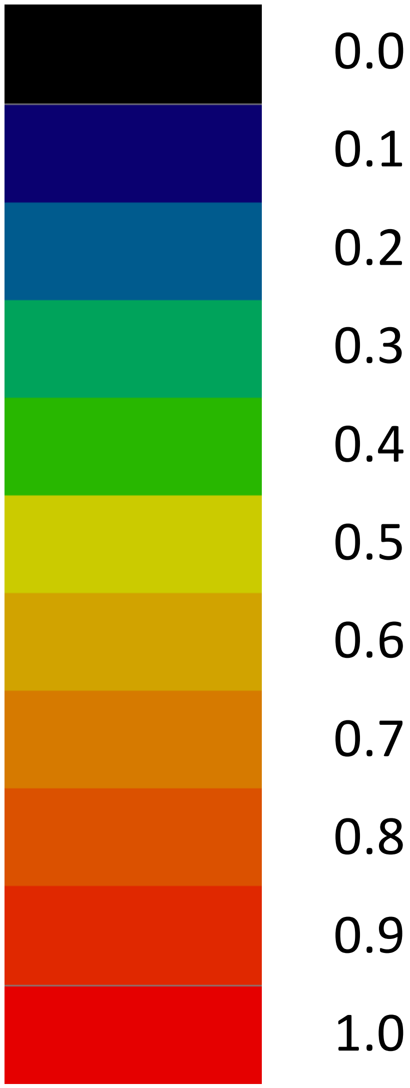


```
A_new[r,c] = 0.2 *
    (A_curr[r,   c  ] +
     A_curr[r,   c-1] +
     A_curr[r-1, c  ] +
     A_curr[r,   c+1] +
     A_curr[r+1, c  ]);
```

- Jacobi Iterations
- Compute new state for each grid point based on current state
- Avoids sequential dependency among grid points

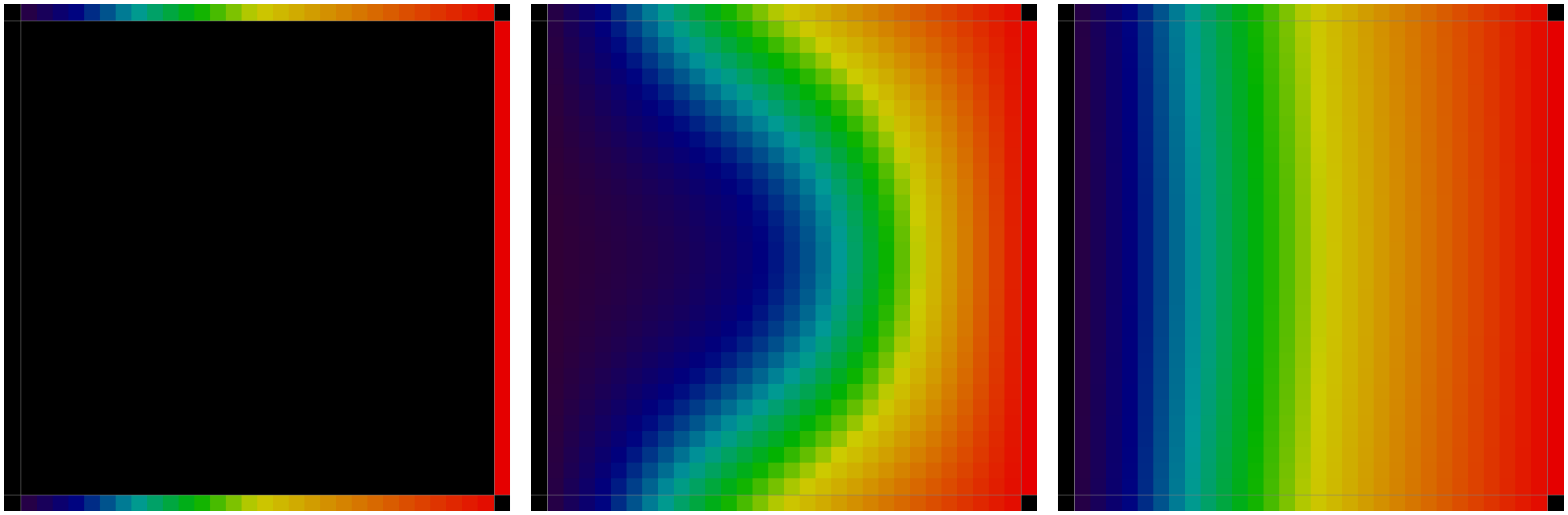
Visualization of Grid Solver





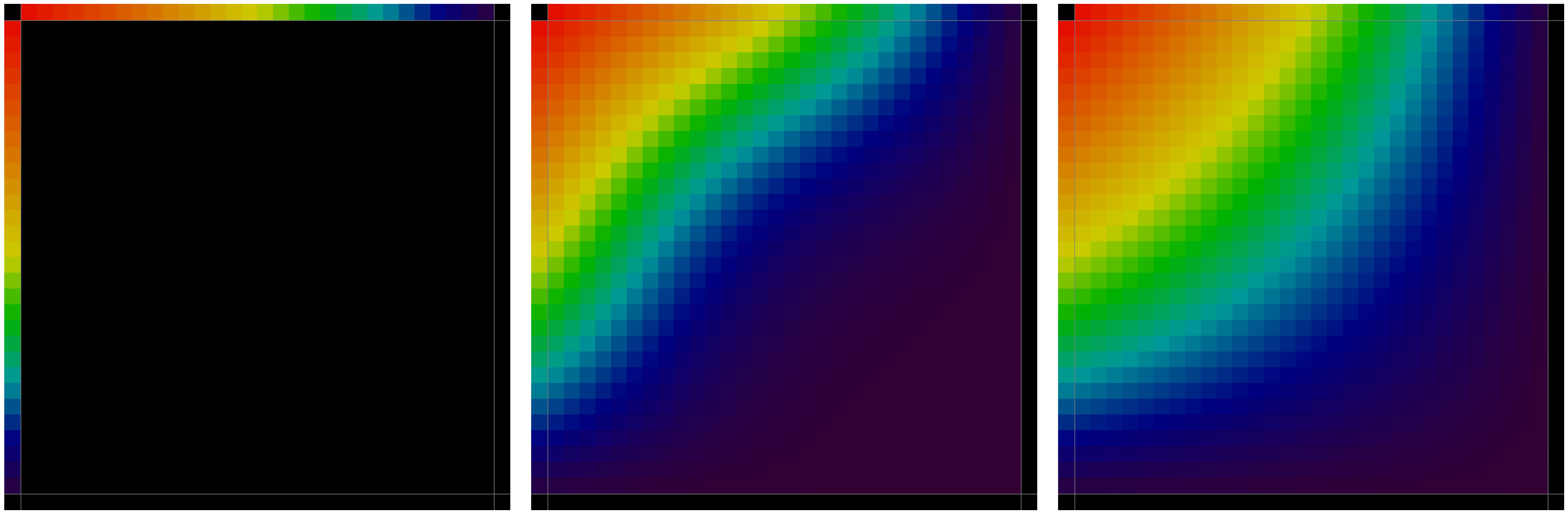
- **Outer ring shows boundary conditions**
- **Inner 30 x 30 grid shows grid values**

