

## Introduction to MPI

Presented by: Kevin Glass  
and Doug Baxter

# Outline

- ▶ MPI basics
- ▶ Matrix multiplication serial code
- ▶ Parallelization assumptions
  - Matrix multiplication decomposition 1
  - Matrix multiplication decomposition 2
  - Matrix multiplication decomposition 3
  - Matrix multiplication decomposition 4
  - Matrix multiplication in groups
- ▶ Problems with matrix multiplication assumptions

# MPI Basics

## ► What is MPI?

- Message Passing Interface
- Supports communications between processors
- Details hidden behind a set of function calls

## ► Architecture

- Sets up remote process communications
- Sets up communications topologies

## ► API

- Provides users with structured communications

# MPI Code Basics

## ► Structure of MPI code

- MPI\_Init initializes() MPI runtime environment
- MPI\_Comm\_size () returns the number of processors used by MPI (important for decomposition)
- MPI\_Comm\_rank () returns unique id for each processor
- MPI\_Finalize () shuts down runtime environment
  - This is crucial, failure to properly shutdown code could (will) cause other nodes in program to hang

## ► NOTE: Processes are spawned when the code is executed, not at MPI\_Init ()

## A few cautions

### ► MPI can't do everything

- Failure to detect and resolve problems will hang the system
- Failure to set up communications properly will hang the system
- Failure to pass “promised” data will hang the system
- **HANGING THE SYSTEM UPSETS ADMINISTRATORS**

# Hello World

- ▶ Starting example: parallel “hello world”
  - As good a place as any
  - Examine basic, important MPI functions

# Hello world—the code

```
#include "mpi.h"
#include <stdio.h>
int
main(int argc, char * argv[]) {
    int rank;
    int nprocs;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD,
                  &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,
                  &rank);

    printf("Hello world from %d of %d
processes\n", rank, nprocs);

    MPI_Finalize();
    return 0;
}
```

# Hello world—the code

```
gcc -o hello hello.c -L${MPI_LIB} -lmpi -I${MPI_INCLUDE}
```

```
mpirun -np 4 ./hello
Hello world from 0 of 4 processes
Hello world from 1 of 4 processes
Hello world from 2 of 4 processes
Hello world from 3 of 4 processes
```

```
mpirun -np 4 ./hello
Hello world from 2 of 4 processes
Hello world from 0 of 4 processes
Hello world from 3 of 4 processes
Hello world from 1 of 4 processes
```

# Outline

- ▶ MPI basics
- ▶ Matrix multiplication serial code
- ▶ Parallelization assumptions
  - Matrix multiplication decomposition 1
  - Matrix multiplication decomposition 2
  - Matrix multiplication decomposition 3
  - Matrix multiplication decomposition 4
  - Matrix multiplication in groups
- ▶ Problems with matrix multiplication assumptions

## Serial *matmul*

- ▶ Restructured and refined version (ref. “Introduction to Parallel Computing”)
  - Divided into small programming units
  - Separated into functional groups
  - Error detection
  - Code is available

# Serial *matmul*—*matmul.c*

```
#include <stdio.h>
#include <stdlib.h>
#include "matrix.h"
#include "dataReader.h"

int
main(int argc, char * argv[]) {
    matrix * A = NULL;
    matrix * B = NULL;
    matrix * C = NULL;

    A = makeMatrix();
    getMatrix(A, argv[1]);
    B = identityMatrix(A->rows, A->cols);
    C = makeMatrix();
    initializeMatrix(C, A->rows, B->cols);

    multiply(A, B, C);
    cleanUp(A, B, C);
    return 0;
}
```

# Serial *matmul*—*matmul.c*

```
#ifndef MATRIX_H
#define MATRIX_H
typedef struct matrix
{
    int                     rows;
    int                     cols;
    int                     size;
    double                 * elements;
} matrix;

matrix * makeMatrix(void);
matrix * initializeMatrix(matrix * mat, int rows, int cols);

int zeroMatrix(matrix * mat);
matrix * identityMatrix(int rows, int cols);
int printMatrix(matrix * mat);
int multiply(matrix * A, matrix * B, matrix * C);
void cleanUp(matrix * A, matrix * B, matrix * C);

#endif
```

## Serial *matmul*—*matmul.c*

```
#include <stdio.h>
#include <stdlib.h>
#include "matrix.h"
#include "dataReader.h"

int
main(int argc, char * argv[]) {
    matrix * A = NULL;
    matrix * B = NULL;
    matrix * C = NULL;

    A = makeMatrix();
    getMatrix(A, argv[1]);
    B = identityMatrix(A->rows, A->cols);
    C = makeMatrix();
    initializeMatrix(C, A->rows, B->cols);

    multiply(A, B, C);
    cleanUp(A, B, C);
    return 0;
}
```

# Serial *matmul*—matrix functions

```
#include "matrix.h"
#include "matErrors.h"
#include <malloc.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

matrix *
makeMatrix(void)
{
    matrix          * mat      = (matrix *) malloc(sizeof(matrix));
    matrixAllocateCheck(mat, __FILE__, __LINE__); // terminates if check
    fails

    mat->rows      = 0;
    mat->cols      = 0;
    mat->size       = 0;
    mat->elements  = NULL;

    return mat;
}
```

# Serial *matmul*—*matmul.c*

```
#include <stdio.h>
#include <stdlib.h>
#include "matrix.h"
#include "dataReader.h"

int
main(int argc, char * argv[]) {
    matrix * A = NULL;
    matrix * B = NULL;
    matrix * C = NULL;

    A = makeMatrix();
    getMatrix(A, argv[1]);
    B = identityMatrix(A->rows, A->cols);
    C = makeMatrix();

    initializeMatrix(C, A->rows, B->cols);
    multiply(A, B, C);
    cleanUp(A, B, C);
    return 0;
}
```

# Serial *matmul*—data access

```
#include "matrix.h"
#include "dataReader.h"
#include <unistd.h>

int
getMatrix(matrix * mat, const char * fileName)
{
    int          fileSize      = 0;
    int          fh            = 0;
    int          err           = 0;

    fileSize = verifyMatrixFile(fileName);
    fh = openMatrixFile(fileName);

    setMatrixSize(fh, mat, fileSize);
    loadMatrix(fh, mat);

    close(fh);

    return err;
}
```

# Serial *matmul*—data access

```
#include "matrix.h"
#include "matErrors.h"
#include "dataReader.h"

#include <unistd.h>
#include <malloc.h>
#include <sys/stat.h>
#include <fcntl.h>

int
verifyMatrixFile(const char * fileName)
{
    struct stat    statInfo;
    int            err        = 0;
    int            size       = 0;

    err = lstat(fileName, & statInfo);

    fileFoundCheck(err, __FILE__, __LINE__);

    size = (int)statInfo.st_size;
    zeroSizeCheck(size, __FILE__, __LINE__);

    return size;
}
```

# Serial *matmul*—data access

```
#include "matrix.h"
#include "dataReader.h"
#include <unistd.h>

int
getMatrix(matrix * mat, const char * fileName)
{
    int          fileSize      = 0;
    int          fh            = 0;
    int          err           = 0;

    fileSize = verifyMatrixFile(fileName);
    fh = openMatrixFile(fileName);

    setMatrixSize(fh, mat, fileSize);
    loadMatrix(fh, mat);

    close(fh);

    return err;
}
```

# Serial *matmul*—data access

```
int openMatrixFile(const char * fileName)
{
    int fh = open(fileName, O_RDONLY);
    fileOpenCheck(fh, __FILE__, __LINE__);
    return fh;
}
```

# Serial *matmul*—data access

```
#include "matrix.h"
#include "dataReader.h"
#include <unistd.h>

int
getMatrix(matrix * mat, const char * fileName)
{
    int          fileSize      = 0;
    int          fh            = 0;
    int          err           = 0;

    fileSize = verifyMatrixFile(fileName);
    fh = openMatrixFile(fileName);

    setMatrixSize(fh, mat, fileSize);
    loadMatrix(fh, mat);

    close(fh);

    return err;
}
```

# Serial *matmul*—data access

```
int
setMatrixSize(int fileHandle, matrix * mat, int fileSize)
{
    size_t    readSize      = 0;
    int       rows         = 0;
    int       cols         = 0;
    size_t    expectedSize = 0;

    readSize = read(fileHandle, &rows, sizeof(int));
    readFailureCheck((int)readSize, sizeof(int), __FILE__, __LINE__);

    readSize = read(fileHandle, &cols, sizeof(int));
    readFailureCheck((int)readSize, sizeof(int), __FILE__, __LINE__);

    initializeMatrix(mat, rows, cols);

    expectedSize = rows*cols*sizeof(double) + 2*sizeof(int);
    fileSizeMismatchCheck((int)expectedSize, fileSize, __FILE__,
                          __LINE__);

    return 0;
}
```

# Serial *matmul*—data access

```
#include "matrix.h"
#include "dataReader.h"
#include <unistd.h>

int
getMatrix(matrix * mat, const char * fileName)
{
    int          fileSize      = 0;
    int          fh            = 0;
    int          err           = 0;

    fileSize = verifyMatrixFile(fileName);
    fh = openMatrixFile(fileName);

    setMatrixSize(fh, mat, fileSize);
    loadMatrix(fh, mat);

    close(fh);

    return err;
}
```

# Serial *matmul*—data access

```
int
loadMatrix(int fileHandle, matrix * mat)
{
    size_t          readSize      = 0;
    int            err           = 0;

    readSize = read(fileHandle, mat->elements, mat-> size*sizeof(double));
    readFailureCheck((int)readSize, mat-> size*sizeof(double), __FILE__,
                      __LINE__);

    return err;
}
```

