Northern Illinois University

Introduction to parallel computing

Jifu Tan, Ph.D.

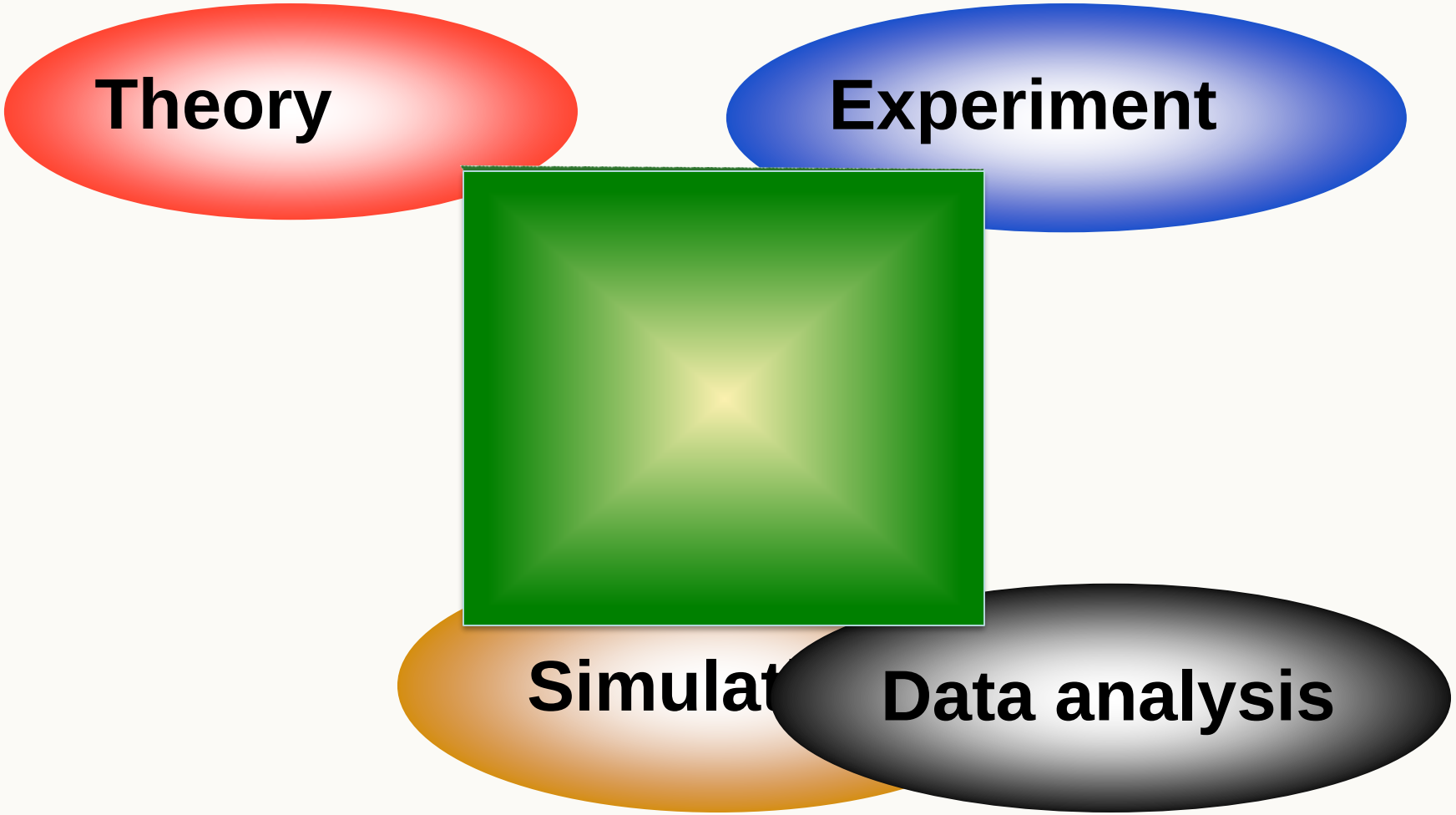
Jifutan@niu.edu

8/16/2023

Outline

- Introduction
  - What is parallel computing?
  - Why is it important?
- How to write a parallel code?
- MPI introduction
  - functions, compiling, running
- Simple MPI code
- How to design a parallel algorithm?
  - Matrix addition/multiplication
  - Sample code analysis
- Reference

The Four Paradigms of Science

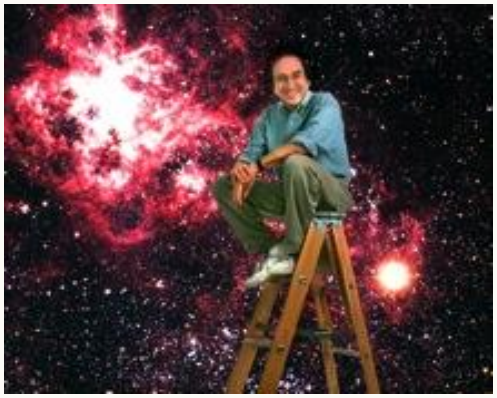


Simulation in Science and Engineering

High performance computing (HPC) simulation to understand things that are:

