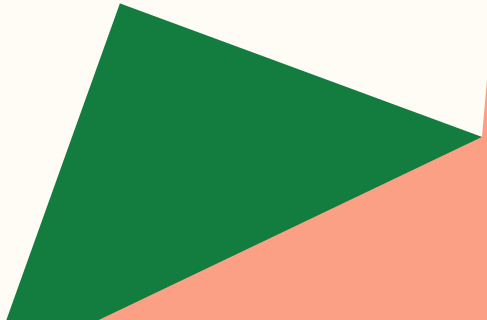




# MPI: Message Passing Interface

Lecture 5  
February 25 & 27, 2025



# To Dos

Reading for next time

Program 2 submission

Program 3



# Algorithm Presentations



MPI

# Parallelizing a Problem



Partition problem solution into tasks



Identify communication between tasks

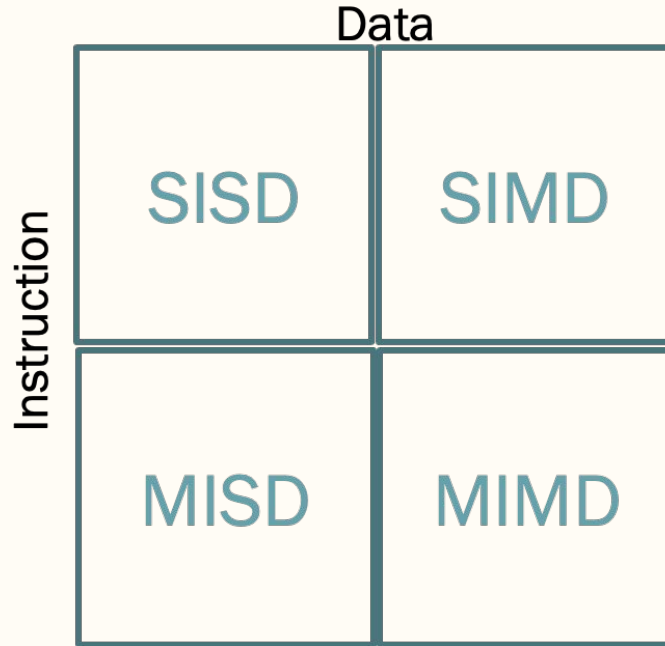


Aggregate tasks into composite tasks



Map composite tasks to cores

# Reminder: Flynn's Taxonomy



SPMD is a special case of MIMD

# Distributed Message Systems

- Each core has its own private memory
- Cores must explicitly communicate to coordinate or share data
- Programs often started by executing multiple processes
  - Processes given identifiers so they can identify each other
- Typically have **send()** and **receive()** functions to communicate between processes
  - May also have collective communication mechanisms like
    - **broadcast()**
    - **reduction()**
- Programmer has to do all the heavy lifting
  - Figure out parallelization
  - Figure out data replication and distribution to different processes

# Message Passing Interface (MPI)

- Defines library of functions called from C, C++, or Fortran programs
- Compilation:
  - `mpicc -g -Wall -o hello hello.c`
  - Calls C compiler with all the libraries needed
- Execution:
  - `mpiexec -n 4 ./hello`
  - Starts up 4 processes, giving them names, and enabling their communication
  - All 4 processes run the same `./hello` executable
  - Processes are given a non-negative rank which they use to differentiate which code to run



# Basic MPI program setup

- `#include <mpi.h>`
- `MPI_Init(int *argc_p, char ***argv_p)`
  - Should be first MPI function called
  - Allocates resources (e.g. buffers) and decides which process gets assigned which rank (i.e. 0, 1, 2, ..., p-1)
- `MPI_Finalize()`
  - Last MPI function called
  - Releases resources

```
#include <mpi.h>

int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);
    ...
    MPI_Finalize();
    return 0;
}
```

# Communicators

- Named collection of processes that can send messages to each other
  - Can have more than one communicator in a program
  - Created by `MPI_Init()`
  - Default communicator named `MPI_COMM_WORLD`
- `MPI_Comm_size(MPI_Comm comm, int *comm_sz_p)`
  - Let's you find out number of processes in communicator
- `MPI_Comm_rank(MPI_Comm comm, int *my_rank_p)`
  - Let's process find out its rank/name in communicator
- Can create new communicators using `MPI_Comm_split()`