# Serial *matmul*—data access

```
#include "matrix.h"
#include "dataReader.h"
#include <unistd.h>

int
getMatrix(matrix * mat, const char * fileName)
{
    int          fileSize      = 0;
    int          fh            = 0;
    int          err           = 0;

    fileSize = verifyMatrixFile(fileName);
    fh = openMatrixFile(fileName);

    setMatrixSize(fh, mat, fileSize);
    loadMatrix(fh, mat);

    close(fh);

    return err;
}
```

# Serial *matmul*—*matmul.c*

```
#include <stdio.h>
#include <stdlib.h>
#include "matrix.h"
#include "dataReader.h"

int
main(int argc, char * argv[]) {
    matrix * A = NULL;
    matrix * B = NULL;
    matrix * C = NULL;

    A = makeMatrix();
    getMatrix(A, argv[1]);
    B = identityMatrix(A->rows, A->cols);
    C = makeMatrix();

    initializeMatrix(C, A->rows, B->cols);
    multiply(A, B, C);
    cleanUp(A, B, C);
    return 0;
}
```

# Serial *matmul*—matrix type

```
matrix * identityMatrix(int rows, int cols){  
    matrix * rtn = makeMatrix();  
    double * e_ptr = elements;  
    int i = 0, j = 0;  
    rtn->rows = rows; rtn->cols = cols; rtn->size = rows*cols;  
    rtn->elements = (double *)malloc(sizeof(double)*size);  
  
    for (i = 0; i < rows; i++) {  
        for(j = 0; j < cols; j++) {  
            if (i == j) {  
                *e_ptr = 1.0;  
            } else {  
                *e_ptr = 0.0;  
            }  
            ++e_ptr;  
        }  
    }  
    return rtn;  
}
```

# Serial *matmul*—*matmul.c*

```
#include <stdio.h>
#include <stdlib.h>
#include "matrix.h"
#include "dataReader.h"

int
main(int argc, char * argv[]) {
    matrix * A = NULL;
    matrix * B = NULL;
    matrix * C = NULL;

    A = makeMatrix();
    getMatrix(A, argv[1]);
    B = identityMatrix(A->rows, A->cols);
    C = makeMatrix();

    initializeMatrix(C, A->rows, B->cols);
    multiply(A, B, C);
    cleanUp(A, B, C);
    return 0;
}
```

# Serial *matmul*—matrix functions

```
matrix *
initializeMatrix(matrix * mat, int rows, int cols)
{
    mat->rows      = rows;
    mat->cols      = cols;
    mat->size       = rows * cols;

    mat->elements   = (double *)malloc(sizeof(double) * mat->size);
    zeroMatrix(mat);
    return mat;
}

int
zeroMatrix(matrix * mat)
{
    int size = mat->rows * mat->cols;
    double * elements = mat->elements;
    int i = 0;

    for (i = 0; i < size; i++)
    {
        elements[i] = 0;
    }
    return 0;
}
```

# Serial *matmul*—*matmul.c*

```
#include <stdio.h>
#include <stdlib.h>
#include "matrix.h"
#include "dataReader.h"

int
main(int argc, char * argv[]) {
    matrix * A = NULL;
    matrix * B = NULL;
    matrix * C = NULL;

    A = makeMatrix();
    getMatrix(A, argv[1]);
    B = identityMatrix(A->rows, A->cols);
    C = makeMatrix();

    initializeMatrix(C, A->rows, B->cols);
    multiply(A, B, C);
    cleanUp(A, B, C);
    return 0;
}
```

# Serial *matmul*—matrix functions

```
int
multiply(matrix * A, matrix * B, matrix * C) {
    double * a = A->elements, * b = B->elements, * c = C->elements;
    double * a_ptr      = a, * b_ptr      = b, * c_ptr      = c;
    int      i = 0, j = 0, k = 0;
    int      aRows = A->rows, middle = A->cols, bCols = B->cols;

    for (j = 0; j < bCols; j++) {
        for (i = 0; i < aRows; i++) {
            a_ptr = a + i;
            for (k = 0; k < middle; k++) {
                *c_ptr += *a_ptr * *(b_ptr + k);
                a_ptr += aRows;
            }
            ++c_ptr;
        }
        b_ptr += middle;
    }
    return 0;
}
```

# Serial *matmul*—*matmul.c*

```
#include <stdio.h>
#include <stdlib.h>
#include "matrix.h"
#include "dataReader.h"

int
main(int argc, char * argv[]) {
    matrix * A = NULL;
    matrix * B = NULL;
    matrix * C = NULL;

    A = makeMatrix();
    getMatrix(A, argv[1]);
    B = identityMatrix(A->rows, A->cols);
    C = makeMatrix();

    initializeMatrix(C, A->rows, B->cols);
    multiply(A, B, C);
    cleanUp(A, B, C);
    return 0;
}
```

## Performance of serial *matmul*

- ▶ Before we proceed we need to understand performance of serial code
  - Is it correct?
  - How well does it scale?

## Performance of serial *matmul*

### ► How do we measure performance?

- Time function
  - Returns execution time of program
  - Make sure that is what you want
- Timer
  - Create timer functions and data structures
  - Place them in code
  - Bisect the code
- Profiler
  - Add compiler flags
  - Run profiler after program completes

## Serial *matmul* timing

- ▶ Two sets of five runs
  - 100 x 100 matrix
    - Executes in 0.03 seconds
  - 1000 x 1000 matrix
    - Executes in 47.27 seconds
- ▶ Did not run larger matrix because of time constraints
  - 5000 x 5000 (estimate 1 hr, 30 min)
  - 10000 x 10000 (estimate 13 hr)

# Outline

- ▶ MPI basics
- ▶ Matrix multiplication serial code
- ▶ Parallelization assumptions
  - Matrix multiplication decomposition 1
  - Matrix multiplication decomposition 2
  - Matrix multiplication decomposition 3
  - Matrix multiplication decomposition 4
  - Matrix multiplication in groups
- ▶ Problems with matrix multiplication assumptions

# What can we parallelize?

## ► Tasks

- Looking at revised code the tasks are
  - Reading the data
  - Computing the matrix product
  - Collecting and displaying the data
- Reading possibilities
  - One reader that distributes
  - Every node reads
- Computing the product possibilities
  - None
- Collecting and displaying data
  - Same as “Reading possibilities”

# What can we parallelize?

## ► Data

- $C = AB$ 
  - Distribute one matrix only
  - Distribute A and C or B and C
  - Distribute A and B
  - Distribute A, B and C

# Initial assumptions?

## ► Tasks

- One reader that distributes data
- One writer that collects and displays results

## ► Data

- Distribute A and C

# Outline

- ▶ MPI basics
- ▶ Matrix multiplication serial code
- ▶ Parallelization assumptions
  - Matrix multiplication decomposition 1
  - Matrix multiplication decomposition 2
  - Matrix multiplication decomposition 3
  - Matrix multiplication decomposition 4
  - Matrix multiplication in groups
- ▶ Problems with matrix multiplication assumptions

# Broadcast Everything

- ▶ Distributing matrix data
  - Rank 0 reads and writes all data
  - Rows and columns broadcast first
  - Matrix elements broadcast second
- ▶ Processes are responsible for “self-identification”
  - Process’ rank used to determine row set
  - Function used by all processes to compute row set
- ▶ Processes send results back to rank 0
- ▶ Ensure the full matrix is available to every processor after the computation

# Data structure changes

## ► Additional information

- Rank, nprocs, startRow and nRows
  - Rank and nprocs are used to determine distribution
  - Distribution parameters computed by each process
- Modify
  - makeMatrix() – include rank and nprocs
    - Meaning of “int size” changes
- Add communications functions

# Parallel *matmul*

```
#include "parMatmul.h"

int main(int argc, char * argv[]) {
    matrix * A = NULL, * B = NULL, * C = NULL;
    int rank, nprocs;

    setupCommunications(argc, argv, &rank, &nprocs);
    A = makeMatrix(rank, nprocs);
    distributeMatrixA(A, argv[1]);
    B = makeMatrix(rank, nprocs);
    distributeMatrixB(B, argv[2]);
    C = makeMatrix(rank, nprocs);
    initializeMatrix(C, A->rows, B->cols);
    multiply(A, B, C);
    collectMatrixC(C);
    cleanUp(A, B, C);
    breakDownCommunications();

    return 0;
}
```

# Setting up MPI runtime environment

```
void setupCommunications(int argc, char * argv[], int * rank, int * procs)
{
    int err = 0;

    err = MPI_Init(&argc, &argv);
    mpiInitFailureCheck(err, __FILE__, __LINE__);

    err = MPI_Comm_size(MPI_COMM_WORLD, procs);
    mpiSizeFailureCheck(err, __FILE__, __LINE__);

    err = MPI_Comm_rank(MPI_COMM_WORLD, rank);
    mpiRankFailureCheck(err, __FILE__, __LINE__);

}
```

# Parallel matmul

```
#include "parMatmul.h"

int main(int argc, char * argv[]) {
    matrix * A = NULL, * B = NULL, * C = NULL;
    int rank, nprocs;

    setupCommunications(argc, argv, &rank, &nprocs);
    A = makeMatrix(rank, nprocs);
    distributeMatrixA(A, argv[1]);
    B = makeMatrix(rank, nprocs);
    distributeMatrixB(B, argv[2]);
    C = makeMatrix(rank, nprocs);
    initializeMatrix(C, A->rows, B->cols);
    multiply(A, B, C);
    collectMatrixC(C);
    cleanUp(A, B, C);
    breakDownCommunications();

    return 0;
}
```

# Decomposing the problem

```
void distributeMatrixA(matrix * A, char * fileName)
{
    int err = 0;

    if (A->rank == 0)
    {
        getMatrix(A, fileName);
    }

    distributeMatrixDimensions(A);

    setRankMatrix(A, A->rows, A->cols);

    err = MPI_Bcast(A->elements, A->size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    mpiBcastFailureCheck(err, __FILE__, __LINE__);
}
```

