OpenMP Synchronization

- Mutual exclusion by critical sections

```
#pragma omp parallel
{
  // …
  #pragma omp critical
  sum += local_sum
}
```

- Named critical sections
  - unnamed sections treated as one
- Critical section is scoped

- Atomic update

```
#pragma omp parallel
{
  // …
  #pragma omp atomic
  sum += local_sum
}
```

- Specialized critical section
- May enable fast HW implementation
- Applies to following statement

OpenMP Synchronization

- Barrier directive
  - Thread waits until all others reach this point
  - Implicit barrier at end of each parallel region

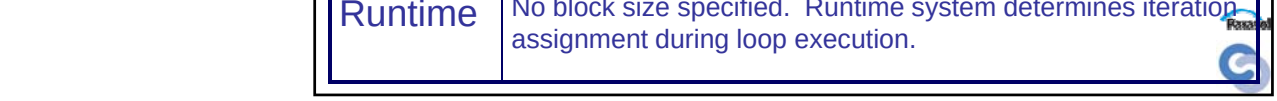
```
#pragma omp parallel
{
  // …
  #pragma omp barrier
  // …
}
```

OpenMP Scheduling

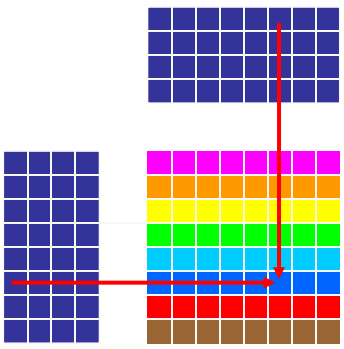
- Load balancing handled by runtime scheduler
- Scheduling policy can be set for each parallel loop

Scheduling Policies

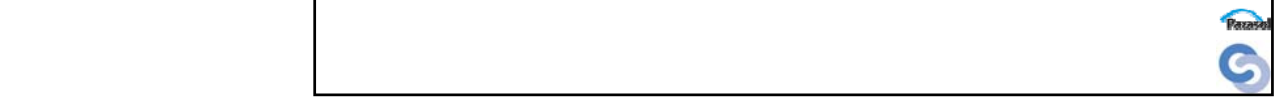
Static	Create blocks of size *chunk* and assign to threads before loop begins execution. Default chunk creates equally-sized blocks.
Dynamic	Create blocks of size *chunk* and assign to threads during loop execution. Threads request a new block when finished processing a block. Default chunk is 1.
Guided	Block size is proportional to number of unassigned iterations divided by number of threads. Minimum block size can be set.
Runtime	No block size specified. Runtime system determines iteration assignment during loop execution.



OpenMP Matrix Multiply

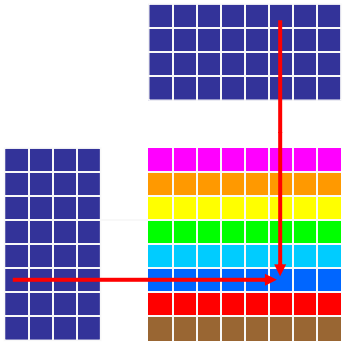


```
#pragma omp parallel for
for(int i=0; i<M; ++i) {
  for(int j=0; j<N; ++j) {
    for(int k=0; k<L; ++k) {
      C[i][j] +=
          A[i][k]*B[k][j];
    }
  }
}
```



OpenMP Matrix Multiply

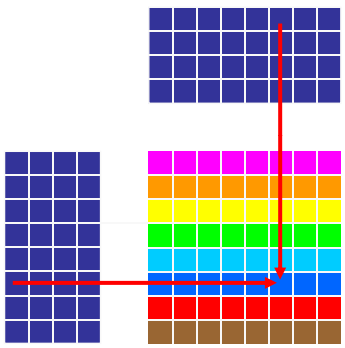
- Parallelizing two loops
  - Uses nested parallelism support
  - Each element of result matrix computed independently



```
#pragma omp parallel for
for(int i=0; i<M; ++i) {
  #pragma omp parallel for
  for(int j=0; j<N; ++j) {
     for(int k=0; k<L; ++k) {
      C[i][j] +=
          A[i][k]*B[k][j];
    }
  }
}
```

