VAMPIR & VAMPIRTRACE
DETAILS AND HANDS-ON

Jens Doleschal, Andreas Knüpfer
ZIH, TU Dresden
jens.doleschal@tu-dresden.de
September 2009
Overview

• Event Tracing in General
• Hands-on: NPB 3.3 BT-MPI
• Finding Performance Bottlenecks
VAMPIR & VAMPIRTRACE
Event Tracing in General
Common Event Types

- enter/leave of function/routine/region
  - time stamp, process/thread, function ID
- send/receive of P2P message (MPI)
  - time stamp, sender, receiver, length, tag, communicator
- collective communication (MPI)
  - time stamp, process, root, communicator, # bytes
- hardware performance counter values
  - time stamp, process, counter ID, value
- etc.
Profiling and Tracing

• **Tracing Advantages**
  – preserve temporal and spatial relationships
  – allow reconstruction of dynamic behavior on any required abstraction level
  – profiles can be calculated from traces

• **Tracing Disadvantages**
  – traces can become very large
  – may cause perturbation
  – instrumentation and tracing is complicated
    • event buffering, clock synchronization, …
Instrumentation

- **Instrumentation**: Process of modifying programs to detect and report events

- There are various ways of instrumentation:
  - manually
    - large effort, error prone
    - difficult to manage
  - automatically
    - via source to source translation
    - via compiler instrumentation
    - Program Database Toolkit (PDT)
    - OpenMP Pragma And Region Instrumenter (Opari)
Open Source Trace File Format (OTF)

- Open source trace file format
- Available at http://www.tu-dresden.de/zh/otf
- Includes powerful libotf for reading/parsing/writing in custom applications
- Multi-level API:
  - High level interface for analysis tools
  - Low level interface for trace libraries
- Actively developed by TU Dresden in cooperation with the University of Oregon and the Lawrence Livermore National Laboratory
Practical Instrumentation

• Instrumentation with VampirTrace
  – hide instrumentation in compiler wrapper
  – use underlying compiler, add appropriate options

  
  CC = mpicc

  CC = vtcc –vt:cc mpicc

• Test Run
  – user representative test input
  – set parameters, environment variables, etc.
  – perform trace run

• Get Trace
Source Code Instrumentation

Manually or automatically

```
int foo(void* arg) {
    if (cond) {
        return 1;
    }
    return 0;
}
```

```
int foo(void* arg) {
    enter(7);
    if (cond) {
        leave(7);
        return 1;
    }
    leave(7);
    return 0;
}
```
VAMPIR & VAMPIRTRACE
HANDS-ON: NPB 3.3 BT-MPI
• Move into tutorial directory in your home directory

    % cd NPB3.3-MPI

• Select the VampirTrace compiler wrappers

    % gedit config/make.def
    -> comment out line 32, resulting in:
    ...
    32: #MPIF77 = mpif77
    ...
    -> remove the comment from line 38, resulting in:
    ...
    38: MPIF77 = vtf77 -vt:f77 mpif77
    ...
    -> comment out line 88, resulting in:
    ...
    88: #MPICC = mpicc
    ...
    -> remove the comment from line 94, resulting in:
    ...
    94: MPICC = vtcc -vt:cc mpicc
    ...
Hands-on: NPB 3.3 BT-MPI

- **Build benchmark**
  ```
  % make clean; make suite
  ```

- **Launch as MPI application**
  ```
  % cd bin.vampir; export VT_FILE_PREFIX=bt_1_initial
  % mpiexec -np 16 bt_W.16
  
  NAS Parallel Benchmarks 3.3 -- BT Benchmark
  
  Size: 24x 24x 24
  Iterations: 200  dt: 0.0008000
  Number of active processes: 16
  
  Time step 1
  ...
  Time step 180
  [0] VampirTrace: Maximum number of buffer flushes reached \n  (VT_MAX_FLUSHES=1)
  [0] VampirTrace: Tracing switched off permanently
  Time step 200
  ...
• Resulting trace files

```
% ls -alh
4,1M bt_1_initial.16
4,9K bt_1_initial.16.0.def.z
29 bt_1_initial.16.0.marker.z
12M bt_1_initial.16.10.events.z
12M bt_1_initial.16.1.events.z
11M bt_1_initial.16.2.events.z
12M bt_1_initial.16.3.events.z
...
11M bt_1_initial.16.c.events.z
12M bt_1_initial.16.d.events.z
12M bt_1_initial.16.e.events.z
12M bt_1_initial.16.f.events.z
66 bt_1_initial.16.otf
```

• Visualization with VampirServer

