Introduction to Performance Analysis for HPC

LLNL-PRES-508651

Analysis Process: Step by Step

1. Optimize single-node code
2. Fix any load imbalance – consider decomposition and rank order
3. Fix your hotspots
    1. Communication
        - Pre-post receives
        - Overlap computation and communication
        - Reduce collectives
        - Adjust MPI environment variables
        - Use rank reordering
    2. Computation
        - Examine the hardware counters and compiler feedback
        - Adjust the compiler flags, directives, or code structure to improve performance
        - Try other compilers (if available)
    3. I/O
        - Stripe files/directories appropriately
        - Use I/O methods that scale
            - MPI-IO or Sub-setting

At each step, check your *results* **and** *performance*.
Between each step, gather your data again.

Analysis Process

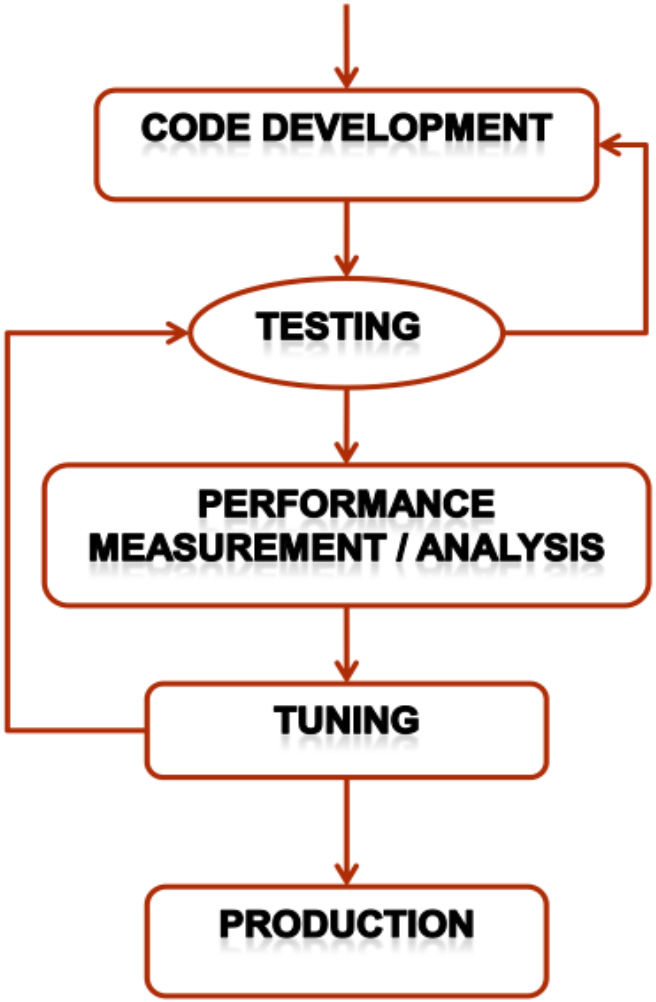
- Tabulate Application & Architecture characteristics
- Determine whether application is:
  - Computation-bound
  - Memory-bound
- Determine application communication characteristics
- Benchmark application on system
  - use test and production data sets
  - Weak-scaling study
  - Strong-scaling study
- Identify limits to scalability
  - Single-node performance
  - Communication
  - I/O

Analysis Process

- Levels of Parallel Performance
  - Inter-Node
    - Message-passing optimization
- Intra-Node
  - Core
  - Vector – vector width, SIMD instructions
  - Pipeline – support for oo instructions, multiple pipes
  - Instruction - max # in-flight instructions
  - Multiple-core optimizations

Basic of Performance Evaluation

- Determine what types of data you need and how much performance data you are willing to (or need to) consume.
- Select a performance tool to instrument your code.
- Submit your jobs.
- View the text report, or use a visual tool (usually comes with the performance tool you pick) to observe the data.



