CIS 631
Parallel Processing

Lecture 13: Parallel Performance Tools

Allen D. Malony
malony@cs.uoregon.edu

Department of Computer and Information Science
University of Oregon
Acknowledgements

- Portions of the lectures slides were adopted from:
Outline

- TAU performance system status
- Vampir trace visualization system
- Automatic performance analysis and Expert
TAU Performance System Status

- Computing platforms
  - IBM SP / Power4, SGI Origin 2K/3K, Intel Teraflop, Cray T3E / SV-1 (X-1 planned), Compaq SC, HP, Sun, Hitachi SR8000, NEX SX-5 (SX-6 underway), Intel (x86, IA-64) and Alpha Linux cluster, Apple (OS X), Windows

- Programming languages
  - C, C++, Fortran 77, F90, HPF, Java, OpenMP, Python

- Communication libraries
  - MPI, PVM, Nexus, shmem, Tulip, ACLMPL, MPIJava

- Thread libraries
  - pthreads, Java, Windows, Tulip, SMARCS, OpenMP
TAU Performance System Status (continued)

- Compilers
  - KAI, PGI, GNU, Fujitsu, Sun, Microsoft, SGI, Cray, IBM, Compaq

- Application libraries
  - Blitz++, A++/P++, PETSc, SAMRAI, Overture, PAWS

- Application frameworks
  - POOMA, POOMA-2, MC++, Conejo, Uintah, VTF, UPS

- Projects
  - Aurora / SCALEA: ACPC, University of Vienna

- TAU full distribution (Version 2.1x, web download)
  - Measurement library and profile analysis tools
  - Automatic software installation and examples
  - TAU User’s Guide
PDT Status

- Program Database Toolkit (Version 2.x, web download)
  - EDG C++ front end (Version 2.45.2)
  - Mutek Fortran 90 front end (Version 2.4.1)
  - C++ and Fortran 90 IL Analyzer
  - DUCTAPE library
  - Standard C++ system header files (KCC Version 4.0f)

- PDT-constructed tools
  - TAU instrumentor (C/C++/F90)
  - Program analysis support for SILOON and CHASM

- Platforms
  - SGI, IBM, Compaq, SUN, HP, Linux (IA32/IA64), Apple, Windows, Cray T3E, Hitachi
Work in Progress

- Trace visualization
  - Event traces with counters (Vampir 3.0 will visualize)
  - EPILOG trace conversion

- Runtime performance monitoring and analysis
  - Online performance data access
  - Performance analysis and visualization in SCIRun

- Performance Database Framework
  - XML parallel profile representation of TAU profiles
  - PostgresSQL performance database

- Next-generation PDT

- Performance analysis for component software (CCA)
Vampir

- Visualization and Analysis of MPI Programs
- Originally developed by Forschungszentrum Jülich
- Current development by Technical University Dresden
- Distributed by PALLAS, Germany
- [http://www.pallas.de/pages/vampir.htm](http://www.pallas.de/pages/vampir.htm)
Vampir: General Description

- Offline trace analysis for message passing trace files
- Convenient user–interface / easy customization
- Scalability in time and processor–space
- Excellent zooming and filtering
- Display and analysis of MPI and application events:
  - user subroutines
  - point–to–point communication
  - collective communication
  - MPI–2 I/O operations
- Large variety of customizable (via context menus) displays for ANY part of the trace
## Vampir: Main Window

- Trace file loading can be
  - interrupted at any time
  - resumed
  - started at a specified time offset
- Provides main menu
  - access to global and process local displays
  - preferences
  - help
- Trace file can be re-written (re-grouped symbols)
Vampir: Timeline Diagram

- Functions organized into groups
- Coloring by group
- Message lines can be colored by tag or size
- Information about states, messages, collective, and I/O operations available by clicking on the representation
Vampir: Timeline Diagram (Message Info)

- Source-code references are displayed if recorded in trace
**Vampir: Support for Collective Communication**

