Debuggers and Performance Tools

February 2015  |  Markus Geimer, Alexandre Strube
Outline

Make it work, make it right, make it fast.

Kent Beck

Debuggers:
- STAT
- TotalView

Performance Tools:
- Score-P
- Scalasca
- Vampir
- TAU
- HPCToolkit

Local module setup
UNITE

- UNiform Integrated Tool Environment
  - Standardizes tool access and documentation
    - Currently in use at JSC, RWTH, ZIH
  - Based on “module” command
    - Standardized tool and version identification
      - `<tool>/<version>-<special>`
      - `<special>`: optional indicator if tool is specific for a MPI library, compiler, or 32/64 bit mode
  - Tools only visible after
    - `module load UNITE` # once per session
  - Basic usage and pointer to tool documentation via
    - `module help <tool>`
% module load UNITE
UNITE loaded
% module help scalasca/1.4.3
Module Specific Help for scalasca/1.4.3:

Scalasca: Scalable Performance Analysis of Large-Scale Parallel Applications
Version 1.4.3

Basic usage:
1. Instrument application with skin
2. Collect & analyze execution measurement with scan
3. Examine analysis results with square

For more information:
- See ${SCALASCA_ROOT}/doc/manuals/QuickReference.pdf
  or type "scalasca -h"
- http://www.scalasca.org
- mailto:scalasca@fz-juelich.de
Documentation

- Use “module avail” to check latest status

- Websites
  - http://www.fz-juelich.de/ias/jsc/juqueen/
    - User Info
    - Debugging
    - Performance Analysis (⚠)
  - http://www.vi-hps.org/training/material/
    - Performance Tools LiveDVD image
    - Links to tool websites and documentation
    - Tutorial slides
Debugging on JUQUEEN

February 2015  |  Alexandre Strube
STAT: Stack Trace Analysis Tool

- Very lightweight helper tool
- Shows merged call tree of whole program
- Useful to detect deadlocks
- Scales to millions of processes
  - [http://www.hpcwire.com/hpcwire/2012-12-03/bug_repellent_for_supercomputers_proves_effective.html](http://www.hpcwire.com/hpcwire/2012-12-03/bug_repellent_for_supercomputers_proves_effective.html)
- Pinpoint individual problems
- **NOT** a real/full debugger
- [http://www.paradyn.org/STAT/STAT.html](http://www.paradyn.org/STAT/STAT.html)
STAT: Main Window
STAT: Zoom

Which ranks are following

- main
  - 512:[0-511]
  - body()
    - 128:[2,6,10,14,...]
    - 128:[0,4,8,12,16,...]
    - 128:[3,7,11,15,...]
    - 128:[1,5,9,13,17,...]
      - south()
        - 128:[2,6,10,14,...]
        - north()
          - 128:[0,4,8,12,16,...]
          - _sleep
            - 128:[2,6,10,14,...]
      - west()
        - 128:[3,7,11,15,...]
        - east()
          - 128:[1,5,9,13,17,...]
          - south()
            - 128:[3,7,11,15,...]
            - north()
              - 128:[1,5,9,13,17,...]
                - _sleep
                  - 128:[3,7,11,15,...]
STAT: Equivalence Classes
STAT: Equivalence Classes (cont.)

Diagram:
- main
  - 512:[0-511]
  - body()
    - 128:[2,6,10,14,...]
    - 128:[0,4,8,12,16,...]
    - 128:[3,7,11,15,...]
    - 128:[1,5,9,13,17,...]
  - south() => north() => _sleep
  - north() => _sleep
  - west() => east() => south() => north() => _sleep
  - east() => south() => north() => _sleep
STAT: Recipe

- Compile and link your program with debug option: -g
- Load modules

```
% ssh -X user@juqueen
[...]
juqueen% module load UNITE stat
UNITE loaded
stat/2.1 loaded
juqueen% STATGUI
```

