Parallel Application
Scaling, Performance, and
Efficiency

NUG 2010
Supplemental Material
(David Skinner and Katie Antypas)
Overview

• Review Some Basic MPI
• Domain Decomposition
• Load Balancing
• Case Study: FLASH Scaling Performance
• Performance Monitoring with IPM
• Many-core chips and the future of parallel programming
Overview and History of MPI

• Library (not language) specification

• Goals
  – Portability
  – Efficiency
  – Functionality (small and large)

• Most basic communications are 2 sided

• Pros
  – Programmer has control at low level
  – Performance model understood
  – Can be very high performing

• Cons
  – Programmer has control at low level
  – Error prone
  – Questions about memory usage as cores/node increase
Generic Message Passing

send(void* sendbuffer, int num_elements, int destination_rank)

receive(void* recvbuffer, int num_elements, int source_rank)

Processor 0
x = 5
send(&x, 1, 1)
x = 7

Processor 1
receive(&x, 1, 0)
print x

What rules are needed so that processor 1 receives “5” and not “7”?"
## Ways to Send Data

<table>
<thead>
<tr>
<th></th>
<th>Blocking</th>
<th>Non-blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Non-Buffered</strong></td>
<td>Sending processor &quot;blocks&quot; or &quot;waits&quot; for receive</td>
<td>Sending processor completes call, but must be careful not to overwrite send buffer until receive operation has completed</td>
</tr>
<tr>
<td><strong>Buffered</strong></td>
<td>Sending process completes call after sendbuf has been copied to another buffer</td>
<td></td>
</tr>
</tbody>
</table>

*Buffered* and *Non-Buffered* are terms used to describe how data is sent. *Buffered* refers to the use of a buffer to hold data before it is sent, while *Non-Buffered* means the data is sent immediately without using a buffer.*
Be careful with buffering …

send(void* sendbuffer, int num_elements, int destination_rank)

receive(void* recvbuffer, int num_elements, int source_rank)

Processor 0
do i=1, 1000
  produce_data(&x)
  send(&x, 1, 1)
end do

Processor 1
do i=1, 1000
  receive(&x, 1, 0)
  consume(&x)
end do

What could go wrong with buffered send?

Example – John Mellor-Crummey, Rice University
Basic Point to Point

• Blocking – Non-buffered
  • MPI_Send()
  • MPI_Recv()

```c
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
    MPI_Send(&work, 1, MPI_INT, dest, TAG, MPI_COMM_WORLD);
} else {
    MPI_Recv(&result, 1, MPI_INT, src, TAG, MPI_COMM_WORLD, &status);
}
```
Non-Blocking Operations

• MPI_Isend()
• MPI_Irecv()
• “I” is for immediate
• Paired with MPI_Test() / MPI_Wait()
Non-Blocking Operations

MPI_Comm_rank(comm,&rank);

if (rank == 0) {
    MPI_Isend(sendbuf,count,MPI_REAL,1,tag,comm,&request);
    /* Do some computation */
    MPI_Wait(&request,&status);
} else {
    MPI_Irecv(recvbuf,count,MPI_REAL,0,tag,comm,&request);
    /* Do some computation */
    MPI_Wait(&request,&status);
}
Collective Operations

- May be layered on point to point
- May use tree communication patterns for efficiency
- Synchronization! (No non-blocking collectives)
Collective Operations

MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, comm);

O(P)  O(log P)
Quiz: MPI_Send()

• After I call MPI_Send()
  – The recipient has received the message
  – I have sent the message
  – I can write to the message buffer without corrupting the message

• I can write to the message buffer
Quiz: MPI_Isend()

• After I call MPI_Isend()
  – The recipient has started to receive the message
  – I have started to send the message
  – I can write to the message buffer without corrupting the message

• None of the above (I must call MPI_Test() or MPI_Wait())
Minimizing Latency

• Collect small messages together (if you can)
  – One 1024-byte message instead of 1024 one-byte messages
• Minimize other overhead (e.g., copying)
• Overlap with computation (if you can)
Example: Domain Decomposition
while (!done) {
    exchange(D, neighbors, myrank);
    dored(D);
    exchange(D, neighbors, myrank);
    doblack(D);
}

void exchange(Array D, int *neighbors, int myrank) {
    for (i = 0; i < 4; i++)
        MPI_send(...);
    for (i = 0; i < 4; i++)
        MPI_recv(...);
}
Naïve Approach