Horizontal Boundary Conditions



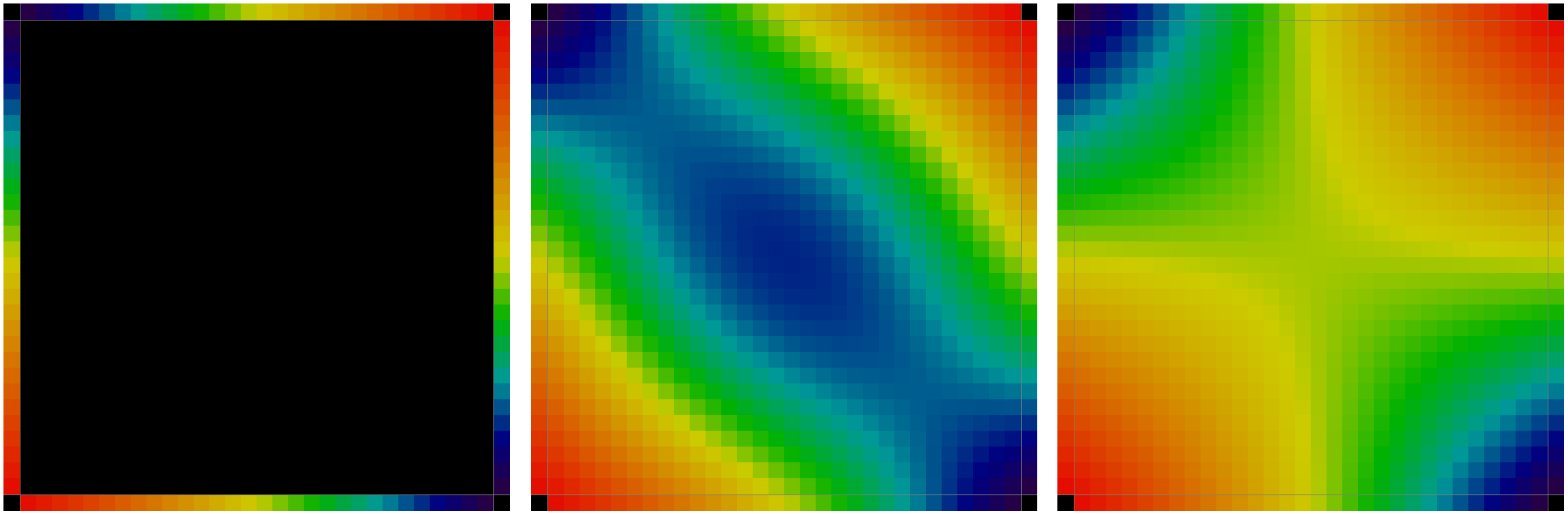
- ./heat.py –V –b h

Corner Boundary Conditions



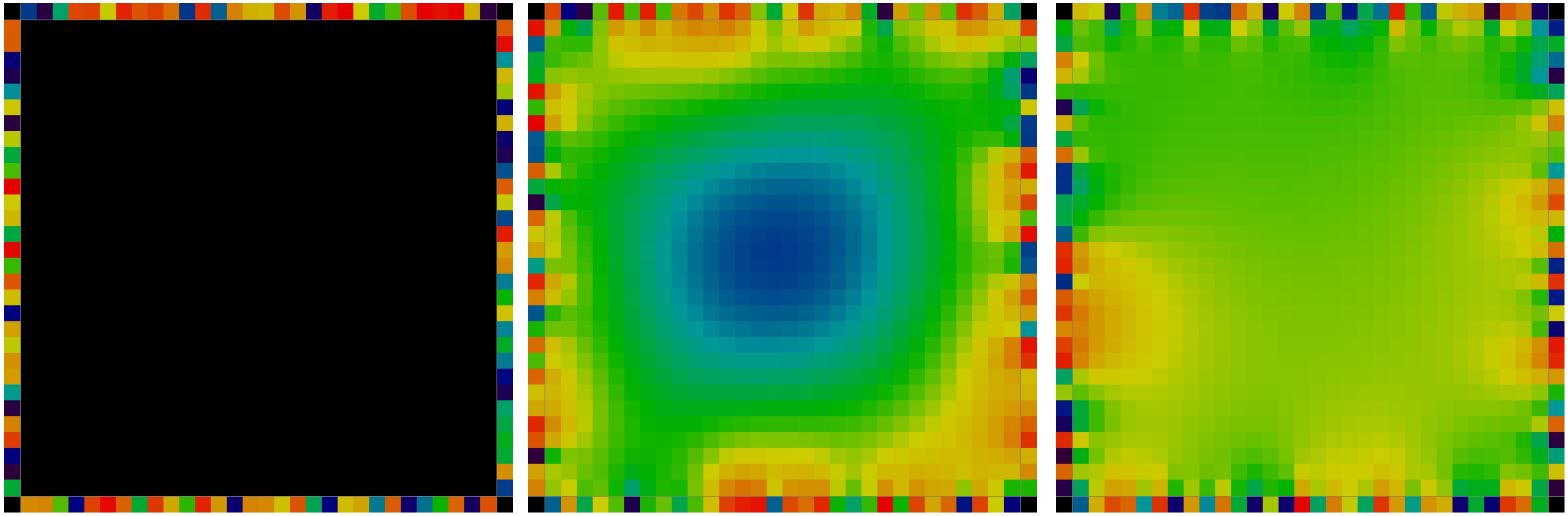
- ./heat.py –V –b c

Diagonal Boundary Conditions



- ./heat.py –V –b d

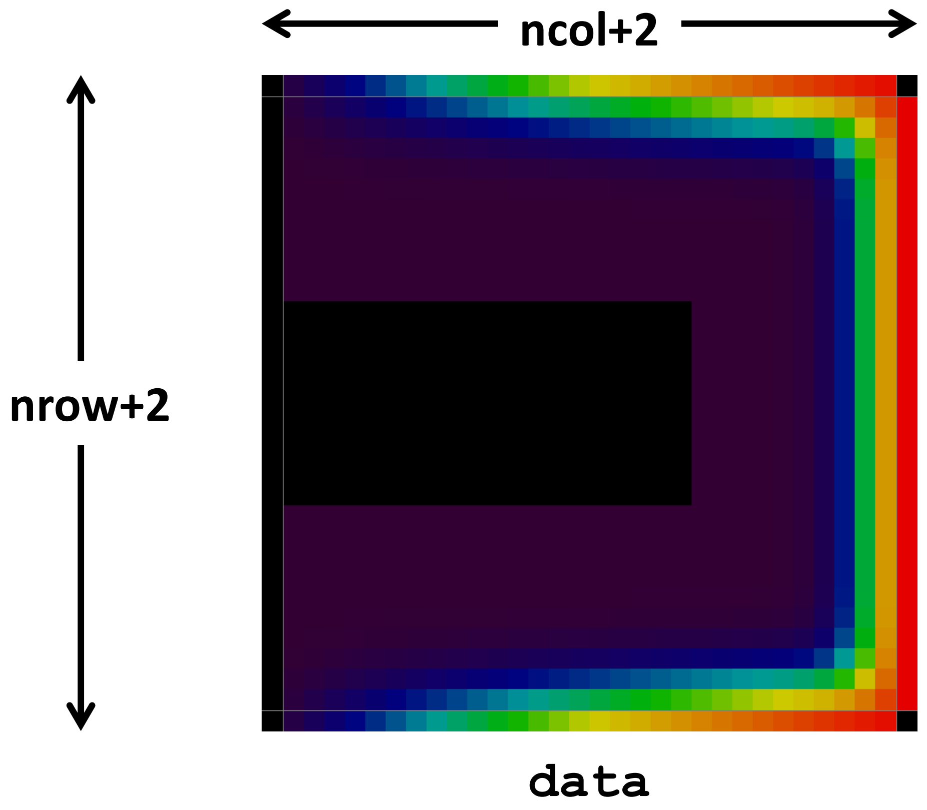
Random Boundary Conditions

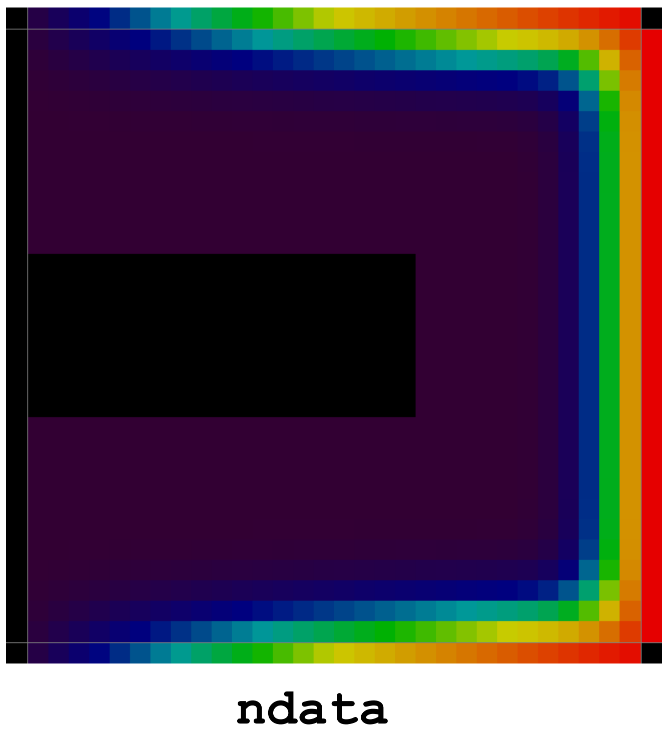


- `./heat.py –V –b r`

Sequential Version: Grid Representation

```
typedef struct {
    int nrow;
    int ncol;
    double *data;  // Size = (nrow+2) X (ncol+2)
    double *ndata; // Size = (nrow+2) X (ncol+2)
} grid_t;
```





Sequential Code Structure

```
static double step_grid(grid_t *g) {
    double maxdiff = 0.0;
    for (int r = 0; r < g->nrow; r++) {
        for (int c = 0; c < g->ncol; c++) {

            Compute g->ndata[r,c]

            Compute diff = |g->data[r,c] - g->ndata[r,c]|
            maxdiff = max(diff, maxdiff)

        }
    }

    Swap g->data and g->ndata

    return maxdiff;
}
```

- Keep stepping until `maxdiff < epsilon`

Computing New State for One Grid Point

- Indexing into grid (row-major order)

```
#define GINDEX(g, r, c) (((r)+1)*((g)->ncol+2)+((c)+1))
```

- Fraction of new state coming from adjacent grid points

```
#define CONDUCTIVITY 0.8
```

- Computation of `g->ndata[GINDEX(g, r, c)]`

```
static inline double new_state(grid_t *g, int r, int c) {
    double ov = g->data[GINDEX(g, r,   c)];
    double nv = g->data[GINDEX(g, r-1, c)];
    double ev = g->data[GINDEX(g, r, c+1)];
    double sv = g->data[GINDEX(g, r+1, c)];
    double wv = g->data[GINDEX(g, r,  c-1)];
    return 0.25 * CONDUCTIVITY * (nv+ev+sv+wv)
            + (1-CONDUCTIVITY) * ov;
}
```

Additional Points

- **Exchanging Data Arrays**

```
    double *tdata = g->data;
    g->data = g->ndata;
    g->ndata = tdata;
```

