Parallel Performance Analysis

Rosa Filgueira
Data Intensive Research, School of Informatics, University of Edinburgh

rosa.filgueira@ed.ac.uk
Goals

- Overview of MPI programming
- Performance problems
- Performance analysis
- Performance tools suite
Parallel Architectures

- **Distributed Memory**
  - Explicit data distribution
  - Explicit communication
  - Programming e.g. via MPI
  - Scalable

- **Shared Memory**
  - Implicit data distribution
  - Implicit communication
  - Programming e.g. via OpenMP
What it is MPI

- MPI means Message Passing Interface
- Basically is a (big) library of communication functions for sending and receiving messages between processes.
- The aim of MPI is explicit the communication among processes:
  - Movement of data between processors
  - Synchronization among processes
- Different implementations:
  - MPICH2, LAM/MPI, OpenMPI
What it is MPI

- Since the finalization of the first standard, MPI has replaced many previous message passing approaches.
- The MPI API is very large (>300 subroutines), but with only 6 – 10 different calls serious MPI applications can be programmed.
- Tools for debugging and runtime analysis are widely available.
Basic concepts of MPI

- Parallel processes with local address spaces
- Processes communication:
  - two-side operation
  - send & receive
  - receiver has to wait until the data has been sent to use it.
- Single Program Multiple Data (SMPD) Paradigm
Basic MPI functions

- **Type of functions:**
  - Send data: MPI_Send, MPI_Scatter, MPI_Isend
  - Receive data: MPI_Recv, MPI_Gather, MPI_Irecv
  - Synchronization: MPI_Wait, MPI_Barrier
  - Read data: MPI_File_read
  - Write: MPI_File_write

- **Number of processes:**
  - Point to Point: e.g. MPI_Send, MPI_Recv, MPI_Isend
  - Collective: e.g. MPI_Bcast, MPI_Scatter, MPI_Gather, MPI_Alltoall

- **Type of communication:**
  - Blocking: MPI_Send, MPI_Recv, MPI_Sendrecv
  - Non-Blocking: MPI_Isend, MPI_Irecv
  - Synchronous: MPI_Ssend, MPI_Issend
  - Asynchronous (buffered): MPI_Bsend, MPI_Ibsend
Type of communications

- **Blocking mode:**
  - The process is blocked until the operation has been realized
  - `MPI_Send`, `MPI_Recv`, `MPI_Sendrecv`

- **Non blocking mode:**
  - The process asks to the system to realize the operation getting the control immediately:
    - Require to determine whether the operation has completed or no.
    - `MPI_Isend`, `MPI_Irecv`

- **When a operation has finished?**
  - Receive: When the message is in the receive buffer
  - Send:
    - When the message has been received in the destine
    - When the message has been copied into a buffer system in the transmitter side
MPI’s send modes

- **Buffering:**
  - The message is copied in a system buffer in the transmission side.
  - The process gets the control after the copy.
  - If the message size is bigger than the buffer, the operation failed.
  - MPI_Bsend, MPI_Ibsend

- **Synchronous:**
  - The process get the control, after the receiver has sent a confirmation about message received.
  - MPI_Ssend, MPI_Issend

- **Basic:**
  - Short messages: Buffering
  - Big messages: Synchronous
  - MPI_Send

- **Ready:**
  - This operation only can be done if before the other end is prepared for an immediate receipt. No copies Additional message.
  - MPI_Rsend, MPI_Risend
MPI example

if (pid==0)
    for ( i=1; i<NumProc; i++)
        MPI_Send();

else MPI_Recv();
Computation

if (pid!=0)
    MPI_Send();
else
    for ( i=1; i<NumProc; i++)
        MPI_Recv();
Performance of parallel applications

- The objective of a parallel program is to solve **bigger problems** and/or in **less time**.
- We need to use the resources in an optimal way, in order to allow that parallel applications exploit efficiently these resources in a parallel system.
- To developing parallel software needs a detail level knowledge about the architecture of the parallel system.
Common performance problems with MPI

- Frequent synchronization
  - Reduction operations
  - Barrier operations
  - Blocking operations

- Load balancing
  - Wrong data decomposition
  - Dynamically changing load
Common performance problems with MPI

- **IO**
  - High data volume
  - Sequential IO due to IO subsystems or sequentialization in the program

- **Excessive communication**
  - Frequent communication
  - High data volume

- **More examples:**
  - Excessive MPI time due to many small messages
  - Excessive MPI time in receive due to late sender
Performance factors of parallel applications