# Distributing matrix details

```
void distributeMatrixDimensions(matrix * mat) {
    int size[2];
    int err;

    if(mat->rank == 0) {
        size[0] = mat->rows;
        size[1] = mat->cols;
    }

    err = MPI_Bcast(size, 2, MPI_INT, 0, MPI_COMM_WORLD);
    mpiBcastFailureCheck(err, __FILE__, __LINE__);

    if(mat->rank != 0)
    {
        mat->rows = size[0];
        mat->cols = size[1];
    }
}
```

## MPI\_Bcast

► MPI allows us to “broadcast” data

- One node sends data to all members of a group
- C function signature
  - `int MPI_Bcast(void * info, int N, MPI_Datatype type, int src, MPI_Comm comm)`
    - `void * info`—a pointer to the information to be broadcast
    - `MPI_Datatype type`—an MPI value describing the type of the data. If the “type” is not consistent with the type of “info”, a serious error is likely to occur
    - `Int N`—number of elements of `MPI_Datatype` “type” contained in the “info” buffer
    - `Int src`—the source “rank” for the message
    - `MPI_Comm`—the communicator for the broadcast message
  - The return value is an error message

# MPI\_Bcast

► MPI allows us to “broadcast” data

- Fortran function signature
  - `MPI_BCAST(INFO, N, DATATYPE, SRC, COMM, IERR)`
    - INFO—an array (or scalar) of a specific type
    - DATATYPE—an MPI value describing the type of the data. If the DATATYPE is not consistent with the type of INFO, a serious error is likely to occur
    - N—number of elements of MPI\_Datatype “type” contained in the “info” buffer
    - SRC—the source “rank” for the message
    - COMM—the communicator for the broadcast message
    - IERR—the error code of the message. Prudent programming practice is to verify the message before it is sent.

# Distributing matrix details

```
void distributeMatrixDimensions(matrix * mat) {
    int size[2];
    int err;

    if(mat->rank == 0) {
        size[0] = mat->rows;
        size[1] = mat->cols;
    }

    err = MPI_Bcast(size, 2, MPI_INT, 0, MPI_COMM_WORLD);
    mpiBcastFailureCheck(err, __FILE__, __LINE__);

    if(mat->rank != 0)
    {
        mat->rows = size[0];
        mat->cols = size[1];
    }
}
```

# Decomposing the problem

```
void distributeMatrixA(matrix * A, char * fileName)
{
    int err = 0;

    if (A->rank == 0)
    {
        getMatrix(A, fileName);
    }

    distributeMatrixDimensions(A);

    setRankMatrix(A, A->rows, A->cols);

    err = MPI_Bcast(A->elements, A->size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    mpiBcastFailureCheck(err, __FILE__, __LINE__);
}
```

# Setting up decomposition parameters

```
void setRankMatrix(matrix * mat, int rows, int cols) {  
    int nRows =  rows/ mat->nprocs;  
    int rem   =  rows - mat->nprocs*nRows;  
  
    if (mat->rank != 0) {  
        initializeMatrix(mat, rows, cols);  
    }  
  
    if (mat->rank < rem) {  
        mat->startRow = mat->rank * (nRows + 1);  
        mat->nRows    = nRows + 1;  
    } else {  
        mat->startRow = mat->rank * nRows + rem;  
        mat->nRows    = nRows;  
    }  
}
```

# Decomposing the problem

```
void distributeMatrixA(matrix * A, char * fileName)
{
    int err = 0;

    if (A->rank == 0)
    {
        getMatrix(A, fileName);
    }

    distributeMatrixDimensions(A);

    setRankMatrix(A, A->rows, A->cols);

    err = MPI_Bcast(A->elements, A->size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    mpiBcastFailureCheck(err, __FILE__, __LINE__);
}
```

# Parallel matmul

```
#include "parMatmul.h"

int main(int argc, char * argv[]) {
    matrix * A = NULL, * B = NULL, * C = NULL;
    int rank, nprocs;

    setupCommunications(argc, argv, &rank, &nprocs);
    A = makeMatrix(rank, nprocs);
    distributeMatrixA(A, argv[1]);
    B = makeMatrix(rank, nprocs);
    distributeMatrixB(B, argv[2]);
    C = makeMatrix(rank, nprocs);
    initializeMatrix(C, A->rows, B->cols);
    multiply(A, B, C);
    collectMatrixC(C);
    cleanUp(A, B, C);
    breakDownCommunications();

    return 0;
}
```

# Matrix multiplication

```
int multiply(matrix * A, matrix * B, matrix * C) {
    double * a = A->elements, * b = B->elements, * c = C->elements;
    double * a_ptr = a, * b_ptr = b, * c_ptr = c;
    int i = 0, j = 0, k = 0;
    int aRows = A->rows, middle = A->cols, bCols = B->cols;
int start = A->startRow, nRows = A->nRows;
    double * c_start = c + start;
    c += start; a += start; c_ptr = c;
    for (j = 0; j < bCols; j++) {
        for (i = 0; i < nRows; i++) {
            a_ptr = a + i;
            for (k = 0; k < middle; k++) {
                *c_ptr += *a_ptr * *(b_ptr + k);
                a_ptr += aRows;
            }
            ++c_ptr;
        }
        c += aRows; c_ptr = c; b_ptr += middle;
    }
    return 0;
};
```

# Parallel matmul

```
#include "parMatmul.h"

int main(int argc, char * argv[]) {
    matrix * A = NULL, * B = NULL, * C = NULL;
    int rank, nprocs;

    setupCommunications(argc, argv, &rank, &nprocs);
    A = makeMatrix(rank, nprocs);
    distributeMatrixA(A, argv[1]);
    B = makeMatrix(rank, nprocs);
    distributeMatrixB(B, argv[2]);
    C = makeMatrix(rank, nprocs);
    initializeMatrix(C, A->rows, B->cols);
    multiply(A, B, C);
    collectMatrixC(C);
    cleanUp(A, B, C);
    breakDownCommunications();

    return 0;
}
```

# Collecting the results

```
void collectMatrixC(matrix * C) {
    int i = 0, j = 0, k = 0;
    int err = 0;
    double * elements = NULL, * c_ptr = NULL, * e_ptr = NULL;
    MPI_Status status;

    if (C->rank == 0) {
        elements = (double *)malloc(sizeof(double)*C->size);
        for (i = 1; i < C->nprocs; i++) {
            MPI_Recv(elements,C->size,MPI_DOUBLE,i,1,MPI_COMM_WORLD,&status);
            c_ptr = C->elements; e_ptr = elements;

            Add values in elements to C matrix

        } else {
            MPI_Send(C->elements, C->size, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD);
        }
    }

    err = MPI_Bcast(C->elements, C->size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    mpiBcastFailureCheck(err, __FILE__, __LINE__);
}
```

# MPI\_Send and MPI\_Recv

## ► MPI send and receive

- C function signature

- `MPI_Send (void * info, int N, MPI_Datatype type, int dest, int tag, MPI_Comm comm)`
- `MPI_Recv (void * info, int N, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Status * status)`
  - `info`, `N`, `type` and `comm` are consistent with the broadcast signature (slide #47)
  - `int dest`—the destination rank for the message
  - `int src`—the source rank for the message
  - `int tag`—a unique identifier for the message (must match on send and receive)
  - `MPI_Status * status`—a data structure containing information regarding the message

# **MPI\_Send and MPI\_Recv**

## ► MPI send and receive

- Fortran function signature
  - MPI\_SEND (INFO, N, TYPE, DEST, TAG, COMM, IERR)
  - MPI\_RECV (INFO, N, TYPE, SRC, TAG, COMM, STATUS, IERR)
    - INFO, N, TYPE and COMM are consistent with the broadcast signature (slide #48)
    - DEST—the destination rank for the message
    - SRC—the source rank for the message
    - TAG—a unique identifier for the message (must match on send and receive)
    - STATUS—an array containing information regarding the message

## Comments about MPI\_Send and MPI\_Recv

- ▶ Blocking communications (MPI\_Send, MPI\_Recv)
  - Send waits until the process has completed its part
  - Receive waits until the process has completed its part
  - Once process has completed buffers are available for reuse
  - Easy to conceptualize
- ▶ Use specific tags
  - Wildcard tags allowed
  - Wildcard tags worse idea than global variables

## Non-blocking send and receive

### ► Nonblocking (MPI\_Isend, MPI\_Irecv)

- Send and receive immediately
- Buffers can be over-written, but will alter what is sent
- MPI\_Wait to prevent overwriting
- More complex to code
- Requires much greater attention to details

# Parallel matmul

```
#include "parMatmul.h"

int main(int argc, char * argv[]) {
    matrix * A = NULL, * B = NULL, * C = NULL;
    int rank, nprocs;

    setupCommunications(argc, argv, &rank, &nprocs);
    A = makeMatrix(rank, nprocs);
    distributeMatrixA(A, argv[1]);
    B = makeMatrix(rank, nprocs);
    distributeMatrixB(B, argv[2]);
    C = makeMatrix(rank, nprocs);
    initializeMatrix(C, A->rows, B->cols);
    multiply(A, B, C);
    collectMatrixC(C);
    cleanUp(A, B, C);
    breakDownCommunications();

    return 0;
}
```

# Finalizing MPI

```
void breakDownCommunications()
{
    MPI_Finalize();
}
```

# Performance of decomposition 1

► Did it speed up?

