

Automatic Performance Analysis

Felix Wolf

fwolf@cs.utk.edu

Outline

- A quick introduction to OpenMP
- Introduction
- Instrumentation
- Analysis of trace data
- Performance algebra
- Topology analysis

A quick introduction to OpenMP

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char* argv[])
{
    printf("Hello parallel world from thread:\n");

    #pragma omp parallel
    {
        printf("%d\n", omp_get_thread_num());
    }
    printf("Back to the sequential world\n");
}
```

Output

```
fwolf@torc0> export OMP_NUM_THREADS=4
fwolf@torc0> ./a.out
Hello parallel world from thread:
1
3
0
2
Back to sequential world
fwolf@torc0>
```

Parallelizing a simple loop

```
void saxpy(double z[], double a, double x[], double y,
           int n)
{
    /* for simplicity, y is a scalar variable */

    int i;

    #pragma omp parallel for
    for ( i = 0; i < n; i++ )
        z[i] = a * x[i] + y;
}
```

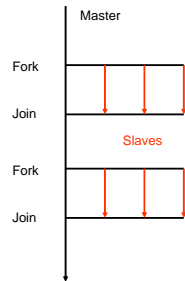
- Only change is addition of the parallel for directive
- Must be followed by a for loop
- Specifies the concurrent execution of the loop
- Runtime system must create a set of threads and distribute iterations across the threads for parallel execution

Fork-join execution model

- Program begins execution as a single process
- This process is called the **master thread** of execution
- The master thread executes until the first parallel construct is encountered (i.e., a parallel directive)
- Then the master thread creates a **team** of threads
- The master becomes the master of the team
- The statements enclosed by the parallel construct are executed in parallel by each thread of the team
- Upon completion of the construct, the threads synchronize and only the master continues execution

Fork-join execution model (2)

- A number of parallel constructs can be specified in a program
- As a result, a program may fork and join many times



KOJAK Project

- Collaborative research project between
 - Forschungszentrum Jülich, Germany
 - University of Tennessee, USA
- Software package for the automatic performance analysis of parallel applications
 - MPI and/or OpenMP
 - Parallel performance
 - CPU and memory performance
- WWW
 - <http://www.fz-juelich.de/zam/kojak/>
 - <http://icl.cs.utk.edu/kojak/>
- Contact
 - kojak@fz-juelich.de
 - kojak@cs.utk.edu



People and sponsors

- People
 - Nikhil Bhatia
 - Marc-André Hermanns
 - Andrej Kühnal
 - Bernd Mohr
 - Shirley Moore
 - Fengguang Song
 - Felix Wolf
 - Brian Wylie
- Funding
 - German Helmholtz Research Programme Scientific Computing
 - U.S. Department of Defense HPCMP PET Program
