Hybrid MPI/OpenMP programming

Martin Čuma
Center for High Performance Computing University of Utah
m.cuma@utah.edu
Overview

• Single and multilevel parallelism.
• Example of MPI-OpenMP buildup.
• Compilation and running.
• Performance suggestions.
• Code examples.
Single level parallelism

- **Shared memory computers**
  - N processors, single system image
  - thread-based parallelism - OpenMP, shmem
  - message-based parallelism - MPI
- **Distributed memory computers**
  - nodes with local memory, coupled via network
  - message-based parallelism – MPI
  - partitioned global space – UPC, Coarray Fortran
- Each node has N processors that share memory
- Nodes loosely connected (network)
- CHPC:
  - 8, 12, 16, 20, 24 core cluster nodes
Multilevel parallelism

- Coarse and fine grain level
  - coarse – nodes, processors, fine – CPU cores
  - MPI - nodes, CPU sockets
    OpenMP, pthreads, shmem – CPU cores
  - OpenMP works best with processing intensive loops

- Multilevel advantages
  - memory limitations – extra memory for each copy of executable on the node
  - process vs. thread overhead
  - message overhead
  - portability, ease to maintain (can disable OpenMP)
• MPI (Message Passing Interface)
  ▪ standardized library (not a language)
  ▪ collection of processes communicating via messages
  ▪ available for most architectures
  ▪ http://www.mpi-forum.org/

• OpenMP
  ▪ API for shared memory programming
  ▪ available on most architectures as a compiler extension (C/C++, Fortran)
  ▪ includes compiler directives, library routines and environment variables
  ▪ www.openmp.org
Processes vs. threads

- **Process**
  - have own address space
  - can have multiple threads

- **MPI**
  - many processes
  - shared-nothing architecture
  - explicit messaging
  - implicit synchronization
  - all or nothing parallelization

- **Thread**
  - execute within process
  - same address space
  - share process’s stack
  - thread specific data

- **OpenMP**
  - 1 process, many threads
  - shared-everything architecture
  - implicit messaging
  - explicit synchronization
  - incremental parallelism
• Calculation of value of π using integral:

\[ \int_{0}^{1} \frac{dx}{x^2 + 1} = \frac{\pi}{4} \]

• trapezoidal rule
• simple loop easy to parallelize both with MPI and OpenMP
```c
#include <stdio.h>
#include <math.h>
#include "timer.h"
int main(int argc, char *argv[]){
    const int N = 10000000000;
    const double h = 1.0/N;
    const double PI = 3.141592653589793238462643;
    double x,sum,pi, error, time; int i;

    time = ctimer();
    sum = 0.0;
    for (i=0;i<=N;i++){
        x = h * (double)i;
        sum += 4.0/(1.0+x*x);
    }
    pi = h*sum;
    time += ctimer();

    error = pi - PI;
    error = error<0 ? -error:error;
    printf("pi = %18.16f +/- %18.16f
",pi,error);
    printf("time = %18.16f sec
",time);
    return 0;
}
```

- User-defined timer
- Calculation loop
- Print out result
```c
#include <stdio.h>
#include <math.h>
#include "timer.h"
int main(int argc, char *argv[]){
    const int N = 10000000000;
    const double h = 1.0/N;
    const double PI = 3.141592653589793238462643;
    double x,sum,pi,error,time; int i;

    time = -ctimer();
    sum = 0.0;
    #pragma omp parallel for shared(N,h),private(i,x),reduction(+:sum)
    for (i=0;i<=N;i++){
        x = h * (double)i;
        sum += 4.0/(1.0+x*x);
    }
    pi = h*sum;
    time += ctimer();

    .......

    return 0;
}
```
```c
#include <stdio.h>
#include <math.h>
#include "timer.h"
int main(int argc, char *argv[]){
    const int N = 10000000000;
    const double h = 1.0/N;
    const double PI = 3.141592653589793238462643;
    double x,sum,pi,error,time,mypi; int i;
    int myrank,nproc;

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);

    time = -ctimer();
    sum = 0.0;
    for (i=myrank;i<=N;i=i+nproc){
        x = h * (double)i;
        sum += 4.0/(1.0+x*x);
    }
    mypi = h*sum;
    MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
    time += ctimer();
    ......
    return 0;
}
```

