# Bag of Tasks Tricks from MPI and OpenMP

Timothy H. Kaiser, Ph.D.
tkaiser@mines.edu
Director - CSM High Performance Computing
Director - Golden Energy Computing Organization

## Miscellaneous Topics

- Some tricks from MPI and OpenMP
- Libraries
- Nvidia node

## Tricks from MPI and OpenMP

- Show that parallel runs can be managed relatively easily using either package
- Show some things that are possible
  - Maybe try on your own
  - Come talk with us about it

# What if you have?

- A bunch of tasks to do
- They are all independent
- Similar, maybe just different input files
- Often called bag of task parallelism or embarrassingly parallel

# Workerbee.c

- Starts MPI
- Splits the processors into two groups/communicators 0-(N-2) and (N-1)
- Processor (N-1) waits for "ready" from other processors, then sends work
- Rest of processors loop
  - send requests for work
  - do work
  - send results

The Full Source for this program is in the slides but we will skip over most of it and look at just the worker and manager subroutines

## Do initialization

```
void init_it(int  *argc, char ***argv) {
  mpi_err = MPI_Init(argc,argv);
  mpi_err = MPI_Comm_size( MPI_COMM_WORLD, &numnodes );
  mpi_err = MPI_Comm_rank(MPI_COMM_WORLD, &myid);
}

int main(int argc,char *argv[]){
    int *will_use;
    MPI_Comm WORKER_WORLD;
    MPI_Group new_group,old_group;
    int ijk,num_used,used_id;
    init_it(&argc,&argv);
    printf("hello from %d\n",myid);
```

## Create a communicator that contains everyone but the last process

```
/* get our old group from MPI_COMM_WORLD */
  mpi_err = MPI_Comm_group(MPI_COMM_WORLD,&old_group);
/* create a new group from the old group that */
/* will contain a subset of the  processors */
  num_used= numnodes-1
  will_use=(int*)malloc(num_used*sizeof(int));
    for (ijk=0;ijk <= num_used-1;ijk++){
     will_use[ijk]=ijk;
    }
    mpi_err = MPI_Group_incl(old_group,num_used,will_use,&new_group);
/* create the new communicator */
    mpi_err=MPI_Comm_create(MPI_COMM_WORLD,new_group,&WORKER_WORLD);
/* find my rank in the new_group. */
/* if not in the new group then used_id will be  MPI_UNDEFINED */
    mpi_err =  MPI_Group_rank(new_group,&used_id);
```

## Manager is not part of "WORKER_WORLD" so she manages

```
 if(used_id == MPI_UNDEFINED){
/* if not part of the new group then do management. */
   manager(num_used);
   printf("manager finished\n");
   mpi_err =  MPI_Barrier(MPI_COMM_WORLD);
   mpi_err =  MPI_Finalize();
   exit(0);
}
```

## Workers work
## Note: we are passing in the ID for the manager, num_used

```
worker(WORKER_WORLD,num_used);
printf("worker finished\n");
mpi_err = MPI_Barrier(MPI_COMM_WORLD);
mpi_err = MPI_Finalize();
exit(0);
```

- Workers tell manager they are ready
- Get work
- Do work
- Send Results

```
void worker(MPI_Comm THE_COMM_WORLD,int managerid) {
 float x;
 MPI_Status status;
 x=0.0;
 while(x > -1.0) {
/* send message says I am ready for data */
   mpi_err= MPI_Send((void*)&x,1,MPI_FLOAT,managerid,1234,MPI_COMM_WORLD);
/* get a message from the manager */
   mpi_err= MPI_Recv((void*)&x,1,MPI_FLOAT,managerid,2345,MPI_COMM_WORLD,&status);
/* process data */
   x=x*2.0;
   sleep(1);
 }
}
```

Here the message to do a task is just a real number and the task is to multiply it by 2. The message could be a text string containing a command and the task could be to run that command as a new process