MPI Performance Issues

MPI Performance Issues

- Know your MPI
  - Understand features of specific MPI implementation
    - Focus on routines where most time is spent
    - Understand how MPI library environment variables affect performance
- xxx

MPI - Short Message Eager Protocol

- Sending rank "pushes" message to receiving rank
  - Sender assumes receiver can handle message and blindly transmits to it
- If matching receive is posted, receiver
  - routes incoming data directly into specified receive buffer
  - posts notification event to other event queue
- If no matching receive is posted, receiver
  - routes incoming data into unexpected message buffer
  - posts two events to unexpected event queue
  - copies data into specified receive buffer when matching receive is posted

MPI - Long Message Rendezvous Protocol

- Receiving rank "pulls" message from sending rank
- Sender notifies receiver about waiting message via a small header packet
- Receiver requests message from sender after matching receive is posted
- Receiver routes incoming data directly into specified receive buffer

MPI - Long Message Eager Protocol

- Sender assumes receiver will handle message appropriately or will request retransmission
  - Sender blindly transmits data to receiver
- If matching receive is posted, receiver
  - routes incoming data directly into specified receive buffer
  - sends completion acknowledgement to sender
- If no matching receive is posted, receiver
  - creates a long protocol match entry
  - requests retransmission when matching receive is posted
  - routes incoming data directly into specified receive buffer

MPI Environment

MPI Environment Variables

- Many environment variables are available to tune MPI performance
  - Usually documented on the MPI man page – Read it!
  - Default settings generally focus on attaining the best performance for "most" cases – not necessarily your application!
  - May need to experiment to find optimum settings for application, data set

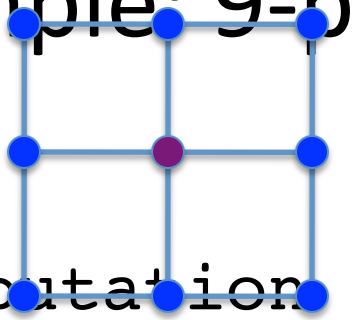
MPI - Rank Placement

- In some cases, changing how the processes are laid out on the machine may affect performance by relieving synchronization/ imbalance time.
- Often default is SMP-style placement. This means that for a multi-node core, sequential MPI ranks are placed on the same node.
  - In general, MPI codes perform better using SMP placement - Nearest neighbor
  - Collectives have been optimized to be SMP aware
- Check your local MPI documentation for options

MPI Programming Techniques
Pre-posting receives

- If possible, pre-post receives before the matching sends
  - Optimization technique for all MPICH installations (not just MPT)
  - Not sufficient to simply put receive immediately before send
  - Put significant amount of computation between receive-send pair
- Do not go crazy pre-posting receives. You will overrun the resources available MPI.
- Code example (From Glenn Brook)
  - Halo update – with four buffers (n,s,e,w), post all receive requests as early as possible. Makes a big difference on CNL.

MPI Programming Techniques
Example: 9-pt stencil pseudo-code



Basic

```
9-pt computation
Update ghost cell
  boundaries
    East/West IRECV,
      ISEND, WAITALL
    North/South IRECV,
      ISEND, WAITALL
```

Maximal Irecv preposting

```
Prepost all IRECV
9-pt computation
Update ghost cell
  boundaries
    East/West ISEND,
      Wait on E/W IRECV
      only
    North/South ISEND,
      Wait on the rest
```

*Makes use of temporary buffers

MPI Programming Techniques Overlapping communication with computation

- Use non-blocking send/recvs to overlap communication with computation whenever possible
  - Typical pattern:
    1. Pre-post non-blocking receive
    2. Compute a “reasonable” amount to ensure effective pre-posting
    3. Post non-blocking send
    4. Compute as much as possible to maximize overlap of comm. and comp.
    5. Wait on communication to finish only when absolutely necessary

MPI Programming Techniques Overlapping communication with computation

- In some cases, it may be better to replace collective operations with point-to-point communications to overlap communication with computation
  - Caution: Do not blindly reprogram every collective by hand
  - Concentrate on the parts of your algorithm with significant amounts of computation that can overlap with the point-to-point communications when a [blocking] collective is replaced

MPI Programming Techniques Reduce Collective Communications

- Avoid using collective communications whenever possible
  - MPI collectives are blocking, leading to large sync times
  - Collective communication can cripple scalability
- Use algorithms that only require local data where possible
  - Consider duplicating computation to reduce communication
- When an algorithm must communicate “globally”:
  - Use MPI collectives that have been optimized in library
  - Minimize the scope of the collective operation
  - Minimize the number of collectives through aggregation
  - Consider implementing a non-blocking collective only if justified after careful analysis

