Bottlenecks in Parallel Applications

Peer into the complexity of Parallel Applications and their serial components

Understand the source of bottlenecks in message passing

Methods & Tools of discovering and capturing performance bottlenecks

Pascal Paschos
pascal.paschos@northwestern.edu
Generic Definition

In a nutshell a bottleneck is a portion of the code that inhibits performance of the overall application.
Amdahl’s law speedup = $1/[(1-P)+P/N]$
Large I/O data operations, like staging, output streaming, lookup tables, precision, check-pointing

Third party libraries and software linked to the application

Typical serial program bottlenecks

Array over-allocation, inefficient, non-cohesive memory use, matrix operations and solvers

Inefficient sorting and indexing algorithms. Abuse of if-then statements, poorly constructed do-loops, complex, nested loops, etc …

Hardware access limits due to latency bandwidth affecting L1/L2… cache, memory and disk read/write

Compiler options, optimization, vendor
Serial bottlenecks can be addressed in the following ways

• Use compiler optimization to increase performance, efficiency and speed
  • Pay attention to warnings during to compilation. They provide valuable insight to portions of the code that might create issues under a set of circumstances
• Upgrade your compilers if old
  • Be suspicious of parts of code that fail compilation when either a newer or different compiler is used or a higher optimization level is used
• Acquire programming experience in designing efficient codes in terms of work flow, memory allocation and handling
• Utilize smart algorithms available through computing community portals
• Run the tests for third party software in order to understand any performance issues that might be intrinsic to them
• Use supporting libraries already installed and optimized on the computing machine
• Collaborate with people that have computing experience
Examples of programming ‘tricks’ to increase speed and performance

**Achieved by optimization**
- Eliminate unnecessary temporary variables
- Loop expansion
- Choose faster operators
- Function inlining

**User dependent**
- Use lookup tables instead of function calls
- Read/write blocks of data from and to files
- Use of vector languages such as Fortran 90 can increase speed and lower complexity (i.e. array slicing) by substituting expensive explicit loops and indexing with direct array operations better suited for implicit parallelization and data manipulation *
Need for parallelization

- Speed. Production turnaround. Problem of the same size needs to execute faster
- Size. Problem needs to fit in distributed memory.
- Addition of physics packages and increased complexity
- Multiple data processing single program, i.e. embarrassingly parallel
- Problem executes at the same time but has increased domain size. Bigger size problem can match real world situations
- Problem has fixed size but attempts higher resolution more closely related to real world (physical) situations
- Dynamically spawned instances like adaptive refinement
Types of Domain Decomposition

General def. any data decomposition

Specific: The domain is the domain of an operator equation or function space

Subdomains may have the same or different dimensionality as the original i.e. ghost zones

Partitioning into tasks

Assign tasks to processes

Orchestrate processes i.e. setup synchronization points, communication pattern etc

Map processes onto processors
Decomposition of \( \hat{S}\Psi = \phi \) in physical domain \( \Omega \)

**Operator space**

\[ \hat{S} = \sum_k \hat{S}_k \]

Global communication (transpose)

**Function space**

\[ \phi = \sum_k \phi_k U_k \]
\[ \Psi = \sum_k \Psi_k U_k \]

Global communication (transform)

**Physical domain (Krylov/Schwarz)**

\[ \Omega = \sum_k \Omega_k \]

Local surface data exchanges plus reduced global communication
Serial I/O operations
- Parallel I/O operations (MPI/IO)
- Utilization of parallel filesystems (GPFS, Lustre)
- parallel high level I/O libraries like HDF5 and netCDF

Instruction Set
- Added instruction set complexity due to message passing directives
- Checks for concurrent computation, communication, synchronization frequency and resource utilization (load balancing)

On-node computation (vertical memory structure)
- Intra-node communication which remotely accesses horizontal memory structure
- network topology (bus, star, torus, mesh, hypercubes)
- Message passing efficiency suffer from latency and bandwidth restrictions
Importance of network topology mapping