- Submit job and attach to it from GUI

**CAVEAT:**
- Job needs to be started by login node where GUI is running
- Add the following entry to the submission script
  ```bash
  #@ requirements = (Machine == "juqueen<n>")
  with <n> being the login node id
  ```
STAT: Attaching to a Job
STAT: Attach TotalView to Subset
Parallel Debugger

- UNIX Symbolic Debugger for C, C++, f77, f90, PGI HPF, assembler programs
- “Standard” debugger
- Special, non-traditional features
  - Multi-process and multi-threaded
  - C++ support (templates, inheritance, inline functions)
  - F90 support (user types, pointers, modules)
  - 1D + 2D Array Data visualization
  - Support for parallel debugging (MPI: automatic attach, message queues, OpenMP, pthreads)
  - Scripting and batch debugging
  - Memory Debugging
- http://www.roguewave.com
TotalView: Recipe

- Compile and link your program with debug option: `-g`
- Use absolute paths for source code info: `-qfullpath`
- In case of optimized codes (XL), keep function call parameters: `-qkeepparm`
- Load modules

```
% ssh -X user@juqueen
[...]
juqueen% module load UNITE totalview
UNITE loaded
totalview/8.14.0-16-mrnet loaded

juqueen% mpixlcxx hello.cpp -qfullpath -qkeepparm -g -o helloworld
juqueen%
```
TotalView: Interactive Startup

- Interactively: call the `lltv` script
  - Creates a LoadLeveler batch script with required TotalView parameters
  - If user cancels the script, it cancels the debugging job (does not eat your computing quota)

**NOTE:**
- License limited to 2048 MPI ranks (shared between all users)
- Attaching to subset is recommended
## TotalView: `lltv` Launch Script

```
% lltv -n <nodes> : -default_parallel_attach_subset=
 <rank-range> runjob -a --exe <program> -p <num>
```

- Starts `<program>` with `<nodes>` and `<num>` processes per node, attaches to `<rank-range>`:
  - Rank: that rank only
  - RankX-RankZ: all ranks, both inclusive
  - RankX-RankZ: stride every `stride`th between RankX and RankZ
- Example:

```
% lltv -n 2 : -default_parallel_attach_subset=
 2-6 runjob -a --exe helloworld -p 64
```

Creating LoadLeveler Job
Submitting LoadLeveler Interactive Job for Totalview
Wait for job juqueen1c1.32768.0 to be started:.........
TotalView: Execution Recipe

- TotalView tries to debug “runjob” and shows no source code
  - Ignore it and press “GO”

- After some seconds, TotalView will detect parallel execution and ask if it should stop. Yes, it should stop.
- To find the correct point file/function to debug, use the “File-Open” command.
- Set your breakpoints, and press “GO” again. Debugging session will then start.

- To see a variable’s contents, double click on it in the source.
TotalView: Main Window

- Stack trace
- Toolbar for common options
- Local variables for selected stack frame
- Source code window
- Break points
TotalView: Tools Menu

- Call Graph
  - Message queue graph
- Data visualization
Performance Analysis Tools on JUQUEEN

February 2015  |  Markus Geimer
Typical Performance Analysis Procedure

- Do I have a performance problem at all?
  - Time / speedup / scalability measurements
- **What** is the key bottleneck (computation / communication)?
  - MPI / OpenMP / flat profiling
- **Where** is the key bottleneck?
  - Call-path profiling, detailed basic block profiling
- **Why** is it there?
  - Hardware counter analysis
  - Trace selected parts (to keep trace size manageable)
- Does the code have scalability problems?
  - Load imbalance analysis, compare profiles at various sizes function-by-function
Remark: No Single Solution is Sufficient!

A combination of different methods, tools and techniques is typically needed!

- Analysis
  - Statistics, visualization, automatic analysis, data mining, ...
- Measurement
  - Sampling / instrumentation, profiling / tracing, ...
- Instrumentation
  - Source code / binary, manual / automatic, ...
Critical Issues

- Accuracy
  - Intrusion overhead
    - Measurement itself needs time and thus lowers performance
  - Perturbation
    - Measurement alters program behavior
    - E.g., memory access pattern
  - Accuracy of timers & counters

- Granularity
  - How many measurements?
  - How much information / processing during each measurement?