MPI Programming Techniques
Aggregating data

- For very small buffers, aggregate data into fewer MPI calls (especially for collectives)
  - 1 all-to-all with an array of 3 reals is clearly better than 3 all-to-alls with 1 real
  - Do not aggregate too much. The MPI protocol switches from a short (eager) protocol to a long message protocol using a receiver pull method once the message is larger than the eager limit. The optimal size for messages most of the time is less than the eager limit.
- Example – DNS
  - Turbulence code (DNS) replaced 3 AllGatherv's by one with a larger message resulting in 25% less runtime for one routine

MPI Programming Techniques
Aggregating data: Example from CFD

Original

```
    for (index = 0; index < No; index++){
       double tmp;
       tmp = 0.0;
       out_area[index] = Bndry_Area_out(A, labels[index]);
       gdsum(&outlet_area[index],1,&tmp);
    }
    for (index = 0; index < Ni; index++){
      double tmp;
      tmp = 0.0;
      in_area[index] = Bndry_Area_in(A, labels[index]);
      gdsum(&inlet_area[index],1,&tmp);
    }


void gdsum (double *x, int n, double *work)
    {
      register int i;
      MPI_Allreduce (x, work, n, MPI_DOUBLE, MPI_SUM,
    MPI_COMM_WORLD);
      /* *x = *work; */
      dcopy(n,work,1,x,1);
      return;
    }
```

Improved

```
       for (index = 0; index < No; index++){
          out_area[index] = Bndry_Area_out(A, labels
    [index]);
       }

       /* Get gdsum out of for loop */
       tmp = new double[No];
       gdsum (outlet_area, No, tmp);
       delete tmp;

       for (index = 0; index < Nin; index++){
         in_area[index] = Bndry_Area_in(A, labels[index]);
       }

       /*  Get gdsum out of for loop */
       tmp = new double[Ni];
       gdsum(inlet_area, Ni, tmp);
       delete tmp;
```


Hybrid – MPI + OpenMP

MPI + OpenMP

- When does it pay to add/use OpenMP in my MPI application?
  - Add/use OpenMP when code is network bound
  - As collective and/or point-to-point time increasingly becomes a problem, use threading to keep number of MPI processes per node to a minimum
  - Be careful adding OpenMP to memory bound codes – can hurt performance
  - Be careful to match memory affinity to thread affinity
    - Pre-touch memory from correct thread after allocation
  - It is code/situation dependent!
  - Consider one MPI process on each CPU and one OpenMP thread per available core within each process
    - Often gives results almost as good as a fully optimized one-process-per-node code (with OpenMP threads across all of the cores on the node) with significantly less development overhead

Closing Remarks

Summary

- Vendor MPI libraries provide optimized, high-performance communication
  - Sometimes requires guidance and tuning – also patience and perseverance
  - Must understand effect of default parameter choices on performance
- Environment variables may be easy way to improve performance
  - Familiarize yourself with 'man mpi' and remain up-to-date
- The is no replacement for good MPI programming practices
  - Pre-posting receives, overlap computation and communication, reduce collective communications, aggregate data for communication
- Rank reordering may significantly improve performance
- Remember your parallel I/O – it can be crippling
- Some of this may not show a benefit at <1K processes, but it can reap huge gains at 10K to 100K processes
- Thanks to Jeff Larkin , Glenn Brook for permission to use their slides

Summary (continued) - Input/Output

- Sometimes I/O causes scalability issues
  - For example, cleaning up some writes improved weak scaling of the CFD code NektarG from 70% to 95% at 1K to 8K cores
- Set file striping appropriately
  - The default stripe count will almost always be suboptimal
  - The default stripe size is usually fine.
  - Once a file is written, the striping information is set
    - Stripe input directories before staging data
    - Stripe output directories before writing data
  - Stripe for your I/O pattern
    - Many-many – narrow stripes Many-one – wide stripes
- Reduce output to stdout
  - Remove debugging reports in production runs (e.g. "Hello from rank n of N")

References

- Vendor Documentation
  - manuals – invaluable source of information
  - man pages
- High Performamnce Computing, J. Levesque & G. Wagenbreth
- Intoduction to High Performance Computing for Scientists and Engineers, G. Hager & G. Wellein, 2011
- Performance Tuning for Scientific Applications, D. bailey, R. Lucas & S. Williams