OpenMP Matrix Multiply

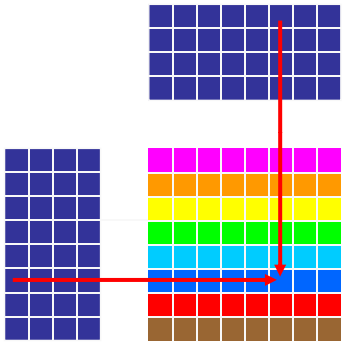
- Parallelizing inner loop
  - Inner loop parallelized instead of outer loop
    - Minimizes work in each parallel loop – for illustration purposes only
  - Multiple threads contribute to each element in result matrix
  - Critical section ensures only one thread updates at a time



```
for(int i=0; i<M; ++i) {
  for(int j=0; j<N; ++j) {
    #pragma omp parallel for
    for(int k=0; k<L; ++k) {
      #pragma omp critical
      C[i][j] +=
          A[i][k]*B[k][j];
    }
  }
}
```

OpenMP Matrix Multiply

- Use dynamic scheduling of iterations

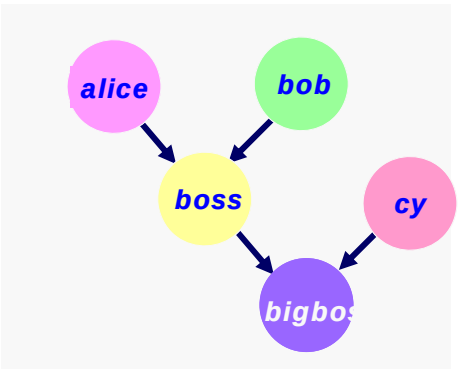


```
#pragma omp parallel for \
schedule(dynamic)
for(int i=0; i<M; ++i) {
  for(int j=0; j<N; ++j) {
    for(int k=0; k<L; ++k) {
      C[i][j] +=
          A[i][k]*B[k][j];
    }
  }
}
```

OpenMP 3.0 – Task Decomposition

```
a = alice();
b = bob();
s = boss(a, b);
c = cy();
printf ("%6.2f\n",
  bigboss(s,c));
```

alice,bob, and cy can be computed in parallel



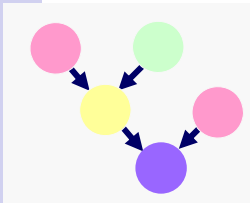
omp sections

- **`#pragma omp sections`**
- Must be inside a parallel region
- Precedes a code block containing of *N* blocks of code that may be executed concurrently by *N* threads
- Encompasses each omp section

- **`#pragma omp section`**
- Precedes each block of code within the encompassing block described above
- May be omitted for first parallel section after the parallel sections pragma
- Enclosed program segments are distributed for parallel execution among available threads

Functional Level Parallelism w sections

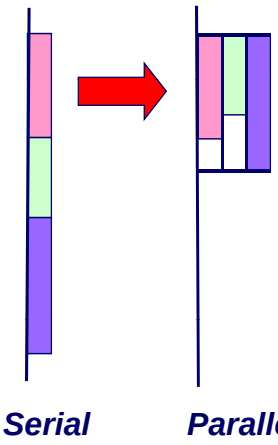
```
#pragma omp parallel sections
{
#pragma omp section  /* Optional */
   a = alice();
#pragma omp section
   b = bob();
#pragma omp section
    c = cy();
}

s = boss(a, b);
printf ("%6.2f\n",
         bigboss(s,c));
```



Advantage of Parallel Sections

- Independent sections of code can execute concurrently – reduce execution time

```
#pragma omp parallel sections
{
    #pragma omp section
    phase1();
    #pragma omp section
    phase2();
    #pragma omp section
    phase3();
}
```

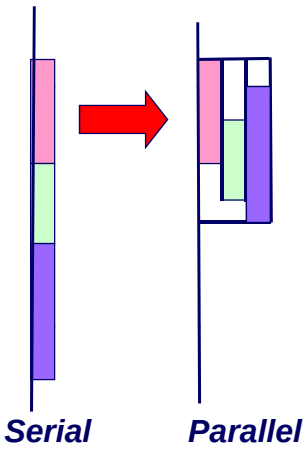


New Addition to OpenMP 3.0

- Tasks – Main change for OpenMP 3.0
- Allows parallelization of irregular problems
  - unbounded loops
  - recursive algorithms
  - producer/consumer

What are tasks?