	100	1000
1	0.0300	43.27
2	0.0142	17.30
4	0.0085	8.680
8	0.0058	4.430
64	0.0018	0.685
512	0.0014	0.225

# Performance of decomposition 1

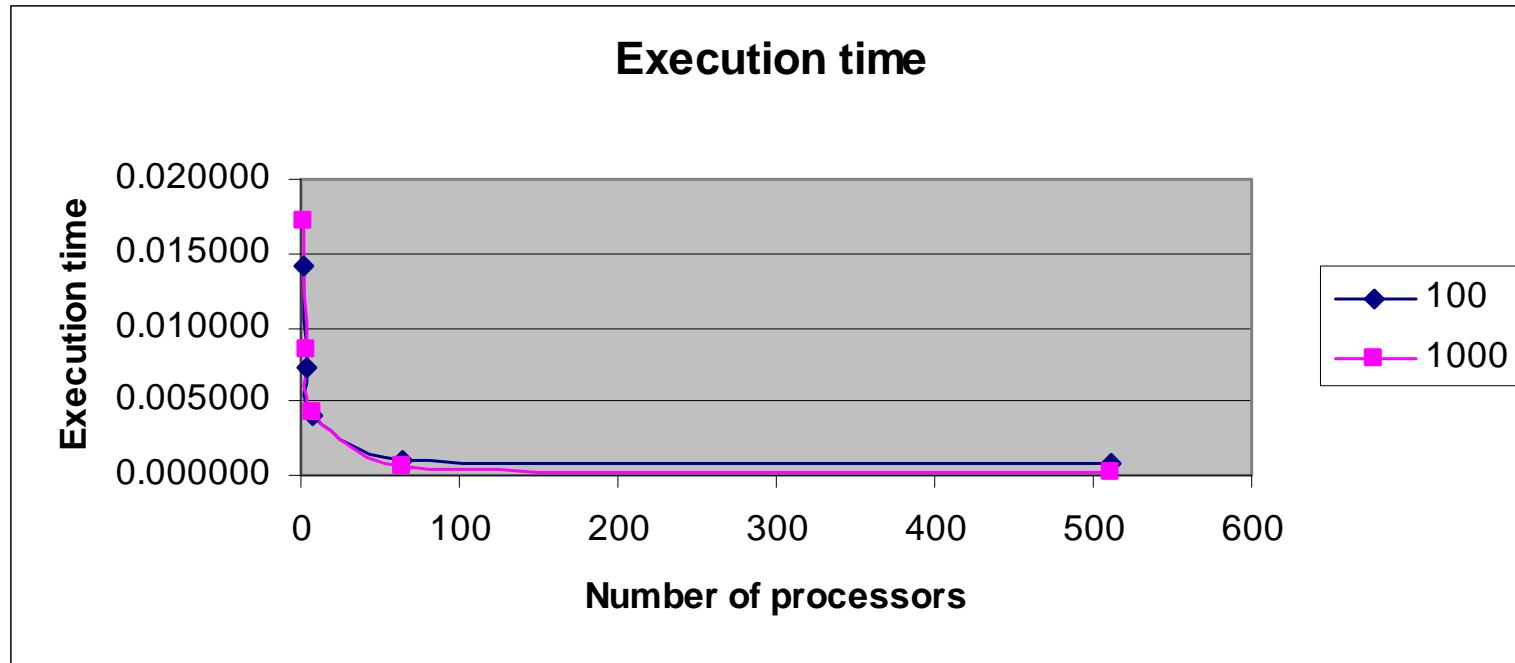
## ► A few questions

- Is the improved speed worth the expense?
  - $100 \times 100$  matrix from 1 to 2 processors?
  - $1000 \times 1000$  matrix from 1 to 2 processors?
  - $100 \times 100$  matrix on 512 processors? Does this even make sense?
- Is the improved speed actually good?
  - What is the speed-up?
  - What influences the speed-up?

## Performance graphs (run time)

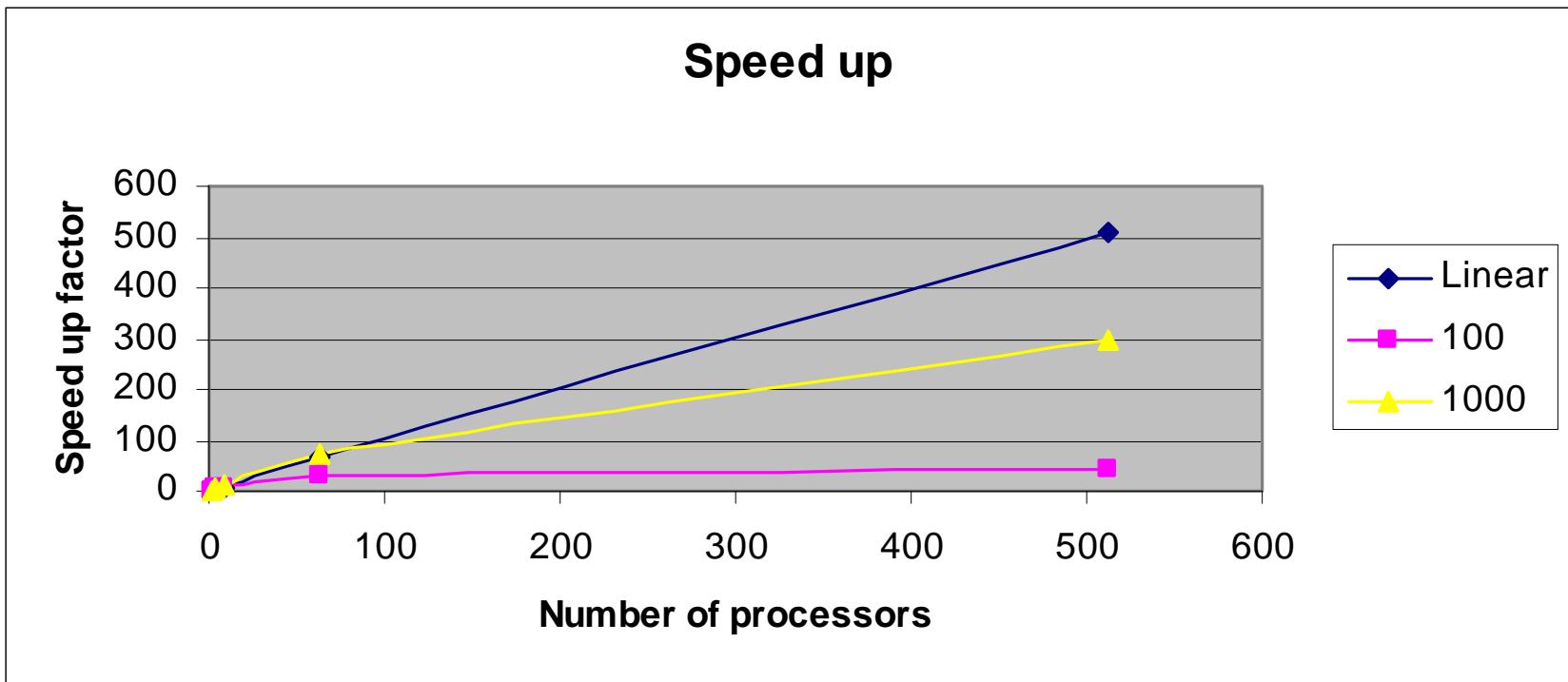
100 x 100 matrix

1000 x 1000 matrix was scaled by a factor of 1000



# Performance graphs (speed-up)

► How well does it scale?



# Source of performance problems

## ► Profiling the data

- Add
  - `sprintf(name, gmon.out.%d.%d", rank, nprocs)`
  - `setenv("GMON_OUT_PREFIX", name, 1)`
- `gcc -o hello hello.c -L${MPI_LIB} -lmpi -I${MPI_INCLUDE} -pg`
- `gprof matmul gmon* > profile`

# gprof output

Each sample counts as 0.000976562 seconds.

%	cumulative	self		self	total	
time	seconds	seconds	calls	s/call	s/call	name
99.07	34.07	34.07	4	8.52	8.52	multiply
0.37	34.20	0.13	8	0.02	0.02	zeroMatrix
0.30	34.30	0.10	4	0.03	0.03	collectMatrixC
0.27	34.40	0.09	4	0.02	0.02	identityMatrix
0.00	34.40	0.00	12	0.00	0.00	makeMatrix
0.00	34.40	0.00	12	0.00	0.00	matrixAllocateCheck
0.00	34.40	0.00	12	0.00	0.00	mpiBcastFailureCheck
0.00	34.40	0.00	9	0.00	0.00	spaceAllocateCheck
0.00	34.40	0.00	9	0.00	0.00	zeroSizeCheck
0.00	34.40	0.00	8	0.00	0.02	initializeMatrix
0.00	34.40	0.00	4	0.00	0.00	breakDownCommunications

# Profiling results

## ► What went wrong?

- Profile data shows
  - multiply function accounted for ~99% of execution on 2-8 processes ( $1000 \times 1000$ )
  - multiply function accounted for 85% on 64 and 45% on 512
  - Distributing A matrix accounted for 9% on 64 and 34% on 512
  - Collecting C matrix accounted for 5% on 64 and 20% on 512
  - Note that the cost of distributing A and collecting C reverted to ~95% on the  $10000 \times 10000$  matrix

