Lecture 10: Introduction to OpenMP (Part 3)

Why Task Parallelism?

```
#include “omp.h”
/* traverse elements in the list */

Void traverse_list(List *L){
   Element *e;
#pragma omp parallel private(e)
   {
       for(e = L->first; e != NULL; e = e->next)
           #pragma omp single nowait
               do_work(e);
   }
}
```

- Poor performance

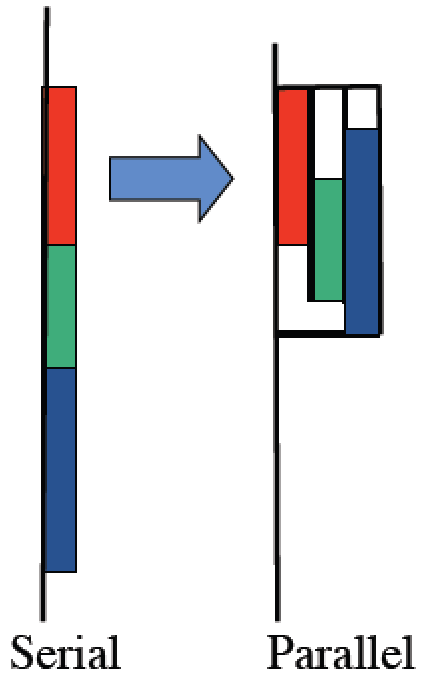
- Improved performance by sections
- Too many parallel regions
    - Extra synchronization
    - Not flexible

```
#include “omp.h”
/* traverse elements in the list */

Void traverse_tree(Tree *T){
#pragma omp parallel sections
   {
        #pragma omp section
           if(T->left)
               traverse_tree(T->left);
        #pragma omp section
           if(T->right)
               traverse_tree(T->right);
   }
   process(T);
}
```

OpenMP 3.0 and Tasks

- 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
- Why task?
  - The basic idea is to set up a task queue: when a thread encounters a task directive, it arranges for some thread to execute the associated block – at some time. The first thread can continue.



OpenMP 3.0 and Tasks

Tasks allow to parallelize irregular problems

- Unbounded loops
- Recursive algorithms
- Manger/work schemes
- …

A task has

- **Code** to execute
- **Data** environment (It owns its data)
- **Internal control variables**
- An assigned thread that executes the code and the data

Two activities: packaging and execution

- Each encountering thread packages a new instance of a task (code and data)
- Some thread in the team executes the task at some later time

- OpenMP has always had tasks, but they were not called “task”.
  - A thread encountering a parallel construct, e.g., “for”, packages up a set of implicit tasks, one per thread.
  - A team of threads is created.
  - Each thread is assigned to one of the tasks.
  - Barrier holds master thread till all implicit tasks are finished.
- OpenMP 3.0 adds a way to create a task explicitly for the team to execute.

Task Directive

```
#pragma omp task [clauses]
                        if( logical expression)
                        untied
                        shared (list)
                        private (list)
                        firstprivate (list)
                        default(shared | none)
     structured block
```

- Each encountering thread creates a task
  - Package code and data environment
  - Can be nested
    - Inside parallel regions
    - Inside other tasks
    - Inside worksharing
- An OpenMP barrier (implicit or explicit):

  All tasks created by any thread of the current team are guaranteed to be completed at barrier exit.
- Task barrier (taskwait):

  Encountering thread suspends until all child tasks it has generated are complete.

Fibonacci series:

f(1) = 1

f(2) = 1

f(n) = f(n-1) + f(n-2)

```
/* serial code to compute Fibonacci */
int fib(int n)
{
   int i, j;
   if(n < 2) return n;
   i = fib(n-1);
   j = fib(n-2);
   return (i+j);
}
int main(){
   int n = 8;
   printf(“fib(%d) = %d\n”, n, fib(n);
}
```

```
/* OpenMP code to compute Fibonacci */
#include <stdlib.h>
#include <stdio.h>
#include "omp.h"
static int fib(int);
int main(){
   int nthreads, tid;
   int n = 8;
   #pragma omp parallel num_threads(4) private(tid)
   {
      #pragma omp single
      {
         tid = omp_get_thread_num();
         printf("Hello world from (%d)\n", tid);
         printf("Fib(%d) = %d by %d\n", n, fib(n), tid);
      }
   } // all threads join master thread and terminates
}

Static int fib(int n){
   int i, j, id;
   if(n < 2)
      return n;
   #pragma omp task shared (i) private (id)
   {
      i = fib(n-1);
   }
   #pragma omp task shared (j) private (id)
   {
      j = fib(n-2);
    }
   return (i+j);
}
```

```
/* Example of pointer chasing  using task*/
Void process_list(elem_t *elem){
   #pragma omp parallel
   {
       #pragma omp single
       {
          while (ele != NULL) {
             #pragma omp task
             {
                 process(elem);
             }
             elem = elem->next;
          }
       }
   }
}
```

