Supercomputing for Everyone Series: Performance Tuning Summer School

L8: Handmade parallelization hurts

Dr. Holger Brunst
Technische Universität Dresden, Germany

August 17–21, 2015
Organization of this class

- course runs August 17-21, from 11 a.m. until 4.30 p.m. (EDT)
- 20 sites, each site should have an instructor/TA/helper
- please put your microphones on mute, if you are not asking questions
- set up in lectures (60 min) and exercises (75 min)
- Q&A time at end of each lecture
- chat for discussions
- class material and links: http://go.iu.edu/CtB
Lunch breaks are important

Monday
11.00-12.30 L1_Performance is ambiguous
12.30-13.15 X1_First steps on Blue Waters
13.15-14.15 Lunch break
14.15-15.15 L2_Evaluation first!
15.15-16.30 X2_Pen, paper, and performance

Tuesday
11.00-12.00 L3_Use simple tools for simple questions
12.00-13.15 X3_The command line is your friend
13.15-14.15 Lunch break
14.15-15.15 L4_Tuning needs persistence
15.15-16.30 X4_Benchmarks provide the baseline

Wednesday
11.00-12.00 L5_Core tuning pays off n-times
12.00-13.15 X5_How to access data efficiently
13.15-14.15 Lunch break

Thursday
11.00-12.00 L7_Sharing may double the sorrow (OpenMP)
12.00-13.15 X7_OpenMP enables quick 'n easy gains
13.15-14.15 Lunch break
14.15-15.15 L8_Hand made parallelization hurts (MPI)
15.15-16.30 X8_From bad MPI to good MPI

Friday
11.00-12.00 L9_Climb the mount Olympus with GPUs
12.00-13.15 X9_Hybrid tuning in practice
13.15-14.15 Lunch break
14.15-15.15 L10_The grand final
15.15-16.30 X10_Useful bits and pieces

* All times are EDT
Goal

- briefly introduce MPI
- discuss MPI patterns, that indicate bad performance
- make general recommendations and “good” MPI usage
Early classics on MPI

- Using MPI, by William Gropp, Ewing Lusk, and Anthony Skjellum
- Designing and Building Parallel Programs, by Ian Foster
- Parallel Programming with MPI, by Peter Pacheco
- High Performance Computing, 2nd Ed., by Dowd and Severence
Warning: where not to use MPI

- you can use OpenMP or POSIX threads
- you don’t need parallelism at all
- you can use libraries (which may be written in MPI)
- you need simple threading in a slightly concurrent environment
Hand made distribution of work
An introduction to message passing

directives-based parallelism

• High Performance Fortran (HPF), OpenMP
• serial code + directives
• compiler distributes the data and work
• hides details about distribution and communication hidden
• usually shared memory

```c
#pragma omp for nowait
for ( k = 0; k < m; k++ ) {
    fn10(k); fn20(k);
}
#pragma omp sections private(y, z)
{
    #pragma omp section
        { y = sectionD(); fn70(y); }
    #pragma omp section
        { z = sectionC(); fn80(z); }
}
```
An introduction to message passing

**explicit parallelism**

- MPI
- explicit parallel code
- programmer encodes the data and work distribution
- manage the communications
- approach is very flexible
- supports distributed memory
Distributed memory

Processor
Memory

Processor
Memory

Processor
Memory

Processor
Memory

...
The message passing model is extremely general

• parallel computation
  – number of processes
  – work on local data only
  – do direct access between processes

• sharing of data
  – message passing
  – sending and receiving of data
Designing an explicit parallel code has issues

- load balancing
  - goal
    - maximize utilization of the resources
    - make uniform progress
    - avoid waiting time at points of exchange

- equally dividing the work
  - easy when all processes carry out same work
  - complicated when processing time depends on data
If one guy is late, others have to wait
Blue is good on 200,000 cores
Minimize the communication

- serial codes do not communicate!
  - speedup
    - “sequential fraction”
- minimizing communication
  - computation time (max)
  - idle time (min)
  - communication time (min)
Communication overlapping hides latencies

- overlapping communication and computation
  - hides latencies
  - do something useful while exchanging data in the background
  - sounds logic but is very complicated in practice
- error prone
  - explicit communication is subject to human errors
MPI is the de facto standard for message passing in HPC
MPI enables portable, efficient, and parallel code

- MPI stands for “Message Passing Interface”
- library of communication functions (# > 400)
- MPI forum
- source code portability
- efficiency across a range of architectures
MPI provides three types of communication

- point-to-point
- one-sided
- collective

- various attributes
  - blocking, non-blocking, synchronous, ...
Chatting over the fence

- point-to-point
- two active parties
- explicit send and receive calls
- if one is missing or late
  - hick ups
One-sided: a bit like sending letters to a mailbox

- one-sided communication
- a.k.a. remote memory access
- put and get
- only one active party
- non-blocking

- can avoid buffering
- handshaking
Processes communicating in a collective

- collective
  - needs to be defined
  - must be complete
- types
  - synchronization
  - data movement
  - collective computation
- traditionally blocking
- why not peer-to-peer?
Bad MPI communication. This is what you try to avoid.
Waiting at a barrier

Time spent waiting in front of a barrier until all members have reached the barrier.
Barrier completion

Time spent in barrier after the first member left the operation.