- For each process: locally mark operation

  - Start of op
  - Data being sent
  - Data being received
  - Stop of op

- Connect start/stop points by lines
Vampir: Collective Communication Display
Vampir: MPI-I/O Support

- MPI I/O operations shown as message lines to separate I/O system time line
Vampir: Execution Statistics Displays

- Aggregated profiling information: execution time, # calls, inclusive/exclusive
- Available for all/any group (activity)
- Available for all routines (symbols)
- Available for any trace part (select in timeline diagram)
**Vampir: Communication Statistics Displays**

- Bytes sent/received for collective operations
- Byte and message count, min/max/avg message length and min/max/avg bandwidth for each process pair
- Message length statistics
  - Available for any trace part
**Vampir: Other Features**

- Parallelism display
- Powerful filtering and trace comparison features
- All diagrams highly customizable (through context menus)

- Dynamic global call graph tree
Vampir: Process Displays

- Activity chart
- Call tree
- Timeline

For all selected processes in the global displays
Vampir: New Features

- New Vampir versions (3 and 4)
  - New core (dramatic timeline speedup, significantly reduced memory footprint)
  - Load–balance analysis display
  - Hardware counter value displays
  - Thread analysis
  - Show hardware and grouping structure
  - Improved statistics displays

- Raised scalability limits: can now analyse 100s of processes/threads
Vampir: Load Balance Analysis

- **State Chart** display
- Aggregated profiling information: execution time, # calls, inclusive/exclusive
- For all/any group (activity)
- For all routines (symbols)
- For any trace part
Vampir: HPM Counter

- Counter Timeline Display

- Process Timeline Display
Vampir: Cluster Timeline

- Display of whole system

![Vampir Cluster Timeline Diagram]

- Parallelism Display
- Communication Volume Display
Vampir: Cluster Timeline

- SMP or Grid Nodes Display

Intra-node Communication Volume

Parallelism Display for each Node
Vampir: Cluster Timeline (2)

- Display of messages between nodes enabled
Vampir: Improved Message Statistics Display

□ Process View

□ NodeView
Release Schedule

- Vampir/SX and Vampirtrace/SX
  - version 1 available via NEC Japan
  - version 2 is ready for release

- Vampir/SC and Vampirtrace/SC
  - version 3 is available from Pallas
  - version 4 scheduled for Q4/2001

- Vampir and Vampirtrace
  - version 3 is scheduled for Q4/2001
  - version 4 will follow in 2002
# Vampir Feature Matrix

<table>
<thead>
<tr>
<th>Feature</th>
<th>Vampir</th>
<th>Vampir/SC</th>
<th>Vampir/SX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>New core</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Load–balance displays</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Counter analysis</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Thread analysis</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Grouping support</td>
<td>no</td>
<td>partial</td>
<td>partial</td>
</tr>
<tr>
<td>Improved statistics displays</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Scalability (processes)</td>
<td>200</td>
<td>500</td>
<td>500</td>
</tr>
</tbody>
</table>
**Vampirtrace**

- Commercial product of Pallas, Germany
- Library for Tracing of MPI and Application Events
  - records MPI point-to-point communication
  - records MPI collective communication
  - records MPI–2 I/O operations
  - records user subroutines (on request)
  - records source–code information (some platforms)
  - support for shmem (Cray T3E)
- Uses the PMPI profiling interface
- [http://www.pallas.de/pages/vampir.htm](http://www.pallas.de/pages/vampir.htm)
Vampirtrace: Usage

- Record MPI–related information
  - Re–link a compiled MPI application (no re-compilation)
    - `{f90,cc,CC} *.o -o myprog
      -L$(VTHOME)/lib -lVT -lpmpi -lmpi
  - Re-link with `-vt` option to MPICH compiler scripts
    - `{mpif90,mpicc,mpiCC} -vt *.o -o myprog
  - Execute MPI binary as usual

- Record user subroutines
  - insert calls to Vampirtrace API (portable, but inconvenient)
  - use automatic instrumentation
    (NEC SX, Fujitsu VPP, Hitachi SR)
  - use instrumentation tool (Cray PAT, dyninst, ...)