Elem is firstprivate by default

```
#include "omp.h"
/* traverse elements in the list */

Void traverse_list(List *L){
   Element *e;

    for(e = L->first; e != NULL; e = e->next)
           #pragma omp task
               do_work(e);
    #pragma omp taskwait

}
```

All tasks guaranteed to be completed here

```
/* Tree traverse using tasks*/

struct node{
   struct node *left, *right;
};
void traverse(struct node *p, int postorder){
   if(p->left != NULL)
       #pragma omp task
        traverse(p->left, postorder);
    if(p->right != NULL)
       #pragma omp task
        traverse(p->right, postorder);
    if(postorder){
       #pragma omp taskwait
    }
    process(p);
}
```

Task Data Scope

Data Scope Clauses

- shared (list)
- private (list)
- firstprivate (list)
- default (shared | none)

If no clause:

- Implicit rules apply: global variables are shared

Otherwise

- Firstprivate
- Shared attribute is lexically inherited

```
int a;
void foo(){
    int  b, c;
    #pragma omp parallel shared (c)
    {
         int d;
          # pragma omp task
          {
                 int e;
                 /*
                  a = shared
                  b = firstprivate
                  c = shared
                  d = firstprivate
                  e = private
                 */
     }
}
```

Task Synchronization

Barriers (implicit or explicit)

- All tasks created by any thread of the current team are guaranteed to be completed at barrier exit

Task Barrier

```
#pragma omp taskwait
```

- Encountering task suspends until child tasks complete

Task Execution Model

- Tasks are executed by a thread of the team
  - Can be executed immediately by the same thread that creates it
- Parallel regions in 3.0 create tasks
  - One implicit task is created for each thread
- Threads can suspend the execution of a task and start/resume another

```
#include “omp.h”
/* traverse elements in the list */
List *L;
…
#pragma omp parallel
 traverse_list(L);
```

Multiple traversals of the same list

```
#include “omp.h”
/* traverse elements in the list */
List *L;
…
#pragma omp parallel
#pragma omp single
 traverse_list(L);
```

Single traversal:

- One thread enters single and creates all tasks
- All the team cooperates executing them

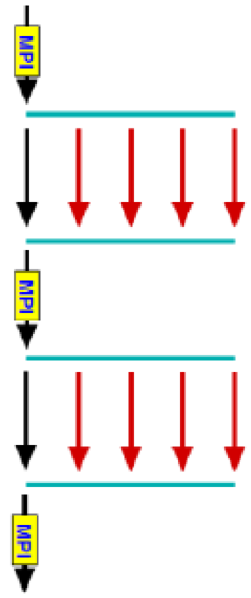
```
#include “omp.h”
/* traverse elements in the list */
List L[N];
…
#pragma omp parallel for
For (i = 0; i < N; i++)
 traverse_list(L[i]);
```

Multiple traversals:

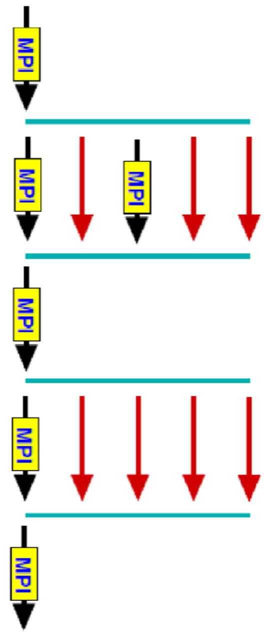
- Multiple threads create tasks
- All the team cooperates executing them