```
% mpirun -np <X> vngd
% vng &
```
Hands-on: NPB 3.3 BT-MPI
Hands-on: NPB 3.3 BT-MPI
Hands-on: NPB 3.3 BT-MPI
Hands-on: NPB 3.3 BT-MPI

- Increase the number of buffer flushes
  
  ```
  % export VT_MAX_FLUSHES=10
  ```

- Set a new file prefix
  
  ```
  % export VT_FILE_PREFIX=bt_2_multiple_flushes
  ```

- Launch as MPI application
  
  ```
  % mpiexec -np 16 bt_W.16
  ```
Hands-on: NPB 3.3 BT-MPI
Hands-on: NPB 3.3 BT-MPI
Hands-on: NPB 3.3 BT-MPI

- Decrease number of buffer flushes by increasing the buffer size

  `% export VT_MAX_FLUSHES=1 VT_BUFFER_SIZE=120M`

- Set a new file prefix

  `% export VT_FILE_PREFIX=bt_3_buffer_120M`

- Launch as MPI application

  `% mpiexec -np 16 bt_W.16`
Hands-on: NPB 3.3 BT-MPI
Function Filtering

• Filtering is one of the ways to reduce trace size
• Environment variable `VT_FILTER_SPEC`
  ```
  % export VT_FILTER_SPEC = /home/user/filter.spec
  ```

• Filter definition file contains a list of filters
  ```
  my_*;test_* -- 1000
  debug_* -- 0
  calculate -- -1
  * -- 1000000
  ```

• See also the `vtfilter` tool
  – can generate a customized filter file
  – can reduce the size of existing trace files
Function Grouping

- Groups can be defined for related functions
  - Groups can be assigned different colors, highlighting different activities
- Environment variable `VT_GROUPS_SPEC`
  ```
  % export VT_GROUPS_SPEC = /home/user/groups.spec
  ```
- Group file contains a list of associated entries
  ```
  CALC=calculate
  MISC=my*;test
  UNKNOWN=*  
  ```
Hands-on: NPB 3.3 BT-MPI

- Generate filter specification file

```
% vtfilter -gen -fo filter.txt -r 10 -stats \ 
  -p bt_3_buffer_120M.otf
% export VT_FILTER_SPEC=filter.txt
```

- Set a new file prefix

```
% export VT_FILE_PREFIX=bt_4_filter
```

- Launch as MPI application

```
% mpiexec -np 16 bt_W.16
```
Hands-on: NPB 3.3 BT-MPI
Hands-on: NPB 3.3 BT-MPI
• PAPI counters can be included in traces
  – If VampirTrace was built with PAPI support
  – If PAPI is available on the platform

• VT_METRICS specifies a list of PAPI counters

```bash
% export VT_METRICS = PAPI_FP_OPS:PAPI_L2_TCM
```

• see also the PAPI commands papi_avail and papi_command_line
Memory Allocation and I/O counters

- Memory allocation counters can be recorded:
  - If VampirTrace build with memory allocation tracing support
  - If GNU glibc is used on the platform
- Intercepts glibc functions like “malloc” and “free”
- Environment variable **VT_MEMTRACE**

```text
% export VT_MEMTRACE = yes
```

- I/O counters can be included in traces
  - If VampirTrace was build with I/O tracing support
- Standard I/O calls like “open” and “read” are recorded
- Environment variable **VT_IOTRACE**

