Hybrid MPI and OpenMP

Monte Lunacek and Timothy Dunn
Outline

• OpenMP
  • multiply.cpp
• Thread affinity/NUMA
• MPI
  • trap.cpp
• Hybrid: use any of the advantages of both
  • Processor affinity
Software

• Gcc 4.5+ OpenMP
  • http://hpc.sourceforge.net/

• OpenMPI
  • http://www.open-mpi.org/software/ompi/v1.6/

• CMake
  • http://www.cmake.org/cmake/resources/software.html
Code Examples

• https://github.com/ResearchComputing/Hybrid-Tutorial
Shared-memory Model

Processors interact and synchronize with each other through shared variables.
Compiling OpenMP

Gnu

gfortran -fopenmp -g basic.f90

gcc -fopenmp -g basic.c

Intel

ifort -openmp -g basic.f90

icc -openmp -g basic.c

Environmental variables

setenv OMP_NUM_THREADS 12

export OMP_NUM_THREADS=12
Functions

Threads

int thread = omp_get_thread_num();

Thread id

int size = omp_get_num_threads();

How many threads?

omp_set_num_threads(4);

What is another way to set the number of threads?

export OMP_NUM_THREADS=12
Directives

A way for the programmer to communicate with the compiler

Compiler free to ignore directives (they are hints)

#pragma directive is used to instruct the compiler to use pragmatic or implementation-dependent features (e.g. OpenMP)

#pragma omp parallel

#pragma omp parallel private(var1, var2,...)
Either loop could be executed in parallel

We prefer to make outer loop parallel, to reduce number of forks/joins

We then must give each thread its own private copy of variable i

```cpp
for(int i=0; i<num_rows; ++i)
{
    for(int j=0; j<num_cols; ++j)
    {
        a(i,j) = a(i,j) + tmp
    }
}
```
Directs compiler to make one or more variables private

```cpp
#pragma omp parallel for private(j)
for(int i=0; i<num_rows; ++i)
{
    for(int j=0; j<num_cols; ++j)
    {
        a(i,j) = a(i,j) + tmp
    }
}
```
Example: Multiply.cpp
NUMA Architecture
Symmetric Multiprocessing (SMP)

Each CPU can access any memory location in the same amount of time as any other CPU in the system.

Fine for a small number of CPUs (< ~ 8 - 16)
Non-Uniform Memory Access (NUMA)

Each socket has a dedicated memory area for high speed access

Also has an interconnect to other sockets for slower access to the other sockets' memory
Affinity

Keeping threads close to the data they access

Problem:

- CPU on socket 0 can take twice as long to access memory on socket 1
- Avoid regularly accessing remote memory in a NUMA topology system

How do we ensure that information travels the shortest path?

- Bind thread to processor (so OS doesn’t move them)
- Bind thread to memory near processor
Two ways to bind processes and memory

numactl

libnuma

Where are processes assigned?

Which memory do they use?
numactl --hardware

- Janus
  available: 2 nodes (0-1)
  node 0 size: 12090 MB
  node 0 free: 8565 MB
  node 1 size: 12120 MB
  node 1 free: 11597 MB
  node distances:
  node 0 1
  0: 10 20
  1: 20 10

- Himem
  available: 4 nodes (0-3)
  node 0 size: 258508 MB
  node 0 free: 254698 MB
  node 1 size: 258560 MB
  node 1 free: 255805 MB
  node 2 size: 258560 MB
  node 2 free: 258437 MB
  node 3 size: 256540 MB
  node 3 free: 256477 MB
  node distances:
  node 0 1 2 3
  0: 10 20 20 20
  1: 20 10 20 20
  2: 20 20 10 20
  3: 20 20 20 10
Quick guide to numactl

