HPCSE - II

«MPI Programming Model: Master-Worker »

Panos Hadjidoukas

Outline

- Hands-on: Task queue in OpenMP
- MPI recap: probing messages
- Hands-on: Task queue in MPI (master-worker)
- Hands-on: MPI server threads and OpenMP tasks

I. Sequential Code

```
void task(double *x, double *y) {
      *y = x[0]+x[1];
}

int main(int argc, char *argv[]) {
      double result[100];

      for (int i=0; i<100; i++) {
          double d[2];
          d[0] = drand48();
          d[1] = drand48();
          task(d, &result[i]);
      }

    /* print results */
    return 0;
}
```

OpenMP Code

```
void task(double *x, double *y) { *y = x[0] + x[1]; }

int main(int argc, char *argv[]) {
      double result[100];

      #pragma omp parallel
      #pragma omp single nowait
      {
          for (int i=0; i<100; i++) {
              double d[2];
              d[0] = drand48();
              d[1] = drand48();
              #pragma omp task firstprivate(d, i) shared(result)
              {
                task(d, &result[i]);
              }
          }
          #pragma omp taskwait
          /* print results */
      }
    return 0;
}
```

OpenMP + Task Queue (1/4)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <omp.h>

void master();
void worker();

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

 #pragma omp parallel
 {
     myrank = omp_get_thread_num();

     if (myrank == 0) {
         master();
     } else {
         worker();
     }
 }
 return 0;

}
```

OpenMP + Task Queue (2/4)

```
typedef struct work_s
{
 int pos;
 double x[2];
 double y;
} work_t;

work_t *workarray;
int work_id, nworks;

void init_work(int n)
{
 int i;

 #pragma omp critical
 {
 workarray = malloc(n*sizeof(work_t));
 work_id = 0;
 nworks = n;
 srand48(1);
 for (i = 0; i < n; i++) {
     workarray[i].pos = i;
     workarray[i].x[0] = drand48();
     workarray[i].x[1] = drand48();
 }
 }
}
```

OpenMP + Task Queue (3/4)

```
work_t *get_next_work_request()
{
 int local_work_id;

 #pragma omp critical
 {
     local_work_id = work_id++;
 }

 if (local_work_id >= nworks) return NULL;

 work_t *w = &workarray[local_work_id];

 return w;
}

void print_work()
{
 int i;
 int n = nworks;

 for (i = 0; i < n; i++) {
     printf("work[%d]: %f %f -> %f\n", i, workarray[i].x[0], workarray[i].x[1],
workarray[i].y);
 }

}
```

OpenMP + Task Queue (4/4)

```
void master()
{
 init_work(10);

 worker();

 /* Print the results */
 print_work();
}

void worker()
{
 work_t               *work;

 #pragma omp barrier

 work = get_next_work_request();
 while (work != NULL)
 {
     work->y  = work->x[0] + work->x[1];
     sleep(1);
     printf("result of work %d on %d : %f\n", work->pos, omp_get_thread_num(), work->y);
     work = get_next_work_request();
 }

 #pragma omp barrier
}
```

II. Probing for messages

- Instead of directly receiving you can probe whether a message has arrived:

```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
// wait for a matching message to arrive

int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)
// check if a message has arrived.
// flag is nonzero if there is a message waiting

int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int* count)
// gets the number of elements in the message waiting to be received
```

- The `MPI_Status` object can be queried for information about the message:

```
MPI_Status status;
int count;

// wait for a message
MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, &status);
std::cout << "A message is waiting from " << status->MPI_SOURCE
          << "with tag "                  << status->MPI_TAG;

// get the element count
MPI_Get_count(&status, MPI_INT, &count)
std::cout << "and assuming it contains ints there are " << count << "elements";
```

MPI_Status and MPI_Recv

- Looking into mpi.h:

```
typedef struct MPI_Status {
    int count_lo;
    int count_hi_and_cancelled;
    int MPI_SOURCE;
    int MPI_TAG;
    int MPI_ERROR;
} MPI_Status;
```

- Example

```
int tag=99;

if(num==0) {
  MPI_Send(ds, N, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
}
else {
 MPI_Recv(dr, N, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);

 int count;
 MPI_Get_count(&status, MPI_CHAR, &count);

 printf("status.MPI_SOURCE = %d, status.MPI_TAG=%d, count = %d\n",
        status.MPI_SOURCE, status.MPI_TAG, count);
}
```

- Output