- Tasks are independent units of work
- Threads are assigned to perform the work of each task
  - Tasks may be deferred
- Tasks may be executed immediately
- The runtime system decides which of the above
  - Tasks are composed of:
    - **code** to execute
    - **data** environment
    - **internal control variables** (ICV)



Simple Task Example

```
#pragma omp parallel
// assume 8 threads
{
  #pragma omp single private(p)
  {
    …
    while (p) {
    #pragma omp task
      {
        processwork(p);
      }
      p = p->next;
    }
  }
}
```

A pool of 8 threads is created here

One thread gets to execute the while loop

The single "while loop" thread creates a task for each instance of processwork()

Task Construct – Explicit Task View

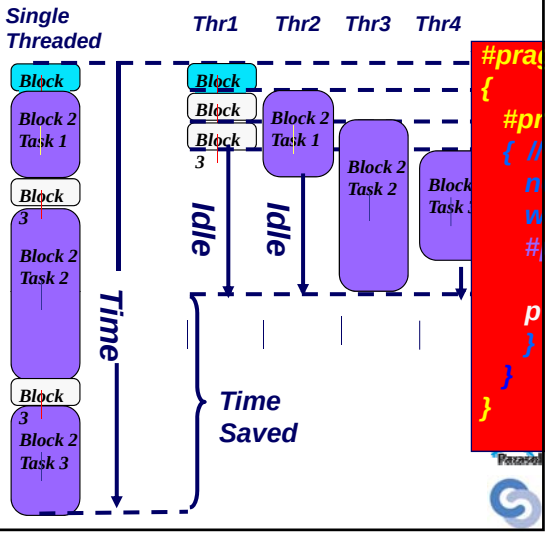
- A team of threads is created at the omp parallel construct
- A single thread is chosen to execute the while loop – lets call this thread "L"
- Thread L operates the while loop, creates tasks, and fetches next pointers
- Each time L crosses the omp task construct it generates a new task and has a thread assigned to it
- Each task runs in its own thread
- All tasks complete at the barrier at the end of the parallel region's single construct

```
#pragma omp parallel
{
   #pragma omp single
   {  // block 1
      node * p = head;
      while (p) {  //block 2
      #pragma omp task
        private(p)
         process(p);
      p = p->next; //block 3
      }
   }
}
```

Why are tasks useful?

Have potential to parallelize irregular patterns and recursive function calls

```
#pragma omp parallel
{
   #pragma omp single
   {  // block 1
      node * p = head;
      while (p) {
//block 2
      #pragma omp task
         process(p);
      p = p->next;
//block 3
      }
   }
}
```



Activity 3 – Linked List using Tasks

- **Objective:** Modify the linked list pointer chasing code to implement tasks to parallelize the application
- Follow the Linked List task activity called LinkedListTask in the student lab doc

```
while(p != NULL){
  do_work(p->data);
  p = p->next;
}
```

When are tasks gauranteed to be complete?

- ***Tasks are gauranteed to be complete:***
- ***At thread or task barriers***
  - ***At the directive:*** `#pragma omp barrier`
  - ***At the directive:*** `#pragma omp taskwait`

Task Completion Example

```
#pragma omp parallel
{
   #pragma omp task
   foo();
   #pragma omp barrier
   #pragma omp single
   {
      #pragma omp task
      bar();
   }
}
```

Multiple foo tasks created here – one for each thread

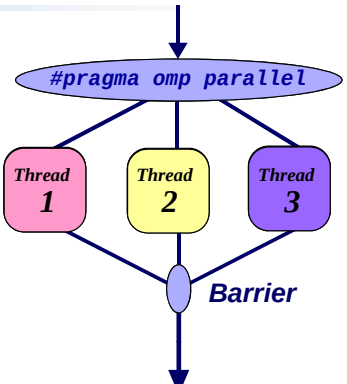
All foo tasks guaranteed to be completed here