<table>
<thead>
<tr>
<th>Socket affinity</th>
<th>-N --cpunodebind=</th>
<th>{0,1}</th>
<th>Execute process on cores of these sockets only</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory policy</td>
<td>-l --localalloc</td>
<td>No argument</td>
<td>Allocate on current socket; fallback to any other if full</td>
</tr>
<tr>
<td>Memory policy</td>
<td>-i --interleave=</td>
<td>{0,1}</td>
<td>Allocate round robin (interleave) on these sockets. No fallback</td>
</tr>
<tr>
<td>Memory policy</td>
<td>--preferred=</td>
<td>{0,1} select one</td>
<td>Allocate on this socket; fallback to any other if full.</td>
</tr>
<tr>
<td>Memory policy</td>
<td>-m --membind=</td>
<td>{0,1}</td>
<td>Allocate only on this (these) socket(s). No fallback.</td>
</tr>
<tr>
<td>Core affinity</td>
<td>-C --physcpubind=</td>
<td>{1,2,3,4,5,6,7,8,9,10,11,12}</td>
<td>Execute process on this (these) core(s) only</td>
</tr>
</tbody>
</table>
Examples

```
numactl --cpunodebind=0 --localalloc application

numactl --cpunodebind=0 --membind=0 application

numactl --cpubind=0 --membind=0,1 application

numactl --physcpubind=3 application

numactl --interleave=all
```
Stream

Simple synthetic benchmark program that measures sustainable memory bandwidth (in MB/s)

Creates three arrays, called a, b, and c, each of a fixed size N with double-precision elements

Initializes these arrays

Performs the following sequence of streaming benchmarks

- **Copy**: $c[k] = a[k]$ (one read and one write)
- **Scale**: $b[k] = \alpha \cdot c[k]$ (one read and one write)
- **Add**: $c[k] = a[k] + b[k]$ (two reads and one write)
- **Triad**: $a[k] = b[k] + \alpha \cdot c[k]$ (two reads and one write)
**OpenMP Stream**

```bash
eexport OMP_NUM_THREADS=6
./stream_omp
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Rate (MB/s)</th>
<th>Avg time</th>
<th>Min time</th>
<th>Max time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy:</td>
<td>24510.1768</td>
<td>0.0033</td>
<td>0.0033</td>
<td>0.0033</td>
</tr>
<tr>
<td>Scale:</td>
<td>24330.6736</td>
<td>0.0033</td>
<td>0.0033</td>
<td>0.0033</td>
</tr>
<tr>
<td>Add:</td>
<td>25141.9392</td>
<td>0.0048</td>
<td>0.0048</td>
<td>0.0049</td>
</tr>
<tr>
<td>Triad:</td>
<td>25423.8763</td>
<td>0.0047</td>
<td>0.0047</td>
<td>0.0047</td>
</tr>
</tbody>
</table>

```bash
numactl --membind=0 ./stream_omp
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Rate (MB/s)</th>
<th>Avg time</th>
<th>Min time</th>
<th>Max time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy:</td>
<td>13867.7600</td>
<td>0.0058</td>
<td>0.0058</td>
<td>0.0058</td>
</tr>
<tr>
<td>Scale:</td>
<td>13500.0732</td>
<td>0.0060</td>
<td>0.0059</td>
<td>0.0061</td>
</tr>
<tr>
<td>Add:</td>
<td>14450.6598</td>
<td>0.0083</td>
<td>0.0083</td>
<td>0.0084</td>
</tr>
<tr>
<td>Triad:</td>
<td>14857.1739</td>
<td>0.0081</td>
<td>0.0081</td>
<td>0.0082</td>
</tr>
</tbody>
</table>
OpenMP Stream

export OMP_NUM_THREADS=12

./stream_numa

<table>
<thead>
<tr>
<th>Function</th>
<th>Rate (MB/s)</th>
<th>Avg time</th>
<th>Min time</th>
<th>Max time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy:</td>
<td>13829.4654</td>
<td>0.0232</td>
<td>0.0231</td>
<td>0.0232</td>
</tr>
<tr>
<td>Scale:</td>
<td>18381.2060</td>
<td>0.0174</td>
<td>0.0174</td>
<td>0.0174</td>
</tr>
<tr>
<td>Add:</td>
<td>18086.5301</td>
<td>0.0266</td>
<td>0.0265</td>
<td>0.0266</td>
</tr>
<tr>
<td>Triad:</td>
<td>18135.7336</td>
<td>0.0265</td>
<td>0.0265</td>
<td>0.0265</td>
</tr>
</tbody>
</table>