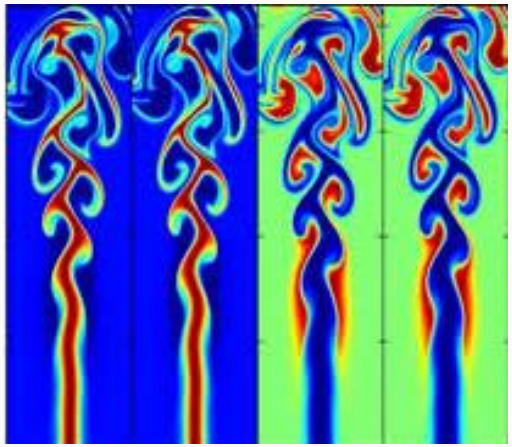
- too big
- too small
- too fast
- too slow
- too expensive or
- too dangerous

for experiments

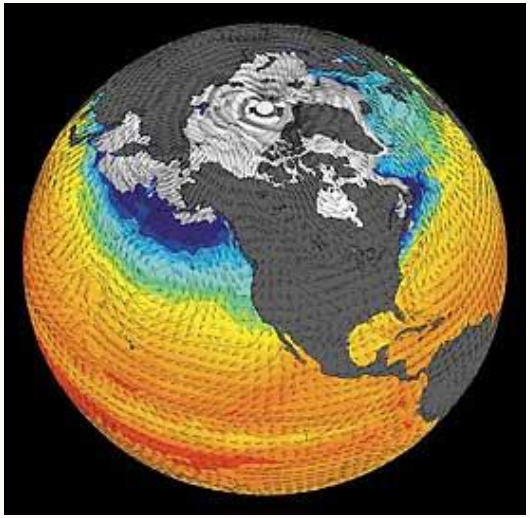


Understanding the universe

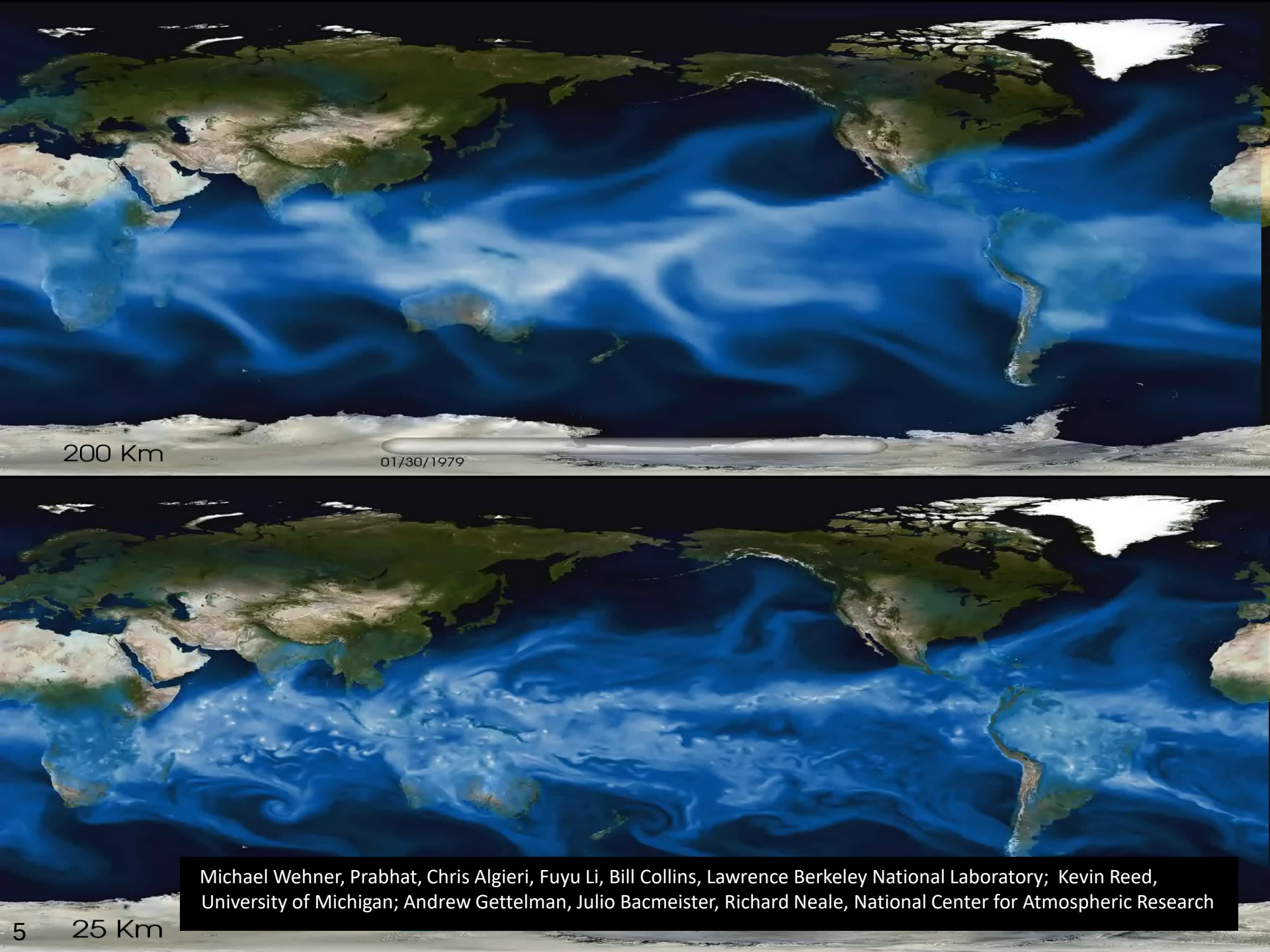
Proteins and diseases



Energy-efficient jet engines



Climate change



Michael Wehner, Prabhat, Chris Algieri, Fuyu Li, Bill Collins, Lawrence Berkeley National Laboratory; Kevin Reed, University of Michigan; Andrew Gettelman, Julio Bacmeister, Richard Neale, National Center for Atmospheric Research