```
$ mpicc -o status0 status0.cpp
$ mpirun -l -n 2 ./status0 1024
[1] status.MPI_SOURCE = 0, status.MPI_TAG = 99, count = 1024
```

MPI_Status and MPI_Recv

- Consider the following code (same as before):

```
 int tag=99;

 if(num==0) {
   MPI_Send(ds, N, MPI_CHAR, 1,tag, MPI_COMM_WORLD);
 }
 else {
  MPI_Recv(dr, N, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
  int count;

  MPI_Get_count(&status, MPI_CHAR, &count);
  printf("status.MPI_SOURCE = %d, status.MPI_TAG=%d, count = %d\n",
          status.MPI_SOURCE, status.MPI_TAG, count);
}
```

- What will happen if:
  1. Rank 0 sends less data?

     (or, equivalently, rank 1 asks for more data)?
  2. Rank 0 sends more data?

     (or, equivalently, rank 1 asks for less data?)

MPI_Status and MPI_Recv

- Case 1: rank 0 sends N/2, rank 1 asks for N

```
  int tag=99;

  if(num==0) {
    MPI_Send(ds, N/2, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
 }
  else {
   MPI_Recv(dr, N, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
   int count;
   MPI_Get_count(&status, MPI_CHAR, &count);
   printf("status.MPI_SOURCE = %d, status.MPI_TAG=%d, count = %d\n",
           status.MPI_SOURCE, status.MPI_TAG, count);
 }
```

```
$ mpirun -l -n 2 ./status0_less 1024
[1] status.MPI_SOURCE = 0, status.MPI_TAG = 99, count = 512
```

- Case 2: rank 0 sends N, rank 1 asks for N/2

```
$ mpirun -l -n 2 ./status0_more 1024
[1] status.MPI_SOURCE = 0, status.MPI_TAG = 99, count = 512
[1] Fatal error in MPI_Recv: Message truncated, error stack:
[1] MPI_Recv(200).....................: MPI_Recv(buf=0x7fe9a8002600, count=512,
MPI_CHAR, src=0, tag=99, MPI_COMM_WORLD, status=0x7fff55005968) failed
[1] MPIDI_CH3U_Receive_data_found(131): Message from rank 0 and tag 99 truncated; 1024
bytes received but buffer size is 512
```

How can we deal with incoming messages of unknown size?

Answer: MPI_Status and MPI_Probe

- Solution 1: Probe, get count, and then receive

status1.c

```
 int tag=99;

 if(num==0) {
   MPI_Send(ds, N/2, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
 }
 else {
  MPI_Probe(0, tag, MPI_COMM_WORLD, &status);
  int count;
  MPI_Get_count(&status, MPI_CHAR, &count);
  printf("status.MPI_SOURCE = %d, status.MPI_TAG=%d, count = %d\n",
          status.MPI_SOURCE, status.MPI_TAG, count);
  MPI_Recv(dr, count, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
}
```

- Solution 2: Probe, get source+tag+count, and then receive

status2.c

```
int tag=99;

if(num==0) {
  MPI_Send(ds, N/2, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
}
else {
  MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
  int count;
  MPI_Get_count(&status,MPI_CHAR,&count);
  printf("status.MPI_SOURCE = %d, status.MPI_TAG = %d, count = %d\n",
              status.MPI_SOURCE, status.MPI_TAG, count);
  MPI_Recv(dr,count, MPI_CHAR, status.MPI_SOURCE, status.MPI_TAG, MPI_COMM_WORLD,&status);
}
```

increased flexibility

Example: Sending data of dynamic size

- In the following example, rank 0 allocates, initializes and finally sends N elements to rank 1
  - N is chosen randomly at runtime
  - The max size is predefined (`MAX_N`)

dynamic1.c