Lecture 14
Vampirtrace Instrumentation API (C / C++)

- Calls for recording user subroutines

```c
#include "VT.h"
/* Symbols defined with/without source information */
VT_symdef(123, "foo", "USER", "foo.c:6");
VT_symdef(123, "foo", "USER");

void foo { 
    VT_begin(123);     /* 1st executable line */
    ...
    VT_end(123);       /* at EVERY exit point! */
}
```

- VT calls can only be used between `MPI_Init` and `MPI_Finalize`!
- Event numbers used must be globally unique
- Selective tracing: `VT_traceoff()`, `VT_traceon()`
**VT++.h – C++ Class Wrapper for Vampirtrace**

```cpp
#ifndef __VT_PLUSPLUS_
#define __VT_PLUSPLUS_
#include "VT.h"

class VT_Trace {
    public:  VT_Trace(int code) {VT_begin(code_ = code);}  
            ~VT_Trace()         {VT_end(code_);} 
    private: int code_; 
};
#endif /* __VT_PLUSPLUS_ */
```

- Same tricks can be used to wrap other C++ tracing APIs
- Usage:

```cpp
VT_symdef(123, "foo", "USER"); // symbol definition as before
void foo(void) {  
    VT_Trace vt(123); // declare VT_Trace object in 1st line
...  
}  // => automatic tracing by ctor/dtor
```
Vampirtrace Instrumentation API (Fortran)

- Calls for recording user subroutines

```fortran
include 'VT.inc'
integer ierr
call VTSYMDEF(123, "foo", "USER", ierr) ! or
call VTSYMDEFL(123, "foo", "USER", "foo.f:8", ierr)
```

```c
SUBROUTINE foo(...)
include 'VT.inc'
integer ierr
call VTBEGIN(123, ierr)
... 
call VTEND(123, ierr);
END
```

- Selective tracing: VTTRACEOFF(), VTTRACEON()
**Vampirtrace: Runtime Configuration**

- Trace file collection and generation can be controlled by using a configuration file
  - Trace file name, location, size, flush behavior
  - Activation/deactivation of trace recording for specific processes, activities (groups of symbols), and symbols

- Activate a configuration file with environment variables
  - `VT_CONFIG` name of configuration file
    (use absolute pathname if possible)
  - `VT_CONFIG_RANK` MPI rank of process which should read and process configuration file

- Reduce trace file sizes
  - restrict event collection in a configuration file
  - use selective tracing functions
## Vampirtrace: Configuration File Example

- # collect traces only for MPI ranks 1 to 5
  - TRACERANKS 1:5:1
- # record at most 20000 records per rank
  - MAX-RECORDS 20000

- # do not collect administrative MPI calls
  - SYMBOL MPI_comm* off
  - SYMBOL MPI_cart* off
  - SYMBOL MPI_group* off
  - SYMBOL MPI_type* off

- # do not collect USER events
  - ACTIVITY USER off
- # except routine foo
  - SYMBOL foo on

- □ Be careful to record complete message transfers!
- □ See Vampirtrace User's Guide for complete description
New Features – Tracing

- New Vampirtrace versions (3 and 4)
  - New core (significantly reduce memory and runtime overhead)
  - Better control of trace buffering and flush files
  - New filtering options
  - Event recording by thread
  - Support of MPI–I/O
  - Hardware counter data recording (PAPI)
  - Support of process/thread groups
### Vampirtrace Feature Matrix

<table>
<thead>
<tr>
<th>Feature</th>
<th>Vampirtrace</th>
<th>Vampirtrace/SC</th>
<th>Vampirtrace/SX</th>
</tr>
</thead>
<tbody>
<tr>
<td>New core</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Buffer control</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Recover trace</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>New filter options</td>
<td>partial</td>
<td>yes</td>
<td>partial</td>
</tr>
<tr>
<td>Thread events</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>MPI–I/O</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Counter data</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Thread/process grouping</td>
<td>no</td>
<td>yes</td>
<td>partial</td>
</tr>
<tr>
<td>Scalability (processes)</td>
<td>200</td>
<td>500</td>
<td>500</td>
</tr>
</tbody>
</table>
Motivation: Current Tool Support