Biofluids modeling group

Dr. Jifu Tan

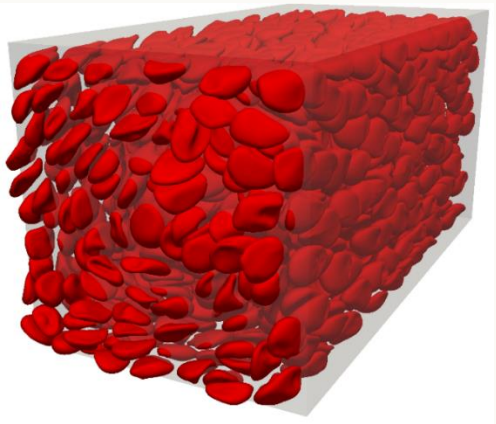
https://sites.google.com/site/tanjifu

Undergraduate research experience
Graduate school

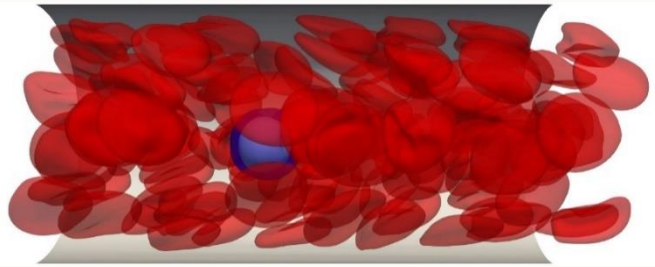
Fluid mechanics
Solid Mechanics
Programming (C++/Matlab)

Applications:

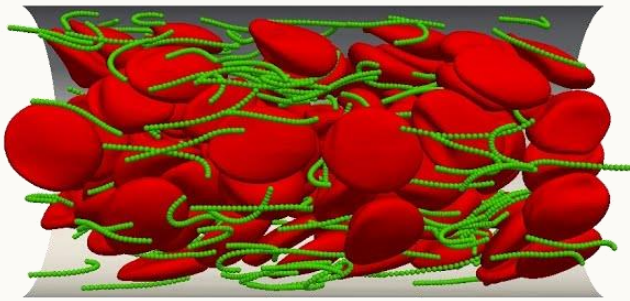
- **Microfluidic device design**
- **Cancer Cell adhesion and transport**
- **Blood rheology**
- **Drug carrier transport**
- **Blood clotting**
- **Cell membrane damage**



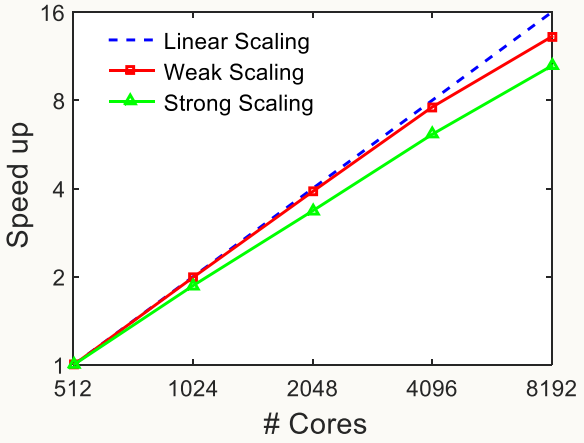
Cancer cell transport



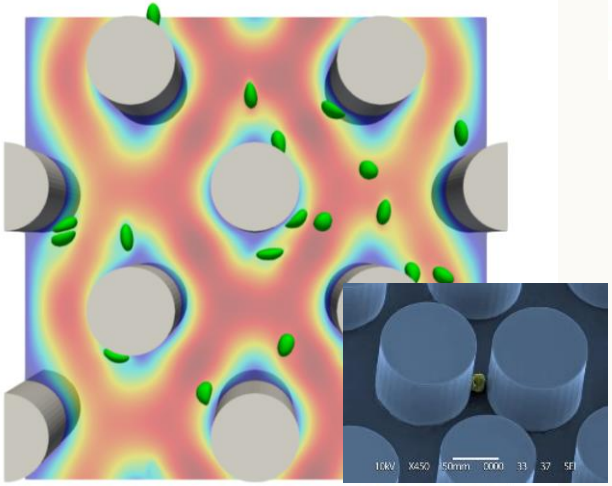
Drug carrier design



Supercomputers



Microfluidics for cancer cell detection



AIP Scilight

25 APRIL 2022 • https://doi.org/10.1063/10.0010412

Capillary model tracks blood flow in healthy and diseased networks

Avery Thompson

3D model of retina capillary networks provides insight on flow of blood cells, impacts of constricted vessels

PREV

PPT | High resolution

Simulation Setup

Flow of red blood cells in the retina network

- Red blood cells generated in the domain
- 1843 cells in the network
- $Ht = 18.4\%$
- 1320 particles per cell
- 120 processors over 9 days
- Inlet flow rate: $2.55 \times 10^{-13}\ m^3/s$
- Maximum shear rate: $2813\ s^{-1}$
- $Re = 0.2$