 - U.S. Department of Energy MICS Program
 - U.S. National Science Foundation



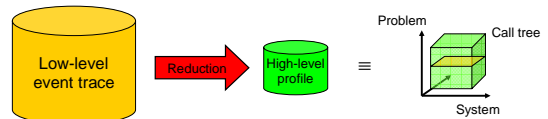
Platforms

- Linux IA-32, IA-64, and Opteron clusters
 - GNU, PGI, or Intel compilers
- IBM Power3 / Power4 based clusters
- SGI Mips (O2K, O3K) and IA-64 based (Altix) clusters
- SUN Sparc based clusters
- Hitachi SR-8000
- Cray T3E
- NEC SX
- IBM BlueGene / L
- Cray X1

Monitoring of parallel applications

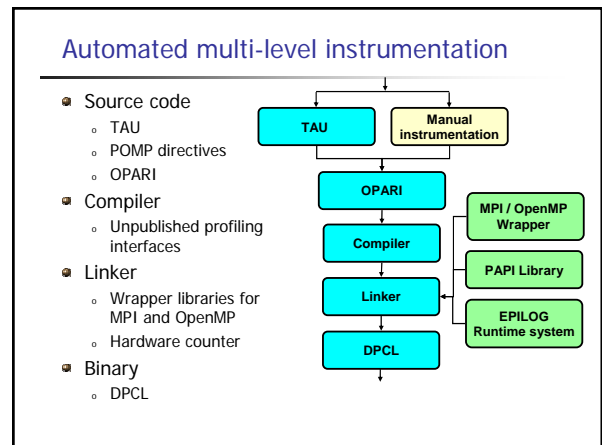
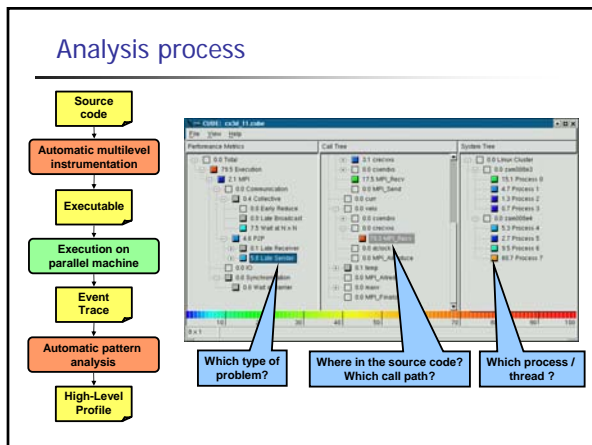
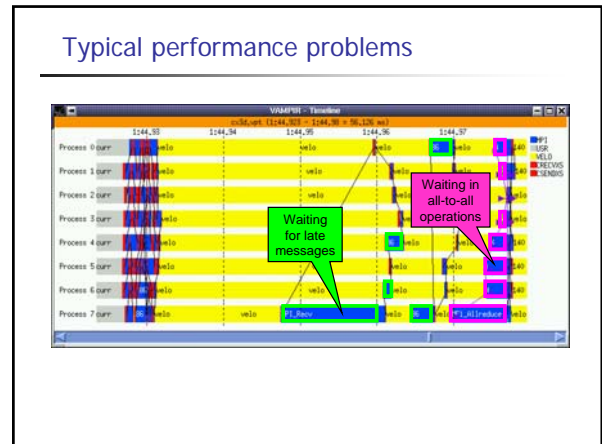
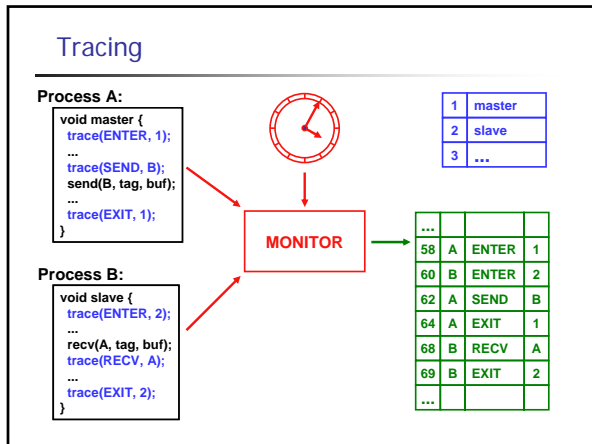
Automatic search for inefficient behavior

- Automatic performance analysis

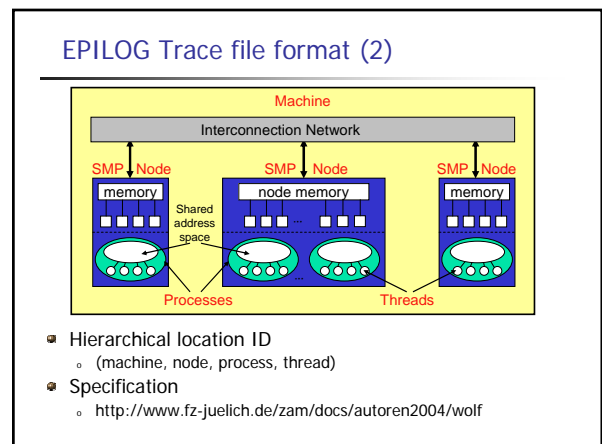


- Take event traces of MPI/OpenMP applications
- Search for execution patterns
 - Complex situations of inefficient behavior
- Calculate high-level call path profile
- Display in performance browser



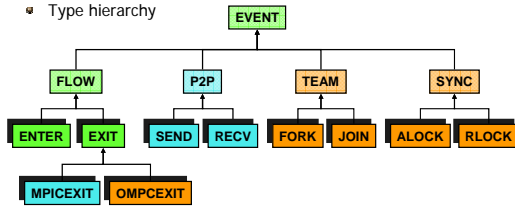


- ### EPILOG Trace file format
- Event Processing, Investigating, and LOGging
 - MPI and OpenMP support (i.e., thread-safe)
 - Region enter and exit
 - Collective region enter and exit (MPI & OpenMP)
 - Message send and receive
 - Parallel region fork and join
 - Lock acquire and release
 - Stores source code + HW counter information
 - Input of the EXPERT analyzer
 - Visualization using VAMPIR
 - EPILOG ⇒ VTF3 converter



KOJAK Event model

- Type hierarchy



- Event type
 - Set of attributes (time, location, ...)
- Event trace
 - Sequence of events in chronological order

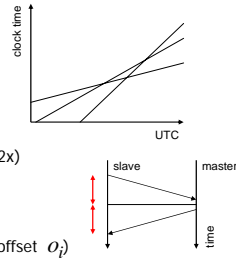
Clock synchronization

- Time ordering of parallel events requires global time
- Accuracy requirements
 - Correct order of message events (latency!)
- Many clusters provide only distributed local clocks
- Local clocks may differ in drift and offset
 - Drift:** clocks may run differently fast
 - Offset:** clocks may start at different times
- Clock synchronization
 - Hardware: cannot be changed by tool builder
 - Software: online / offline
- Online: (X)NTP accuracy usually too low

Offline clock synchronization

- Model

- Different offset
- Different but constant drift
 - Approximation!
- One master clock



- Algorithm

- Measure offset slave ↔ master (2x)
 - Request time from master (N x)
 - Take shortest propagation time
 - Assume symmetric propagation
- Get two pairs of (slave time s_i , offset o_i)
- Master time

$$m(s) := s + \frac{(o_2 - o_1)}{(s_2 - s_1)} * (s - s_1) + o_1$$

Compiler-supported instrumentation

- Put **kinst** in front of every compile and link line in your makefile