# Determining Your Rank in A Communicator

- A process can dynamically determine it's rank in a communicator
- `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
  - Rank will be placed into second argument's memory location

# MPI\_Send

```
MPI_Send(void*msg_buf_p, int msg_size, MPI_Datatype msg_type,  
         int dest, int tag, MPI_Comm communicator)
```

- msg\_size: number of elements to be sent
- msg\_type: can send different types of data
- dest: rank of process to send to
- tag: mechanism for identifying message if multiple messages will be sent from sender to receiver (non-negative int)

MPI datatype
MPI_CHAR
MPI_SHORT
MPI_INT
MPI_LONG
MPI_FLOAT
MPI_DOUBLE
...

Implementation dependent:

May buffer message and not block if size < threshold; if size > threshold, blocks until completes

# MPI\_Recv

```
MPI_Recv(void*msg_buf_p, int buf_size, MPI_Datatype buf_type,  
         int src, int tag, MPI_Comm communicator,  
         MPI_Status *status_p)
```

- **status\_p:**
  - Stores info about MPI\_SOURCE, MPI\_TAG, MPI\_ERROR
    - ex.) `MPI_Get_count(&status, recv_type, &count)`
      - `count` tells you amount of data received
  - Can specify MPI\_STATUS\_IGNORE as argument if you don't care
- **src:**
  - Indicates sender
  - Can be MPI\_ANY\_SOURCE to deal with blocking issues on multiple sources

Always blocks

# Sends and Receives Must Match

- Communicator
- Tag
- Source
- Destination
- Type
- Receiver buffer size  $\geq$  sender buffer size

# SPMD

```
#include <mpi.h>
int main(int argc, char **argv)
{
    char greeting[100];
    int comm_sz;
    int my_rank;

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if(my_rank != 0){
        sprintf("Greetings from process %d of %d\n", comm_sz, my_rank);
        MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    }
    else{
        printf("Greetings from %d\n", my_rank);
        for(int q = 0; q < comm_sz; q++){
            MPI_Recv(greeting, 100, MPI_CHAR, q, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("%s\n", greeting);
        }
    }
    MPI_Finalize();
    return 0;
}
```

# Collective Communication

- Sometimes you want group communication – either sending or receiving
- There are some well-defined group communications to make programming easier than all point-to-point communication
  - Can send/receive to all members of communicator
- Underlying implementation figures out best way to do group communications (e.g., trees)



# MPI\_Reduce

```
MPI_Reduce(void *input_data_p, void *output_data_p, int count,  
           MPI_Datatype datatype, MPI_Op operator,  
           int dest_process, MPI_Comm communicator)
```

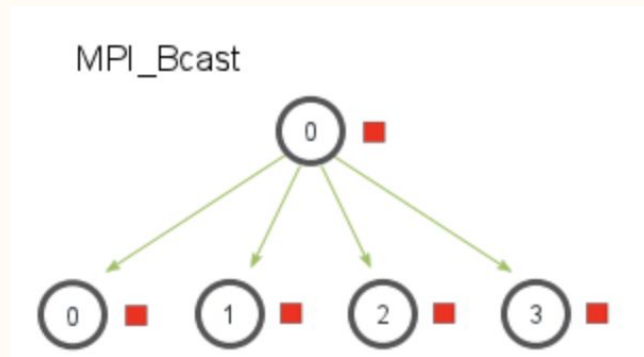
- Construct that collects information from all processes and combines results together
  - All processes in communicator must execute **MPI\_Reduce()** with compatible arguments
    - They match up by order of **MPI\_Reduce()** calls across different processes if more than 1 call
- **input\_data\_p**: data from each process
- **output\_data\_p** : global result used only in dest\_process
- **count**: If > 1, indicates number of elements in array
- **operator**: Max, Min, Sum, Product, Logical AND/OR/XOR, Bitwise AND/OR/XOR, Max and location of max, Min and location of min