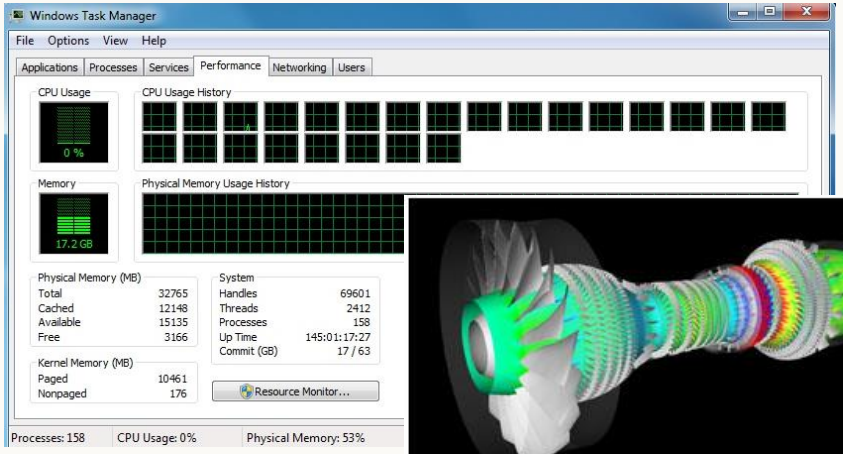
K Ostalowski, J Tan
Physics of Fluids 34 (4), 041912, 2022

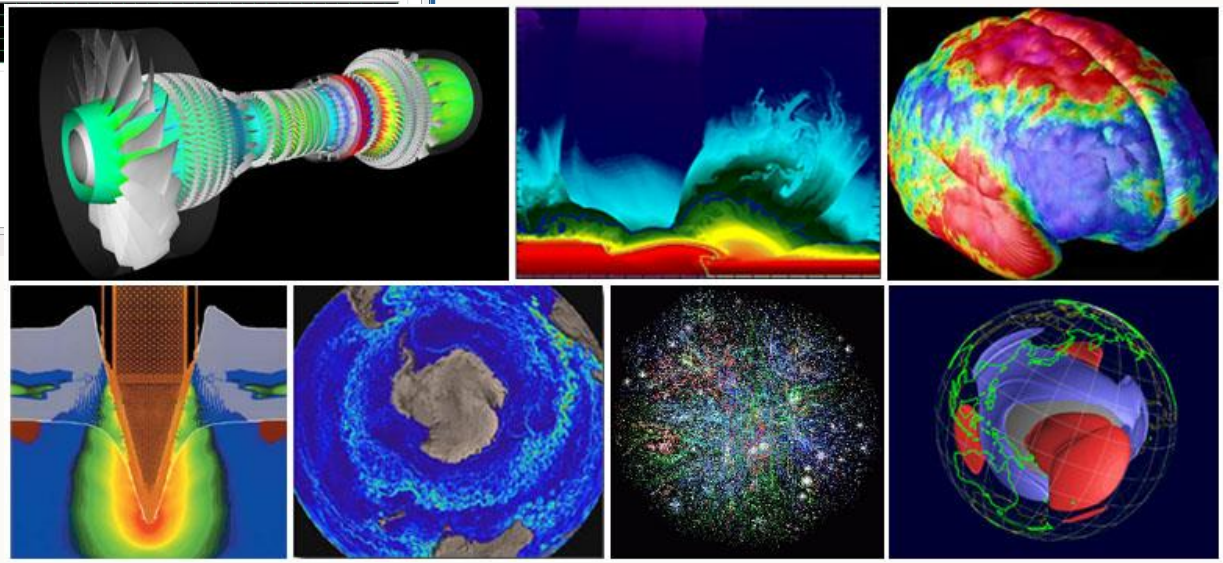
Parallel is everywhere

Rush Hour Traffic

Plate Tectonics

Weather



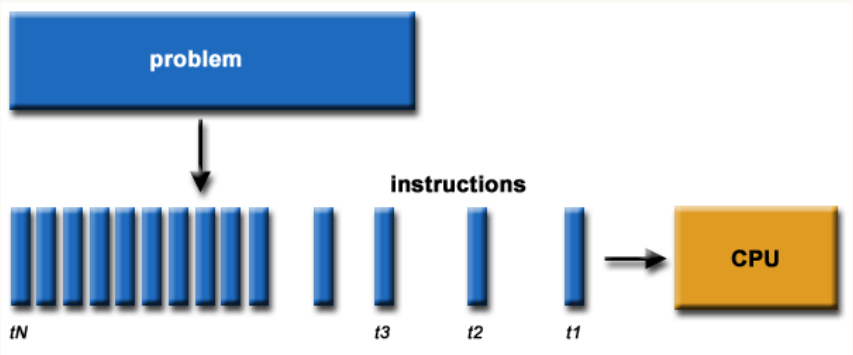


https://computing.llnl.gov/tutorials/parallel_comp/

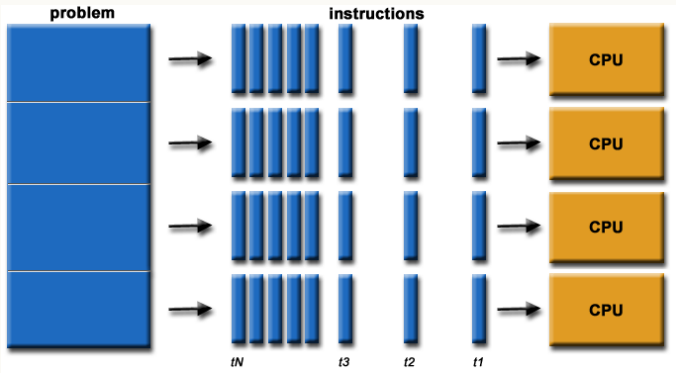
What is parallel computing

- a form of computation in which many calculations are carried out simultaneously
  - use of multiple processors or computers working together on a common task
  - Multithreads
  - Graphics processing units