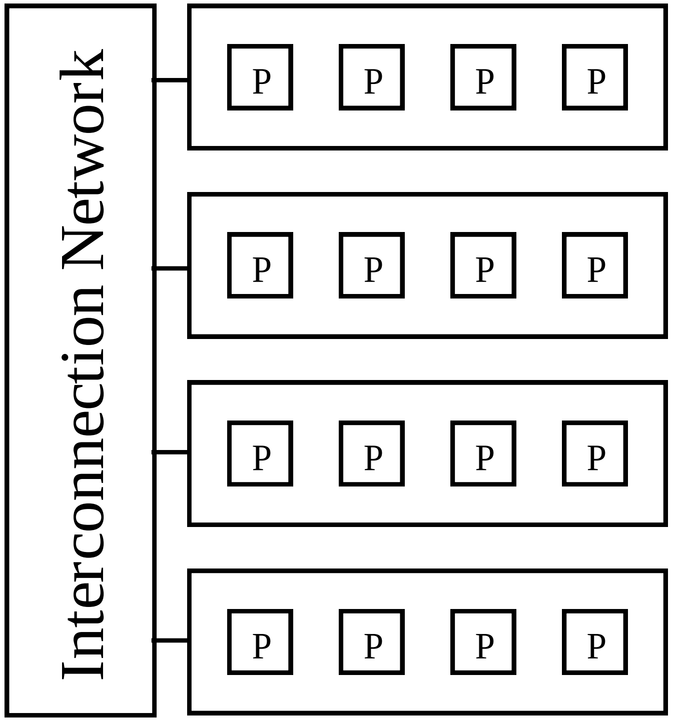
Hybrid MPI/OpenMP

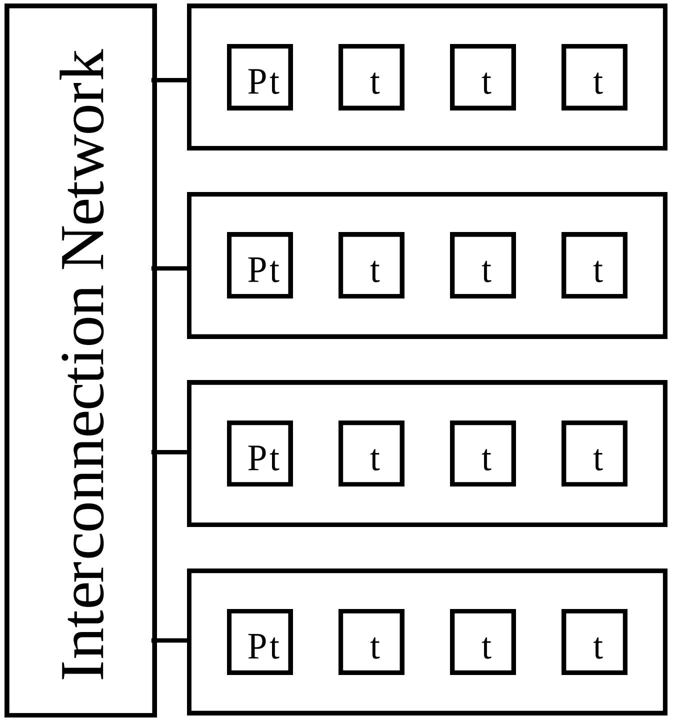
- **Vector mode:** MPI is called only outside OpenMP parallel regions.



- **Task mode:** One or more threads in the parallel region are dedicated to special tasks, like doing communication in the background.







C+MPI

C+MPI+OpenMP

Basic Hybrid Framework

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

#define _NUM_THREADS 4

/*  Each MPI process spawns a distinct OpenMP
 *  master thread; so limit the number of MPI
 *  processes to one per node
 */

int main (int argc, char *argv[]) {
  int p,my_rank;

  /* set number of threads to spawn */
  omp_set_num_threads(_NUM_THREADS);

  /* initialize MPI stuff */
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD,&p);
  MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);

  /* the following is a parallel OpenMP
   * executed by each MPI process
   */
  int c;
  #pragma omp parallel reduction(+:c)
  {
    c = omp_get_num_threads();
  }

  /* expect a number to get printed for each MPI process */
  printf("%d\n",c);
  /* finalize MPI */
  MPI_Finalize();
  return 0;
}
```

Compileing: mpicc –fopenmp test.cc

Concept 1: ROOT MPI Process Controls Communication

- Map one MPI process to one SMP node.
- Each MPI process fork a fixed number of threads.
- Communication among MPI process is handled by main MPI process only.