# MPI\_Allreduce

```
MPI_Allreduce(void *input_data_p, void *output_data_p,  
              int count, MPI_Datatype datatype,  
              MPI_Op operator, MPI_Comm communicator)
```

- Result stored on all processes

# MPI\_Bcast



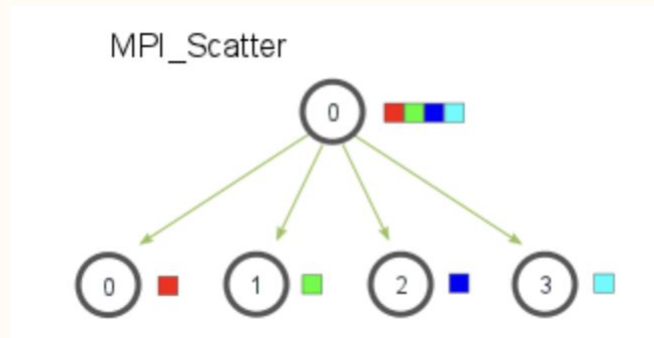
```
MPI_Bcast(void *data_p, int count, MPI_Datatype datatype,  
          int src_process, MPI_Comm communicator)
```

- Sends data to all processes in communicator
  - All processes in communicator must execute **MPI\_Bcast()** with compatible arguments
  - Data will be taken from src\_process's **data\_p**
  - Data will be put into all other process' **data\_p**

# Example: Find Min in Large Array

- How do we distribute the data?

# MPI\_Scatter



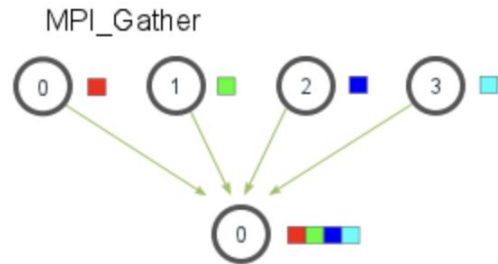
```
MPI_Scatter(void *send_data_p, int send_count,  
            MPI_Datatype send_type, void *recv_data_p,  
            int recv_count, MPI_Datatype recv_type,  
            int src_proc, MPI_Comm communicator)
```

- Construct that sends information selectively to other processes
  - Divides total data in send\_data\_p amongst comm\_sz processes
- send\_data\_p : data to be distributed
- send\_count : amount of data going to each process

# Ways to Distribute Data

- Block partition
- Cyclic partition
- Block-cyclic partition

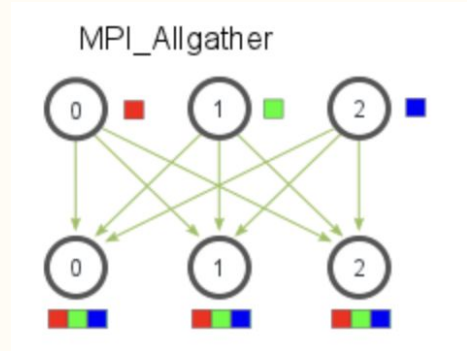
# MPI\_Gather



```
MPI_Gather(void *send_buf_p, int send_count,  
           MPI_Datatype send_type, void *recv_buf_p,  
           int recv_count, MPI_Datatype recv_type,  
           int dest_proc, MPI_Comm communicator)
```

- Construct that collects information from other processes and stores consecutively
- `send_buf_p` : data to be collected from given process
- `send_count` : amount of data coming from each process
- `recv_buf_p` : location to store all collected data (`comm_size * send_count`)
- `recv_count` : amount of data from each process

# MPI\_Allgather



```
MPI_Allgather(void *send_buf_p, int send_count,  
              MPI_Datatype send_type, void *recv_buf_p,  
              int recv_count, MPI_Datatype recv_type,  
              MPI_Comm communicator)
```

- Construct that collects information from other processes and stores consecutively AT ALL processes
- `recv_buf_p` : location to store all collected data (`comm_size * send_count`)
- `recv_count` : amount of data from each process