- Graphical presentation of enormous amounts of low-level data
- Manual search for performance problems
  - Different aspects over different views
- Too simple performance metrics
  - Counter, rates
  - Low-level view of behavior
- Complex to use, Hard to learn

Needed: Automatic Performance Analysis
⇒ Transformation of large volumes of low-level data into small volumes of high-level data
KOJAK

- Kit for Objective Judgement and Automatic Knowledge-based detection of bottlenecks
- Forschungszentrum Jülich, Germany
- Goal: Automatic, portable and generic performance-analysis environment
- Current focus
  - Event traces
  - Parallel computers with SMP nodes
  - MPI, OpenMP, hybrid (OpenMP + MPI)
- http://www.fz-juelich.de/zam/kojak/
APART

- European IST Working Group APART (since 1999)
- Automatic Performance Analysis: Real Tools
- Forum for scientists and vendors of hardware/software
- About 20 partners in Europe and the U.S.
- http://www.fz-juelich.de/apart/
Performance Analysis: APART Terminology

- Performance property
  - Aspect of performance behavior
    - Example: communication dominated by waiting time
  - Specification
    - As condition referring to performance data
  - Quantification
    - Severity ::= execution time spent on specific behavior
    - Allows for normalization

- Performance problem
  - “Negative” performance property

- Performance bottleneck
  - Performance problem with highest severity
Performance Property: Message in Wrong Order

Location

B

C

A

Time

waiting time

SEND

RECV

RECV
More Examples of Performance Properties

- **Message Passing (MPI)**

  - Late Receiver
  - Messages in Wrong Order
  - Wait at N x N
  - ...

  # Blocked sender
  # Waiting for new messages although older messages ready to be received
  # Waiting for last participant in N-to-N operation

- **Shared Memory (OpenMP)**

  - Barrier Synchronization
  - Lock Synchronization
  - Idle Threads
  - ...

  # Waiting time in explicit or implicit barriers
  # Waiting for lock owned by another thread
  # Time on unused CPUs while executing in sequential mode
Event-Based Specification

- Basic event model
  - Hierarchy of event types

- Event ::= Set of attributes
- Event trace ::= Sequence of events in chronological order
Event-Based Specification

Problem: Event-based specification of properties?
- Complex relationships among constituents
- May depend on execution state of the program
- May depend on programming model

Solution: Definition of a enhanced event model by encapsulation of complex relationships
- State sequences (state ::= set of the triggering events)
  - Inductively defined by transition operators
  - e.g., message queue, region stack, ...
- Pointer attributes (link between corresponding events)
  - Defined as function over attributes and state sequences
  - e.g., sender, enclosing region, …
Event-Based Specification

- Property represented as a compound event (event pattern)
  - Set of primitive events (constituents)
  - Distinguished root event
    - Starting point of search for constituents
  - Relationships between constituents and other conditions
    - Defined based on state sequences and pointer attributes

Implementation

- Enhanced event model implemented as EARL (Python/C++)
- Compound event specified as Python class
  - Implements interface (Pattern)
  - Overrides call-back method for root event type
    - Locates constituents
    - Checks conditions
    - Calculates severity per call tree node and location
      \[ \Rightarrow \text{severity matrix} \]
  - Registers call-back method with analysis process
  - All properties arranged in a specialization hierarchy
- Analysis process
  - Walks sequentially through event trace
  - Invokes registered call-back methods for each event
Example

**Diagram Description:**

- **Location:** A, B, C
- **Action:**
  - **RECV**
  - **SEND**
- **Time:**
  - **ENTER**
  - **EXIT**
  - **SEND**
  - **RECV**
- **Waiting Time:**
  - Indicated by an arrow labeled **waiting time**
- **Call Back:** Indicated by an arrow labeled **call back**