- When a code is executed in parallel there are more things to do apart from the calculation. Among all these things, the principal is the communication between the processes (and IO operations).

\[ T_p(N,P) = T_{calc}(N,P) + T_{com}(N,P) + (T_{io}) \]

- \( ts + N/B \)
  - \( ts = \text{Latency}(\text{start-up}) \)
  - \( B = \text{Bandwidth} \)
  - \( N = \text{Number of data to transfer} \)
  - \( P = \text{Number of processes} \)
Basic tips to improve the performance

- It is necessary to execute as much as possible the calculation and communication at the same time:
  - Use non-blocking and asynchronous communications.

- Try to reduce the latency in the communication:
  - Grouping data: To send less messages with more data
  - Increasing the grain size of the tasks
  - Take account the buffering limitations
Basic tips to improve the performance

- **Load Balancing:**
  - A multiprocess job completes only when the process with the most work has finished.

- **Cost of Synchronization**
  - Sources of Sync: MPI_Barrier, MPI_Bcast, MPI_Reduce, Synchronous MPI point to point.
  - Reduce the number of message-passing calls
  - Specifically reduce the amount of explicit synchronization
  - Post sends as early as possible and receives as late as possible
MPI’s Profiling Interface

- Every MPI function also exists in the library under the name PMPI_
- PMPI allows selective replacement of MPI routines at link time (no need to recompile)
- This feature can be used by profiling tools to instrument MPI calls
This program runs perfectly. But can we improve the performance?
Performance Example

Execution time of each interaction: 0.9 ms

Yes we can! Using non-blocking communication to replace the usage of MPI_Sendrecv and to avoid the serialization in this way.
The 80-20 rule

- **Pareto principle (80-20 rule):** For any phenomenon, 80% of the consequences come from 20% of the causes.
- We see this phenomenon in software engineering where 80% of the time is spent in only 20% of the code.
- When we optimize our applications, we know to focus on that 20% of the code.
  - Know when to stop!
  - Don’t optimize what does not matter
Performance Analysis

- The performance analysis is the method that allows a programmer to evaluate and analyze and (sometimes) predict the performance of an application in a specific parallel system.

- The aim of the performance analysis is to determine which is the correct way of functionality of an application in a specific system and also give us the maximum productivity possible.

- The performance analysis has to provide to the user the needed information for diagnosing the behavior of an application.
Performance Analysis

- To improve the performance of a parallel program
  - Identify which are the tasks which consume the most time of the execution time of an application (rule 80-20).
- Measurement is better than guessing
  - To determine performance bottlenecks
  - To compare alternatives
  - To validate tuning decisions and optimizations
    - After each step.
Performance analysis challenges

- Complex architectures are hard to use efficiently
  - Multi-level parallelism: multi-core, ILP, SIMD instructions
  - Multi-level memory hierarchy
  - Result: gap between typical and peak performance is huge

- Complex applications present challenges
  - For measurement and analysis
  - For understanding and tuning

Performance tools can play an important role as guide
Performance Analysis Workflow

- Coding
- Performance Analysis
- Program Tuning
- Production

\{ Measurement \rightarrow Analysis \rightarrow Examination \rightarrow Optimization \}
Performance Measurement

- **Metrics and Parameters:**
  - **Metrics:** Magnitudes used for evaluating some events related with performance.
  - What we can measure?
    - A **count** of how often an event occurs
      - E.g. the number of MPI point-to-point messages sent
    - The **duration** of some interval
      - E.g. the time spent these send calls
    - The **size** of some parameters
      - E.g. the number of bytes transmitted by these calls
    - Derived metrics
      - E.g. rates/throughput/efficiency/speedup
  - **Parameter:** Features that affect the performance of an application
    - Number of processes
    - Number of processors
    - Parameters of an application
    - Network Type
  - The execution time metric, is the most important:
    - Minimize the running time of an application is usually the main objective of the performance analysis
Performance Measurement Techniques

- **Classification:**
  - By the measurement event moment:
    - Methods by sampling
    - Methods by code instrument
  - By record data:
    - Profiling: Recording accumulated performance data for events
    - Trace: Recording performance data of individual events
Sampling

```c
int main()
{
    int i;

    for (i=0; i < 3; i++)
        foo(i);

    return 0;
}

void foo(int i)
{
    if (i > 0)
        foo(i - 1);
}
```

- Statistical inference of program behaviour
- Not very detailed information
- Only for long-running applications
- Unmodified executables
Most of the Performance Tools use this method!