-tradeoff: Accuracy vs. Expressiveness of data
Score-P

- Community instrumentation and measurement infrastructure
  - Developed by a consortium of performance tool groups
  - Next generation measurement system of
    - Scalasca 2.x
    - Vampir
    - TAU
    - Periscope
- Common data formats improve tool interoperability
- http://www.score-p.org
Collection of trace-based performance analysis tools

- Specifically designed for large-scale systems
- Unique features:
  - Scalable, automated search for event patterns representing inefficient behavior
  - Scalable identification of the critical execution path
  - Delay / root-cause analysis
- Based on Score-P for instrumentation and measurement
  - Includes convenience / post-processing commands providing added value

http://www.scalasca.org
What is the Key Bottleneck?

- Generate flat MPI profile using Score-P/Scalasca
  - Only requires re-linking
  - Low runtime overhead

- Provides detailed information on MPI usage
  - How much time is spent in which operation?
  - How often is each operation called?
  - How much data was transferred?

- Limitations:
  - Computation on non-master threads and outside of MPI_Init/MPI_Finalize scope ignored
Flat MPI Profile: Recipe

1. Prefix your *link command* with “scorep --nocompiler”

2. Prefix your MPI *launch command* with “scalasca -analyze”

3. After execution, examine analysis results using “scalasca -examine scorep_<title>”
Flat MPI Profile: Example

% module load UNITE scorep scalasca
% mpixlf90 -O3 -qsmp=omp -c foo.f90
% mpixlf90 -O3 -qsmp=omp -c bar.f90
% scorep --nocompiler \
   mpixlf90 -O3 -qsmp=omp -o myprog foo.o bar.o

###############################################
## In the job script:  ##
###############################################

module load UNITE scalasca
scalasca -analyze \
   runjob --ranks-per-node P --np n [...] --exe ./myprog

###############################################
## After job finished:  ##
###############################################

% scalasca -examine scorep_myprog_Ppnx_t_sum
Flat MPI Profile: Example (cont.)

Aggregate execution time on master threads

Time spent in a particular MPI call

Time spent in selected call as percentage of total time
Where is the Key Bottleneck?

- Generate **call-path profile** using Score-P/Scalasca
  - Requires re-compilation
  - Runtime overhead depends on application characteristics
  - Typically needs some care setting up a good measurement configuration
    - Filtering
    - Selective instrumentation

- Option 1 (recommended):
  Automatic compiler-based instrumentation

- Option 2:
  Manual instrumentation of interesting phases, routines, loops
Call-path Profile: Recipe

1. Prefix your *compile & link commands* with "scorep"

2. Prefix your MPI *launch command* with "scalasca -analyze"

3. After execution, compare overall runtime with uninstrumented run to determine overhead

4. If overhead is too high
   1. Score measurement using "scalasca -examine -s scorep_<title>"
   2. Prepare filter file
   3. Re-run measurement with filter applied using prefix "scalasca -analyze -f <filter_file>"

5. After execution, examine analysis results using "scalasca -examine scorep_<title>"
Call-path Profile: Example

```bash
% module load UNITE scorep scalasca
% scorep mpxlf90 -O3 -qsmp=omp -c foo.f90
% scorep mpxlf90 -O3 -qsmp=omp -c bar.f90
% scorep \
  mpxlf90 -O3 -qsmp=omp -o myprog foo.o bar.o

##########################
## In the job script:  ##
##########################

module load UNITE scalasca
calasca -analyze \
  runjob --ranks-per-node "P" --np n [...] --exe ./myprog
```
Call-path Profile: Example (cont.)