One bar task created here

bar task guaranteed to be completed here

Parallel Construct – Implicit Task View

- Tasks are created in OpenMP even without an explicit task directive.
- Lets look at how tasks are created implicitly for the code snippet below
  - Thread encountering parallel construct packages up a set of *implicit* tasks
  - Team of threads is created.
  - Each thread in team is assigned to one of the tasks (and *tied* to it).
  - Barrier holds original master thread until all implicit tasks are finished.




```
#pragma omp parallel
   {
      int mydata;
      code…
   }
```

Task Construct

```
#pragma omp task [clause[[,]clause] ...]
         structured-block
```

where clause can be one of:

```
if (expression)
untied
shared (list)
private (list)
firstprivate (list)
default( shared | none  )
```

Tied & Untied Tasks

- Tied Tasks:
  - A tied task gets a thread assigned to it at its first execution and the same thread services the task for its lifetime
  - A thread executing a tied task, can be suspended, and sent of to execute some other task, but eventually, the same thread will return to resume execution of its original tied task
  - Tasks are tied unless explicitly declared untied
- Untied Tasks:
  - An united task has no long term association with any given thread. Any thread not otherwise occupied is free to execute an untied task. The thread assigned to execute an untied task may only change at a "task scheduling point".
  - An untied task is created by appending “untied” to the task clause
  - Example: #pragma omp task untied

Task switching

- **task switching** The act of a *thread* switching from the execution of one *task* to another *task.*
- The purpose of task switching is distribute threads among the unassigned tasks in the team to avoid piling up long queues of unassigned tasks
- Task switching, for tied tasks, can only occur at task scheduling points located within the following constructs
  - encountered **task** *constructs*
  - encountered **taskwait** *constructs*
  - encountered **barrier** *directives*
  - implicit **barrier** *regions*
  - at the end of the *tied task region*
- Untied tasks have implementation dependent scheduling points

Task switching example

- ***The thread executing the "for loop" , AKA the generating task, generates many tasks in a short time so...***
- ***The SINGLE generating task will have to suspend for a while when "task pool" fills up***
  - ***Task switching is invoked to start draining the "pool"***
  - ***When the "pool" is sufficiently drained – then the single task can being generating more tasks again***

```
#pragma omp single
{
    for (i=0; i<ONEZILLION; i++)
     #pragma omp task
         process(item[i]);
    }
}
```

Table of Contents

Pthreads

- Specification part of larger IEEE POSIX standard
  - POSIX is the **P**ortable **O**perating **S**ystem **I**nterface
  - Standard C API for threading libraries
    - IBM provides Fortran API
  - Introduced in 1995
- Explicit threading of application
  - User calls functions to create/destroy threads

Materials from http://www.llnl.gov/computing/tutorials/pthreads/

The Pthreads Model

- Execution Model
  - Explicit parallelism
  - Explicit synchronization
- Productivity
  - Not a primary objective
  - Library for existing language
  - Low level of abstraction
  - Uses opaque objects – prevents expansion

The Pthreads Model

- Performance
  - No attempts to manage latency
  - Load balancing left to OS
  - Developer responsible for creating high degree of parallelism by spawning threads
- Portability
  - Library widely available

Pthreads Thread Management

- User creates/terminates threads
- Thread creation
  - `pthread_create`
  - Accepts a single argument (void *)
- Thread termination
  - `pthread_exit`
  - Called from within terminating thread

Pthreads Synchronization

Mutual Exclusion Variables (mutexes)

•Mutexes must be initialized before use
•Attribute object can be initialized to enable error checking

```
pthread_mutex_t mutexsum;
void *dot_product(void *arg) {
  …
  pthread_mutex_lock (&mutexsum);
  sum += mysum;
  pthread_mutex_unlock (&mutexsum);
  …
}
int main() {
  pthread_mutex_init(&mutexsum, NULL);
  …
  pthread_mutex_destroy(&mutexsum);
}
```

Pthreads Synchronization

Condition Variables

- Allows threads to synchronize based on value of data
- Threads avoid continuous polling to check condition
- Always used in conjunction with a mutex
  - Waiting thread(s) obtain mutex then wait
    - pthread_cond_wait() function unlocks mutex
    - mutex locked for thread when it is awakened by signal
  - Signaling thread obtains lock then issues signal
    - pthread_cond_signal() releases mutex