numactl --interleave=all ./stream_numa

<table>
<thead>
<tr>
<th>Function</th>
<th>Rate (MB/s)</th>
<th>Avg time</th>
<th>Min time</th>
<th>Max time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy:</td>
<td>22200.8945</td>
<td>0.0144</td>
<td>0.0144</td>
<td>0.0145</td>
</tr>
<tr>
<td>Scale:</td>
<td>25356.6327</td>
<td>0.0127</td>
<td>0.0126</td>
<td>0.0128</td>
</tr>
<tr>
<td>Add:</td>
<td>29345.3331</td>
<td>0.0164</td>
<td>0.0164</td>
<td>0.0165</td>
</tr>
<tr>
<td>Triad:</td>
<td>30245.5670</td>
<td>0.0159</td>
<td>0.0159</td>
<td>0.0160</td>
</tr>
</tbody>
</table>
export KMP_AFFINITY=verbose,compact

export OMP_NUM_THREADS=6

./stream_kmp

   Number of Threads requested = 6
   OMP: Info #147: KMP_AFFINITY: Internal thread 1 bound to OS proc set {1}
   OMP: Info #147: KMP_AFFINITY: Internal thread 2 bound to OS proc set {2}
   OMP: Info #147: KMP_AFFINITY: Internal thread 3 bound to OS proc set {3}
   OMP: Info #147: KMP_AFFINITY: Internal thread 4 bound to OS proc set {4}
   OMP: Info #147: KMP_AFFINITY: Internal thread 5 bound to OS proc set {5}
export KMP_AFFINITY=verbose,scatter

export OMP_NUM_THREADS=6

./stream_kmp

OMP: Info #147: KMP_AFFINITY: Internal thread 0 bound to OS proc set {0}
Number of Threads requested = 6
OMP: Info #147: KMP_AFFINITY: Internal thread 1 bound to OS proc set {6}
OMP: Info #147: KMP_AFFINITY: Internal thread 2 bound to OS proc set {1}
OMP: Info #147: KMP_AFFINITY: Internal thread 3 bound to OS proc set {7}
OMP: Info #147: KMP_AFFINITY: Internal thread 4 bound to OS proc set {2}
OMP: Info #147: KMP_AFFINITY: Internal thread 5 bound to OS proc set {8}
When is this important?

Intel MKL DGEMM (matrix multiply)

if the data are not distributed, numactl --interleave=all a higher performance compared to default mode

Distributing matrices in the default mode has almost the same effect as applying NUMA interleave memory policy.

Similar findings for Linpack

NUMA helps when the whole matrix is initialized (and previously allocated) in the master thread
Quick MPI
Getting Started

#include <iostream>
#include <mpi.h>

int main(int argc, char ** argv)
{
    MPI_Init(&argc,&argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    std::cout << "rank " << rank << " of " << size << std::endl;

    MPI_Finalize();

    return 0;
}
Example: Trap.cpp
Hybrid OpenMP and MPI
Shared-memory vs. Message Passing

Shared-memory model

Number active threads 1 at start and finish of program, changes dynamically during execution

Incrementally make your program parallel

Programs may only have a single parallel loop

Message-passing model

All processes active throughout execution of program

Sequential-to-parallel transformation requires major effort
What is a hybrid model?

Use of inherently different models of programming in a complimentary manner, in order to achieve some benefit not possible otherwise.

A way to use different models of parallelization in a way that takes advantage of the good points of each;
Motivation

Introducing MPI into OpenMP applications can help scale across multiple nodes.

Introducing OpenMP into MPI applications can help make more efficient use of the shared memory on nodes mitigating the need for explicit intra-node communication.

Introducing MPI and OpenMP during the design/coding of a new application can help maximize efficiency, performance, and scaling.
Minimize...