- Deadlock! (*Maybe*)
- Can fix with careful coordination of receiving versus sending on alternate processes
- But this can still serialize
while (!done) {
    exchange(D, neighbors, myrank);
    dored(D);
    exchange(D, neighbors, myrank);
    doblack(D);
}

void exchange(Array D, int *neighbors, int myrank) {
    for (i = 0; i < 4; i++) {
        MPI_Sendrecv(...);
    }
}
while (!done) {
    exchange(D, neighbors, myrank);
    dored(D);
    exchange(D, neighbors, myrank);
    doblack(D);
}

void exchange(Array D, int *neighbors, int myrank) {
    for (i = 0; i < 4; i++) {
        MPI_Isend(...);
        MPI_Irecv(...);
    }
    MPI_Waitall(...);
}
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Init</td>
<td>Initializes MPI</td>
</tr>
<tr>
<td>MPI_Comm_size</td>
<td>Returns number of tasks in communicator</td>
</tr>
<tr>
<td>MPI_Comm_rank</td>
<td>Returns ID of current processor</td>
</tr>
<tr>
<td>MPI_Send</td>
<td>Sends data</td>
</tr>
<tr>
<td>MPI_Recv</td>
<td>Receives data</td>
</tr>
<tr>
<td>MPI_Reduce</td>
<td>Reduce data to single processor</td>
</tr>
<tr>
<td>MPI_Allreduce</td>
<td>Reduce all processes data to all processes</td>
</tr>
<tr>
<td>MPI_Bcast</td>
<td>Broadcasts to all processes</td>
</tr>
<tr>
<td>MPI_Finalize</td>
<td>Closes MPI</td>
</tr>
</tbody>
</table>
Load Balancing
The Universal Parallel Science App

Unbalanced:

Task 1
Task 2
Task 3
Task 4

Balanced:

Task 1
Task 2
Task 3
Task 4

~All apps come down to the same basic pattern. Ok, Maybe there is no I/O.
Load Balance: real code
Load Balance: performance data

Communication Time: 64 tasks show 200s, 960 tasks show 230s

MPI ranks sorted by total communication time
while(1) {
    do_flops(N_i);
    MPI_Alltoall();
    MPI_Allreduce();
}

960  64
   x  x

960  64
   x  x

Load Balance: ~code
Load Balance : analysis

- The 64 slow tasks (with more compute work) cause 30 seconds more “communication” in 960 tasks
- This leads to 28800 CPU*seconds (8 CPU*hours) of unproductive computing
- All imbalance requires is one slow task and a synchronizing collective!
- Parallel computers allow you to scale both your computation and your load imbalance.
Load Balance : FFT

When is imbalance good?
Dynamical Load Balance:
Motivation

Time ➔

MPI Rank ➔

Flops
Exchange
Sync

Time ➔
Load Balance: Summary

• Imbalance most often a byproduct of data decomposition
• Must be addressed **before** further MPI tuning can happen
• Good software exists for graph partitioning / remeshing

• Dynamical load balance may be required for adaptive codes
Scaling Study
Scaling: definitions

• Scaling studies involve changing the degree of parallelism.

• Strong scaling
  – Fixed problem size, more computer

• Weak scaling
  – Problem size grows with concurrency
Parallel Performance Measurements

- **Speed up** = $T_{\text{serial}} / T_{\text{parallel}}(n)$
  - $T_{\text{serial}} = 100$ secs
  - $T_{\text{parallel}}(2) = 80$ secs
  - 25% speed up

- **Efficiency** = $T_{\text{serial}} / (n \times T_{\text{parallel}}(n))$
  - $100 / (2 \times 80) =$
  - 62% efficiency

- **Perfect Scaling?**
  
Be aware there are multiple definitions for these terms
FLASH Sedov 3d problem with Particles

FLASH Sedov 3d Strong Scaling

- total time
- hydro
- particles advance
- update refinement

Seconds

Processor Cores

384 512 1024 2048 4096 8192
FLASH Sedov 3d problem with Particles

Scaling of FLASH Sedov 3d

- Perfect Scaling
- Simulation Time

Seconds

Processors
Scaling: Analysis

• What is happening in the 8192 case?
  – Compute per core decreasing
  – Synchronization rate increasing
  – Surface to volume ratio increasing

• What else could happen?
  – Algorithmic scaling may change
  – Maybe we hit an architectural boundary in the machine (switch level, mid-plane, queue, etc.)
  – Maybe depleted some buffer space resource
  – Many more things…performance debugging at scale is detective work in the application + architecture space
Parallel programs are easier to mess up than serial ones. Here are a couple common pitfalls.
What’s wrong here?