```
# compiler
CC      = kinst pgcc ...
F90     = kinst pgf90 ...

# compiler MPI
MPICC   = kinst mpicc ...
MPIF90  = kinst mpif90 ...
```

- Build as usual, everything else is taken care off
 - Instrumentation of MPI / OpenMP constructs
 - Instrumentation of user functions

Manual instrumentation

- Instrumentation of user-specified arbitrary (non-function) code regions

- C/C++

```
#pragma pomp inst begin(name)
...
[ #pragma pomp inst altend(name) ]
...
#pragma pomp inst end(name)
```

- Fortran

```
!$POMP INST BEGIN(name)
...
[ !$POMP INST ALTEND(name) ]
...
!$POMP INST END(name)
```

Manual instrumentation (2)

- Insert once as the first executable line of the main

```
#pragma pomp inst init
!$POMP INST INIT
```

- Put **kinst-pomp** in front of every compile and link line in your makefile

```
# compiler
CC      = kinst-pomp pgcc ...
F90     = kinst-pomp pgf90 ...

# compiler MPI
MPICC   = kinst-pomp mpicc ...
MPIF90  = kinst-pomp mpif90 ...
```

EXPERT: Automated trace analysis

- Classification and quantification of performance behavior
- Hierarchy of patterns that model inefficient behavior
- Inefficient use of
 - MPI
 - OpenMP
- Single-node performance
 - CPU
 - Memory
- Efficient detection
 - Exploit specialization
 - Publish / subscribe



Profiling patterns (samples)

- Execution time

Total Execution	# Execution time including idle threads
	# Execution time

- CPU and memory performance

L1 Data Cache	# L1 data cache misses
Floating Point	# Floating point instructions

- MPI and OpenMP

MPI	# MPI API calls
OpenMP	# OpenMP API calls
Idle Threads	# Time lost on unused CPUs during OpenMP sequential execution

Complex patterns (samples)

- MPI

Late Sender	# Blocked receiver
Late Receiver	# Blocked sender
Messages in Wrong Order	# Waiting for new messages although older messages ready to be received
Wait at N x N	# Waiting for last participant in N-to-N operation
Late Broadcast	# Waiting for sender in broadcast operation

- OpenMP

Wait at Barrier	# Waiting time in explicit or implicit barriers
Lock Synchronization	# Waiting for lock owned by another thread

Running EXPERT is simple...

- Run your instrumented application
- Application will generate a trace file `a.elg`
- Run analyzer with trace file as input
- Generate CUBE input file `a.cube`

```
> expert a.elg
Total number of events: 11063530
100 %
Elapsed time (h:m:s): 0 : 3 : 34
Events per second: 51698
```

- Invoke CUBE