- Topology aware of task mapping
- Placing frequently communicating tasks i.e. controlling interfaces (ghost zones) on nodes in physical proximity
- “Nearest neighbor communication” of the domain is mapped on “nearest neighbor message passing” on the network minimizing latency and maximizing bandwidth use
- Objective is to minimize ‘hop- bytes’ the aggregate number of hops between source to destination for all tasks
- A topology manager can query communication mapping at runtime
- An example is the portable hardware locality (hwloc) tool available through OpenMPI
- Information of application-to-network mapping can also be obtained by performance event-collector tools such as PAPI, Perfsuite, IPM and others
MPI include file

*Declarations, prototypes, etc.*

Program Begins

- Serial code

Initialize MPI environment

*Parallel code begins*

Do work & make message passing calls

Terminate MPI environment

*Parallel code ends*

- Serial code

Program Ends
Blocking Point-to-Point Communication

- Returns when system buffer is safe to use again
- For every `MPI_Send()` there must be a `MPI_Recv()`
- Avoid sending/receiving at the same time with blocking calls. Instead, use special communication patterns e.g. checked, odd-even …
- Care should be given to deadlock creation when code is scaled up
  - Significant overhead associated with time waiting for Recv’s to get data. Pending communication consumes system memory resources.
  - A deadlock is lack of matching Recv to requesting Send
  - Some MPI libraries will let deadlock code run until message size reaches the `MP_EAGER_LIMIT`

**Standard send (MPI_SEND):** Synchronous or Buffered depending on buffer size and `MP_EAGER_LIMIT`

**Synchronous send (MPI_SSEND):** Only completes when the receive has started. High latency, best bandwidth. Risk of idle times, serialization, deadlocks

**Buffered send (MPI_BSEND):** Low latency, low bandwidth

**Ready send (MPI_RSEND):** Use only if the logical flow of your parallel program permits it, i.e. if it is guaranteed that the receiving process is ready for the message
Blocking versus non-blocking point-to-point operations MPI_Isend()…

- ‘I’ stands for immediate communication
- Routine returns before communication completes
- A blocking send can be used with a non-blocking receive and vice versa
- Application can perform work while waiting
- Non-blocking sends can use any mode, just like the blocking counterparts
- Synchronous mode refers to the completion of the send not to the initiation
- A non-blocking operation immediately followed by a matching MPI_WAIT() is equivalent to the blocking operation
- Initiation of communication should be placed as early as possible
- Synchronization/completion should be placed as late as possible
- Buffer must NOT be accessed before communication is confirmed
One-sided communication – MPI-2 compliant

• Two-sided communication needs two explicit communication calls

• Problem: target does not know it needs to participate in a data transfer in advance therefore the problem requires dynamically spawned processes

• Solution: An one-sided data transfer from origin to target

• Target process provides access to its memory through a defined window

• Accessing is called MPI RMA (remote memory access) operations
A few tips to increase communication performance in parallel codes.

- Avoid collective calls: MPI_Barrier() MPI_Alltoall() MPI_Allreduce() MPI_Allreduce()
- Aggregate messages up to the defined MP_EAGER_LIMIT
- Prefer MPI_Bcast() rather than MPI_Scatter()
- Replace MPI_All*() with MPI_*() + MPI_Bcast()
A few **more** tips to increase communication performance in parallel codes.

- Place receives before sends
- Avoid MPI_Iprobe() and MPI_Probe
- Use non-blocking calls to overlap communication with computation
- Choose compute over communication if latter has excessive overhead
- Load balancing – distribute the work equally between the tasks
- RMA should be avoided if possible
- Non-contiguous data communication has significant overhead. Choose to pack beforehand otherwise MPI will do that for you on the source and target. Use derived MPI datatypes.
Load Balancing