- MPI initialization
- Distributed loop
- Global reduction
MPI-OpenMP code

```c
#include <stdio.h>
#include <math.h>
#include "timer.h"
int main(int argc, char *argv[]){
const int N = 10000000000;
const double h = 1.0/N;
const double PI = 3.141592653589793238462643;
double x,sum,pi,error,time,mypi; int i;
int myrank,nproc;

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
MPI_Comm_size(MPI_COMM_WORLD,&nproc);

time = -ctimer();
sum = 0.0;

#pragma omp parallel for shared(N,h,myrank,nproc),private(i,x),reduction(+:sum
for (i=myrank;i<=N;i=i+nproc){
    x = h * (double)i;
    sum += 4.0/(1.0+x*x);
}
mypi = h*sum;
MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
time += ctimer();
......
return 0;
```
- GNU, PGI, Intel compilers, OpenMP with `-fopenmp, -mp, -openmp` switch

- MPICH2, MVAPICH2, OpenMPI or Intel MPI

```bash
module load mpich2 MPICH2
module load mvapich2 MVAPICH2
module load openmpi OpenMPI
module load impi Intel MPI
```

```bash
mpicc -mp=numa source.c -o program.exe (PGI)
mpif90 -openmp source.f -o program.exe (Intel)
mpif90 -fopenmp source.f -o program.exe (GNU)
```
Third party libraries

- BLASes and FFTW are threaded
- Intel compilers:
  -I/uufs/chpc.utah.edu/sys/pkg/fftw/std_intel/include
  -lfftw3 -lfftw3_omp
  -L/uufs/chpc.utah.edu/sys/pkg/fftw/std_intel/lib
  -Wl,-rpath=/uufs/chpc.utah.edu/sys/installdir/intel/mkl/lib/intel64
  -L/uufs/chpc.utah.edu/sys/installdir/intel/mkl/lib/intel64
  -lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core -liomp5 -lpthread
- PGI compilers:
  -I/uufs/chpc.utah.edu/sys/pkg/fftw/std_pgi/include
  -lfftw3 -lfftw3_omp
  -L/uufs/chpc.utah.edu/sys/pkg/fftw/std_pgi/lib -lacml_mp
- MKL ScaLAPACK w/ Intel
  -Wl,-rpath=/uufs/chpc.utah.edu/sys/installdir/intel/mkl/lib/intel64
  -L/uufs/chpc.utah.edu/sys/installdir/intel/mkl/lib/intel64
  -lmkl_scalapack_ilp64 -lmkl_intel_ilp64 -lmkl_core
  -lmkl_intel_thread -lmkl_blacs_intelmpi_ilp64 -liomp5 -lpthread -lm

• Ask for #MPI processes
• Use SLURM environment variables to get OpenMP thread count
• Interactive batch (asking for 2 nodes, 2 tasks/node)

```bash
srun -n 4 -N 2 -t 1:00:00 -p kingspeak -A chpc -pty
/bin/tcsh -l
... wait for prompt ...
```

```bash
set TPN=`echo $SLURM_TASKS_PER_NODE | cut -f 1 -d '('
set PPN=`echo $SLURM_JOB_CPUS_PER_NODE | cut -f 1 -d '('
@ THREADS = ( $PPN / $TPN )
mpirun -genv OMP_NUM_THREADS=$THREADS -np $SLURM_NTASKS
./program.exe
```

• Non-interactive batch
  • same thing, except in a Slurm script
Running – process pinning

- Current NUMA architectures penalize memory access on neighboring CPU sockets
- Distribute and bind processes to CPU sockets