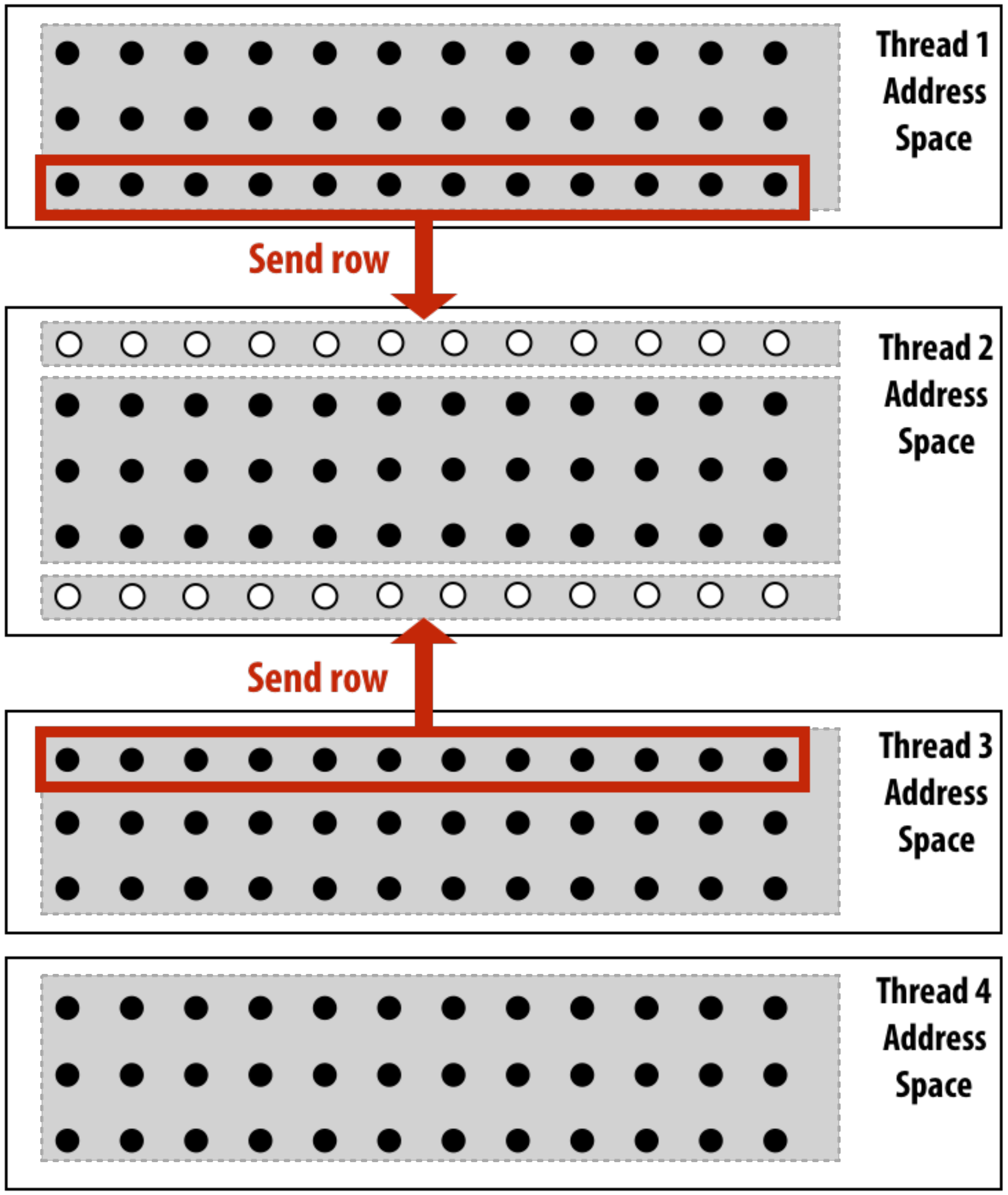
- **Properties of Computation**
  - On each step, **`g->data`** is read-only, **`g->ndata`** is write-only
  - Requires storage for 16 bytes / grid point
    - GHC machines have 12MiB L3 cache
      - Up to 886 X 886 array
    - Latedays nodes have 15MiB L3 cache
      - Up to 991 X 991 array

Using OpenMP for Parallelism

```
static double step_grid(grid_t *g) {
    double maxdiff = 0.0;

#pragma omp parallel for schedule(static) reduction(max:maxdiff)
    for (int r = 0; r < g->nrow; r++) {
        for (int c = 0; c < g->ncol; c++) {

            Compute g->ndata[r,c]

            Compute diff = |g->data[r,c] - g->ndata[r,c]|
            maxdiff = max(diff, maxdiff)

        }
    }

    Swap g->data and g->ndata

    return maxdiff;
}
```

- Easy!

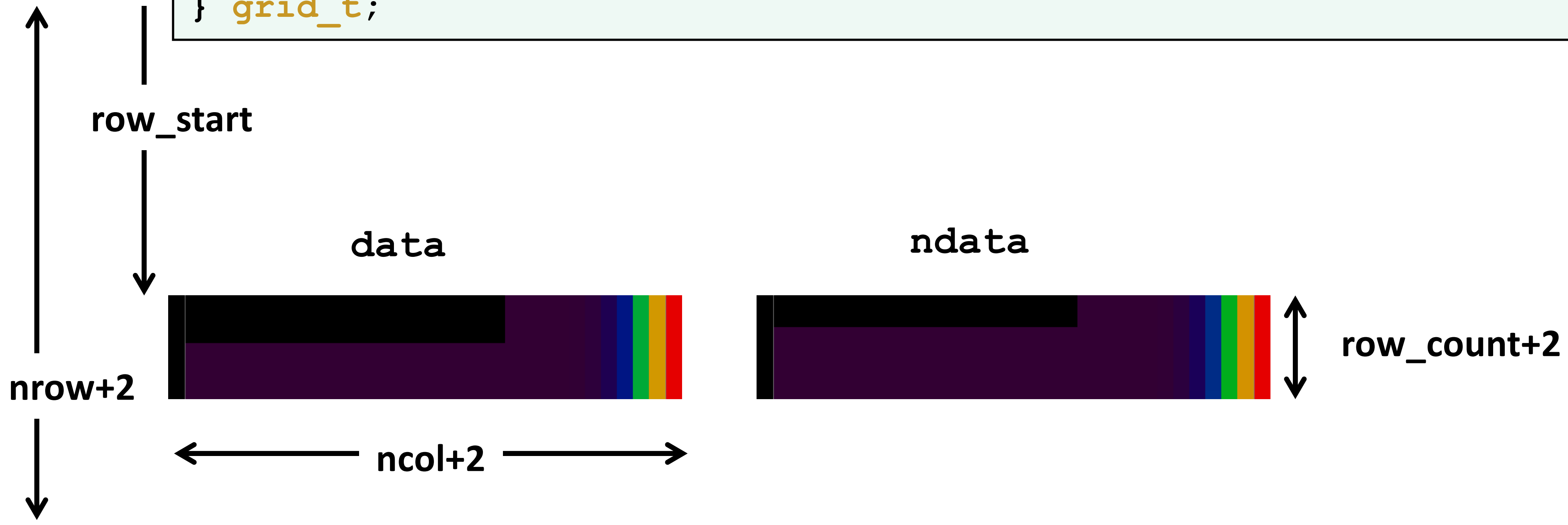
Message-Passing Decomposition
From Lecture #5



- Each process *p* maintains state for subrange of rows
  - Plus storage for "ghost cells" above and below
- Exchanges boundary data with processes *p*–1 and *p*+1 on each step
  - Fill in ghost cells
- Computes local value of maxdiff
- Globally: find max of maxdiff's