```
…
#pragma omp master
{
    if(0== my_rank)
        // some MPI call as root process
    else
        // some MPI call as non-root process
} // end of omp master
```

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

#define _NUM_THREADS 4

int main (int argc, char *argv[]) {
  int p,my_rank;

  /* set number of threads to spawn */
  omp_set_num_threads(_NUM_THREADS);

  /* initialize MPI stuff */
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD,&p);
  MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);

  /* the following is a parallel OpenMP
   * executed by each MPI process
   */
  #pragma omp parallel
  {
    #pragma omp master
    {
      if ( 0 == my_rank)
        // some MPI_ call as ROOT process
      else
        // some MPI_ call as non-ROOT process
    }
  }

  /* expect a number to get printed for each MPI process */
  printf("%d\n",c);
  /* finalize MPI */
  MPI_Finalize();
  return 0;
}
```

Concept 2: Master OpenMP Thread Controls Communication

- Each MPI process uses its own OpenMP master thread to communicate.
- Need to take more care to ensure efficient communications.

```
…
#pragma omp master
{
    some MPI call as an MPI process
} // end of omp master
```

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

#define _NUM_THREADS 4

int main (int argc, char *argv[]) {
  int p,my_rank;

  /* set number of threads to spawn */
  omp_set_num_threads(_NUM_THREADS);

  /* initialize MPI stuff */
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD,&p);
  MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);

  /* the following is a parallel OpenMP
   * executed by each MPI process
   */
  #pragma omp parallel
  {
    #pragma omp master
    {
      // some MPI_ call as an MPI process
    }
  }

  /* expect a number to get printed for each MPI process */
  printf("%d\n",c);
  /* finalize MPI */
  MPI_Finalize();
  return 0;
}
```

Concept 3: All OpenMP Threads May Use MPI Calls

- This is by far the most flexible communication scheme.
- Great care must be taken to account for explicitly which thread of which MPI process communicates.
- Requires an addressing scheme that denotes which MPI process participates in communication and which thread of MPI process is involved, e.g., <my_rank, omp_thread_id>.
- Neither MPI nor OpenMP have built-in facilities for tracking communication.
- Critical sections may be used for some level of control.

```
…
#pragma omp critical
{
    some MPI call as an MPI process
} // end of omp critical
```

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

#define _NUM_THREADS 4

int main (int argc, char *argv[]) {
  int p,my_rank;

  /* set number of threads to spawn */
  omp_set_num_threads(_NUM_THREADS);

  /* initialize MPI stuff */
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD,&p);
  MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);

  /* the following is a parallel OpenMP
   * executed by each MPI process
   */
  #pragma omp parallel
  {
    #pragma omp critical /* not required */
    {
      // some MPI_ call as an MPI process
    }
  }

  /* expect a number to get printed for each MPI process */
  printf("%d\n",c);
  /* finalize MPI */
  MPI_Finalize();
  return 0;
}
```