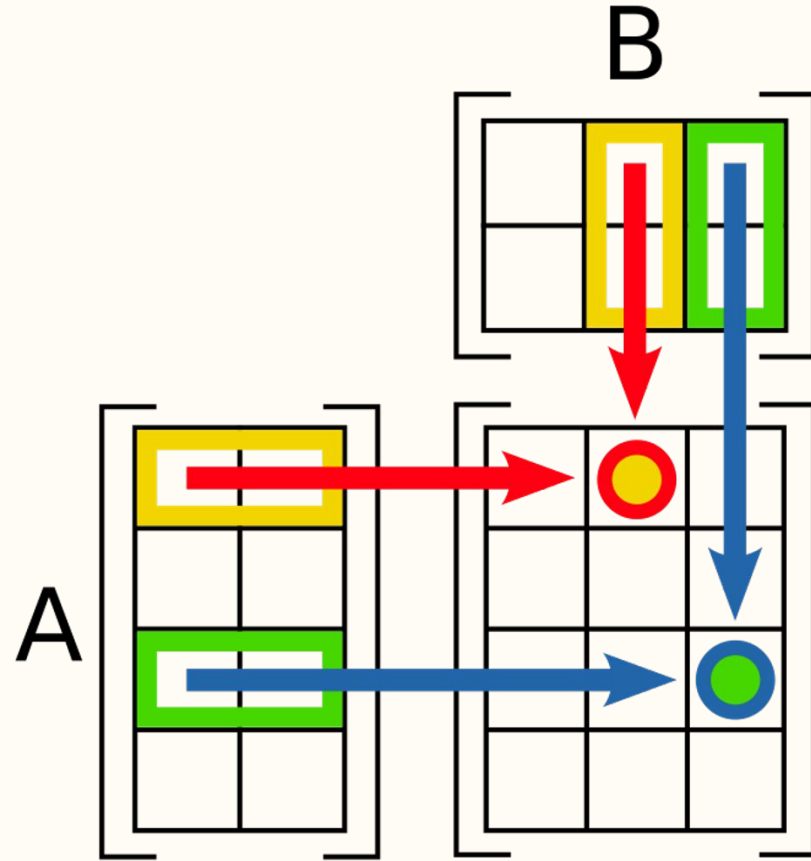


# Dave Patterson's take on Matrix Multiply



- Dave Patterson - Distinguished Engineer, Google; ACM A.M. Turing Award Laureate
- ACM Tech Talk
- Title: A New Golden Age for Computer Architecture
- In the 1980s, Mead and Conway democratized chip design and high-level language programming surpassed assembly language programming, which made instruction set advances viable. Innovations like Reduced Instruction Set Computers (RISC), superscalar, and speculation ushered in a Golden Age of computer architecture, when performance doubled every 18 months. The ending of Dennard Scaling and Moore's Law crippled this path; microprocessor performance improved only 3% last year! In addition to poor performance gains of modern microprocessors, Spectre recently demonstrated timing attacks that leak information at high rates. The ending of Dennard scaling and Moore's law and the deceleration of performance gains for standard microprocessors are not problems that must be solved but facts that if accepted offer breathtaking opportunities. We believe high-level, domain-specific languages and architectures, freeing architects from the chains of proprietary instruction sets and the demand from the public for improved security will usher in a new Golden Age. Aided by open source ecosystems, agilely developed chips will convincingly demonstrate advances and thereby accelerate commercial adoption. The instruction set philosophy of the general-purpose processors in these chips will likely be RISC, which has stood the test of time. We envision the same rapid improvement as in the last Golden Age, but this time in cost, energy, and security as well as in performance. Like in the 1980s, the next decade will be exciting for computer architects in academia and in industry!

# Matrix Multiply



# Matrix Multiply – Ways to Parallelize

- What is a task?
- What data is needed for each task?



# Parallel Performance

# Limitations to Performance Improvements

- Amdahl's Law

$$T_{enhanced} = (1 - fraction_{enhanced}) \times T_{unenhanced} + (\frac{fraction_{enhanced}}{Speedup_{enhancement}} \times T_{unenhanced})$$

- Inherently sequential parts of code
- Overhead in parallel parts of code
  - Communication of data between processes
  - Load balancing
  - Synchronization