Traditional computing



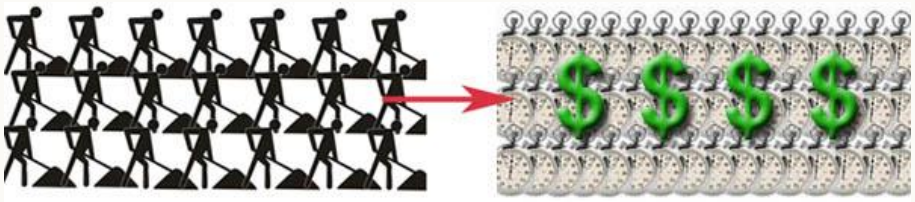
Parallel computing



Gottlieb, Allan; Almasi, George S. (1989). Highly parallel computing. Redwood City, Calif.: Benjamin/Cummings. ISBN 0-8053-0177-1
https://computing.llnl.gov/tutorials/parallel_comp/

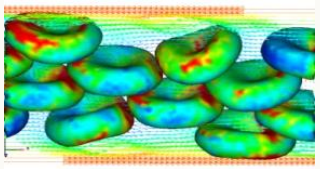
Why?

Save time and/or money



Solve larger problems

50 um



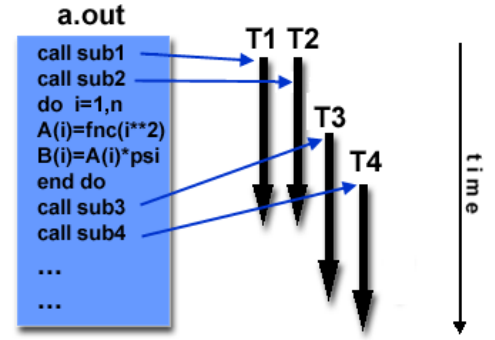
Use of non-local resources

Limits to serial computing

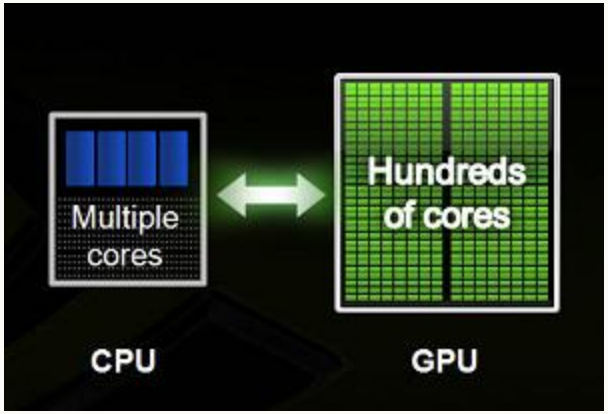
How?

Several parallel programming models

Shared memory: openmp

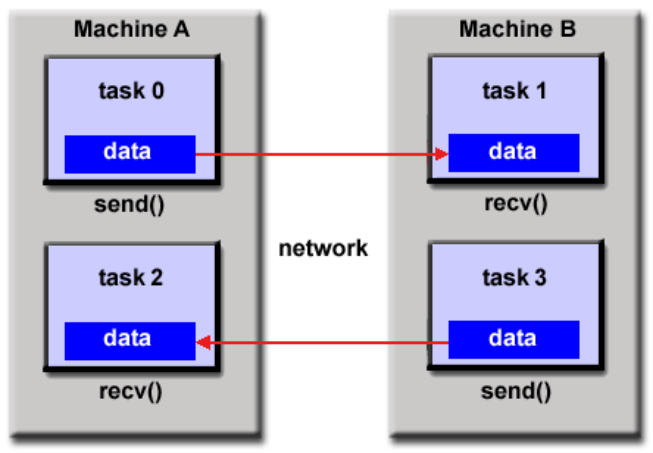


GPU computing



Distributed Memory / Message Passing Model (MPI)

MPI_Send
MPI_Recv



Message Passing Interface

- Why MPI ?
  - One of the oldest libraries
  - Wide-spread adoption. Portable.
  - Minimal requirements on the underlying hardware
  - Explicit parallelization
    - Intellectually demanding
    - Achieves high performance
    - Scales to large number of processors
- MPI is 125 functions
- MPI has 6 most used functions

MPI_Init
MPI_Finalize
MPI_Comm_size
MPI_Comm_rank
MPI_Send
MPI_Recv

MPI_Init, MPI_Finalize

int MPI_Init(int *argc, char ***argv)

Initialize the MPI execution environment.
It has to be called only once in main thread, it has to call MPI_Finalize.

int MPI_Finalize(void)

Terminates MPI execution environment

```
#include "mpi.h"

int main(int argc, char* argv[])
{
  MPI_Init(&argc, &argv);
  MPI_Finalize();

  return 0;
}
```

MPI_Comm_size / MPI_Comm_rank

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

#define HOSTNAMELEN 256

int main(int argc, char* argv[]){
  int rank, size;
  char hostname[HOSTNAMELEN];

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  gethostname(hostname, HOSTNAMELEN);
  printf("Hello, world! I am %d of %d
from %s\n", rank, size,hostname);
  MPI_Finalize();
  return 0;}
```

int MPI_Comm_size(
MPI_Comm comm, int *size)

Determines the size of the group associated with a communicator

int MPI_Comm_rank(
MPI_Comm comm, int *rank)

Determines the rank of the calling process in the communicator

MPI Features

- Communicator Information
- Point to Point communication
- Collective Communication

MPI_Send(void* *data*, int *count*, MPI_Datatype *datatype*,
int *destination*, int *tag*, MPI_Comm *comm*)
MPI_Recv(void* *data*, int *count*, MPI_Datatype *datatype*, int *sourc*
int *tag*, MPI_Comm *com,* MPI_Status* *status*)

```
for (i = 1; i < nnodes; i++)
MPI_Send(overallmin,2,MPI_INT,i,OVRLMIN_MSG,MPI_COMM_WORLD
);
```

MPI_Bcast()