Communication overhead
   e.g., messages inside of one node

Synchronization overhead
   e.g. OpenMP fork/join

Load imbalance
   e.g. using OpenMP schedules

Memory consumption
   e.g., replicated data in MPI parallelization

Computation overhead
   e.g., duplicated calculations in MPI parallelization
Details

OpenMP

Launch one process per node

Have each process fork one thread per core

Share data using shared memory

MPI

Launch one process per core

Pass messages among processes without concern for location

Hybrid OpenMP/MPI

we want each MPI process to launch multiple OpenMP threads that can share local memory
Configurations

Treat each node as an SMP

- launch a single MPI process per node
- create parallel threads sharing full-node memory
- 12 threads per node on JANUS

Treat each socket as an SMP

- launch one MPI process on each socket
- create parallel threads sharing same-socket memory
- 6 threads per socket on JANUS
Creating Configurations

To achieve configurations like these, we must be able to:

Assign to each process/thread an affinity for some set of cores

Make sure the allocation of memory is appropriately matched
Hello World!

#include <iostream>
#include <mpi.h>
#include <omp.h>

int main(int argc, char ** argv) {
    int provided;
    MPI_Init_thread(&argc, &argv, MPI_THREAD_SINGLE, &provided);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int thread_id, thread_size;
    #pragma omp parallel private(thread_id, thread_size)
    {
        thread_id = omp_get_thread_num();
        thread_size = omp_get_num_threads();
        #pragma omp critical
        std::cout << rank << " " << thread_id << " " << thread_size << std::endl;
    }
}
MPI

export OMP_NUM_THREADS=1

mpirun -np 24 ./hello_world

0 0 1
3 0 1
5 0 1
7 0 1
10 0 1
20 0 1
14 0 1
...

...
Processor Affinity

- Affinity can improve performance by inhibiting excessive process movement
- Judicious bindings can improve performance by
  - reducing resource contention (by spreading processes apart from one another)
  - improving interprocess communications (by placing processes close to one another).
  - improve performance reproducibility by eliminating variable process placement.
- Unfortunately, binding can also degrade performance by inhibiting the OS capability to balance loads.
Bindings

- man mpirun
- -bind-to-core, --bind-to-core
  - Bind processes to cores.
- -npernode, --npernode <N>
  - On each node, launch N processes.
- -pernode, --pernode
  - On each node, launch one process -- equivalent to -npernode 1.
- -bynode, --bynode
  - Launch processes one per node, cycling by node in a round-robin fashion. This spreads processes evenly among nodes and assigns ranks in a round-robin, "by node" manner.
- -report-bindings, --report-bindings
  - Report any bindings for launched processes.
bash-janus> mpirun -np 4 -bind-to-core -report-bindings ./hello_world
[node1080:28636] [[55081,1],0] to cpus 0001
[node1080:28636] [[55081,1],1] to cpus 0002
[node1080:28636] [[55081,1],2] to cpus 0004
[node1080:28636] [[55081,1],3] to cpus 0008

bash-janus> mpirun -np 4 -bynode -bind-to-core -report-bindings ./hello_world
[node1080:28733] [[51400,1],0] to cpus 0001
[node1080:28733] [[51400,1],2] to cpus 0002
[node1077:27981] [[51400,1],1] to cpus 0001
[node1077:27981] [[51400,1],3] to cpus 0002

bash-janus> mpirun -npernode 2 -bind-to-core -report-bindings ./hello_world
[node1080:30505] [[53212,1],0] to cpus 0001
[node1080:30505] [[53212,1],1] to cpus 0002
[node1077:29623] [[53212,1],2] to cpus 0001
[node1077:29623] [[53212,1],3] to cpus 0002
Summary

On NUMA systems like Janus, placement and binding of processes and their associated memory are important performance considerations.

Process Affinity and Memory Policy have a significant effect on MPI, OpenMP, and hybrid codes.

Simple `numactl` commands and APIs allow users to control affinity of processes and threads and memory assignments.

Future prospects for hybrid programming:

12-core and 16-core socket systems are on the way, so even more effort will be focused on process scheduling and data locality.
Example: Trap.cpp