Parallel Version: Grid Representation

```
typedef struct {
    int nrow;                // Total rows in grid
    int ncol;                // Columns in grid
    int process_id;
    int process_count;
    int row_count;           // Number of rows in local region
    int row_start;           // Offset of local rows from global rows
    double *data;            // Size = (row_count+2) X (ncol+2)
    double *ndata;           // Size = (row_count+2) X (ncol+2)
} grid_t;
```



Message Passing Code Structure

```
static double step_grid(grid_t *g) {
    double local_maxdiff = 0.0;
    for (int r = 0; r < g->row_count; r++) {
        for (int c = 0; c < g->ncol; c++) {

            Compute g->ndata[r,c]

            Compute diff = |g->data[r,c] - g->ndata[r,c]|
            local_maxdiff = max(diff, local_maxdiff)
        }
    }

    Swap g->data and g->ndata

    Exchange rows with neighbors

    double maxdiff = global_maximum(local_maxdiff)

    return maxdiff;
}
```

Exchanging Row Data

```
/* Exchange rows with neighbors to north and south */
static void exchange_rows(grid_t *g) {
    if (g->row_count == 0)
        return;
    int process_id = g->process_id;
    bool north_neighbor = process_id > 0;
    bool south_neighbor = process_id < g->process_count - 1;

    if (north_neighbor)
        Start sending row 0 north
    if (south_neighbor)
        Start sending row row_count–1 to south
    if (north_neighbor)
        Receive row –1 from north
    if (south_neighbor)
        Receive row row_count from south
    if (north_neighbor)
        Wait until finished sending data north
    if (south_neighbor)
        Wait until finished sending data south
}
```

Asynchronous Send

Synchronous Receive

Send/Receive Functions

- Keeping record of asynchronous send or receive

```
MPI_Request north_request;
```

- Asynchronous send

```
double *north_boundary = &g->data[GINDEX(g, 0, 0)];
start_send_data(north_boundary, g->ncol, process_id - 1,
                &north_request);
```

- Synchronous receive

```
double *north_ghost = &g->data[GINDEX(g, -1, 0)];
receive_data(north_ghost, g->ncol, process_id - 1);
```

- Completion of asynchronous send

```
finish_data(&north_request);
```

Asynchronous Send Function

- **Call to wrapper function**

```
double *north_boundary = &g->data[GINDEX(g, 0, 0)];
start_send_data(north_boundary, g->ncol, process_id - 1,
                &north_request);
```

- **Wrapper Implementation**

```
static void start_send_data(double *data, int count, int process_id,
                            MPI_Request *request) {
    MPI_Isend(data, count, MPI_DOUBLE, process_id,
              0, MPI_COMM_WORLD, request);
}
```

  - Send **`count`** double's at **`data`** to **`process_id`** and track with **`request`**
  - Note how send buffer is a portion of the grid data array

Synchronous Receive Function

- Call to wrapper function

```
double *north_ghost = &g->data[GINDEX(g, -1, 0)];
receive_data(north_ghost, g->ncol, process_id - 1);
```

- Wrapper Implementation

```
static void receive_data(double *data, int count, int process_id) {
    MPI_Recv(data, count, MPI_DOUBLE, process_id,
             0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

  - Receive **`count`** double's into **`data`** from **`process_id`**
  - Note how receive buffer is a portion of the grid data array

Asynchronous Completion Function

- **Call to wrapper function**

```
finish_data(&north_request);
```

- **Wrapper Implementation**

```
static void finish_data(MPI_Request *request) {
    MPI_Wait(request, MPI_STATUS_IGNORE);
}
```

  - Wait until communication tracked with `request` has completed
  - Can also be used to wait for completion of asynchronous receive

MPI Send/Receive APIs

```
int MPI_Isend(const void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm, MPI_Request *request)

int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm, MPI_Status *status)

int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

Argument	Meaning
buf	Send / receive buffer
count	Number of items
datatype	Data type of items
source / dest	Rank of source or destination
tag	Integer identifier to distinguish messages
comm	Subset of processes
request	Struct for tracking send/receive status
status	Struct for recording communication status

Choosing Synchronous vs. Asynchronous

- **When exchanging data, best if at least one of send / receive is asynchronous**
  - Avoids deadlock
- **Asynchronous lets more things happen simultaneously**
  - Especially, to start sending messages to neighbors
  - Possible to overlap with grid computation
    - E.g., process grid points along boundaries first, then overlap data exchange with internal grid point computation
- **Synchronous is simpler**
- **Experiments**
  - Found no performance difference using asynch send + asynch receive, vs. asynch send + synch receive
  - Did not try overlapping with grid computation

Broadcasting

- **Same function for broadcaster and receivers**

```
static void broadcast_receive_data(double *data, int count) {
    /* Master process will send.  Others will receive */
    MPI_Bcast(data, count, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

- **Process 0 broadcasts**
  - Send contents of buffer **`data`** to all other processes
- **Other processes receive**
  - Store received data to buffer **`data`**
- **In grid solver**
  - Process 0 broadcasts boundary condition data before starting solver
  - All others update their boundary data

Global Reduction