# Outline

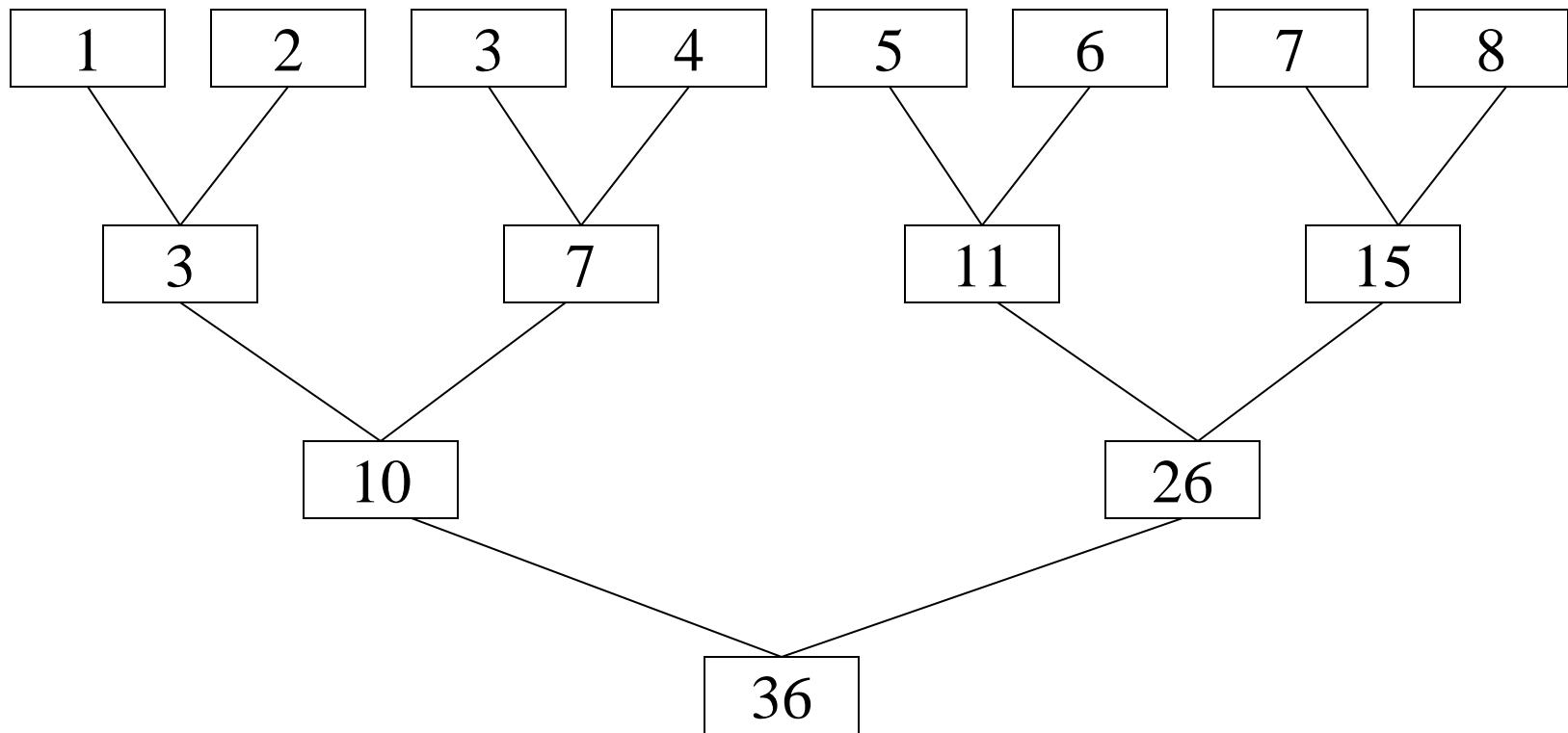
- ▶ MPI basics
- ▶ Matrix multiplication serial code
- ▶ Parallelization assumptions
  - Matrix multiplication decomposition 1
  - Matrix multiplication decomposition 2
  - Matrix multiplication decomposition 3
  - Matrix multiplication decomposition 4
  - Matrix multiplication in groups
- ▶ Problems with matrix multiplication assumptions

## Reduce the results

- ▶ Distributing matrix data
  - Same as previous example
- ▶ Use MPI\_reduce to collect results
  - Nodes pass data to “neighbors” e.g., rank1 passes to rank 0, rank 3 passes to rank 2
  - Perform the operation, e.g. rank 0 SUMS its results with rank 1
  - Pass results to “neighbors” and perform operation, e.g., rank 2 passes results to rank 0
  - Repeat until all data is collected on one node

## Reduce the results

- ▶ Sum becomes  $O(\log n)$  rather than  $O(n)$



# Parallel matmul

```
#include "parMatmul.h"

int main(int argc, char * argv[]) {
    matrix * A = NULL, * B = NULL, * C = NULL;
    int rank, nprocs;

    setupCommunications(argc, argv, &rank, &nprocs);
    A = makeMatrix(rank, nprocs);
    distributeMatrixA(A, argv[1]);
    B = makeMatrix(rank, nprocs);
    distributeMatrixB(B, argv[2]);
    C = makeMatrix(rank, nprocs);
    initializeMatrix(C, A->rows, B->cols);
    multiply(A, B, C);
    collectMatrixC(C);
    cleanUp(A, B, C);
    breakDownCommunications();

    return 0;
}
```

# Reducing the result

```
include "parMatmul.h"
#include <malloc.h>

void collectMatrixC(matrix * C)
{
    double * elements = (double *)malloc(sizeof(double)*C->size);
    double * temp;

    MPI_Reduce(C->elements, elements, C->size, MPI_DOUBLE, MPI_SUM, 0,
               MPI_COMM_WORLD);

    temp = C->elements;
    C->elements = elements;
    free(temp);
}
```

# MPI\_Reduce

## ► MPI supports reduction operation

- Tree based reduction scheme
- C function signature
  - `void MPI_Reduce(void * src_info, void * recv_info, int N, MPI_Datatype type, MPI_Op op, int root)`
    - `void * src_info`—pointer to the information to be transmitted
    - `void * recv_info`—pointer to memory used to store received message
    - `MPI_Datatype type`—the MPI value of the data type sent in the message
    - `int N`—the number of data points sent in the message
    - `MPI_Op op`—the MPI value for type of the reduction operation
    - `int root`—the rank of the process the stores the final result

# MPI\_Reduce

## ► MPI supports reduction operation

- Tree based reduction scheme
- Fortran function signature
  - `MPI_REDUCE(SRC_INFO, RECV_INFO, N, TYPE, OP, ROOT, COMM, IERR)`
    - SRC\_INFO—pointer to the information to be transmitted
    - RECV\_INFO—pointer to memory used to store received message
    - TYPE—the MPI value of the data type sent in the message
    - N—the number of data points sent in the message
    - OP—the MPI value for type of the reduction operation
    - ROOT—the rank of the process the stores the final result
    - IERR—the error code for the operation

## Performance graphs (run time)

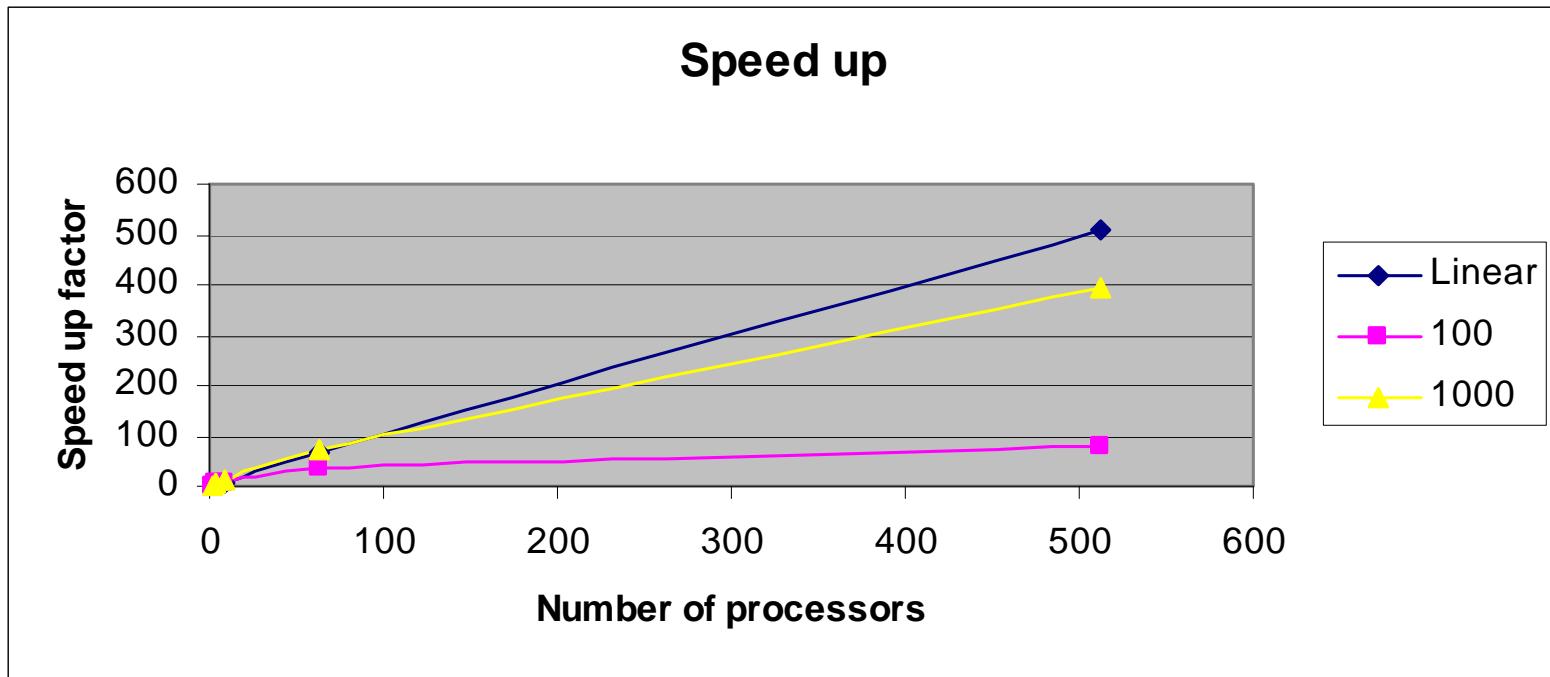
100 x 100 matrix

1000 x 1000 matrix was scaled by a factor of 1000



# Performance graphs (speed-up)

► How well does it scale?