**Legend:**
- Red: **ENTER**
- Orange: **EXIT**
- Blue: **SEND**
- Green: **RECV**

**Annotations:**
- **??**
- **message**
- **enterptr**
- **sendptr**
KOJAK Architecture

user program → OPARI / TAU instr. → modified program → Compiler / Linker → EPILOG trace library → POMP+PMPI libraries

-executable-

execute

EPILOG event trace

EXPERT Analyzer → analysis result → EXPERT Presenter

→ Automatic Analysis

trace converter → VTF3 event trace → VAMPIR

Manual Analysis
KOJAK Tool Components

- Instrument user application with **EPILOG** tracing library calls
  - User functions and regions:
    - Automatically by compiler (PGI, Hitachi)
    - Automatically by **TAU** source instrumentor
  - MPI calls:
    - Automatically by **PMPI** Wrapper Library (MPI 1.2 only)
  - OpenMP:
    - Automatically by **OPARI** source instrumentor

- Analyze measured event trace
  - Automatically with **EXPERT** trace analyzer
    (based on EARL trace analysis language)
  - Manually with **VAMPIR**
    (using EPILOG-VTF3 converter)
EXtensible PERformance Tool

- Representation of results (severity matrix) along three hierarchical axes
  - Performance property (general → specific)
  - Region tree path
  - Location (machine → node → process → thread)

- Three coupled tree browsers
- Each node displays severity
  - As color: for easy identification of bottlenecks
  - As value: for precise comparison
- Displayed severity depends on state
  - Collapsed (inclusive severity)
  - Expanded (exclusive severity)
Performance Property
Which type of behavior caused the problem?

Color Scale
How severe is the problem?

Call Path Tree
Where in the source code? In which context?

Location
How is the problem distributed across the machine?
Expert Conclusion

- Formal characterization of inefficient behavior
  - Automatic detection in event traces
  - Simple specification through abstraction mechanisms

- Specification of common performance problems
  - MPI, OpenMP, and hybrid applications

- Integrated multi-dimensional representation of performance behavior

- Extensible modular tool architecture
Semantic Performance Mapping

- Associate performance measurements with high-level semantic abstractions
- Need mapping support in the performance measurement system to assign data correctly
Semantic Entities/Attributes/Associations (SEAA)

- New dynamic mapping scheme (S. Shende, Ph.D. thesis)
  - Contrast with ParaMap (Miller and Irvin)
  - Entities defined at any level of abstraction
  - Attribute entity with semantic information
  - Entity-to-entity associations

- Two association types (implemented in TAU API)
  - Embedded – extends associated object to store performance measurement entity
  - External – creates an external look-up table using address of object as key to locate performance measurement entity
Hypothetical Mapping Example

- Particles distributed on surfaces of a cube

```c
Particle* P[MAX]; /* Array of particles */
int GenerateParticles() {
    /* distribute particles over all faces of the cube */
    for (int face=0, last=0; face < 6; face++) {
        /* particles on this face */
        int particles_on_this_face = num(face);
        for (int i=last; i < particles_on_this_face; i++) {
            /* particle properties are a function of face */
            P[i] = ... f(face);
            ...
        }
        last += particles_on_this_face;
    }
}
```
Hypothetical Mapping Example (continued)

```c
int ProcessParticle(Particle *p) {
    /* perform some computation on p */
}
int main() {
    GenerateParticles();
    /* create a list of particles */
    for (int i = 0; i < N; i++)
        /* iterates over the list */
        ProcessParticle(P[i]);
}
```

- How much time is spent processing face $i$ particles?
- What is the distribution of performance among faces?
No Performance Mapping versus Mapping

- Typical performance tools report performance with respect to routines
- Does not provide support for mapping

- Performance tools with SEAA mapping can observe performance with respect to scientist’s programming and problem abstractions
Midterm

☐ Four sections
  ☐ Analytical modeling and performance analysis
  ☐ Parallel programming
  ☐ Parallel algorithms
  ☐ Parallel architecture and systems (maybe)

☐ Open notes

☐ Mostly, the test will be self-contained

☐ Suggested preparation
  ☐ Read through lecture notes
  ☐ Practice thinking in parallel