```
static double global_maximum(double local_max) {
    double result = 0.0;
    MPI_Allreduce(&local_max, &result, 1, MPI_DOUBLE,
                  MPI_MAX, MPI_COMM_WORLD);
    return result;
}
```

  - Find global maximum for one value of type double

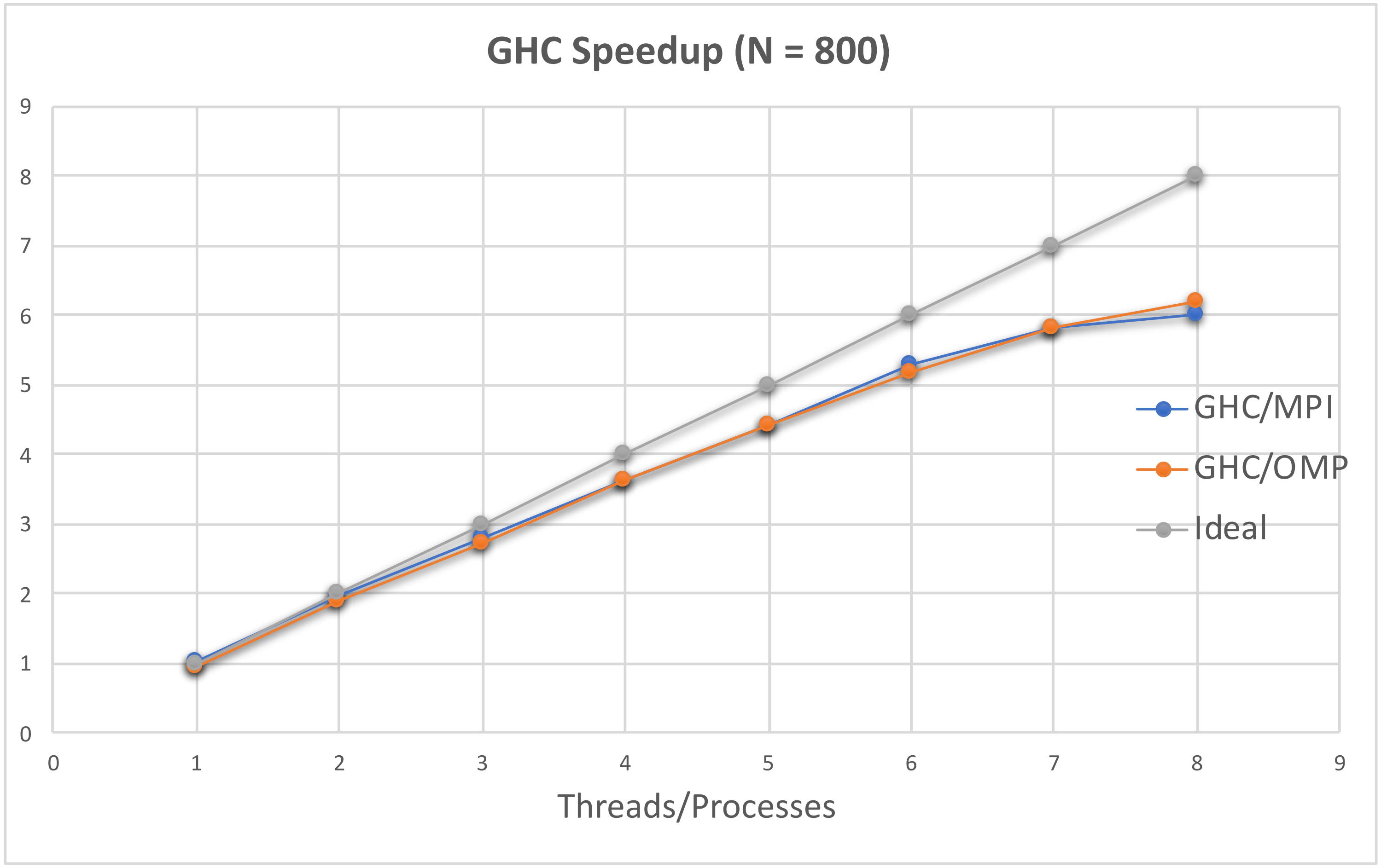
- **API**

```
int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count,
                  MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

  - Perform reduction operation for each of the elements in the sendbuf's. Distribute the results to the recvbuf's.

Performance on GHC Machines

- **Speedups for OMP & MPI with 800 X 800 grid**



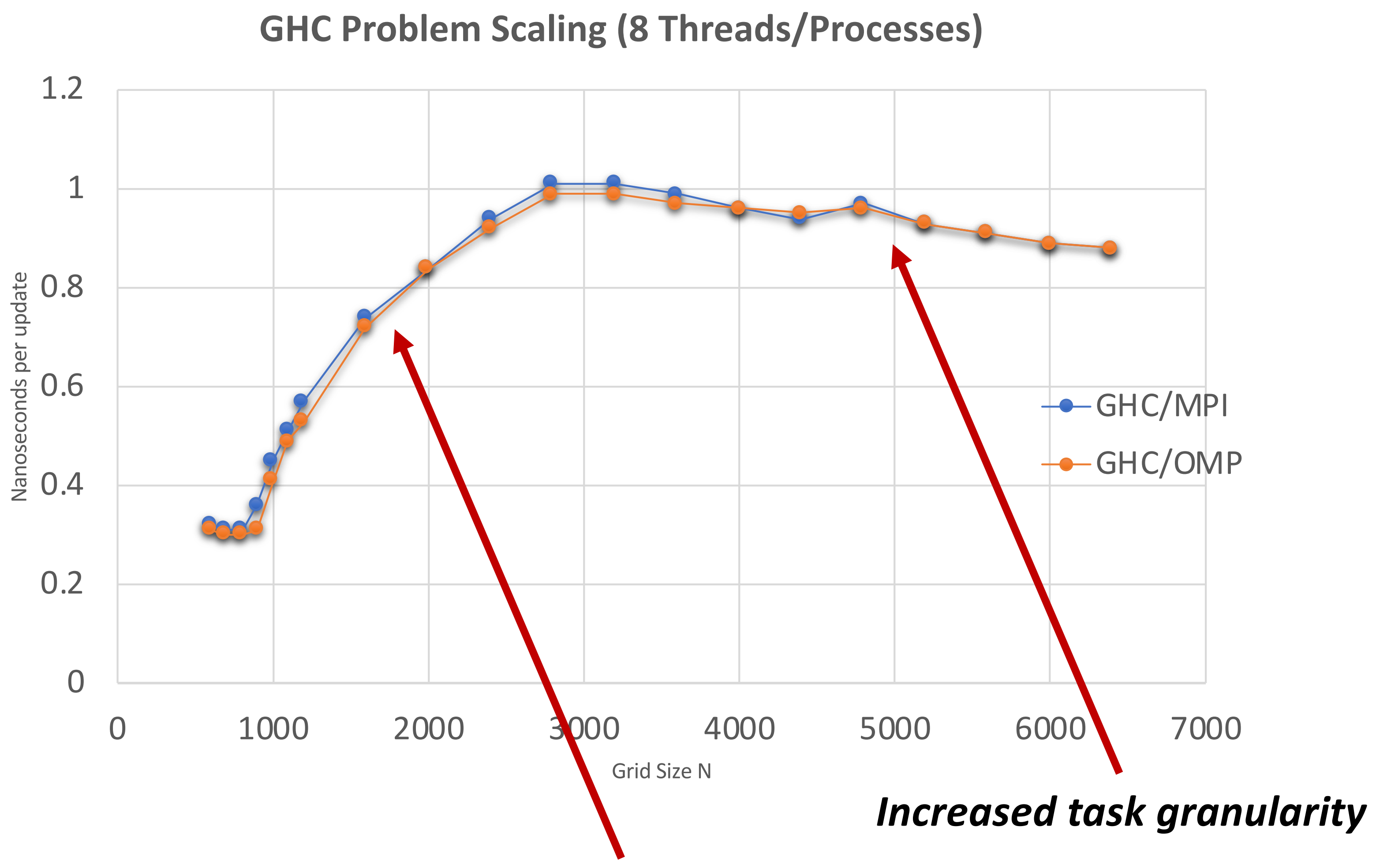
- **Nearly the same. Reasonably good**

Performance on GHC Machines

- **Problem scaling with 8 threads / processes**
- **Express in units of *Nanoseconds per update***
  - N x N grid. S steps. Time T seconds
    - NPU = $T * 10^9 / (N * N * S)$
  - Describes time / work performed

