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.
• 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:
  - 12, 16, 20, 24, 28, 32, 64 core cluster nodes
Multilevel parallelism

- Coarse and fine grain level
  - coarse – nodes, processors (sockets)
  - 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
  ▪ executes 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 \( \pi \) 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
#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\n",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;}
```
#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
OK here, inefficient for vectors due to strided memory access

• Global reduction
#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;}

• OpenMP directive to parallelize each MPI task loop using threads

• Sum MPI task local values of π
- GNU, PGI, Intel compilers, OpenMP with \(-\text{fopenmp, -mp, -qopenmp}\) switch

- MPICH, MVAPICH2, OpenMPI or Intel MPI

  module load mpich MPICH
  module load mvapich2 MVAPICH2
  module load openmpi OpenMPI
  module load impi Intel MPI

  mpicc \(-mp=\text{numa}\) source.c \(-o\) program.exe (PGI)
  mpif90 \(-\text{fopenmp}\) source.f \(-o\) program.exe (Intel gfortran)
  mpiifort \(-q\text{openmp}\) source.f \(-o\) program.exe (Intel ifort)
  mpif90 \(-\text{fopenmp}\) source.f \(-o\) program.exe (GNU)
• BLASes and FFTW are threaded

• Intel compilers:
  -I$FFTW_INCDIR -lfftw3 -lfftw3_omp -L$FFTW_LIBDIR
  -Wl,-rpath=$MKLROOT/lib/intel64 -L$MKLROOT/lib/intel64
  -lmkl_intel_lapack_lp64 -lmkl_intel_thread -lmkl_core -liomp5 -lpthread

• PGI compilers:
  -I$FFTW_INCDIR -lfftw3 -lfftw3_omp -L$FFTW_LIBDIR -lacml_mp

• MKL ScaLAPACK w/ Intel
  -Wl,-rpath=$MKLROOT/lib/intel64 -L$MKLROOT/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)
  
  `salloc -n 4 -N 2 -t 1:00:00 -p kingspeak -A chpc ... wait for prompt ...

  `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
• 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

```bash
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

or use I_MPI_PIN_DOMAIN=socket
```

• Intel MPI binds processes to sockets by default

```bash
module load intel impi
mpirun -x KMP_AFFINITY granularity=fine,compact,1,0
        -genv OMP_NUM_THREADS 8 -np 4
```
• Default pinning policies for compilers and MPI distributions vary.


• Some applications can gain up to 30% performance with pinning processes AND threads.

• Using pinthreads.sh script from the article with common compilers (Intel, PGI, GNU) and MPIs (MPICH, MVAPICH2, IMPI, OpenMPI) on a 24 core node, 8 MPI tasks 3 threads each:

  mpirun -np 8 -genv OMP_NUM_THREADS 3 -bind-to socket -map-by socket ./pinthreads.sh ./myprogram

• Check the pinning by this bash one-liner:

  for i in $(pgrep myprogram); do for tid in $(ps --no-headers -mo tid -p $i |grep -v -); do taskset -cp "$\{tid\}"; done ;
• 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 Advisor can be helpful (module load advisor)
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
  - be aware of thread safety in MPI when using MPI_THREAD_MULTIPLE
• Defines if it is safe to use program or library with parallel threads
• Most libraries these days are thread safe
  – But it’s good to check, usually there is some note in the documentation or in the build scripts
  – Some libraries have threaded and non-threaded versions
• Most often thread safety relates to the concurrent access of shared data
• MPI defines several threading models, some allow communication from threads, some don’t
Four MPI threading models

- **MPI_THREAD_SINGLE**
  - only non-threaded section communicates (default)
- **MPI_THREAD_FUNNELED**
  - process may be multithreaded but only master thread communicates
- **MPI_THREAD_SERIALIZED**
  - multiple threads may communicate but only one at a time
- **MPI_THREAD_MULTIPLE**
  - all threads communicate (fully thread safe)
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();
```
• 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

#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)
    {
        ... calculate pieces of large distributed vector Ebb as a local vector ne
        if (cpuinfo[0] == iip)
        {
            MPI_Isend(&ne[0], Nxy, MPI_DOUBLE_COMPLEX, Dp[0], comlab, MPI_COMM_WORLD, reqx[Nreqi[0]]);
            Nreqi[0]++; comlab++;
        }
        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]++; comlab++;
        }
        MPI_Waitall(Nreqi[0], &reqx[0], &statx[0]);
    }

    free(ne);
}

Start parallel OpenMP section
Data structures for non-blocking communication
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

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
Find thread # and # of threads
Allocate local thread arrays
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
  - One sided MPI communication further improves parallel efficiency
• Single and multilevel parallelism
• Simple MPI-OpenMP example
• Compilation, running
• A few advices

http://www.chpc.utah.edu/short_courses/mpi_omp
References

• MPI
  http://www.mpi-forum.org/

• OpenMP
  http://www.openmp.org/

• MPI+OpenMP
  Pacheco – Introduction to Parallel Programming

• XSEDE HPC Summer Boot Camp
  OpenMP, OpenACC, MPI
  https://www.youtube.com/XSEDETraining