Condition Variable Example

Two threads update a counter
Third thread waits until counter reaches a threshold

```
pthread_mutex_t mtx;
pthread_cond_t cv;

int main() {
…
pthread_mutex_init(&mtx, NULL);
pthread_cond_init (&cv, NULL);
…
pthread_create(&threads[0], &attr,
               inc_count, (void *)&thread_ids[0]);
pthread_create(&threads[1], &attr,
               inc_count, (void *)&thread_ids[1]);
pthread_create(&threads[2], &attr,
               watch_count, (void *)&thread_ids[2]);
…
}
```

Condition Variable Example

Incrementing Threads

```
void *inc_count(void *idp) {
…
for (i=0; i<TCOUNT; ++i) {
  pthread_mutex_lock(&mtx);
  ++count;
  if (count == LIMIT)
    pthread_cond_signal(&cv);
  pthread_mutex_unlock(&mtx);
  …
}
…
}
```

Waiting Thread

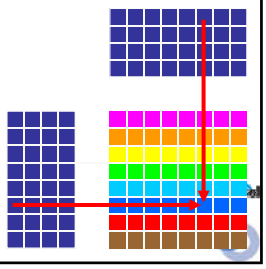
```
void *watch_count(void *idp) {
…
pthread_mutex_lock(&mtx);
while (count < COUNT_LIMIT) {
  pthread_cond_wait(&cv, &mtx);
}
pthread_mutex_unlock(&mtx);
…
}
```

pthread_cond_broadcast() used if multiple threads waiting on signal

Pthreads Matrix Multiply

```
int tids[M];
pthread_t threads[M];
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_attr_setdetachstate(
  &attr,
  PTHREAD_CREATE_JOINABLE);

for (i=0; i<M; ++i) {
  tids[i] = i;
  pthread_create(&threads[i],
   &attr, work, (void *) &tids[i]);
}

for (i=0; i<M; ++i) {
  pthread_join(threads[i], NULL);
}
```

```
void* work(void* tid) {
  for(int j=0; j<N; ++j) {
    for(int k=0; k<L; ++k) {
      C[tid][j] +=
        A[tid][k]*B[k][j];
    }
  }
  pthread_exit(NULL);
}
```



References

OpenMP

http://www.openmp.org

http://www.llnl.gov/computing/tutorials/openMP

Pthreads

http://www.llnl.gov/computing/tutorials/pthreads

"Pthreads Programming". B. Nichols et al. O'Reilly and Associates.

"Programming With POSIX Threads". D. Butenhof. Addison Wesley

Table of Contents

Message Passing Model

- Large scale parallelism (up to 100k+ CPUs)
- Multiple (possibly heterogeneous) system images
- Distributed memory
  - Nodes can only access local data
  - Application (User) responsible for:
    - Distributing data
    - Redistributing data (when necessary)
    - Maintaining memory coherent

Message Passing Model

- Explicit communication
  - Two-sided P2P:
    - Communication initiated on one side requires matching action on the remote side
    - E.g. MPI_Send – MPI_Recv
  - One-sided P2P:
    - Communication is initiated on one side and no action is required on the other
    - E.g. MPI_Get/Put, gasnet_get/put ...
- Implicit synchronization with two-sided communication
  - The matching of communication operations from both sides ensures synchronization

Message Passing Model

- Objectives of the model
  - Enabling parallelization on highly scalable hardware
  - Support for heterogeneous systems
  - Often coarse-grained parallelism
- Main issues
  - Communication
  - Synchronization
  - Load balancing

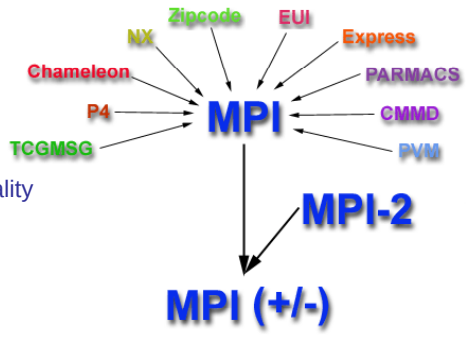
Projects of Interest

- Message Passing Interface (MPI)
  - De facto standard for this model
  - Deemed low level and difficult to program
  - Two-sided and one-sided communication
- Charm++
  - Asynchronous Remote Method Invocation (RMI) communication
  - Split-phase programming model
    - No synchronous communication
    - Caller provides a callback handler to asynchronously process “return” value