```
int MAX_N = 100; // max number of elements
int N;

if (rank == 0) {
    N = lrand48()%MAX_N;
    double *x = (double *)calloc(1, N*sizeof(double));
    init_data(x, N);
    MPI_Send(x, N, MPI_DOUBLE, 1, 123, MPI_COMM_WORLD);
}
if (rank == 1) {
   double *y = (double *)calloc(1, MAX_N*sizeof(double));
    MPI_Recv(y, MAX_N, MPI_DOUBLE, 0, 123, MPI_COMM_WORLD, &status);
    MPI_Get_count(&status, MPI_DOUBLE, &N);
    print_data(y, N);
}
```

Rank 1 allocates and asks for MAX_N elements.
Can we avoid this?

Based on: http://mpitutorial.com/tutorials/dynamic-receiving-with-mpi-probe-and-mpi-status/

Using MPI_Probe

- Get the number of elements, allocate and receive

dynamic2.c

```
int MAX_N = 100;
int N;

if (rank == 0) {
    N = lrand48()%MAX_N;
    double *x = (double *)calloc(1, N*sizeof(double));
    init_data(x, N);
    MPI_Send(x, N, MPI_DOUBLE, 1, 123, MPI_COMM_WORLD);
}
if (rank == 1) {
    MPI_Probe(0, 123, MPI_COMM_WORLD, &status);
    MPI_Get_count(&status, MPI_DOUBLE, &N);
    double *y = (double *)calloc(1, N*sizeof(double));
    MPI_Recv(y, N, MPI_DOUBLE, 0, 123, MPI_COMM_WORLD, &status);
    print_data(y, N);
}
```

How can we do this without using MPI_Probe?

Sending Count (# Elements)

- Use an extra message to send first the size and then the elements

dynamic3.c

```
int MAX_N = 100;
int N;

if (rank == 0) {
    N = lrand48()%MAX_N;
    double *x = (double *)calloc(1, N*sizeof(double));
    init_data(x, N);
    MPI_Send(&N, 1, MPI_INT, 1, 123, MPI_COMM_WORLD);
    MPI_Send(x, N, MPI_DOUBLE, 1, 124, MPI_COMM_WORLD);
}
if (rank == 1) {
     MPI_Recv(&N, 1, MPI_INT, 0, 123, MPI_COMM_WORLD, &status);
    double *y = (double *)calloc(1, N*sizeof(double));
    MPI_Recv(y, N, MPI_DOUBLE, 0, 124, MPI_COMM_WORLD, &status);
    print_data(y, N);
}
```

What if we do not know the data type?

Dynamic Size and Datatype

- Send size and MPI datatype

dynamic4.c

```
if (rank == 0) {
    N = lrand48()%MAX_N;
    double *x = (double *)calloc(1, N*sizeof(double));
    init_data(x, N);
    MPI_Send(&N, 1, MPI_INT, 1, 123, MPI_COMM_WORLD);

    long type = (long) MPI_DOUBLE;
    MPI_Send(&type, sizeof(long), MPI_BYTE, 1, 124, MPI_COMM_WORLD);

    MPI_Send(x, N, MPI_DOUBLE, 1, 125, MPI_COMM_WORLD);
}
if (rank == 1) {
    MPI_Recv(&N, 1, MPI_INT, 0, 123, MPI_COMM_WORLD, &status);
    double *y = (double *)calloc(1, N*sizeof(double));
    long type;
    MPI_Recv(&type, sizeof(long), MPI_BYTE, 0, 124, MPI_COMM_WORLD, &status);
    if (type == MPI_DOUBLE) {
        MPI_Recv(y, N, MPI_DOUBLE, 0, 125, MPI_COMM_WORLD, &status);
    } else {
        MPI_Abort(MPI_COMM_WORLD, 911); // from Deino MPI
    }
    print_data(y, N);
}
```

This code here is not safe!

The code above will work in many cases but it is not considered valid / safe.
Rule: "You must never send MPI datatypes as messages"
This should not be enough for us, we need to know why!

Dynamic Size and Datatype

- What is an MPI datatype? Looking again into `mpi.h`
  - MPICH
    - `typedef int MPI_Datatype;`
    - `MPI_DOUBLE` is a predefined constant integer
    - It has the same value on all processes
  - OpenMPI
    - `typedef struct ompi_datatype_t *MPI_Datatype;`
    - `MPI_DOUBLE` is a pointer to a memory location
    - It depends on the memory layout of the process and thus it might be different for each process (rank) *[see code below]*