```
MPI_Bcast(overallmin,2,MPI_INT,0,MPI_COMM_WORLD);
```

MPI_Reduce() / MPI_Allreduce()

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

example

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

#define HOSTNAMELEN 256

int main(int argc, char* argv[]){
  int rank, size,number;
  char hostname[HOSTNAMELEN];

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  gethostname(hostname, HOSTNAMELEN);
  //printf("Hello, world! I am %d of %d from %s\n", rank, size,hostname)
     if (world_rank == 0) {
        number = -1;
        MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
      } else if (world_rank == 1) {
        MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
        printf("Process 1 received number %d from process 0\n",
                number); }
  MPI_Finalize();
  return 0;}
```

Compiling & run MPI Programs

```
mpicc -O -o foo foo.c
```

- **mpicc/mpiCC**: script to compile and link c/c++MPI programs
- Flags: same meaning as C compiler
  - **`-O`** — optimize
  - **`-o <file>`** — where to put executable
- **`mpirun -np <p> <exec> <arg1>`** …
  - **`-np <p>`** — number of processes
  - **`<exec>`** — executable
  - **`<arg1>`** … — command-line arguments

How to design parallel computing programs?

- *Partitioning.*
  - The problems are decomposed into small tasks, seeking opportunities for parallel execution.
- *Communication.*
  - The communication required to coordinate task execution is determined, and appropriate communication structures and algorithms are defined.
- *Agglomeration.*
  - The task and communication structures defined in the first two stages of a design are evaluated with respect to performance requirements and implementation costs. If necessary, tasks are combined into larger tasks to improve performance or to reduce development costs.
- *Mapping.*
  - Each task is assigned to a processor in a manner that attempts to satisfy the competing goals of maximizing processor utilization and minimizing communication costs. Mapping can be specified statically or determined at runtime by load-balancing algorithms.

Matrix addition and multiplication

cpu1 cpu2

$$\underline{A} = \left(\begin{array}{cc|cc} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{array}\right) \quad and \quad \underline{B} = \left(\begin{array}{cc|cc} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ \hline b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{array}\right)$$

cpu3 cpu4

$$\underline{A} = \begin{pmatrix} \underline{A}_{11} & \underline{A}_{12} \\ \underline{A}_{21} & \underline{A}_{22} \end{pmatrix} \quad and \quad \underline{B} = \begin{pmatrix} \underline{B}_{11} & \underline{B}_{12} \\ \underline{B}_{21} & \underline{B}_{22} \end{pmatrix}$$

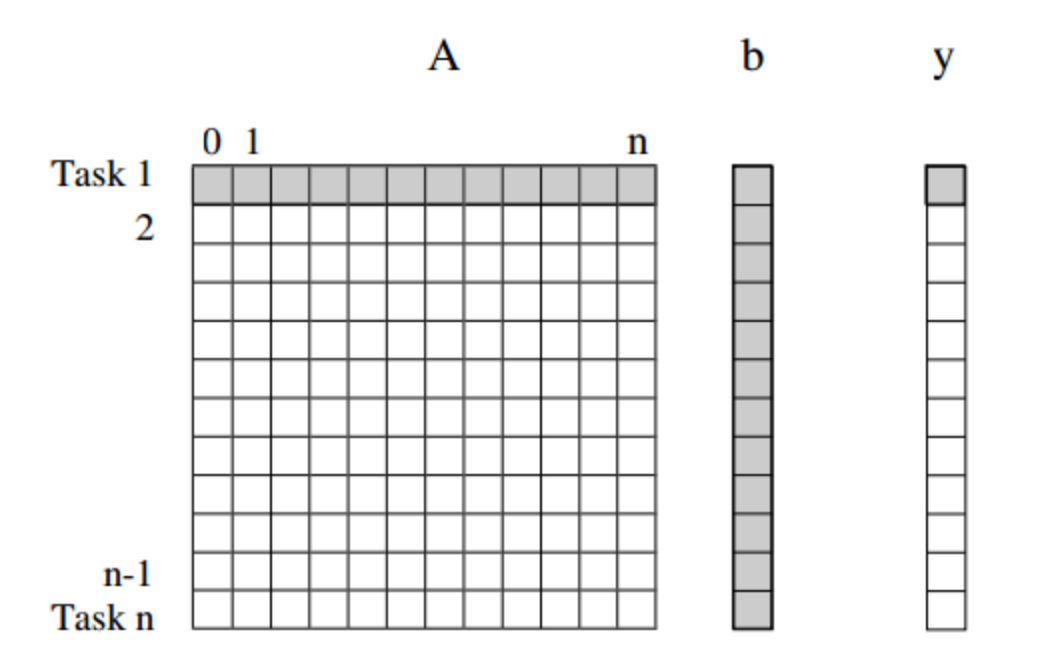
addition

$$\underline{A} + \underline{B} = \begin{pmatrix} \underline{A}_{11} + \underline{B}_{11} & \underline{A}_{12} + \underline{B}_{12} \\ \underline{A}_{21} + \underline{B}_{21} & \underline{A}_{22} + \underline{B}_{22} \end{pmatrix}$$

multiplication

$$\underline{A}\,\underline{B} = \begin{pmatrix} matmul(A_{11}, B_{11}) + matmul(A_{12}, B_{21}) & matmul(A_{11}, B_{12}) + matmul(A_{12}, B_{22}) \\ matmul(A_{21}, B_{11}) + matmul(A_{22}, B_{21}) & matmul(A_{21}, B_{12}) + matmul(A_{22}, B_{22}) \end{pmatrix}$$