Is a property of the MPI implementation.
Late Broadcast

MPI_Bcast

MPI_Bcast (root)

MPI_Bcast

MPI_Bcast
Early Reduce
Late Sender
Late Sender 2
Late sender, wrong order
Analysis results in three columns (Scalasca)

<table>
<thead>
<tr>
<th>Metric</th>
<th>Call Site</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00 Time</td>
<td>0.00 main</td>
<td>- IBM AIX POWER (32-bit)</td>
</tr>
<tr>
<td>131.30 Execution</td>
<td>0.00 Init</td>
<td>- jump06m</td>
</tr>
<tr>
<td>0.00 MPI</td>
<td>0.00 MPI_Init</td>
<td>4.99 Process 0</td>
</tr>
<tr>
<td>0.00 Synchronization</td>
<td>0.00 MPI_Init</td>
<td>0.60 Process 1</td>
</tr>
<tr>
<td>0.00 Collective</td>
<td>0.00 MPI_Init</td>
<td>0.03 Process 2</td>
</tr>
<tr>
<td>0.00 Communication</td>
<td>0.00 MPI_Init</td>
<td>0.05 Process 3</td>
</tr>
<tr>
<td>0.29 Point-to-point</td>
<td>0.00 MPI_Init</td>
<td>0.12 Process 4</td>
</tr>
<tr>
<td>5.95 Collective</td>
<td>0.00 MPI_Init</td>
<td>0.04 Process 5</td>
</tr>
<tr>
<td>0.00 File I/O</td>
<td>0.00 MPI_Init</td>
<td>0.04 Process 6</td>
</tr>
<tr>
<td>0.95 Init/Exit</td>
<td>0.00 MPI_Init</td>
<td>0.06 Process 7</td>
</tr>
<tr>
<td>0.34 Overhead</td>
<td>0.00 MPI_Init</td>
<td>0.09 Process 8</td>
</tr>
<tr>
<td>5.21e4 Visits</td>
<td>0.00 MPI_Init</td>
<td>0.10 Process 9</td>
</tr>
<tr>
<td>5.94 MPI_Allreduce</td>
<td>0.00 MPI_Init</td>
<td>0.11 Process 10</td>
</tr>
</tbody>
</table>

- IBM AIX POWER (32-bit)
MPI needs to be treated like a difficult diva

OUTLINE
Tuning for distributed computing

- aggregation
- decomposition
- load Balancing
- changing the algorithm
Aggregation reduces latency

• collect many small messages into a single large message
• example: sending components separately vs. sending them together
  – packing cost (usually) outweighed by latency
• cache effects suggest keeping components together
• use of collective when many copies bcast/gather
Aggregation in collective operations

- combining collective operations
- much cheaper to do one 2-element allreduce than two 1-element allreduce
Issues in choosing a decomposition

- 1 vs 2 vs 3d
- minimize surface to volume (latency vs. bandwidth tradeoff)
- more complex decompositions (e.g., hexagons in 2-d) possible, but usually not worth it
- careful with relatively small problems - latency may dominate
Changing the algorithm

- some algorithms are just not good candidates for parallelization
- If an approximation, another approximation may be a better choice (different physical model)
- If part of an iterative method, another iteration may be better (different numerical model)
- Need not be less efficient
Trading communication for computation

- For slowly converging algorithms, taking a number of steps and then checking convergence (rather than checking at each iteration) is another form of blocking.
- Can trade bandwidth/computation for latency (unroll a compute loop once, do a single send of more data, do duplicate computation).
Aggregation in loop unrolling

- Blocking is another form of loop unrolling
- Loop unrolling can be used to get longer messages

```do timestep=1,n
    compute
    exchange
  do timestep=1,n,2
    compute(timestep) (and some of timestep-1)
    compute(timestep+1)
    exchange 2 timesteps```
Load balancing

- Small amounts of work imbalance lead to large losses in performance
- First question is: is load balancing central to the algorithm or part of performance tuning?
- Central - master/slave models, multilevel work masters
- Tuning - decomposition tuning
Identifying load imbalances

- Identifying (distinguishing from latency/synchronization overhead)
- Poor load balance focuses attention on collective operations because the implicit synchronization of the collective operation “equalizes” the time taken on each process
- Can generate the appearance of good load balance if not timed correctly
Different send/recv modes

- MPI provides many different ways to perform a send/recv
- Choose different ways to manage buffering (avoid copying) and synchronization
Scheduling for contention

- Many programs alternate between communication and computation phases
- Contention can reduce effective bandwidth
- Consider restructuring program so that some nodes communicate while others compute
MPI implementation parameters

- Each MPI implementation makes different choices about protocols, allocation of memory, etc
- Some of these parameters can be adjusted for each run
Wrap-up

- Select algorithm for effective parallelism
- Choose effective data decompositions
- Tune to problem sizes of interest
- Look carefully at parts of code that do not scale with p
- Model and measure
- Look for underperforming sections of code
Wrap-up

- Apply aggregation to reduce latency
- Increase grain, combine elements
- Unroll loops, block operations
- Manage message flow to avoid handshake delays
- Use MPI message modes
- MPI-2 features (one-sided, I/O)
QUESTIONS?

https://connect.iu.edu/ptune15
Class evaluation

• we appreciate your opinion and feedback
• please consider filling-in our class evaluation questionnaire

https://www.surveymonkey.com/r/ptune15