```
long type  = (long) MPI_DOUBLE;
int main(int argc, char** argv)
{
    MPI_Init(&argc, &argv);

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    printf("rank %d: type = %ld, &type = %p\n",
       rank, type, &type);

    MPI_Finalize();
    return 0;
}
```

Processes will not always print the same values
- True: MacOS + MPICH, Euler + OpenMPI
- False: Euler + MVAPICH2 (= MPICH)

```
[chatzidp@eu-login-06 ~]$ mpirun -n 2 ./hello_datatype
I am rank 0 of 2: type = 47390899143520, &type = 0x601060
I am rank 1 of 2: type = 47347376649056, &type = 0x601060
```

Dynamic Size and Datatype

- Solution: use your own mapping for MPI datatypes, e.g.:

```
const long MY_MPI_DOUBLE = 100;

if (rank == 0) {
    N = lrand48()%MAX_N;
    double *x = (double *)calloc(1, N*sizeof(double));
    init_data(x, N);
    MPI_Send(&N, 1, MPI_INT, 1, 123, MPI_COMM_WORLD);

    long type = (long) MY_MPI_DOUBLE;
    MPI_Send(&type, 1, MPI_LONG, 1, 124, MPI_COMM_WORLD);

    MPI_Send(x, N, MPI_DOUBLE, 1, 125, MPI_COMM_WORLD);
}
if (rank == 1) {
    MPI_Recv(&N, 1, MPI_INT, 0, 123, MPI_COMM_WORLD, &status);
    double *y = (double *)calloc(1, N*sizeof(double));
    long type;
    MPI_Recv(&type, 1, MPI_LONG, 0, 124, MPI_COMM_WORLD, &status);
    if (type == MY_MPI_DOUBLE) {
        MPI_Recv(y, N, MPI_DOUBLE, 0, 125, MPI_COMM_WORLD, &status);
    } else {
        MPI_Abort(MPI_COMM_WORLD, 911); // from Deino MPI
    }
    print_data(y, N);
}
```

Take-home message:

Your code must not be based on assumptions about the underlying software (and hardware)

III. MPI Master-Worker (1/7)

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

#define WORKTAG        1
#define DIETAG         2

void master();
void worker();

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

MPI_Init(&argc, &argv);                  /* initialize MPI */
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);      /* process rank, 0 to N-1 */
if (myrank == 0) {
    master();
} else {
    worker();
}
MPI_Finalize();        /* cleanup MPI */

return 0;
}
```

MPI Master-Worker (2/7)

```
/* A very naive work queue */

typedef struct work_s
{
int pos;
double x[2];
double y;
} work_t;

work_t *workarray;
int work_id, nworks;

void init_work(int n)
{
  int i;

  workarray = malloc(n*sizeof(work_t));
  work_id = 0;
  nworks = n;
  for (i = 0; i < n; i++) {
    workarray[i].pos = i;
    workarray[i].x[0] = drand48();
    workarray[i].x[1] = drand48();
  }
}
```

MPI Master-Worker (3/7)

```
work_t *get_next_work_request()
{
  if (work_id >= nworks) return NULL;

  work_t *w = &workarray[work_id];
  work_id++;

  return w;
}

void print_work()
{
  int i;
  int n = nworks;

  for (i = 0; i < n; i++) {
    printf("work[%d]: %f %f -> %f\n", i, workarray[i].x[0],
           workarray[i].x[1], workarray[i].y);
  }
}
```

MPI Master-Worker (4/7)

```
typedef struct result_s
{
  int pos;
  double y;
} result_t;


void worker()
{
  int rank;
  result_t            result;
  work_t              work;
  MPI_Status          status;

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  for (;;) {
    MPI_Recv(&work, sizeof(work_t), MPI_CHAR, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

    /* Check the tag of the received message */
    if (status.MPI_TAG == DIETAG) {
        return;
    }

    result.pos = work.pos;
    result.y  = work.x[0] + work.x[1];
    sleep(1);

    printf("result of work %d on %d : %f\n", result.pos, rank, result.y);
    MPI_Send(&result, sizeof(result_t), MPI_CHAR, 0, 0, MPI_COMM_WORLD);
  }
}
```