Performance on GHC Machines

- Problem scaling with 8 threads / processes

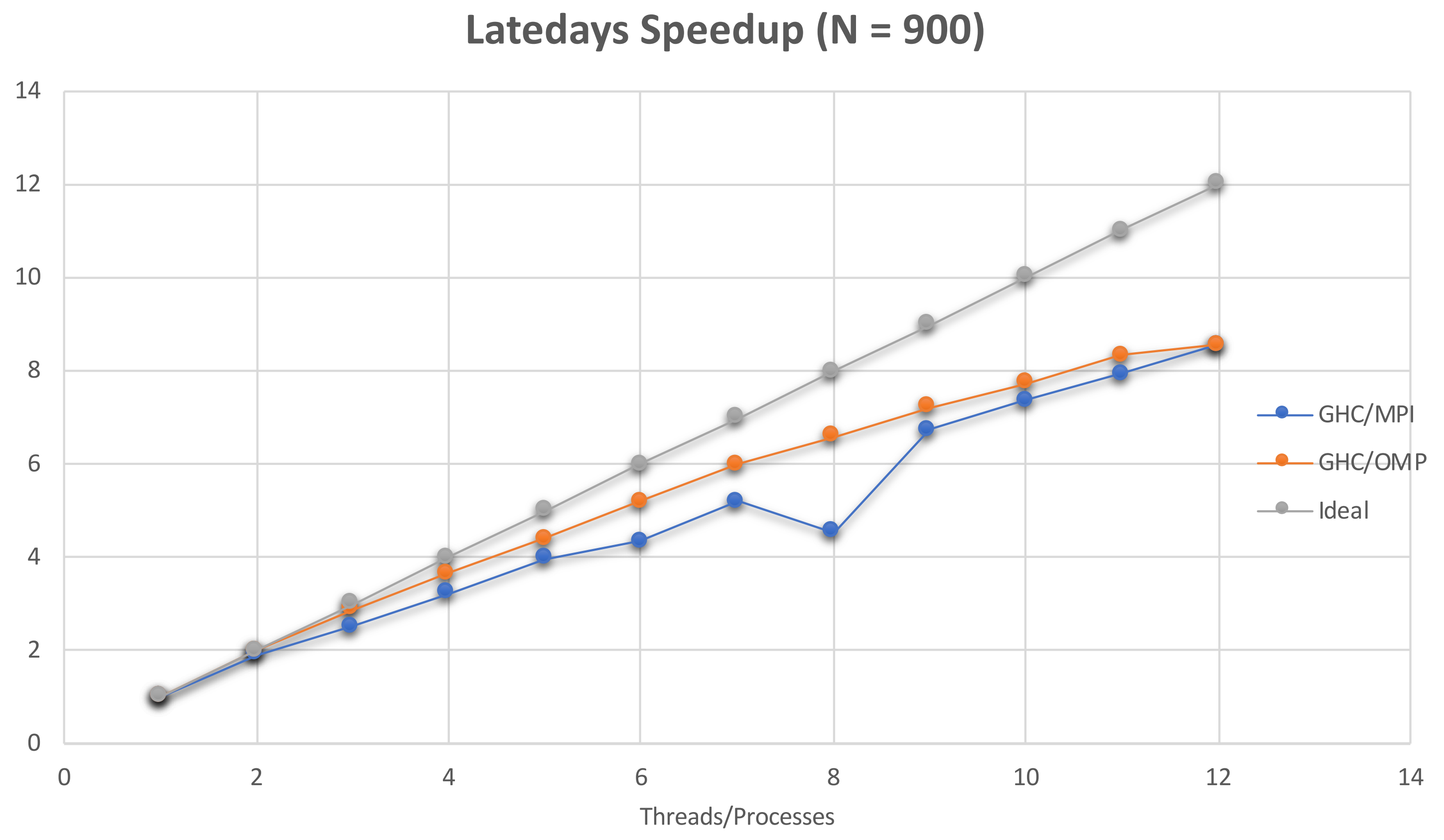


Exceeds L3 cache capacity

- OMP / MPI nearly the same.

Performance on Latedays Node

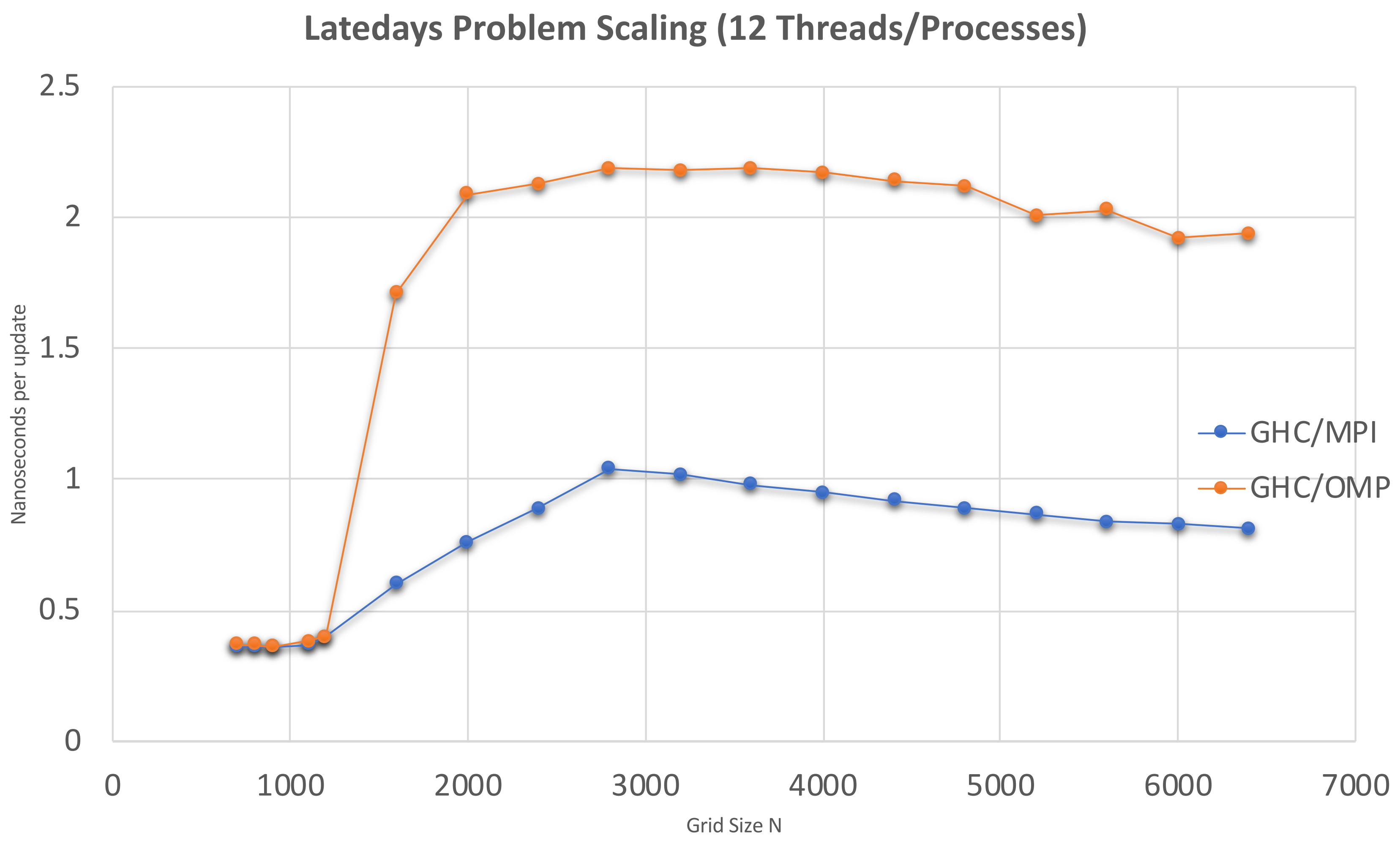
- Speedups for OMP & MPI with 900 X 900 grid