- Manager waits for ready message
- Sends work
- Tells everyone to quit when work is finished

```
#define TODO 100
void manager(int num_used){
     int igot,isent,gotfrom,sendto,i;
     float inputs[TODO];
     float x;
     MPI_Status status;
     int flag;
     igot=0;   isent=0;
     for(i=0;i<TODO;i++) {
          inputs[i]=i+1;
     }
     while(igot < TODO) { /* wait for a request for work */
          mpi_err = MPI_Iprobe(MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,&flag,&status);
          if(flag){
               /* where is it comming from */
               gotfrom=status.MPI_SOURCE;
               sendto=gotfrom;
               mpi_err = MPI_Recv((void*)&x,1,MPI_FLOAT,gotfrom,1234,MPI_COMM_WORLD,&status);
              printf("worker %d sent %g\n",gotfrom,x);
               if(x > 0.0) { igot++; }
               if(isent < TODO){ /* send real data */
                    x=inputs[isent];
                    mpi_err = MPI_Send((void*)&x,1, MPI_FLOAT,sendto,2345,MPI_COMM_WORLD);
                    isent++;
               }
          }
     }
/* tell everyone to quit */
     for (i=0;i<num_used;i++){
          x=-1000.0;
          mpi_err = MPI_Send((void*)&x,1, MPI_FLOAT,i,2345,MPI_COMM_WORLD);
     }

}
```

# OpenMP

- Each core is running a different thread
- Threads are numbered 0 to (N-1)
- Threads share memory
- Messages between threads are passed via the shared memory
- Number of threads is limited to cores on a node
- Parallelism is suggested to the compiler via directives

# Four Independent Matrix Inversions

```
#pragma omp parallel sections
 {
#pragma omp section
        {
       system_clock(&t1_start);
       over(m1,n);
       over(m1,n);
       system_clock(&t1_end);
       e1=mcheck(m1,n,1);
       t1_start=t1_start-t0_start;
       t1_end=t1_end-t0_start;
        }
#pragma omp section
        {
       system_clock(&t2_start);
       over(m2,n);
       over(m2,n);
       system_clock(&t2_end);
       e2=mcheck(m2,n,2);
       t2_start=t2_start-t0_start;
       t2_end=t2_end-t0_start;
       }
#pragma omp section
        {
         system_clock(&t3_start);
         over(m3,n);
         over(m3,n);
         system_clock(&t3_end);
         e3=mcheck(m3,n,3);
         t3_start=t3_start-t0_start;
         t3_end=t3_end-t0_start;
        }
#pragma omp section
        {
         system_clock(&t4_start);
         over(m4,n);
         over(m4,n);
         system_clock(&t4_end);
         e4=mcheck(m4,n,4);
         t4_start=t4_start-t0_start;
         t4_end=t4_end-t0_start;
        }
}
```

- The “over” routines could be any independent operations
- Became parallel by adding 5 directive lines

# Four Independent Matrix Inversions

```
-bash-3.2$ export OMP_NUM_THREADS=1
-bash-3.2$ ./invertc
section 1 start time= 0.00095582   end time=    0.32442  error= 6.00659e-06
section 2 start time=    0.32478   end time=    0.64667  error= 0.000453301
section 3 start time=    0.64702   end time=    0.96885  error= 8.78033e-05
section 4 start time=     0.9692   end time=     1.2911  error= 0.000873184
```

```
-bash-3.2$ export OMP_NUM_THREADS=2
-bash-3.2$ ./invertc
section 1 start time=  0.0013599   end time=    0.32445  error= 6.00659e-06
section 2 start time=    0.32481   end time=    0.64647  error= 0.000453301
section 3 start time=  0.0013621   end time=    0.36006  error= 8.78033e-05
section 4 start time=    0.36042   end time=    0.71921  error= 0.000873184
```

```
-bash-3.2$ export OMP_NUM_THREADS=4
-bash-3.2$ ./invertc
section 1 start time=   0.001534   end time=    0.32544  error= 6.00659e-06
section 2 start time=   0.001538   end time=    0.32929  error= 0.000453301
section 3 start time=   0.002799   end time=    0.32627  error= 8.78033e-05
section 4 start time=   0.002799   end time=     0.3263  error= 0.000873184
-bash-3.2$
```

# N Independent Matrix Inversions

- OpenMP distributes do (for) loop iterations across the cores
- Here we call the matrix inversion routine DGESV nrays times
- Each inversion works on different data stored in tarf(:,:,i)
- Each thread does nrays/ncores calls and we get near linear speedup

```
!$OMP PARALLEL DO PRIVATE(twod)
      do i=1,nrays
        twod=>tarf(:,:,i)
        call my_clock(cnt1(i))
        CALL DGESV( N, NRHS, twod, LDA, IPIVs(:,i), Bs(:,i), LDB, INFOs(i) )
        call my_clock(cnt2(i))
        write(*,'(i5,i5,3(f12.3))')i,infos(i),cnt2(i),cnt1(i),real(cnt2(i)-cnt1(i),b8)
      enddo
```