Matrix multiplication

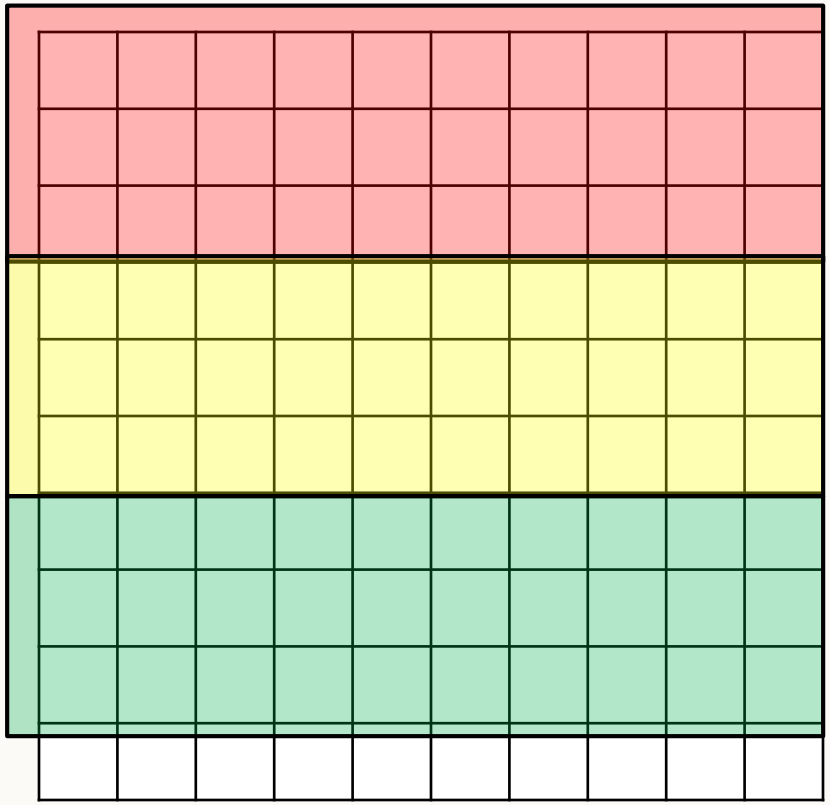


Computation of each element of output vector y is independent of other elements. Based on this, a dense matrix-vector product can be decomposed into n tasks. The figure highlights the portion of the matrix and vector accessed by Task 1

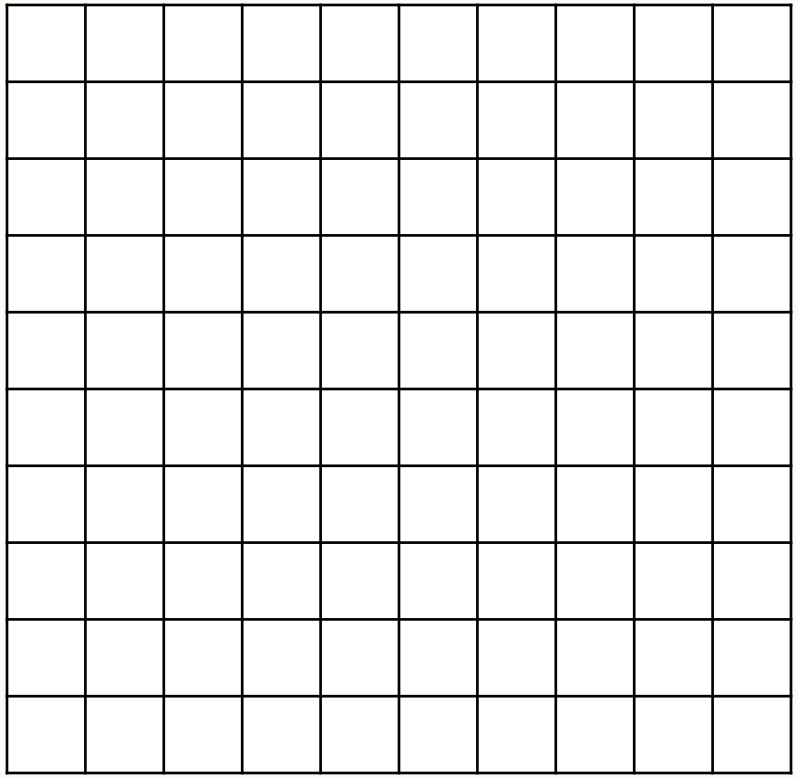
Is it a good approach?

Matrix-Matrix multiplication C= A*B

A, C



B



Example code

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

#define TAG 13

int main(int argc, char *argv[]) {
  double **A, **B, **C, *tmp;
  double startTime, endTime;
  int numElements, offset, stripSize, myrank, numnodes, N, i, j, k;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &numnodes);

  N = atoi(argv[1]);

  // allocate A, B, and C --- note that you want these to be
  // contiguously allocated.  Workers need less memory allocate
```

```
  if (myrank == 0) {
    tmp = (double *) malloc (sizeof(double ) * N * N);
    A = (double **) malloc (sizeof(double *) * N);
    for (i = 0; i < N; i++)
      A[i] = &tmp[i * N];
  }
  else {
    tmp = (double *) malloc (sizeof(double ) * N * N / numnodes);
    A = (double **) malloc (sizeof(double *) * N / numnodes);
    for (i = 0; i < N / numnodes; i++)
      A[i] = &tmp[i * N];
  }
```

Only Master (myrank=0) has a full copy of A, other processors only store a strip of A

Each processor has a copy of B