Communication

% of MPI Time

Communication Event Statistics (100.00% detail)

<table>
<thead>
<tr>
<th>Event</th>
<th>Buffer Size</th>
<th>Ncalls</th>
<th>Total Time</th>
<th>Min Time</th>
<th>Max Time</th>
<th>%MPI</th>
<th>%Wall</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Allreduce</td>
<td>8</td>
<td>3278848</td>
<td>124132.547</td>
<td>0.000</td>
<td>114.920</td>
<td>59.35</td>
<td>16.88</td>
</tr>
<tr>
<td>MPI_Comm_rank</td>
<td>0</td>
<td>35173439489</td>
<td>43439.102</td>
<td>0.000</td>
<td>41.961</td>
<td>20.77</td>
<td>5.91</td>
</tr>
<tr>
<td>MPI_Wait</td>
<td>98304</td>
<td>13221888</td>
<td>15710.953</td>
<td>0.000</td>
<td>3.586</td>
<td>7.51</td>
<td>2.14</td>
</tr>
<tr>
<td>MPI_Wait</td>
<td>196608</td>
<td>13221888</td>
<td>5331.236</td>
<td>0.000</td>
<td>5.716</td>
<td>2.55</td>
<td>0.72</td>
</tr>
<tr>
<td>MPI_Wait</td>
<td>589824</td>
<td>206848</td>
<td>5166.272</td>
<td>0.000</td>
<td>7.265</td>
<td>2.47</td>
<td>0.70</td>
</tr>
</tbody>
</table>
Is MPI_Barrier time bad? Probably. Is it avoidable?
~three cases:
1) The stray / unknown / debug barrier
2) The barrier which is masking compute imbalance
3) Barriers used for I/O ordering
How to use IPM: basics

Many of the graphs in this talk were generated with a tool called IPM – Integrated Performance Monitoring: free and easy to install

http://ipm-hpc.sourceforge.net/

On galera

1) `>> mpicc test.c -lipm`
2) Run job as usual
3) Appended to your output file
How to use IPM: basics

## IPMv0.982

- **command**: unknown (completed)
- **host**: n9-1-6/x86_64_Linux
- **mpi_tasks**: 4 on 1 nodes
- **start**: 07/19/10/14:34:24
- **wallclock**: 0.013101 sec
- **stop**: 07/19/10/14:34:24
- **%comm**: 11.34
- **gbytes**: 0.00000e+00 total
- **gflop/sec**: 0.00000e+00 total

### region: *

- **ntasks**: 4

#### [total] <avg> min max
- **entries**: 4 1 1 1
- **wallclock**: 0.0453982 0.0113496 0.00918508 0.0131011
- **user**: 0.161974 0.0404935 0.035994 0.045993
- **system**: 0.123979 0.0309947 0.023996 0.034994
- **mpi**: 0.00594119 0.0014853 0.000761837 0.00176443
- **%comm**: 11.3372 5.82577 19.2098
- **gflop/sec**: 0 0 0 0
- **gbytes**: 0 0 0 0

#### [time] [calls] <mpi> <wall>
- **MPI_Allreduce**: 0.00343561 372 57.83 7.57
- **MPI_Recv**: 0.00194091 558 32.67 4.28
- **MPI_Send**: 0.000562498 558 9.47 1.24
- **MPI_Comm_size**: 1.21177e-06 4 0.02 0.00
- **MPI_Comm_rank**: 9.65199e-07 4 0.02 0.00
Profiling Codes Using IPM
An Unbalanced Code
A Balanced Code
Communication Patterns
Communication Patterns
P2P Topology Overview
Parallel Programming in the future with many-core architectures.

Can MPI everywhere survive?