- Every event is captured
- Detailed information
- Processing of source-code or executable
- Overhead

```c
int main()
{
    int i;
    Enter("main");
    for (i=0; i < 3; i++)
    {
        foo(i);
        Leave("main");
    }
    return 0;
}

void foo(int i)
{
    Enter("foo");
    if (i > 0)
    {
        foo(i - 1);
        Leave("foo");
    }
}
Profiling

- Record of aggregated information
  - Total, maximum ...
- For measurements
  - Time
  - Counts
    - Function calls
    - Bytes transferred
    - Hardware counters
  - Functions, call sites
  - Processes, threads

```
[rosa@moore mpi_exercise]$ pprof
Reading Profile files in profile.*

NODE 0;CONTEXT 0;THREAD 0:

<table>
<thead>
<tr>
<th>%Time</th>
<th>Exclusive msec</th>
<th>Inclusive total msec</th>
<th>#Call</th>
<th>#Subrs</th>
<th>Inclusive Name usec/call</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0</td>
<td>100</td>
<td>379</td>
<td>1</td>
<td>11</td>
<td>379098 .TAU application</td>
</tr>
<tr>
<td>71.0</td>
<td>269</td>
<td>269</td>
<td>1</td>
<td>0</td>
<td>269001 MPI_Init()</td>
</tr>
<tr>
<td>2.3</td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>8640 MPI_Finalize()</td>
</tr>
<tr>
<td>0.1</td>
<td>0.441</td>
<td>0.441</td>
<td>1</td>
<td>0</td>
<td>441 MPI_Comm_split()</td>
</tr>
<tr>
<td>0.1</td>
<td>0.216</td>
<td>0.216</td>
<td>1</td>
<td>0</td>
<td>216 MPI_Intercomm_create()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.037</td>
<td>0.037</td>
<td>1</td>
<td>0</td>
<td>37 MPI_Send()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.003</td>
<td>0.003</td>
<td>2</td>
<td>0</td>
<td>2 MPI_Comm_rank()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.003</td>
<td>0.003</td>
<td>2</td>
<td>0</td>
<td>2 MPI_Comm_size()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.002</td>
<td>0.002</td>
<td>2</td>
<td>0</td>
<td>1 MPI_Comm_free()</td>
</tr>
</tbody>
</table>
```
**Tracing**

- Recording all the events for the demanded code
  - Enter/leave of a region
  - Send/receive a message

- Extra information in event record
  - Timestamp, location, event type
  - Event-related info (e.g. communicator, sender/receiver)

- Chronologically ordered sequence of event records

---

### Example 1

<table>
<thead>
<tr>
<th>Name</th>
<th>Total</th>
<th>NumSam</th>
<th>MaxVal</th>
<th>MinVal</th>
<th>MeanVa</th>
<th>MeanLo</th>
<th>Std. Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message size received from all nodes</td>
<td>50,311,648</td>
<td>12,284</td>
<td>4,096</td>
<td>4,096</td>
<td>4,096</td>
<td>4,096</td>
<td>0</td>
</tr>
<tr>
<td>Message size received from node 1</td>
<td>16,777,216</td>
<td>4,096</td>
<td>4,096</td>
<td>4,096</td>
<td>4,096</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Message size received from node 2</td>
<td>16,777,216</td>
<td>4,096</td>
<td>4,096</td>
<td>4,096</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Message size received from node 3</td>
<td>16,777,216</td>
<td>4,096</td>
<td>4,096</td>
<td>4,096</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Message size sent to all nodes</td>
<td>50,311,648</td>
<td>12,284</td>
<td>4,096</td>
<td>4,096</td>
<td>4,096</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Message size sent to node 1</td>
<td>16,777,216</td>
<td>4,096</td>
<td>4,096</td>
<td>4,096</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Message size sent to node 2</td>
<td>16,777,216</td>
<td>4,096</td>
<td>4,096</td>
<td>4,096</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Message size sent to node 3</td>
<td>16,777,216</td>
<td>4,096</td>
<td>4,096</td>
<td>4,096</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Profiling vs Tracing

- **Profiling:**
  - recording summary information (time, \#calls, \#misses)
  - about program entities
  - very good for quick, low cost overview
  - implemented through sampling or instrumentation
  - moderate amount of performance data

- **Tracing**
  - recording information about events
  - trace record typically consist of timestamp
  - output is a trace file with trace records sorted by time
  - can be used to reconstruct the dynamic behavior
  - creates huge amounts of data
  - needs selective instrumentation
Analysis Techniques

- Offline vs Online Analysis
  - Offline: First generate data then analyze