- Intel compilers can also pin threads to cores
  
  module load intel mvapich2
  
  mpirun -genv KMP_AFFINITY granularity=fine,compact,1,0 -genv MV2_BINDING_POLICY scatter -genv MV2_BINDING_LEVEL socket -genv OMP_NUM_THREADS 8 -np 4

- Intel MPI binds processes to sockets by default
  
  Module load intel impi
  
  mpirun -x KMP_AFFINITY granularity=fine,compact,1,0 -genv OMP_NUM_THREADS 8 -np 4
  
  or use I_MPI_PIN_DOMAIN=socket
• Parallelize main problem using MPI
  ▪ task decomposition
    ▪ frequencies in wave solvers
  ▪ domain decomposition
    ▪ distribute atoms in molecular dynamics
    ▪ distribute mesh in ODE/PDE solvers
• Exploit internal parallelism with OpenMP
  ▪ use profiler to find most computationally intense areas
    ▪ internal frequency loop in wave solvers
    ▪ local force loop in MD
    ▪ local element update loop in ODE/PDE solvers
  ▪ measure the efficiency to determine optimal number of threads to use
  ▪ Intel AdvisorXE can be helpful (module load advisorxe)
Things to be aware of

• Not every MPI program will benefit from adding threads
  ▪ not worth with loosely parallel codes (too little communication)
  ▪ overhead with thread creation about $10^4$ flops
  ▪ time with different node/thread count to get the best performing combination

• MPI communication within OpenMP
  ▪ can be tricky if each thread communicates
  ▪ Some MPI implementations still have trouble with MPI_THREAD_MULTIPLE
Four MPI threading models

- **MPI_THREAD_SINGLE**
  - only non-threaded section communicates

- **MPI_THREAD_FUNNELLED**
  - process may be multithreaded but only master thread communicates

- **MPI_THREAD_SERIALIZED**
  - multiple threads may communicate but only one at time

- **MPI_THREAD_MULTIPLE**
  - all threads communicate
Example of single thread communication.

- **Complex norm routine**

```c
int main(int argc, char **argv) {
    ........
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    ........

double _Complex stabWmnorm(double *Wm, double _Complex *stab, int size) {
    double _Complex norm, vec, norml;
    int i;
    
    norml = 0 + I*0;
    #pragma omp parallel for private(i,vec) reduction(+:norml) 
    for (i=0;i<size;i++)
    {
        vec = stab[i]*Wm[i];
        norml = norml + vec*conj(vec);
    }
    MPI_Allreduce(&norml,&norm,1,MPI_DOUBLE_COMPLEX,MPI_SUM,MPI_COMM_WORLD);
    
    return sqrt(norm);
}

MPI_Finalize();
```

Parallel OpenMP for loop

MPI communication outside OpenMP
Multiple threads comm. - initialization

- Special MPI_Init
  - Returns variable thread_status which indicates what level of threading is supported

```c
int thread_status;

MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &thread_status);
if (thread_status!=MPI_THREAD_MULTIPLE)
{
    printf("Failed to initialize MPI_THREAD_MULTIPLE\n");
    exit(-1);
}

...

MPI_Finalize();
```
Multiple threads point-to-point communication

```c
#pragma omp parallel private(iis,niip,iip,iisf)
{
    double _Complex *ne, *nh; int comlab, mythread, nthreads;
    MPI_Status statx[fwdd->Nz];
    MPI_Request reqx[fwdd->Nz];

    #ifdef _OPENMP
    mythread = omp_get_thread_num(); nthreads = omp_get_num_threads();
    #endif

    ne = (double _Complex *)malloc(sizeof(double _Complex)*3*Nxy);
    comlab = mythread*10000; // different tag for each proc/thread

    for (iis=mythread; iis < Ncp[0]; iis+=nthreads)
    {
        if (cpuinfo[0] == iip)
        {
            MPI_Isend(&ne[0], Nxy, MPI_DOUBLE_COMPLEX, Dp[0], comlab, MPI_COMM_WORLD, reqx[Nreqi[0]]);
            Nreqi[0]++;
        }
        else if (cpuinfo[0] == Dp[0])
        {
            MPI_Irecv(&Ebb[ie[0]*Nxy], Nxy, MPI_DOUBLE_COMPLEX, iip, comlab, MPI_COMM_WORLD, reqx[Nreqi[0]]);
            Nreqi[0]++;
        }
        MPI_Waitall(Nreqi[0], &reqx[0], &statx[0]);
    }

    free(ne);
}
```