• Non-concurrent synchronization leading to one worker idling while other are doing useful work. Under-utilization of resources
• Equally partition the work each task receives
• For array/matrix operations evenly distribute the data set among the tasks
• For loop iterations evenly distribute the iterations across the tasks
• Use performance analysis tools to detect any load imbalances
• Use dynamic work assignment. Certain classes of problems result in load imbalances even if data is evenly distributed among tasks
  • Sparse arrays - some tasks will have actual data to work on while others have mostly "zeros"
  • N-body simulations - where some particles may migrate to/from their original task domain to another task's; where the particles owned by some tasks require more work than those owned by other tasks
  • Adaptive grid methods - some tasks may need to refine their mesh while others don't.
• Use a scheduler - task pool approach. As each task finishes its work, it queues to get a new piece of work
• Design or use an algorithm which detects and handles load imbalances as they occur dynamically within the code. Adaptive MPI (AMPI) – built around MPI-2 - supports dynamical load balancing but is NOT part of the standard.
Granularity

- The measure of the ratio in the time spent in computation versus communication
- Fine Grain: small granularity facilitates load balancing at increased communication overhead and less performance optimization
- Coarse Grain: large granularity leads to harder load balancing but increased performance enhancement

Scalability

- Fill up memory of arbitrarily large machines to increase resolution, while preserving nearly constant running times with respect to proportionally smaller problem on one processor
- Strong Scaling: execution time is inversely proportional to the number of processors for a fixed size problem
- Weak Scaling: execution time remains constant as problem size and processor count increases in proportion for a fixed size per CPU (Gustafson scaling)
Profiling: Aggregates statistics at run time -- e.g., total amount of time spent in MPI, total number of messages or bytes sent, etc. Data volumes are small.

Tracing: Collect an event history. It is common to display such event history on a timeline display. Tracing data can provide much interesting detail, but data volumes are large.
Application Instrumentation Objectives

- Capture messages during execution (tracing)
- Debugging
- Obtain Hardware Performance counters e.g. clock cycles, cache misses …
- Data logging of major events at runtime of the application.
Types of Instrumentation

• Manual instruction insertion to explicitly calculate runtimes and collect count events
• Automatic source level insertion by a tool i.e. TAU
• Direct instrumentation of a binary – compiled library
• Compiler assisted: Example: "gcc -pg ..." for gprof, "quantify g++ ..." for Quantify
• Instrumentation at runtime
• Runtime injection: Code is interrupted at runtime to jump to helper functions
MPI Standard Profiling interface

There is a corresponding PMPI routine for each MPI routine, with the same functionality. Therefore, the user can replace an MPI routine with own routine that does profiling

MPI_Send(buffer,count,datatype,...) 
{
<tool specific action>
PMPI_Send(buffer,count,datatype,...); 
<more tool specific action>
}

Control of the PMPI interface is handled by a dedicated routine
MPI_Pcontrol(int level,...)
level = 0: no profiling
level = 1: default level
level = 2: verbose

Compile with mpicc into libmyprof.c

Link mpicc –lmyprof –lpmpi –lmpi ...
An example of tracing each communication operation:
Trace time spent and total data exchanged

```c
MPI_Init(int argc, ...) {
    prof_sent = 0; prof_sendtime = 0.0;
    prof_recv = 0; prof_recvtime = 0.0;
    <and other initialization...>
    PMPI_Init(argc,argv);
}
```

```c
MPI_Send(buffer, count, datatype,...) {
    MPI_Type_size(datatype,&typesize);
    prof_sent += count*typesize;
    start = PMPI_Wtime();
    PMPI_Send(buffer,count,datatype,...);
    prof_sendtime += (PMPI_Wtime()-start);
}
```

```c
MPI_Finalize() {
    if (rank==0) {
        <gather results from all processes>
        for (i=0; i<size; i++) {
            fprintf(stderr,"Proc %i:\n"); fprintf
            (stderr,"Sent %d\n",proc_sent_all
            [i]); ... 
        } else { ... 
    }
    PMPI_Finalize(); }
```
Hardware Locality (HWLOC) is a C API that consists of command line tools that can access information about the machine: NUMA memory nodes, shared caches, processor sockets, processor cores, processing units.

HWLOC can collect and assemble the machine fabric topologies so that applications can consult it. This allows it to offer affinity information about the CPUs, the network interface, and/or GPUs. It enables better I/O data transfer by ensuring that processes and data are placed on the host part that is closer to devices.