Table of Contents

Message Passing Interface (MPI)

- 1980s – early 1990s
  - Distributed memory, parallel computing develops
  - Many incompatible software tools
  - Usually tradeoffs between portability, performance, functionality and price
- Recognition of the need for a standard arose.



Material from: http://www.llnl.gov/computing/tutorials/mpi/

Message Passing Interface (MPI)

- Standard based on the consensus of the MPI Forum
  - Not sanctioned by any major standards body
  - Wide practical acceptance
  - No effective alternative to date
- First draft of the MPI-1 standard presented at Supercomputing 1993
- Current standard MPI-2 developed between 1995 and 1997
- Standardization committee open to all members of the HPC community

Further reading and standard documents: http://www.mpi-forum.org/

Message Passing Interface (MPI)

- Objectives
  - High performance and scalability
  - Portability
  - Productivity is not an objective (actually it was)
- Used as communication layer for higher-level libraries
  - Often for more productivity-oriented libraries
  - ISO/OSI layer-5 interface
    - Communication is reliable and sessions are managed internally
    - Data is not structured

MPI: Specification, not Implementation

- Language Independent Specification (LIS)
- Library implementations of MPI vary in:
  - Performance
    - Target or rely on specific hardware (RDMA, PIM, Coprocessors …)
    - Provide load-balancing and processor virtualization (e.g., AMPI)
  - Functionality
    - Support for parallel I/O
    - Support for multithreading within MPI processes
- Standard provides language bindings for Fortran, C and C++
- Implementations usually provide APIs for C, C++ and Fortran
- Project implementations for Python, OCaml, and Java

MPI – Programming Model

Execution Model

- Explicitly parallel
  - Programmer responsible for correctly identifying parallelism and for implementing parallel algorithms using MPI constructs
  - Multi-threaded view
- SPMD
- Explicit data distribution
- Flat parallelism
  - Number of tasks dedicated to run a parallel program is static
- Processor Consistency (one-sided communication)

MPI – Programming Model

Productivity

- Not a principal objective
  - Low level of abstraction
  - Communication is not structured (marshalling done by the user)

Performance

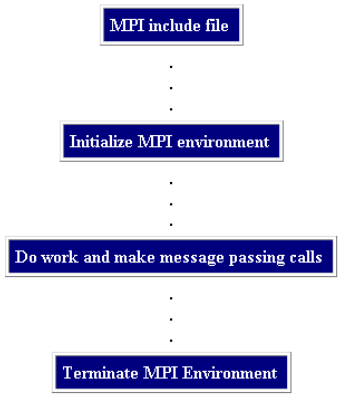
- Vendor implementations exploit native hardware features to optimize performance

Portability

- Most vendors provide an implementation
  - E.g., Specialized open source versions of MPICH, LAM or OpenMPI
- Standard ensures compatibility

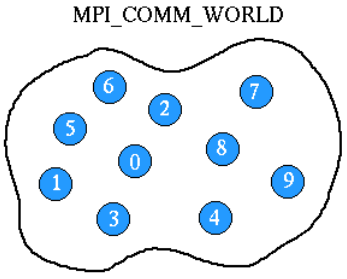
MPI – Program Structure

General program structure



Communicators and groups

- Collection of processes that may communicate
- Unique rank (processor ID) within communicator
- Default communicator: MPI_COMM_WORLD



Materials from: http://www.llnl.gov/computing/tutorials/mpi/

MPI – Point to Point Communication

Types of Point-to-Point Operations:

- Message passing between two, and only two, different MPI tasks
  - One task performs a send operation
  - The other task matches with a receive operation
- Different types of send/receive routines used for different purposes
  - Synchronous send
  - Blocking send / blocking receive
  - Non-blocking send / non-blocking receive
  - Buffered send
  - Combined send/receive
  - "Ready" send
- Any type of send can be paired with any type of receive
- Test and Probe routines to check the status of pending operations