# What went wrong?

- Profile data shows
  - Same story as decomposition 1 on 2-8 processors
  - multiply function accounted for 93% on 64 and 65% on 512
  - Distributing A matrix accounted for 6% on 64 and 38% on 512
  - Collecting C matrix accounted for ~0% on 64 and ~0% on 512
- Why wasn't it better
  - REMEMBER AMDAHL

# Outline

- ▶ MPI basics
- ▶ Matrix multiplication serial code
- ▶ Parallelization assumptions
  - Matrix multiplication decomposition 1
  - Matrix multiplication decomposition 2
  - Matrix multiplication decomposition 3
  - Matrix multiplication decomposition 4
  - Matrix multiplication in groups
- ▶ Problems with matrix multiplication assumptions

## Send what is needed

- ▶ Send just rows of A to processors
  - For each process rank 0 creates array
  - Array is sized for the target rank
  - Rows for each rank are copied to array
  - Array is sent to target rank
- ▶ Rank 0 receives rows from each rank
  - Ranks 1 through n send rows of C to rank 0
  - Rank 0 remixes rows back into full C matrix

# Parallel matmul

```
#include "parMatmul.h"

int main(int argc, char * argv[]) {
    matrix * A = NULL, * B = NULL, * C = NULL;
    int rank, nprocs;

    setupCommunications(argc, argv, &rank, &nprocs);
    A = makeMatrix(rank, nprocs);
    distributeMatrixA(A, argv[1]);
    B = makeMatrix(rank, nprocs);
    distributeMatrixB(B, argv[2]);
    C = makeMatrix(rank, nprocs);
    initializeMatrix(C, A->rows, B->cols);
    multiply(A, B, C);
    collectMatrixC(C);
    cleanUp(A, B, C);
    breakDownCommunications();

    return 0;
}
```

# Distribute submatrices

```
void distributeMatrixA(matrix * A, char * fileName)
{
    MPI_Status status;

    if (A->rank == 0)
    {
        getMatrix(A, fileName);
        distributeMatrixDimensions(A);
        distributeRank0(A);
    }
    else
    {
        distributeMatrixDimensions(A);
        setRankMatrix(A, A->rows, A->cols);
        MPI_Recv(A->elements, A->size, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD,
                 &status);
    }
}
```

# Decompose matrix

```
void distributeRank0(matrix * A) {
    int i = 0, sentRows = 0;
    matrix * send      = makeMatrix(0, A->nprocs);
    double * elements = NULL;
    elements = A->elements;
    A->elements = NULL;

    for (i = 1; i < A->nprocs; i++) {
        send->rank = i;
        setRankMatrix(send, A->rows, A->cols);
        if (sentRows != send->rows) {
            send->elements = (double *)malloc(sizeof(double)*send->size);
            sentRows = send->rows;
        }
        copyElementsToMat(send, elements);
        MPI_Send(send->elements, send->size, MPI_DOUBLE, i, 1, MPI_COMM_WORLD);
    }
    setRankMatrix(A, A->rows, A->cols);
    copyElementsToMat(A, elements);
}
```

# Distribute submatrices

```
void distributeMatrixA(matrix * A, char * fileName)
{
    MPI_Status status;

    if (A->rank == 0)
    {
        getMatrix(A, fileName);
        distributeMatrixDimensions(A);
        distributeRank0(A);
    }
    else
    {
        distributeMatrixDimensions(A);
        setRankMatrix(A, A->rows, A->cols);
        MPI_Recv(A->elements, A->size, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD,
        &status);
    }
}
```

# Parallel matmul

```
#include "parMatmul.h"

int main(int argc, char * argv[]) {
    matrix * A = NULL, * B = NULL, * C = NULL;
    int rank, nprocs;

    setupCommunications(argc, argv, &rank, &nprocs);
    A = makeMatrix(rank, nprocs);
    distributeMatrixA(A, argv[1]);
    B = makeMatrix(rank, nprocs);
    distributeMatrixB(B, argv[2]);
    C = makeMatrix(rank, nprocs);
    initializeMatrix(C, A->rows, B->cols);
    multiply(A, B, C);
    collectMatrixC(C);
    cleanUp(A, B, C);
    breakDownCommunications();

    return 0;
}
```

# Collecting submatrices

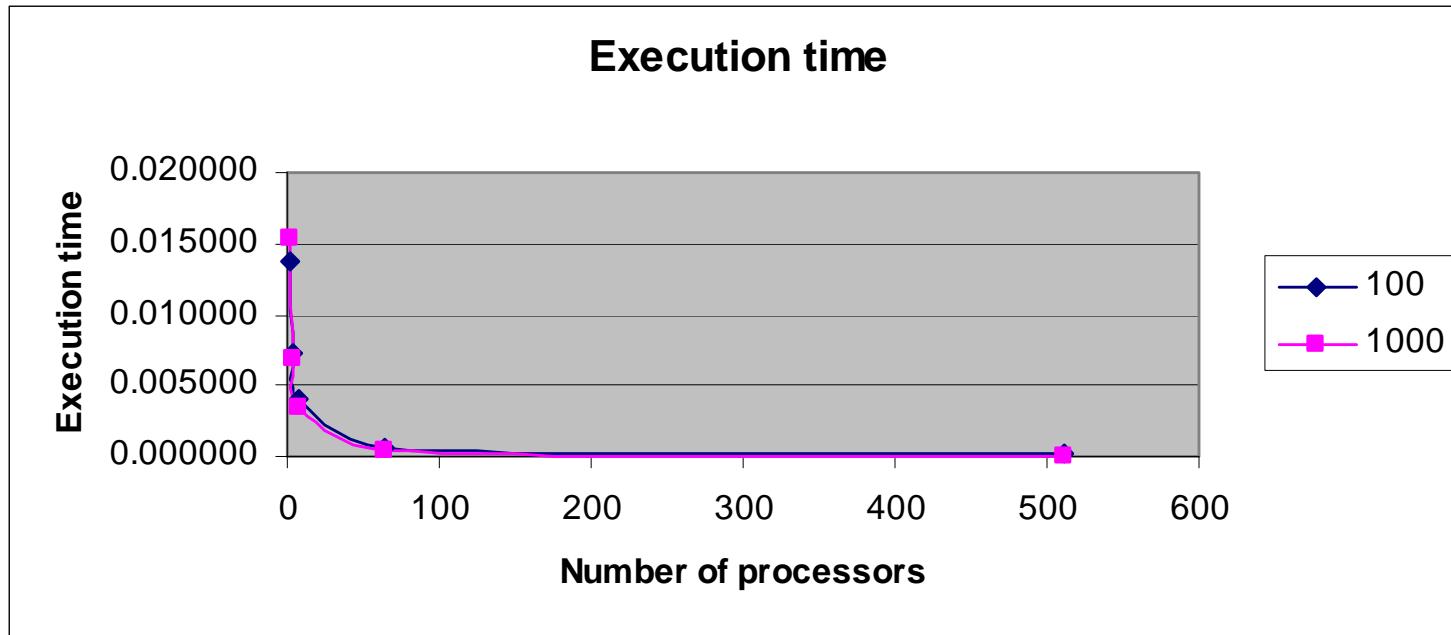
```
void collectMatrixC(matrix * C) {
    int i = 0;
    matrix * recv = makeMatrix(0, 0);
    MPI_Status status;
    double * elements, * temp;

    if (C->rank == 0) {
        elements = (double *)malloc(sizeof(double)*C->rows * C->cols);
        mixMatrix(elements, C);
        for (i = 1; i < C->nprocs; i++) {
            recv->rank = i; recv->nprocs = C->nprocs;
            setRankMatrix(recv, C->rows, C->cols);
            MPI_Recv(recv->elements, recv->size, MPI_DOUBLE, i, 1,
                     MPI_COMM_WORLD, &status);
            mixMatrix(elements, recv);
        }
        swap C and temp
    } else {
        MPI_Send(C->elements, C->size, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD);
    }
}
```