Conjugate Gradient

- Algorithm
  - Start with MPI program
  - MPI_Send/Recv for communication
  - OpenMP “for” directive for matrix-vector multiplication

```
Init.: x(0) =0, d(0) = 0, g(0) = -b;
Step 1. Compute the gradient: g(t) =Ax(t-1)-b
Step 2. Compute the direction vector:
      d(t) = -g(t)+(g(t)^Tg(t))/(g(t-1)^Tg(t-1))d(t-1)
Step 3. Compute the step size:
       s(t) = -(d(t)^Td(t))/(d(t)^TAd(t));
Step 4. Compute the new approximation of x:
       x(t) = x(t-1) + s(t) d(t).
```

```
#include <stdlib.h>
#include <stdio.h>
#include “MyMPI.h”
int main(int argc, char *argv[]){
   double **a, *astorage, *b, *x;
   int p, id, m, n, nl;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD, &p);
   MPI_Comm_rank(MPI_COMM_WORLD, &id);
read_block_row_matrix(id,p,argv[1],(void*)(&a),(void*)(&astorage),MPI_DOUBLE,&m,&n);
   nl = read_replicated_vector(id,p,argv[2],(void**)(&b),MPI_DOUBLE);
   if((m!=n) ||(n != nl)) {
       printf(“Incompatible dimensions %d %d time %d\n”, m,n,nl);
   }
   else{
       x = (double*)malloc(n*sizeof(double));
       cg(p,id,a,b,x,n);
       print_replicated_vector(id,p,x,MPI_DOUBLE,n);
   }
   MPI_Finalize();
}
```

```
#define EPSILON 1.0e-10
Double *piece;
cg(int p, int id, double **a, double *b, double *x, int n){
   int  i, it;
   double   *d, *g, denom1, denom2, num1, num2, s, *tmpvec;
    d = (double*)malloc(n*sizeof(double));
    g =  (double*)malloc(n*sizeof(double));
    tmpvec = (double*)malloc(n*sizeof(double));
    piece = (double*)malloc(BLOCK_SIZE(id,p,n)*sizeof(double));
     for(i=0; i<n; i++){
          d[i] = x[i] = 0.0;
          g[i] = -b[i];
     }
      for(it=0; it<n; it++){
           denom1 = dot_product(g,g,n);
           matrix_vector_product(id,p,n,a,x,g);
           for(i=0;i<n;i++)  g[i]-=b[i];
           num1 = dot_product(g,g,n);
           if(num1<EPSILON) break;
           for(i=0;i<n;i++) d[i]=-g[i]+(num1/denom1)*d[i];
           num2 = dot_product(d,g,n);
           matrix_vector_product(id,p,n,a,d,tmpvec);
           denom2=dot_product(d,tmpvec,n);
           s=-num2/denom2;
           for(i=0;i<n;i++) x[i] += s*d[i];
      }
}
```

```
double dot_product(double *a, double *b, int n)
{
    int i;
    double answer=0.0;
     for(i=0; i<n;i++)
         answer+=a[i]*b[i];
     return answer;
}
double matrix_vector_product(int id, int p, int n, double **a, double *b, double *c){
    int i, j;
    double tmp;
    #pragma omp parallel for private (I,j,tmp)
    for(i=0; i<BLOCK_SIZE(id,p,n);i++){
        tmp=0.0;
         for(j=0;j<n;j++)
             tmp+=a[i][j]*b[j];
         piece[i] = tmp;
    }
    new_replicate_block_vector(id,p,piece,n, c, MPI_DOUBLE);
}
void new_replicate_block_vector(int id, int p, double  *piece, int n, double *c, MPI_Datatype  dtype)
{
   int *cnt, *disp;
   create_mixed_xfer_arrays(id,p,n,&cnt,&disp);
   MPI_Allgatherv(piece,cnt[id], dtype, c, cnt, disp, dtype, MPI_COMM_WORLD);
}
```

Steady-State Heat Distribution

Solve $u_{xx} + u_{yy} = f(x, y),\ \ 0 \leq x \leq a, 0 \leq y \leq b$

With $u(x, 0) = G_1(x), u(x, b) = G_2(x),\ \ \ 0 \leq x \leq a$

$\quad u(0, y) = G_3(y), u(a, y) = G_4(y), \qquad\quad 0 \leq y \leq b$

- Use row-decomposition.