Material from: http://www.llnl.gov/computing/tutorials/mpi/

MPI – Point to Point Communication

Blocking vs. Non-blocking

- Most routines can be used in either blocking or non-blocking mode
- Blocking communication routines
  - Blocking send routines only return when it is safe to reuse send buffer
    - Modifications to send buffer will not affect data received on the remote side
      - Data already sent
      - Data buffered in a system buffer
  - Blocking send calls can be synchronous
    - Handshaking with the receiver
  - Blocking send calls can be asynchronous
    - System buffer used to hold the data for eventual delivery to the receiver
  - Blocking receive calls only return after the data has arrived and is ready for use by the program

Materials from: http://www.llnl.gov/computing/tutorials/mpi/

MPI – Point to Point Communication

Blocking communication example

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

int main(int argc,char *argv[]) {
  int numtasks, rank, dest, source, rc, count, tag=1;
  char inmsg, outmsg='x';
  MPI_Status Stat;

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

  if (rank == 0) {
    dest = 1;
    source = 1;
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  }
  else if (rank == 1) {
    dest = 0;
    source = 0;
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }

  rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
  printf("Task %d: Received %d char(s) from task %d with tag %d \n",
         rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);

  MPI_Finalize();
}
```

Materials from: http://www.llnl.gov/computing/tutorials/mpi/

MPI – Point to Point Communication

Blocking vs. Non-blocking

- Non-blocking communication routines
  - Send and receive routines behave similarly
    - Return almost immediately
    - Do not wait for any communication events to complete
      - Message copying from user memory to system buffer space
      - Actual arrival of message
  - Operations "request" the MPI library to perform an operation
    - Operation is performed when its requirements are met (e.g., message arrives)
    - User cannot predict when that will happen
  - Unsafe to modify the application buffer until completion of operation
    - Wait and Test routines used to determine completion
- Non-blocking communications primarily used to overlap computation with communication and exploit possible performance gains

Material from: http://www.llnl.gov/computing/tutorials/mpi/

MPI – Point to Point Communication

Non-blocking communication example

```
MPI_Request reqs[4];
MPI_Status stats[4];

prev = rank-1;
next = rank+1;
if (rank == 0) prev = numtasks - 1;
if (rank == (numtasks - 1)) next = 0;

MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

  {
    // do some work
    // work will overlap with previous communication
  }

MPI_Waitall(4, reqs, stats);
```

Materials from: http://www.llnl.gov/computing/tutorials/mpi/

MPI – Point to Point Communication

Order and Fairness

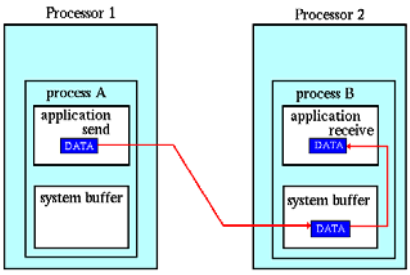
- Message Ordering
  - Messages do not overtake each other
    - If a sender sends two messages (Message 1 and Message 2) in succession to the same destination, and both match the same receive, the receive operation will receive Message 1 before Message 2.
    - If a receiver posts two receives (Receive 1 and Receive 2), in succession, and both match the same message, Receive 1 will receive the message before Receive 2.
  - Ordering is not thread-safe
    - If multiple threads participate in the communication, no order is guaranteed
- Fairness of Message Delivery
  - No fairness guarantee
    - Programmer responsible for preventing operation starvation
  - Example: task 0 sends a message to task 2. However, task 1 sends a competing message that matches task 2's receive. Only one of the sends will complete.

Material from: http://www.llnl.gov/computing/tutorials/mpi/

MPI – Point to Point Communication

Buffering when tasks are out of sync

- If a receive operation is not ready, sent data is buffered
  - On receiving side, sending side or both
- User can manage buffering memory on sending side



Path of a message buffered at the receiving process

Material from: http://www.llnl.gov/computing/tutorials/mpi/

MPI – Collective Communication

- All or None
  - Must involve **all** processes in the scope of the used communicator
  - User responsible to ensure all processes within a communicator participate in any collective operation
- Types of Collective Operations
  - Synchronization (barrier)
    - Processes wait until all members of the group reach the synchronization point
  - Data Movement
    - Broadcast, scatter/gather, all to all
  - Collective Computation (reductions)
    - One member of the group collects data from the other members and performs an operation (e.g., min, max, add, multiply, etc.) on that data

Material from: http://www.llnl.gov/computing/tutorials/mpi/

MPI – Collective Communication

Programming Considerations and Restrictions

- Collective operations are blocking
- Collective communication routines do not take message tag arguments
- Collective operations within subsets of processes
  - Partition the subsets into new groups
  - Attach the new groups to new communicators
- Can only be used with MPI predefined data types
  - Not with MPI Derived Data Types

Material from: http://www.llnl.gov/computing/tutorials/mpi/

MPI – Matrix Multiply (master task)

- Initialization

Common to both master and worker processes