## Performance graphs (run time)

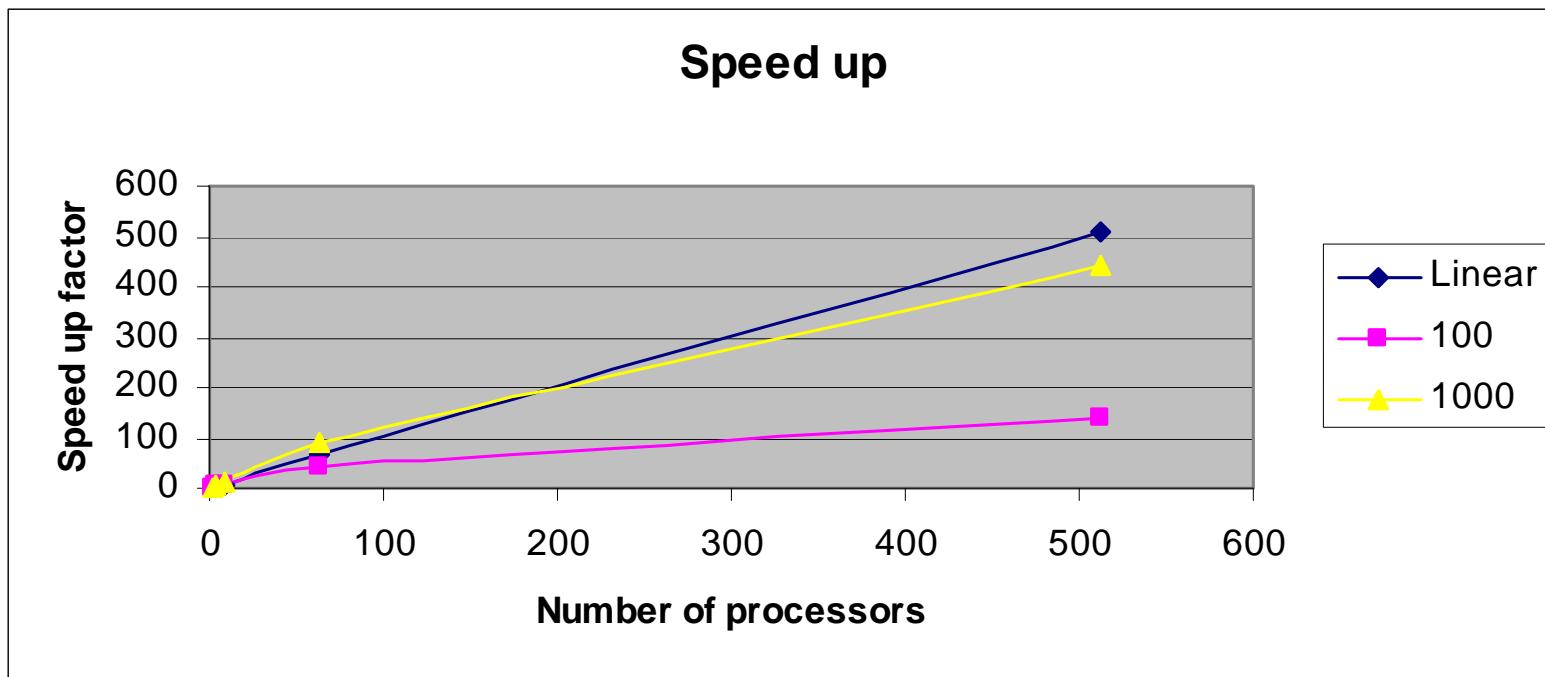
100 x 100 matrix

1000 x 1000 matrix was scaled by a factor of 1000



# Performance graphs (speed-up)

► How well does it scale?



# What went wrong?

- Profile data shows
  - Not a lot
  - Distribution costs were 7% on 64 processors 30% on 512

# Outline

- ▶ MPI basics
- ▶ Matrix multiplication serial code
- ▶ Parallelization assumptions
  - Matrix multiplication decomposition 1
  - Matrix multiplication decomposition 2
  - Matrix multiplication decomposition 3
  - Matrix multiplication decomposition 4
  - Matrix multiplication in groups
- ▶ Problems with matrix multiplication assumptions

# Scatter and Gather

## ► Scatter A

- Decompose and broadcast
- Requires restructuring matrix

## ► Gather C

- Like reduce

## ► Processes send results back to rank 0

## ► Ensure the full matrix is available to every processor after the computation

# Parallel matmul

```
#include "parMatmul.h"

int main(int argc, char * argv[]) {
    matrix * A = NULL, * B = NULL, * C = NULL;
    int rank, nprocs;

    setupCommunications(argc, argv, &rank, &nprocs);
    A = makeMatrix(rank, nprocs);
    distributeMatrixA(A, argv[1]);
    B = makeMatrix(rank, nprocs);
    distributeMatrixB(B, argv[2]);
    C = makeMatrix(rank, nprocs);
    initializeMatrix(C, A->rows, B->cols);
    multiply(A, B, C);
    collectMatrixC(C);
    cleanUp(A, B, C);
    breakDownCommunications();

    return 0;
}
```

# Distribute sub-matrices

```
void distributeMatrixA(matrix * A, char * fileName) {
    matrix * send = makeMatrix(A->rank, A->nprocs);

    if (A->rank == 0) {
        getMatrix(A, fileName);
        distributeMatrixDimensions(A);
        restructure(send, A);
    } else {
        distributeMatrixDimensions(A);
        setRankMatrix(A, A->rows, A->cols);
    }

    MPI_Scatter(send->elements, A->nRows * A->cols, MPI_DOUBLE,
               A->elements, A->nRows * A->cols, MPI_DOUBLE,
               0, MPI_COMM_WORLD);
}
```

## **MPI\_Scatter**

### ► MPI scatter

- Tree based distribution scheme
- MPI\_Scatter distributes buffer among communications group

# MPI\_Scatter

## ► MPI scatter

- C function signature

- **MPI\_Scatter(void \* send, int nSend, MPI\_Type sType, void \* recv, int nRecv, int rType, int root, MPI\_Comm comm)**
  - **void \* send**—a pointer to the information sent in the message
  - **int nSend**—the number of elements of **MPI\_Type sType** sent
  - **int sType**—the **MPI\_Type** of the data sent
  - **void \* recv**—a pointer to the storage location for the data
  - **int nRecv**—the number of elements of type **MPI\_Type rType** received
  - **int root**—the rank of the source of the scattered data
  - **MPI\_Comm comm**—the communicator involved in the message

# MPI\_Scatter

## ► MPI scatter

- Fortran function signature
  - **MPI\_SCATTER(SENDF, NSEND, STYPE, RECV, NRECV, RTYPE, ROOT, COMM, IERR)**
    - SEND—a pointer to the information sent in the message
    - NSEND—the number of elements of MPI\_Type sType sent
    - STYPE—the MPI\_Type of the data sent
    - RECV—a pointer to the storage location for the data
    - NRECV—the number of elements of type MPI\_Type rType received
    - ROOT—the rank of the source of the scattered data
    - COMM—the communicator involved in the message
    - IERR—the error code for the message

# Parallel matmul

```
#include "parMatmul.h"

int main(int argc, char * argv[]) {
    matrix * A = NULL, * B = NULL, * C = NULL;
    int rank, nprocs;

    setupCommunications(argc, argv, &rank, &nprocs);
    A = makeMatrix(rank, nprocs);
    distributeMatrixA(A, argv[1]);
    B = makeMatrix(rank, nprocs);
    distributeMatrixB(B, argv[2]);
    C = makeMatrix(rank, nprocs);
    initializeMatrix(C, A->rows, B->cols);
    multiply(A, B, C);
    collectMatrixC(C);
    cleanUp(A, B, C);
    breakDownCommunications();

    return 0;
}
```

# Distribute sub-matrices

```
void collectMatrixC(matrix * C, matrix * result)
{
    int nRows = 0;
    int rem = 0;

    double * elements = (double *)malloc(sizeof(double) * C->size * C->nprocs);

distributionInfo(C->rows, C->nprocs, &nRows, &rem);

MPI_Gather(C->elements, C->size, MPI_DOUBLE, elements,
           C->size, MPI_DOUBLE, 0, MPI_COMM_WORLD);

if (C->rank == 0)
{
    compress(result, elements, nRows, rem);
}

free(elements); // this is not necessary
}
```

## **MPI\_Gather**

### ► MPI gather

- Tree based reduction scheme
- MPI\_Gather collects buffer from communications group

# MPI\_Gather

## ► MPI gather

- C function signature

- **MPI\_Gather (void \* send, int nSend, MPI\_Type sType, void \* recv, int nRecv, int rType, int root, MPI\_Comm comm)**
  - **void \* send**—a pointer to the information sent in the message
  - **int nSend**—the number of elements of **MPI\_Type sType** sent
  - **int sType**—the **MPI\_Type** of the data sent
  - **void \* recv**—a pointer to the storage location for the data
  - **int nRecv**—the number of elements of type **MPI\_Type rType** received
  - **int root**—the rank of the source of the gathered data
  - **MPI\_Comm comm**—the communicator involved in the message

# MPI\_Gather

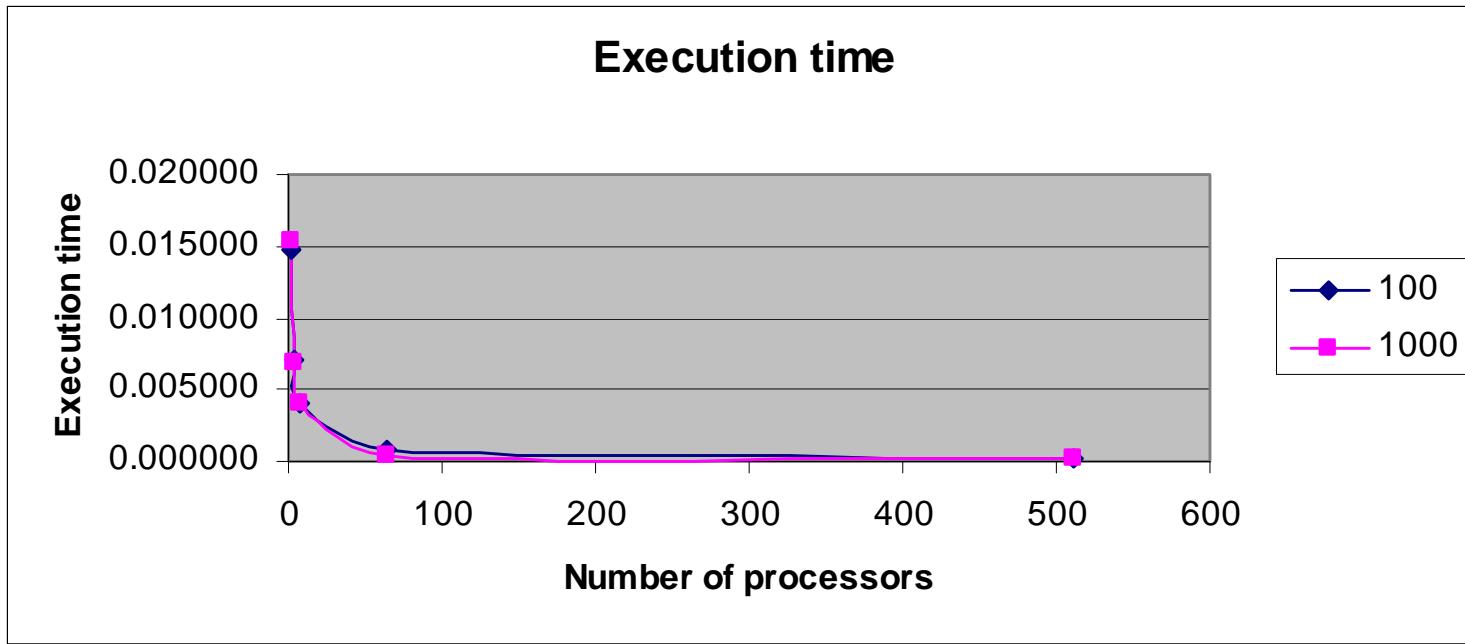
## ► MPI gather

- Fortran function signature
  - **MPI\_GATHER (SEND, NSEND, STYPE, RECV, NRECV, RTYPE, ROOT, COMM, IERR)**
    - SEND—a pointer to the information sent in the message
    - NSEND—the number of elements of MPI\_Type sType sent
    - STYPE—the MPI\_Type of the data sent
    - RECV—a pointer to the storage location for the data
    - NRECV—the number of elements of type MPI\_Type rType received
    - ROOT—the rank of the source of the gathered data
    - COMM—the communicator involved in the message
    - IERR—the error code for the message