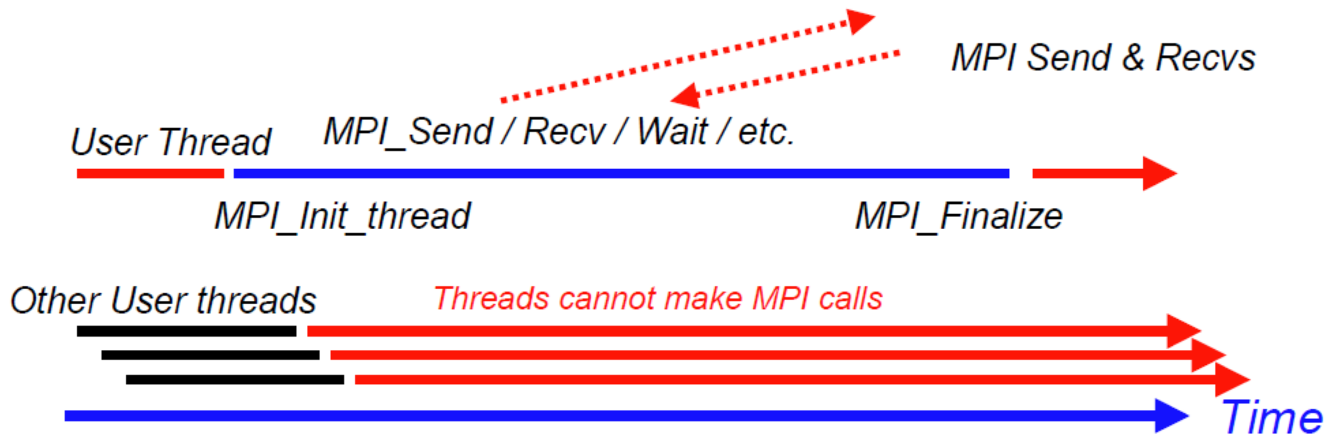
```
int find_steady_state(int p, int id, iny my_rows, double **u, double **w)
{
    double diff, global_diff, tdiff; int its;
    MPI_Status   status; int i,j;
    its = 0;
    for(;;) {
        if(id>0) MPI_Send(u[1], N, MPI_DOUBLE, id-1,0,MPI_COMM_WORLD);
        if(id < p-1) {
            MPI_Send(u[my_rows-2],N,MPI_DOUBLE,id+1,0,MPI_COMM_WORLD);
            MPI_Recv(u[my_rows-1],N,MPI_DOUBLE,id+1,0,MPI_COMM_WORLD,&status);
        }
        if(id>0) MPI_Recv(u[0],N,MPI_DOUBLE,id-1,0,MPI_COMM_WORLD,&status);
        diff = 0.0;
#pragma omp parallel private (I,j,tdiff)
        {
            tdiff = 0.0;
             #pragma omp for
             for(i=1;i<my_rows-1;i++)
                 for(j=1;j<N-1;j++){
                     w[i][j]=(u[i-1][j]+u[i+1][j]+u[i][j-1]+u[i][j+1])/4.0;
                     if(fabs(w[i][j]-u[i][j]) >tdiff)  tdiff = fabs(w[i][j]-u[i][j]);
                  }
             #pragma omp for nowait
              for(i=1;i<my_rows-1;i++)
                  for(j=1;j<N-1;j++)
                      u[i][j] = w[i][j];
             #pragma omp critical
            if(tdiff > diff) diff = tdiff;
        }
        MPI_Allreduce(&diff,&global_diff,1,MPI_DOUBLE,MPI_MAX,MPI_COMM_WORLD);
        if(global_diff <= EPSILON) break;
        its++;
    }
}
```

OpenMP multithreading in MPI

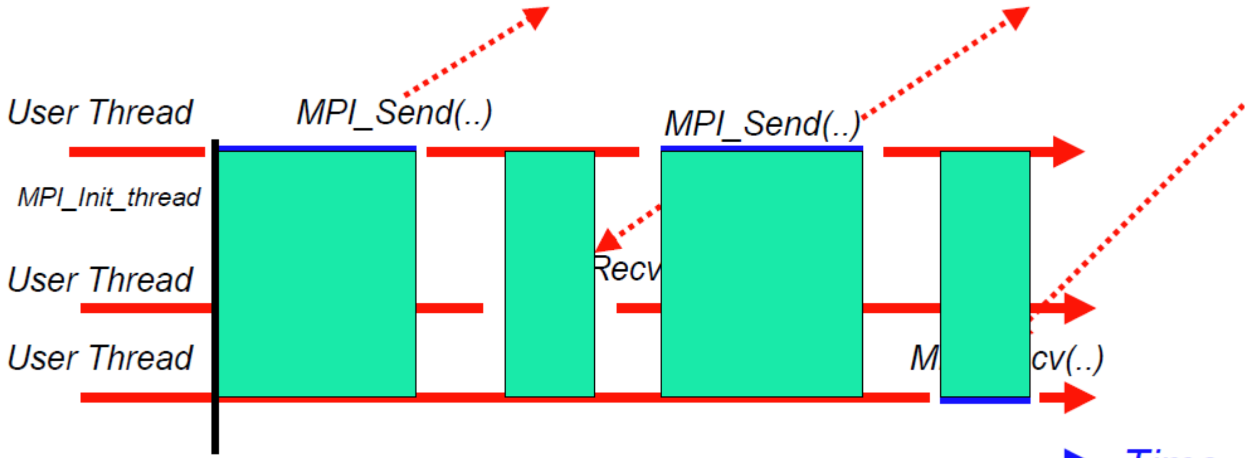
- MPI-2 specification
  - Does not mandate thread support
  - Does define what a "thread compliant MPI" should do
  - 4 levels of thread support
    - MPI_THREAD_SINGLE: There is no OpenMP multithreading in the program.
    - MPI_THREAD_FUNNELED: All of the MPI calls are made by the master thread.