```
#define NRA 15    // Number of rows in matrix A
#define NCA 25    // Number of columns in A
#define NCB 10    // Number of columns in B
#define TAG 0     // MPI communication tag
// Data structures
double A[NRA][NCA];// matrix A to be multiplied
double B[NCA][NCB];// matrix B to be multiplied
double C[NRA][NCB];// result matrix C
```

- Distribute data to workers

```
avgNumRows    = NRA/numWorkers;
remainingRows = NRA%numWorkers;
offset        = 0;
for (dest = 1; dest <= numWorkers; ++dest) {
  rows = (dest <= remainingRows) ? avgNumRows + 1 : avgNumRows;
  MPI_Send(&offset, 1, MPI_INT, dest, TAG, MPI_COMM_WORLD);
  MPI_Send(&rows, 1, MPI_INT, dest, TAG, MPI_COMM_WORLD);
  count = rows * NCA;
  // Send horizontal slice of A
  MPI_Send(&A[offset][0], count, MPI_DOUBLE, dest, TAG, MPI_COMM_WORLD);
  // Send matrix B
  count = NCA * NCB;
  MPI_Send(&B, count, MPI_DOUBLE, dest, TAG, MPI_COMM_WORLD);
  offset += rows;
}
```

- Wait for results from workers

```
for (i = 1; i <= numworkers; ++i)       {
  source = i;
  MPI_Recv(&offset, 1, MPI_INT, source, TAG, MPI_COMM_WORLD, &status);
  MPI_Recv(&rows, 1, MPI_INT, source, TAG, MPI_COMM_WORLD, &status);
  count = rows * NCB;
  MPI_Recv(&C[offset][0], count, MPI_DOUBLE, source, TAG,MPI_COMM_WORLD,&status);
}
```

MPI – Matrix Multiply (worker task)

- Receive data from master

```
source = 0;
MPI_Recv(&offset, 1, MPI_INT, source, TAG, MPI_COMM_WORLD, &status);
MPI_Recv(&rows,   1, MPI_INT, source, TAG, MPI_COMM_WORLD, &status);
// Receive horizontal slice of A
count = rows * NCA;
MPI_Recv(&A, count, MPI_DOUBLE, source, TAG, MPI_COMM_WORLD, &status);
// Receive matrix B
count = NCA * NCB;
MPI_Recv(&B, count, MPI_DOUBLE, source, TAG, MPI_COMM_WORLD, &status);
```

- Process data

```
// Compute the usual matrix multiplication on the slice of matrix A and matrix B
for (k = 0; k < NCB; ++k) {
  for (i = 0; i < rows; ++i) {
    C[i][k] = 0.0;
    for (j = 0; j < NCA; ++j) {
      C[i][k] += A[i][j] * B[j][k];
    }
  }
}
```

- Send results back to master

```
destination = 0;
MPI_Send(&offset, 1, MPI_INT, destination, TAG, MPI_COMM_WORLD);
MPI_Send(&rows, 1, MPI_INT, destination, TAG, MPI_COMM_WORLD);
count = rows * NCB;
// Send horizontal slice of result matrix C computed on this node
MPI_Send(&C, count, MPI_DOUBLE, destination, TAG, MPI_COMM_WORLD);
```

Table of Contents