## Performance graphs (run time)

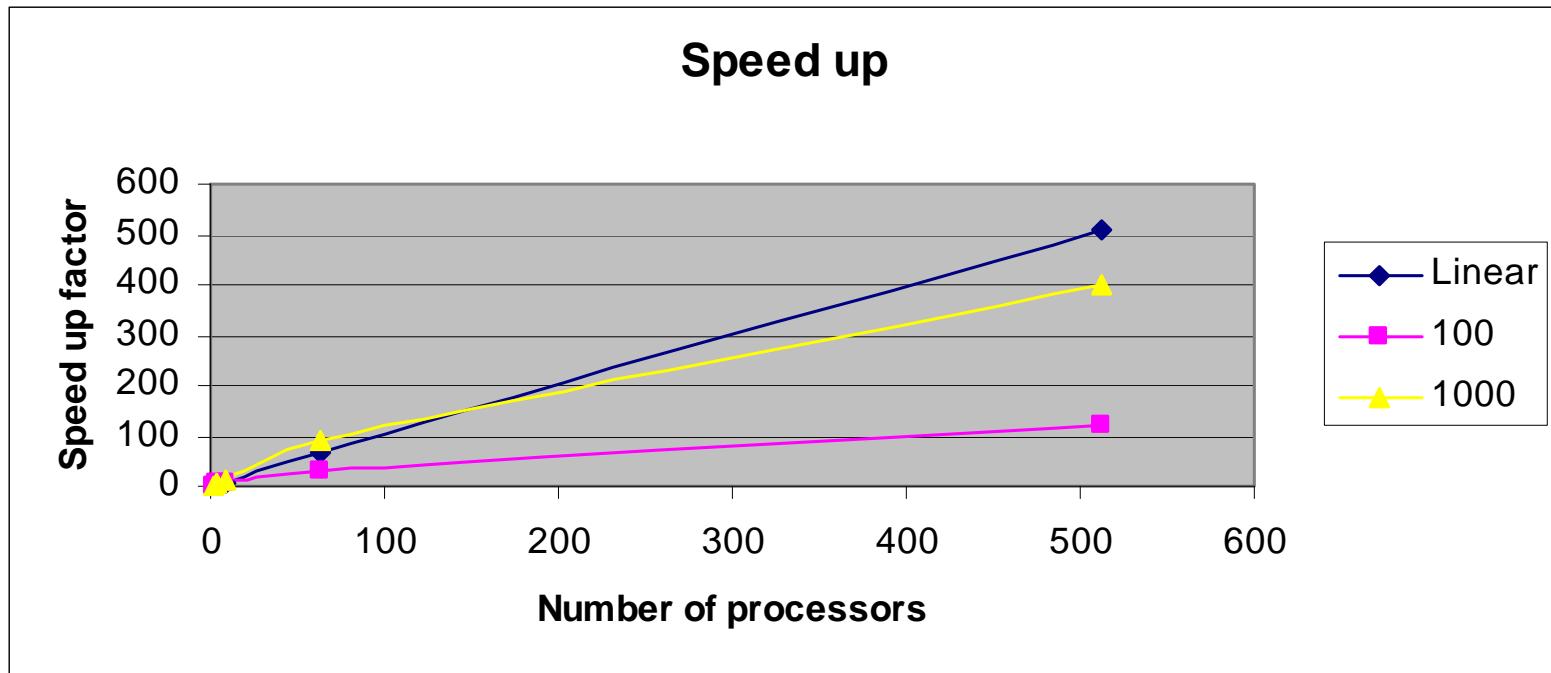
100 x 100 matrix

1000 x 1000 matrix was scaled by a factor of 1000



# Performance graphs (speed-up)

► How well does it scale?



# What went wrong?

- Profile data shows
  - Basically same results as decomposition 3
  - Setup for scatter and gather was more expensive

# Outline

- ▶ MPI basics
- ▶ Matrix multiplication serial code
- ▶ Parallelization assumptions
  - Matrix multiplication decomposition 1
  - Matrix multiplication decomposition 2
  - Matrix multiplication decomposition 3
  - Matrix multiplication decomposition 4
  - Matrix multiplication in groups
- ▶ Problems with matrix multiplication assumptions

## Coordinating parallel tasks

- ▶ Parallel tasks decomposed into groups
  - Groups have their own communicators and ranks
  - MPI communications routines refer to group communicators rather than MPI\_COMM\_WORLD
  - MPI\_Comm\_group(MPI\_COMM\_WORLD, &worldGroup);
  - MPI\_Group\_incl(worldGroup, nLap, lapRank, group);
  - MPI\_Comm\_create(MPI\_COMM\_WORLD, \*group, communicator);

## An example using matmul

- ▶ Two groups
  - Brute force (BRU) uses matmul algorithm
  - LAPACK (LAP) uses dgemm
- ▶ matmul sets up the groups
  - Calls matmulBRU for ranks in BRU group
  - Calls matmulLAP for ranks in LAP group

# Setting up groups—matrix information

```
typedef struct matrix
{
    int                     rows;
    int                     cols;
    int                     size;
    double                 * elements; // treat as private data
    int                     rank;
    int                     nprocs;
    int                     startRow;
    int                     endRow;
    int                     groupRank;
    int                     groupProcs;
    MPI_Comm                communicator;
    MPI_Group               group;
} matrix;
```

# Setting up groups—constructing groups

```
void setupGroups(int rank, int procs, MPI_Comm * communicator, MPI_Group *  
    group, int * groupProcs)  
{  
    int i = 0. nLap = 0. nBru = 0;  
    int * lapRank. * bruRank;  
    MPI_Group worldGroup;  
  
        set up group membership criteria  
    // setup procs for communications group  
    lapRank = (int *)malloc(sizeof(int)*nLap);  
    bruRank = (int *)malloc(sizeof(int)*nBru);  
  
        define process groups  
    MPI_Comm_create(MPI_COMM_WORLD, *group, communicator);  
}
```

# Setting up groups—constructing groups

```
MPI_Comm_group(MPI_COMM_WORLD, &worldGroup);
if (rank < nLap) {
    for(i = 0; i < nLap; i++) {
        lapRank[i] = i;
    }
    *groupProcs = nLap;
    MPI_Group_incl(worldGroup, nLap, lapRank, group);
} else {
    for(i = 0; i < nBru; i++) {
        bruRank[i] = nLap + i;
    }
    *groupProcs = nBru;
    MPI_Group_incl(worldGroup, nBru, bruRank, group);
}
```

## The rest of the code

- ▶ Remainder of the code is basically the same
  - matrixLAP uses dgemm
  - matrixBRU uses brute force method
  - MPI\_COMM\_WORLD is replaced with mat->communicator
- ▶ Full code set is provided in handout

# Outline

- ▶ MPI basics
- ▶ Matrix multiplication serial code
- ▶ Parallelization assumptions
  - Matrix multiplication decomposition 1
  - Matrix multiplication decomposition 2
  - Matrix multiplication decomposition 3
  - Matrix multiplication decomposition 4
  - Matrix multiplication in groups
- ▶ Problems with matrix multiplication assumptions

## Some comments on initial assumptions

- ▶ Were the initial assumptions good?
  - Bad question at this point, but this is when it is usually asked
  - No they weren't
  - Distributing A was complex, B would have been easier
  - Collection C was probably unnecessary
- ▶ Simple assumptions are almost always bad assumptions in parallelizing code

## What to do about it?

### ► Examine alternatives

- Distribution
  - Distributing B
  - Distributing A and B
- Non-blocking send and receive when collecting C
- Each node reads from and writes to file

## Some final thoughts

- ▶ MPI is very useful and helpful but...
  - Think before you start:
    - About communications –  
Number of messages matters a lot, collect several and only send one when possible. ( $< \text{Log}_2(P)$ )
    - About data layout, which data on which processors
  - PROFILE! Know what matters before tuning.
    - Check your scaling – different parts will scale differently
  - Check error return codes on all communication and IO calls. Use MPI\_Abort not exit/stop. Flush after print.

## Some final thoughts

- ▶ MPI is very useful and helpful but...
  - Avoid use of wildcard destinations or message tags  
This increases overhead incurred on most systems.
  - ISOLATE communication, computation, and IO pieces  
**SMALL PROGRAMMING UNITS**
  - Debugging: First check your assumptions.
  - This stuff is complex, USE LIBRARIES when available.
  - The future – variable resources, task scheduling, global resources