```
> cube a.cube&
```

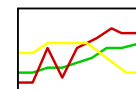
Cross-experiment analysis

- Different** code versions
 - Different algorithm
- Different** execution configurations
 - Different domain decomposition
- Different** input data
- Different** performance data
 - Different monitoring tools that cannot be applied simultaneously
 - Different data that cannot be collected simultaneously
 - Example: L1 cache misses and floating-point instructions on Power4
- Different** random errors
 - System noise
- Different** analysis approaches
 - Modeling, simulation

Comparing multiple experiments

Traditional approach

- Multiple single-experiment views
 - Cumbersome to use
- Overlay diagrams
 - Hierarchical nature of performance space ignored
 - Not equally well-suited for all analyses



CUBE Performance Algebra

- Ability to combine multiple experiments into a single one

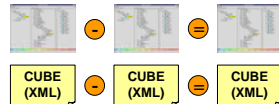
- Extension of



K. Karavanic, B. Miller: A Framework for Multi-Execution Performance Tuning, Parallel and Distributed Computing Practices, 4(3), September 2001.

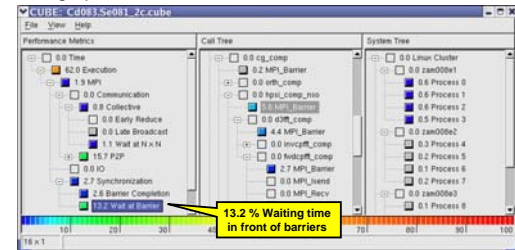
CUBE Performance Algebra

- Abstract data model describing performance experiments
- Closed arithmetic operations on entire experiments yielding entire experiments
 - Difference
 - Mean
 - Merge
- Results are called **derived experiments**
- Generic display component to view experiments and results of operations likewise



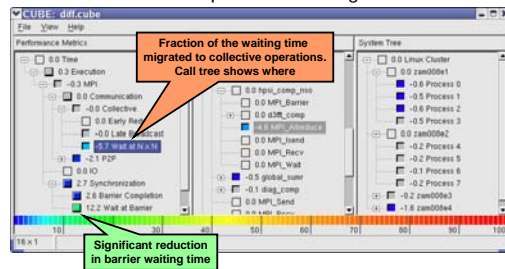
Nano-particle simulation PESCAN

- Application Lawrence Berkeley National Lab
- Numerous barriers to avoid buffer overflow when using large processor counts – not needed for smaller counts

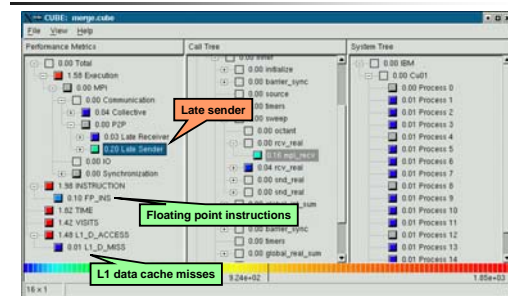


Before/after comparison

- Difference between before / after barrier removal
- Raised relief shows performance improvement
- Sunken relief shows performance degradation

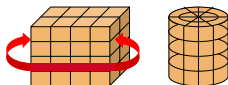


Combining trace with profiling data



Virtual topologies

- Mapping of the processes/threads onto application domain
- Can include processes, threads or a combination of both
 - MPI, OpenMP, hybrid applications
- Can be specified as a graph (e.g., a ring)
- Very common case: Cartesian topologies



Motivation

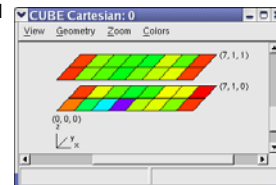
- MPI process topologies
 - Efficient mapping of virtual topologies onto the physical topology of the underlying machine
 - Optimization of communication between neighbors
 - Convenient process naming
- Topology analysis in KOJAK
 - Idea: map performance data onto topology
 - Detect higher-level events related to the parallel algorithm
 - Link occurrence of patterns to such higher-level events
 - Visually expose correlations of performance problems with topological characteristics of affected processes

Recording topological information

- Data format
 - Records to define Cartesian grids
- MPI wrapper functions
 - Applications using MPI process topologies can generate topology information automatically
- Instrumentation API
 - Applications not using MPI process topologies can generate the information by including API calls in the source code (C/C++ and Fortran)

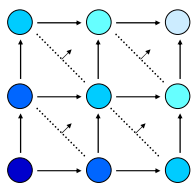
Topology visualization

- Example: environmental science application TRACE
- Simulates spread of pollutants in subsurface water
- Three dimensional domain
- Three-dimensional virtual topology (8 x 2 x 2)
- Wait states resulting from all-to-all communication in corners of topological grid



Analysis of wave-front processes

- Parallelization scheme used for particle transport problems
- Example: ASCI benchmark SWEEP3D
 - Three-dimensional domain (i,j,k)
 - Two-dimensional domain decomposition (i,j)



```

DO octants
DO angles in octant
DO k planes
! block i-inflows
IF neighbor(E/W) MPI_RECV(E/W)
! block j-inflows
IF neighbor(N/S) MPI_RECV(N/S)
... compute grid cell ...
! block i-outflows
IF neighbor(E/W) MPI_SEND(E/W)
! block j-outflows
IF neighbor(N/S) MPI_SEND(N/S)
END DO k planes
END DO angles in octant
END DO octants
    
```

Pipeline refill

- Wave-fronts from different directions
- Limited parallelism upon pipeline refill
- Four new late-sender patterns
 - Refill from NW, NE, SE, SW
 - Requires recognition of direction change \Rightarrow topological knowledge