  - Online: generate and analyze data while application is running
  - Online requires automation \(\rightarrow\) limited to standard bottlenecks
  - Offline suffers more from size of measurement information

- Three techniques to support user in analysis
  - Source-level presentation of performance data
  - Graphical visualization
  - Ranking of high-level performance properties
No single solution

- Combination of different methods, tools and techniques is typically needed
  - Analysis
  - Measurement
  - Instrumentation
Typical performance analysis procedure

- **Do** I have a performance problem at all?
  - Time/ speedup/ scalability ?
- **What** is the key bottleneck
  - Computation /Communication ?
  - MPI flat profiling
- **Where** is the key bottleneck
  - Call-path profiling, detailed basic block profiling
- **Why** is it there ?
  - How counter analysis, trace selected parts to keep trace size manageable
- **Does the code have scalability problems ?**
  - Load imbalance analysis, compare profiles at various size function by function
Productivity tools

- Marmot/MUST
  - MPI correctness checking
- PAPI
  - Interfacing to hardware performance counters
- Periscope
  - Automatic analysis via an on-line distributed search
- **Scalasca**
  - Large scale parallel performance analysis
- TAU
  - Integrated parallel performance system
- **Vampir/Vampir Trace**
  - Event tracing and graphical trace visualization & analysis
- Jumpshot
  - Visualization tool for doing postmortem performance analysis.
- Score-P
  - Common instrumentation & measurement infrastructure
Productivity tools

- KCachegrind
  - Callgraph-based cache analysis [x86 only]
- MAQAO
  - Assembly instrumentation & optimization [x86 only]
- mpiP/mpipview
  - OpenMP profiling tool
- Open MPI
  - Integrated memory checking
- Openl Speedshop
  - Integrated parallel performance analysis environment
- Paraver/Extrae
  - Event tracing and graphical trace visualization & analysis
Integration Tools

- KCACHEGRIND: Hardware monitoring
- PAPI: Error detection
- MARMOT / MUST: Execution
- PERISCOPE: Automatic profile & trace analysis
- SCALASCA: Visual trace analysis
- TAU: Optimization
- JUMPSHOT: Integration Tools
Important

- Tool will **not** automatically make you, your application or computer system more productive.

- However, they can help you understand **how** your parallel code executes and **when/where** it’s necessary to work on correctness and performance issues.
**TAU**

**Instrumentation:** Adds probes to perform measurements

- Source code instrumentation using pre-processors and compiler scripts
- Wrapping external libraries (I/O, MPI, Memory, CUDA, OpenCL, pthread)
- Rewriting the binary executable

**Measurement:** Profiling or Tracing using wallclock time or hardware counters

- Direct instrumentation (Interval events measure exclusive or inclusive duration)
- Indirect instrumentation (Sampling measures statement level contribution)
- Throttling and runtime control of low-level events that execute frequently
- Per-thread storage of performance data
- Interface with external packages (e.g. PAPI hw performance counter library)

**Analysis:** Visualization of profiles and traces

- 3D visualization of profile data in paraprof, perfexplorer tools
- Trace conversion & display in external visualizers (Vampir, Jumpshot, ParaVer)
Interactive event trace analysis

- Alternative & supplement to automatic trace analysis
- Visual presentation of dynamic runtime behaviour
  - event timeline chart for states & interaction of processes/threads
  - communication statistics, summaries & more
- Interactive browsing, zooming, selecting
  - linked displays & statistics adapt to selected time interval (zoom)
  - scalable server runs in parallel to handle larger traces

Developed by TU Dresden ZIH

- Open-source VampirTrace library
  - http://www.tu-dresden.de/zih/vampirtrace
- Vampir Server & GUI have a commercial license
  - http://www.vampir.eu
Vampir & VampirTrace
Jumpshot

- Visualize trace tool for doing postmortem performance analysis
  - Last realize Jumpshot-4
  - Provide scalable log file format. Allows zoom-level Timeline and Histograms
  - Graphical analysis of MPI overhead

- Developed by Laboratory for advanced numerical software