Slides for John Shalf’s Future Architecture’s Talk
Basic Multi-core Compute Node Architecture
Trend #3: Multicore / Manycore

- Power density limit single processor clock speeds
- Cores per chip is growing
- Simple doubling of cores is not enough to reach exascale
  - Also a problem in data centers, laptops, etc.
- Two paths to exascale:
  - Accelerators (GPUs)
  - Low power embedded cores
  - (Not x86 clusters)
What’s Wrong with MPI Everywhere

• We can run 1 MPI process per core (flat model for parallelism)
  – This works now and will work for a while
  – But this is wasteful of intra-chip latency and bandwidth (100x lower latency and 100x higher bandwidth on chip than off-chip)
  – Model has diverged from reality (the machine is *NOT* flat)

• How long will it continue working?
  – Depends on performance expectations

• What is the problem?
  – Latency: some copying required by semantics
  – Memory utilization: partitioning data for separate address space requires some replication
    • How big is your per core subgrid? At 10x10x10, over 1/2 of the points are surface points, probably replicated
  – Memory bandwidth: extra state means extra bandwidth
  – Weak scaling: success model for the “cluster era;” will not be for the many core era -- not enough memory per core
  – Heterogeneity: MPI per CUDA thread-block?
However: MPI will likely persist

- Obviously MPI will not disappear in five years
- By 2014 there will be 20 years of legacy software in MPI
- Thus far, new systems are not sufficiently different to lead to new programming model
- MPI can evolve – (like Fortran, the Fortran from 50 years ago is very different from the Fortran used today)
Why use Hierarchical (hybrid) model for parallelism?

• The machine is not flat
  – We lose a lot of performance by lying to ourselves
• Target: Get Strong scaling on-chip and weak-scaling off-chip
  – 100x higher bandwidth between cores on chip
  – 100x lower latency between cores on chip
  – If you pretend that every core is a peer (each is just a generic MPI rank) you are leaving a lot of performance on the table
  – You cannot domain-decompose things forever (cannot weak-scale forever)
• Potentially MPI between nodes and X within node
  – Where X could be OpenMP, UPC, OpenCL, CUDA, etc…
What is X?

• X is it OpenMP?
  – Lots of synchronization
  – Poor expression of locality (will not scale)

• X might be UPC or PGAS language
  – Explicit definition of local vs. remote
  – Very lightweight communication

• X might be CUDA or OpenCL
  – OpenCL is very CUDA-like cross-platform extension to C language
  – CUDA is also being extended to also target multicore

• For all X
  – Define better way to express fine-grained parallelism on-chip
  – must rigorously determine semantics for interoperation with MPI
  – Must interoperate with numerical methods that target strong scaling
MPI+X: Requirements for X

• **Must be able to write once and run everywhere**
  – Cannot develop architecture-specific code
  – Don’t want to write code for each target! (just once please)

• **Needs to be ubiquitous**
  – Most people start a new code on a laptop and graduate to HPC systems
  – The complete development environment must be in both places (freely available)

• **Must emphasize ability to deliver strong-scaling on-chip to replace clock-frequency scaling**
  – Data parallelism might not be sufficient
  – We cannot rely on domain-decomposition for speed-up ad-infinitum (nothing to take up slack for CFL)
  – **Functional partitioning** (Happening at macro-scale with frameworks At micro-scale, requires bounded side-effects! it's not magic)
Summary

• **Strong scaling on chip**
  – Memory is shrinking per chip and clocks stalled
  – Solutions: UPC on-chip, OpenCL, domain-specific code-generation
  – Not-solutions: CUDA, OpenMP (not locality aware)

• **Weak scaling between chips**
  – Memory size is staying same per node
  – Probably MPI, but could be UPC, PGAS or other distributed memory locality aware models

• **Frameworks for managing big programming teams**
  – Should focus on modularity and agreement on interfaces
  – Benefits from functional semantics

• **Languages for fine-grained parallelism + correctness**
  – Defining exec model for fine-grained explicit parallelism is the challenge of our decade
More Info

• The Berkeley View/Parlab
  – http://view.eecs.berkeley.edu
  – http://parlab.eecs.berkeley.edu/

• NERSC System Architecture Group
  – http://www.nersc.gov/projects/SDSA

• LBNL Future Technologies Group
  http://crd.lbl.gov/ftg
NERSC is enabling new high quality science across disciplines, with over 1,600 refereed publications last year.