- Generally comparable

Performance on Latedays Node

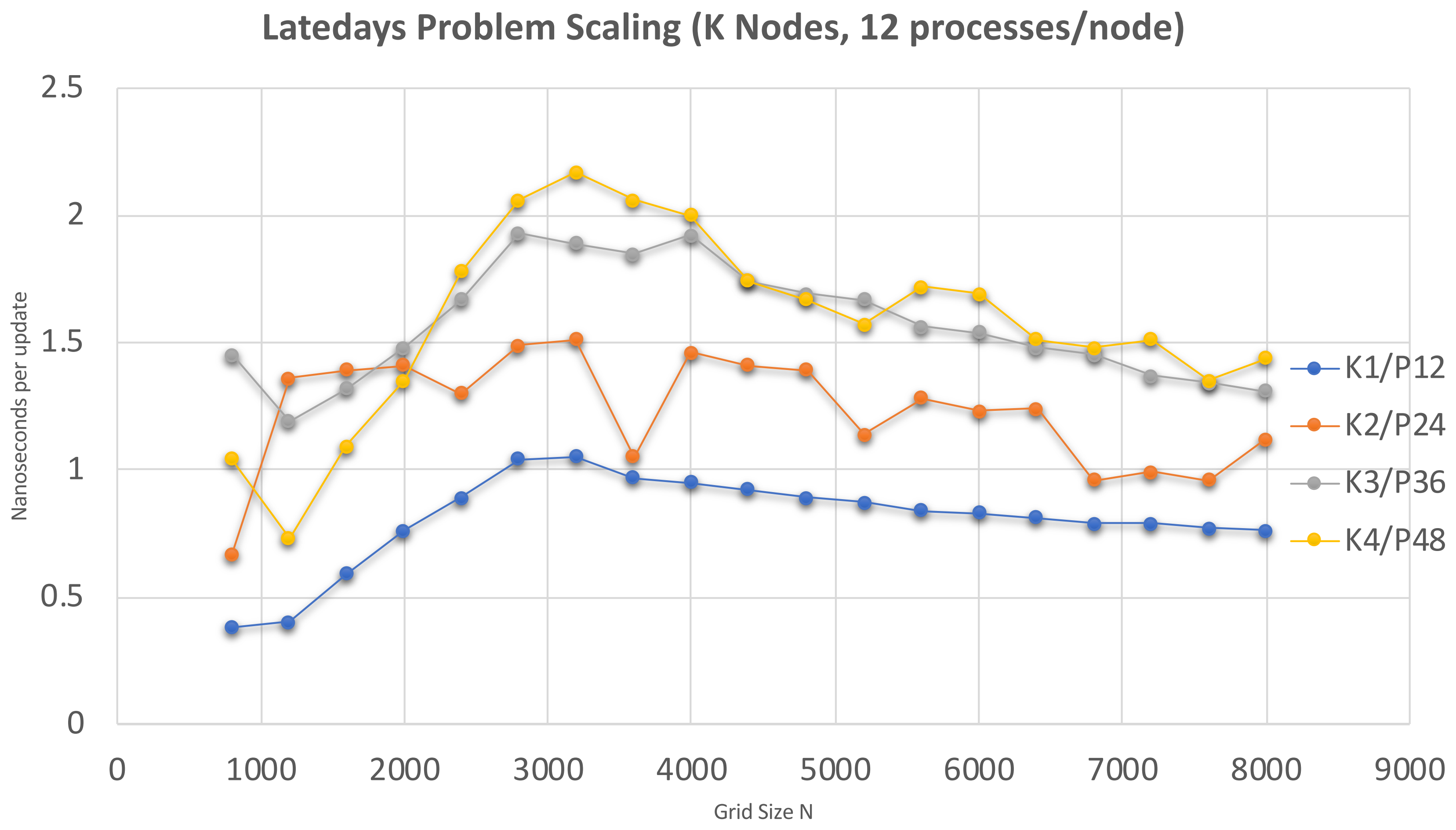
- **Problem scaling with 12 threads / processes**



- **OMP much worse than MPI once exceed L3 cache**
  - Less read/write conflict?

Using Multiple Latedays Nodes

- Run on K nodes with 12 * K processes



- Internode communication has too much overhead to give any speedup

Some Key Points

- **MPI is a large and complex standard, but can do lots with only core set of operations**
- **Well designed for bulk synchronous computations**
  - Repeated steps:
    - Each process updates its portion of state
    - Each process communicates boundary information with neighbors
    - Collectively test for convergence
- **Users must explicitly manage buffers**
  - Can extract data from or insert data into state data arrays
  - Don't overwrite while waiting for asynchronous operation to complete

Useful Resources

- **General Tutorial**
  - Good coverage of concepts and basics
  - https://mpitutorial.com/tutorials/
- **Longer MPI Tutorial**
  - More comprehensive and detailed
  - https://computing.llnl.gov/tutorials/mpi/
- **Documentation for MPI 1.6**
  - Default version running on Latedays
  - https://www.open-mpi.org/doc/v1.6/