```text
% export VT_IOTRACE = yes
```
Hands-on: NPB 3.3 BT-MPI

• Record PAPI hardware counters

% papi_avail
% papi_event_chooser PRESET PAPI_FP_OPS
% export VT_METRICS=PAPI_FP_OPS:PAPI_L2_TCM

• Set a new file prefix

% export VT_FILE_PREFIX=bt_5_papi

• Launch as MPI application

% mpiexec -np 16 bt_W.16
Hands-on: NPB 3.3 BT-MPI
Hands-on: NPB 3.3 BT-MPI
Hands-on: NPB 3.3 BT-MPI
control options by environment variables:

- VT_PFORM_GDIR: Directory for final trace files
- VT_PFORM_LDIR: Directory for intermediate files
- VT_FILE_PREFIX: Trace file name
- VT_BUFFER_SIZE: Internal trace buffer size
- VT_MAX_FLUSHES: Mac number of buffer flushes
- VT_MEMTRACE: Enable memory allocation tracing
- VT_IOTRACE: Enable I/O tracing
- VT_MPITRACE: Enable MPI tracing
- VT_FILTER_SPEC: Name of filter definition file
- VT_GROUPS_SPEC: Name of grouping definition file
- VT_METRICS: PAPI counter selection
VAMPIR & VAMPIRTRACE
Finding Performance Bottlenecks
Finding Bottlenecks

• Trace Visualization
  – Vampir provides a number of display types
  – each allows many different options

• Advice
  – identify essential parts of an application (initialization, main iteration, I/O, finalization)
  – identify important components of the code (serial computation, MPI P2P, collective MPI, OpenMP)
  – make a hypothesis about performance problems
  – consider application’s internal workings if known
  – select the appropriate displays
  – use statistic displays in conjunction with timelines
Finding Bottlenecks

• Communication
• Computation
• Memory, I/O, etc.
• Tracing itself
Bottlenecks in Communication

- communications as such (dominating over computation)
- late sender, late receiver
- point-to-point messages instead of collective communication
- unmatched messages
- overcharge of MPI’s buffers
- bursts of large messages (bandwidth)
- frequent short messages (latency)
- unnecessary synchronization (barrier)

all of the above usually result in high MPI time share
Bottlenecks in Communication

Example: prevalent communication
Bottlenecks in Communication

prevalent communication: MPI_Allreduce
Bottlenecks in Communication

prevalent communication: timeline view
Bottlenecks in Communication

Propagated Delays in MPI_SendReceiveReplace
Bottlenecks in Communication

unnecessary MPI_Barsriers
Patterns of successive MPI_Allreduce calls
Further Bottlenecks

- unbalanced computation
  - single late comer

- strictly serial parts of program
  - idle processes/threads

- very frequent tiny function calls
- sparse loops
Further Bottlenecks

Example: Idle OpenMP threads
Bottlenecks in Computation

- **memory bound computation**
  - inefficient L1/L2/L3 cache usage
  - TLB misses
  - detectable via HW performance counters

- **I/O bound computation**
  - slow input/output
  - sequential I/O on single process
  - I/O load imbalance

- **exception handling**
Bottlenecks in Computation

low FP rate due to heavy cache misses
Bottlenecks in Computation

low FP rate due to heavy FP exceptions
Bottlenecks in Computation

irregular slow I/O operations
Effects due to Tracing

- Measurement overhead
  - Especially grave for tiny function calls
  - Solve with selective instrumentation

- Long/frequent/asynchronous trace buffer flushes
- Too many concurrent counters

- Heisenbugs
Trace buffer flushes are explicitly marked in the trace. It is rather harmless at the end of a trace as shown here.
Conclusions and Outlook

• performance analysis very important in HPC

• use performance analysis tools for profiling and tracing
• do not spend effort in DIY solutions, e.g. like printf-debugging

• use tracing tools with some precautions
  – overhead
  – data volume

• let us know about problems and about feature wishes
• vampirsupport@zih.tu-dresden.de
Vampir and VampirTraces are available at http://www.vampir.eu and http://www.tu-dresden.de/zih/vampirtrace/, get support via vampirsupport@zih.tu-dresden.de
Wrapper Functions

• provide wrapper functions
  – call instrumentation function for notification
  – call original target for functionality
  – via preprocessor directives:

```c
#define MPI_Init WRAPPER_MPI_Init
#define MPI_Send WRAPPER_MPI_Send
```

• via library preload:
  – preload instrumented dynamic library
• suitable for standard libraries (e.g. MPI, glibc)
Each MPI function has to names:
- MPI_xxx and PMPI_xxx
Replacement of MPI routines at link time
Compiler Instrumentation

gcc –finstrument-functions –c foo.c

void __cyg_profile_func_enter( <args> );
void __cyg_profile_func_exit( <args> );

- many compilers support this: GCC, Intel, IBM, PGI, NEC, Hitachi, Sun Fortran, ...
- no source code modification necessary
Dynamic Instrumentation

- modify executable in file or binary in memory
- insert instrumentation calls
- very platform/machine dependent, expensive

DynInst project (http://www.dyninst.org)
  - common interface
  - supported platforms: Alpha/Tru64, MIPS/IRIX, PowerPC/AIX, Sparc/Solaris, x86/Linux x86/Windows, ia64/Linux
Partners with pictures

Forschungszentrum Jülich
- Jülich Supercomputing Centre

RWTH Aachen University
- Center for Computing and Communication

Technical University of Dresden
- Center for Information Services and High Performance Computing

University of Tennessee
- Innovative Computing Laboratory

Technical University of München
- Chair for Computer Architecture

University of Stuttgart
- High Performance Computing Centre