      This will happen if all MPI calls are outside OpenMP parallel regions or are in master regions.

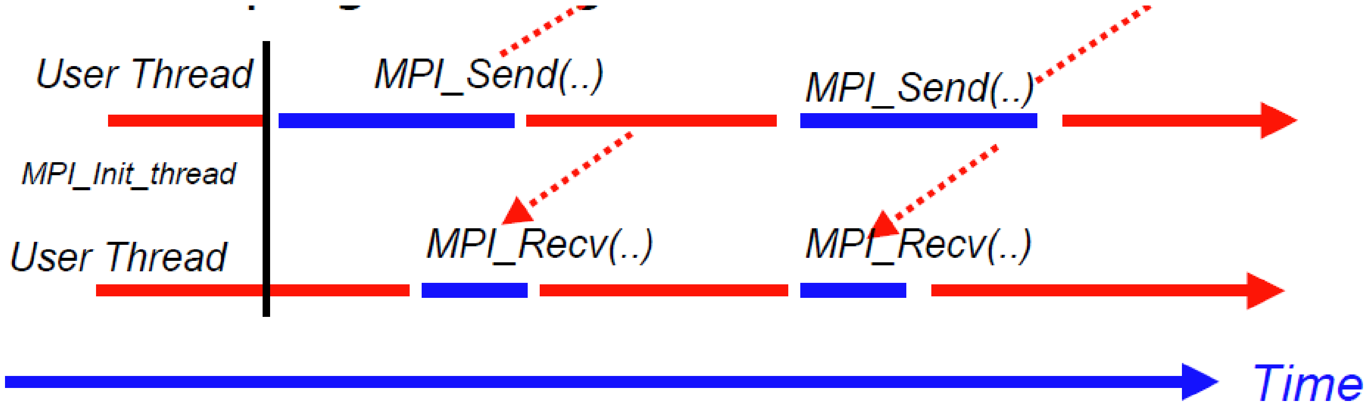
      A thread can determine whether it is the master thread by calling MPI_Is_thread_main



- MPI_THREAD_SERIALIZED: Multiple threads make MPI calls, but only one at a time.



- MPI_THREAD_MULTIPLE: Any thread may make MPI calls at any time.



- Threaded MPI Initialization

Instead of starting MPI by MPI_Init,

int MPI_Init_thread(int *argc, char ***argv, int required, int *provided)

required: the desired level of thread support.

provided: the actual level of thread support provided by the system.

Thread support at levels MPI_THREAD_FUNNELED or higher allows potential overlap of communication and computation.

http://www.mpi-forum.org/docs/mpi-20-html/node165.htm

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

int main(int argc, char *argv[])
{
   int rank, omp_rank, mpisupport;

   MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &mpisupport);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

#pragma omp parallel private(omp_rank)
{
   omp_rank = omp_get_thread_num();
   printf("Hello. This is process %d, thread %d\n",
        rank, omp_rank);
}
   MPI_Finalize();
}
```

References:

– http://bisqwit.iki.fi/story/howto/openmp/
– http://openmp.org/mp-documents/omp-hands-on-SC08.pdf
– https://computing.llnl.gov/tutorials/openMP/
– http://www.mosaic.ethz.ch/education/Lectures/hpc
– R. van der Pas. An Overview of OpenMP
– B. Chapman, G. Jost and R. van der Pas. Using OpenMP: Portable Shared Memory Parallel Programming. The MIT Press, Cambridge, Massachusetts, London, England
– B. Estrade, Hybrid Programming with MPI and OpenMP