Start parallel OpenMP section

Data structures for non-blocking communication

Allocate local thread arrays

Find thread # and # of threads

Allocate local thread arrays

Each thread does different iteration of this loop

Each communication pair has unique tag

Finalize non-blocking communication

Free local thread arrays

End OpenMP parallel section

-> use message tag to differentiate between threads
Multiple threads collective communication

```c
MPI_Comm comm_thread[NOMPCPUS];

#pragma omp parallel private(iis,niip,iip,iisf)
{
    double _Complex *ne; int mythread, nthreads

#ifdef _OPENMP
    mythread = omp_get_thread_num(); nthreads = omp_get_num_threads();
#endif

    ne = (double _Complex *)malloc(sizeof(double _Complex)*3*Nxy);

    for(ithr=0; ithr < nthreads; ithr++)
    {
        #pragma omp barrier // synchronize so that each process gets the right thread
        if (ithr==mythread) MPI_Comm_dup(comm_domain,&comm_thread[mythread]);
    }

    for (iis=mythread; iis < Ncp[0]; iis+=nthreads)
    {
        ... calculate ne ...
        MPI_Gatherv( &ne[indgbp[iic]],Nxy_loc,MPI_DOUBLE_COMPLEX, &Gb[ie[ic]*Nxy2 + iit2], Nxy_rec, Nxy_disp, MPI_DOUBLE_COMPLEX, Dp[ic],comm_thread[mythread]);
    }

    for(ithr=0; ithr < nthreads; ithr++)
    {
        if (ithr==mythread) MPI_Comm_free(&comm_thread[mythread]);
    }

    free(ne);
}
```

- Start parallel OpenMP section
- Local thread variables
- Allocate local thread arrays
- Find thread # and # of threads
- Per thread communicator
- Each thread does different iteration of this loop
- Thread communicator
- Free thread communicators
- Free local thread arrays
- End OpenMP parallel section

-> use communicators to differentiate between threads
Future outlook

- Mixed MPI-OpenMP has become commonplace
  - reduces memory footprint per core
  - better locality of memory access per core
  - faster inter-node communication – larger messages, smaller overhead
Another MPI-OpenMP example

- Master-worker code
  - good for parallelization of problems of varying run time
  - master feeds workers with work until all is done
- Disadvantage – master does not do any work
- Run two OpenMP threads on the master
  - distribute work
  - do work
- Critical section at the work selection
- Can run also on single processor nodes
int main(int argc, char **argv){
    ......
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    ......
    master = numprocs - 1;
    ......
    if (myid == master) {
        ......
        omp_set_num_threads(2);
        #pragma omp parallel sections private(request) {
            #pragma omp section {
                ......
            #pragma omp critical (gen_work) {
                work = generate_work(&work_data, num_tasks, work_array, job_flag);
            }
            ......
        }#pragma omp section{
            ......
        }#pragma omp critical (gen_work){
            work = generate_work(&work_sl_data, num_tasks, work_array, job_flag);
        }
        ......
        #pragma omp barrier
        ......
    }
    else {
        ......
    }
    MPI_Barrier(world); MPI_Finalize();}
• Single and multilevel parallelism
• Simple MPI-OpenMP example
• Compilation, running
• A few advices

http://www.chpc.utah.edu/short_courses/mpi_omp
• MPI

http://www.mpi-forum.org/
Pacheco - Parallel Programming with MPI
Gropp, Lusk, Skjellum – Using MPI 1, 2

• OpenMP

http://www.openmp.org/
Chandra, Dagum, Kohr,… - Parallel Programming in OpenMP

• MPI+OpenMP

Pacheco – Introduction to Parallel Programming