```
tmp = (double *) malloc (sizeof(double ) * N * N);
B = (double **) malloc (sizeof(double *) * N);
for (i = 0; i < N; i++)
  B[i] = &tmp[i * N];
```

```
if (myrank == 0) {
  tmp = (double *) malloc (sizeof(double ) * N * N);
  C = (double **) malloc (sizeof(double *) * N);
  for (i = 0; i < N; i++)
    C[i] = &tmp[i * N];
}
else {
  tmp = (double *) malloc (sizeof(double ) * N * N / numnodes);
  C = (double **) malloc (sizeof(double *) * N / numnodes);
  for (i = 0; i < N / numnodes; i++)
    C[i] = &tmp[i * N];
}
```

```
if (myrank == 0) {
    // initialize A and B
    for (i=0; i<N; i++) {
      for (j=0; j<N; j++) {
        A[i][j] = 1.0;
        B[i][j] = 1.0;
      }
    }
  }

  // start timer
  if (myrank == 0) {
    startTime = MPI_Wtime();
  }
```

Only Master (myrank=0) has a full copy of C, other processors only store a strip of C

```
stripSize = N/numnodes;

  // send each node its piece of A -- note
could be done via MPI_Scatter
  if (myrank == 0) {
    offset = stripSize;
    numElements = stripSize * N;
    for (i=1; i<numnodes; i++) {
      MPI_Send(A[offset], numElements,
MPI_DOUBLE, i, TAG, MPI_COMM_WORLD);
      offset += stripSize;
    }
  }
  else {  // receive my part of A
    MPI_Recv(A[0], stripSize * N, MPI_DOUBLE,
0, TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  }
```

Master: Send a strip of A to other processors
Others: receive part of A.

```
  // everyone gets B
  MPI_Bcast(B[0], N*N, MPI_DOUBLE, 0,
MPI_COMM_WORLD);
```

Every processor gets a copy of B through MPI_Bcast

```
  // Let each process initialize C to zero
  for (i=0; i<stripSize; i++) {
    for (j=0; j<N; j++) {
      C[i][j] = 0.0;
    }
  }
```

```
// do the work
  for (i=0; i<stripSize; i++) {
    for (j=0; j<N; j++) {
      for (k=0; k<N; k++) {
          C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
```

Standard matrix multiplication

```
  // master receives from workers  -- note
could be done via MPI_Gather
  if (myrank == 0) {
    offset = stripSize;
    numElements = stripSize * N;
    for (i=1; i<numnodes; i++) {
      MPI_Recv(C[offset], numElements,
MPI_DOUBLE, i, TAG, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
      offset += stripSize;
    }
  }
  else { // send my contribution to C
    MPI_Send(C[0], stripSize * N, MPI_DOUBLE,
0, TAG, MPI_COMM_WORLD);
  }
```

Others: send C back to master
Master: receive C from others

```
  // stop timer
  if (myrank == 0) {
    endTime = MPI_Wtime();
    printf("Time is %f\n", endTime-startTime);
  }
```

```
 // print out matrix here, if I'm
the master
  if (myrank == 0 && N < 10) {
    for (i=0; i<N; i++) {
      for (j=0; j<N; j++) {
        printf("%f ", C[i][j]);
      }
      printf("\n");
    }
  }

  MPI_Finalize();
  return 0;
}
```

Gauss elimination

Ax=b

A=

2	2	2	2	2	2	2	2	2	2
2	4	4	4	4	4	4	4	4	4
2	4	6	6	6	6	6	6	6	6
2	4	6	8	8	8	8	8	8	8
2	4	6	8	10	10	10	10	10	10
2	4	6	8	10	12	12	12	12	12
2	4	6	8	10	12	14	14	14	14
2	4	6	8	10	12	14	16	16	16
2	4	6	8	10	12	14	16	18	18
2	4	6	8	10	12	14	16	18	20

A[i][j] = ((j < i)? 2*(j+1) : 2*(i+1))

Load balancing: cyclic

b=

0
1
2
3
4
5
6
7
8
9

X=

-0.5
0
0
0
0
0
0
0
0
0.5

A=

2	2	2	2	2	2	2	2	2	2
2	4	4	4	4	4	4	4	4	4
2	4	6	6	6	6	6	6	6	6
2	4	6	8	8	8	8	8	8	8
2	4	6	8	10	10	10	10	10	10
2	4	6	8	10	12	12	12	12	12
2	4	6	8	10	12	14	14	14	14
2	4	6	8	10	12	14	16	16	16
2	4	6	8	10	12	14	16	18	18
2	4	6	8	10	12	14	16	18	20

Performance

Time: (seconds) = calculation time + communication time

Matrix dimension

computers	16	256	512	1024	2048	4096
4	0.004193	0.087752	0.223287	0.717674	3.268051	19.295917
8	0.00895	0.203581	0.404254	1.01758	3.226628	14.37058
16	0.01194	0.232125	0.510776	1.244048	3.571331	12.825494
single computer						
1	0.000067	0.066108	0.51573	4.167458	33.232826	280.781368

GPU

Time (micro seconds)

Matrix A size	Matrix B size	sequential code	cuda naïve	cuda optimized
4096x4096	4096x4096	968317169	54	10
1024x1024	1024x1024	9076053	31	60
512x1024	1024x1024	4475937	28	58
512x512	512x512	357706	26	53

Reference

- https://computing.llnl.gov/tutorials/mpi/samples/C/mpi_mm.c
- http://www.mcs.anl.gov/~itf/dbpp/text/node14.html
- http://www.mcs.anl.gov/research/projects/mpi/www/www3/
- http://www.cs.arizona.edu/classes/cs522/fall12/examples/mpi-mm.c