The hwloc command-line tool "lstopo" produces human-readable topology maps that may be consulted for efficient task mapping.

http://www.open-mpi.org/projects/hwloc/
Performance API – Tracks Hardware Events – signals on the processor

Provides consistent high-level interface (C and Fortran) to CPU and GPU (CUDA) performance data

High level functions (out of 60 low level) – Insert into code

- PAPI_library_init()
- PAPI_num_counters()
- PAPI_flops()
- PAPI_read_counters()
- PAPI_start_counters()
- PAPI_query_event()

Link to PAPI library with “-lpapi”

http://icl.cs.utk.edu/papi  Current Version PAPI 3 OS should be patched with Perfctr

- Cycle count
- Instruction count
- Floating point instruction count
- Pipeline stalls
- L1/L2 cache hits/misses
- TLB misses
- Hardware interrupts
TAU performance evaluation tool
http://tau.uoregon.edu/

It supports parallel profiling and tracing

TAU performs automatic source code instrumentation using a package called PDT. It can automatically instrument routines, loops, I/O, memory, etc.

Profiling and tracing can measure time as well as hardware performance counters from your CPU – coupled with PAPI

TAU has instrumentation, measurement and analysis tools

paraprof is a 3D profile browser for viewing TAU’s output

TAU runs on all HPC platforms and it is free (BSD style license)

Automatic source instrumentation is enabled through setting environmental variables and substitute the name of your compiler with a TAU shell script
export TAU_MAKEFILE=$TAULIBDIR/Makefile.tau-papi-mpi-pdt-pgi

export TAU_OPTIONS= ‘-optVerbose …’ (see tau_compiler.sh -help)

Use tau_f90.sh, tau_cxx.sh or tau_cc.sh as the Fortran, C++ or C compilers:

mpif90 foo.f90 changes to tau_f90.sh foo.f90

Execute application and analyze performance data:

- pprof (for text based profile display)
- paraprof (for GUI)
Supports both direct and indirect performance observation

- Direct instrumentation of program (system) code (probes)
- Indirect mode supports sampling based on periodic timer or hardware performance counter overflow based interrupts

Support for standard program events

- Routines, classes and templates
- Statement-level blocks and loops
- Begin/End events (Interval events)

Support for user-defined events

- Begin/End events specified by user
- Atomic events (e.g., size of memory allocated/freed)
- Flexible selection of event statistics
Rudimentary Profilers

- Compile with `–p` option to obtain runtime statistics in `mon.out.taskid`. View statistics with `prof –m mon.out.*`
  - The percentage of the program's CPU time used by each procedure
  - The execution time in seconds for all references by each procedure.
  - The number of times the procedure was called.
  - The average time in milliseconds for a call to each procedure.
  - The parent of each procedure.
    - An index number for each procedure
    - The percentage of CPU time taken by that procedure and all procedures it calls (the calling tree)
  - A breakdown of time used by the procedure and its descendent
  - The number of times the procedure was called
  - The direct descendants of each procedure.
PGPROF MPI Profiler – part of PGDBG

- Feature: It supports profiling PGI Accelerator and CUDA Fortran codes on NVIDIA CUDA-enabled GPU accelerators
- MPI and thread communication profiling
Performance Analysis and Tuning Tools

• Tools NOT for production but for development due to scalability limit caused by the tool’s generation of unmanageably large trace files.
• KOJAK http://www2.fz-juelich.de/jsc/kojak/
• Vampir: http://vampir.eu
• Perfsuite: http://perfsuite.ncsa.illinois.edu/
• Scalasca: http://www.scalasca.org/
• mpiP: http://mpip.sourceforge.net/
• TAU: http://www.cs.uoregon.edu/Research/tau
• MPE: http://www.mcs.anl.gov/research/projects/perfvis/download
• SvPablo: http://www.renci.org/focus-areas/project-archive/pablo
• FPMPI: http://www.mcs.anl.gov/research/projects/fpmpi/WW
• IPM: http://ipm-hpc.sourceforge.net/ (NERSC/SDSC)
• Autopilot: http://www.tierdata.com/products/autopilot-platform.html
Sub Parallel Region

OpenMP directives

Parallel Region

MPI directives