MPI Master-Worker (5/7)

```
void master()
{
int  ntasks, rank;
double       result;
MPI_Status     status;
MPI_Comm_size(MPI_COMM_WORLD, &ntasks);          /* #processes in application */

init_work(10);

/*
 * Seed the workers.
 */

work_t *work;
for (rank = 1; rank < ntasks; ++rank) {
     work = get_next_work_request();

     printf("sending work %d\n", work->pos);

     MPI_Send(work,         /* message buffer */
     sizeof(work_t),        /* one data item */
     MPI_CHAR,          /* data item is a struct */
     rank,              /* destination process rank */
     WORKTAG,      /* user chosen message tag */
     MPI_COMM_WORLD);  /* always use this */
}
```

MPI Master-Worker (6/7)

```
/*
 * Receive a result from any worker and dispatch a new work
 * until work requests have been exhausted.
 */

result_t res;
work = get_next_work_request();
while (work != NULL) {
     MPI_Recv(&res,              /* message buffer */
     sizeof(result_t),      /* one data item .. */
     MPI_CHAR,              /* of a struct  */
     MPI_ANY_SOURCE,             /* receive from any sender */
     MPI_ANY_TAG,           /* any type of message */
     MPI_COMM_WORLD,             /* always use this */
     &status);              /* received message info */

     workarray[res.pos].y = res.y;

     printf("sending work %d\n", work->pos);
     MPI_Send(work, sizeof(work_t), MPI_CHAR, status.MPI_SOURCE, WORKTAG, MPI_COMM_WORLD);
     work = get_next_work_request();
}
```

MPI Master-Worker (7/7)

```
/*
 * Receive results for pending work requests.
 */
for (rank = 1; rank < ntasks; ++rank) {
  MPI_Recv(&res,sizeof(result_t),MPI_CHAR,MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
  workarray[res.pos].y = res.y;
}

/*
 * Tell all the workers to exit.
 */
for (rank = 1; rank < ntasks; ++rank) {
    MPI_Send(0, 0, MPI_CHAR, rank, DIETAG, MPI_COMM_WORLD);
}


/* Print the results */
print_work();
}
```

IV. Exam Question

Consider the following MPI code:

```
1  // int myrank: MPI rank
2  // int nprocs: number of MPI processes
3  // double A: array of size nprocs-1
4
5  if (myrank < nprocs-1)
6  {
7    double a = prepare_data();     // variable execution time
8    MPI_Send(&a, 1, MPI_DOUBLE, size-1, myrank+100, MPI_COMM_WORLD);
9  }
10 else
11 {
12    for (int i = 0; i < nprocs-1; i++)
13    {
14       MPI_Status status;
15       MPI_Recv(&A[i], 1, MPI_DOUBLE, i, i+100, MPI_COMM_WORLD, &status);
16       process(A[i]);
17   }
18 }
```

a) Parallelize the for loop executed by the last rank.

Possible but inefficient solution

```
if (myrank < nprocs-1)
{
   double a = myrank; // prepare_data();
   MPI_Send(&a, 1, MPI_DOUBLE, nprocs-1, myrank+100, MPI_COMM_WORLD);
}
else
{
  double A[nprocs-1];

  #pragma omp parallel
  {
  #pragma omp master
  {
    for (int i = 0; i < nprocs-1; i++)
    {
      #pragma omp task firstprivate(i) shared(A)
      {
        MPI_Status status;
        MPI_Recv(&A[i], 1, MPI_DOUBLE, i, i+100, MPI_COMM_WORLD, &status);
        process(A[i]);
      }
    }
  } // master
  } // parallel

}
```

Not efficient if
incoming messages < # available workers

Possible solution

```
if (myrank < nprocs-1)
{
   double a = myrank; // prepare_data();
   MPI_Send(&a, 1, MPI_DOUBLE, nprocs-1, myrank+100, MPI_COMM_WORLD);
}
else
{
  double A[nprocs-1];

  #pragma omp parallel
  {
  #pragma omp master
  {
    for (int i = 0; i < nprocs-1; i++)
    {
      MPI_Status status;
      double buf;
      MPI_Recv(&buf, 1, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      int source = status.MPI_SOURCE;
      A[source]=buf;

     #pragma omp task firstprivate(source) shared(A)
     {
       process(A[source]);
     }
   }
  } // master
  } // parallel

}
```