- Estimates trace buffer requirements
- Allows to identify candidate functions for filtering
  - Computational routines with high visit count and low time-per-per-visit ratio
- Region/call-path classification
  - MPI (pure MPI library functions)
  - OMP (pure OpenMP functions/regions)
  - USR (user-level source local computation)
  - COM (“combined” USR + OpenMP/MPI)
  - ANY/ALL (aggregate of all region types)

% scalasca -examine -s epik_myprog_Ppnx_t_sum
scorep-score -r ./epik_myprog_Ppnx_t_sum/profile.cubex
INFO: Score report written to ./scorep_myprog_Ppnx_t_sum/scorep.score
Call-path Profile: Example (cont.)

% less scorep_myprog_Ppdx_t_sum/scorep.score

Estimated aggregate size of event trace: 162GB
Estimated requirements for largest trace buffer (max_buf): 2758MB
Estimated memory requirements (SCOREP_TOTAL_MEMORY): 2822MB
(hint: When tracing set SCOREP_TOTAL_MEMORY=2822MB to avoid intermediate flushes or reduce requirements using USR regions filters.)

<table>
<thead>
<tr>
<th>flt type</th>
<th>max_buf[B]</th>
<th>visits</th>
<th>time[s]</th>
<th>time[%]</th>
<th>time/visit[us]</th>
<th>region</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td>2,891,417,902</td>
<td>6,662,521,083</td>
<td>36581.51</td>
<td>100.0</td>
<td>5.49</td>
<td>ALL</td>
</tr>
<tr>
<td>USR</td>
<td>2,858,189,854</td>
<td>6,574,882,113</td>
<td>13618.14</td>
<td>37.2</td>
<td>2.07</td>
<td>USR</td>
</tr>
<tr>
<td>OMP</td>
<td>54,327,600</td>
<td>86,353,920</td>
<td>22719.78</td>
<td>62.1</td>
<td>263.10</td>
<td>OMP</td>
</tr>
<tr>
<td>MPI</td>
<td>676,342</td>
<td>550,010</td>
<td>208.98</td>
<td>0.6</td>
<td>379.96</td>
<td>MPI</td>
</tr>
<tr>
<td>COM</td>
<td>371,930</td>
<td>735,040</td>
<td>34.61</td>
<td>0.1</td>
<td>47.09</td>
<td>COM</td>
</tr>
</tbody>
</table>

USR   921,918,660 2,110,313,472 3290.11 9.0 1.56 matmul_sub
USR   921,918,660 2,110,313,472 5914.98 16.2 2.80 binvcrhs
USR   921,918,660 2,110,313,472 3822.64 10.4 1.81 matvec_sub
USR     41,071,134 | 87,475,200 | 358.56 1.0 4.10 lhsinit
USR     41,071,134 | 87,475,200 | 145.42 0.4 1.66 binvrhs
USR    29,194,256 | 68,892,672  | 86.15 0.2 1.25 exact_solution
OMP    3,280,320 3,293,184 15.81 0.0 4.80 !$omp parallel [...]

[...]
Call-path Profile: Filtering

- In this example, the 6 most frequently called routines are of type USR
- These routines contribute around 35% of total time
  - However, much of that is most likely measurement overhead
    - Frequently executed
    - Time-per-visit ratio in the order of a few microseconds

☞ Avoid measurements to reduce the overhead
☞ List routines to be filtered in simple text file
Filtering: Example

% cat filter.txt
SCOREP_REGION_NAMES_BEGIN
  EXCLUDE
    binvcrhs
    matmul_sub
    matvec_sub
    binvrhs
    lhsinit
    exact_solution
SCOREP_REGION_NAMES_END

- Score-P filtering files support
  - Wildcards (shell globs)
  - Blacklisting
  - Whitelisting
  - Filtering based on filenames
## To verify effect of filter:

% scalasca -examine -s -f filter.txt \
   scorep_myprog_Ppnx_t_sum

#################################
## In the job script:  ##
#################################

module load UNITE scalasca
scalasca -analyze -f filter.txt \
   runjob --ranks-per-node P --np n [...] --exe ./myprog

#################################
## After job finished:  ##
#################################

% scalasca -examine scorep_myprog_Ppnx_t_sum
Call-path Profile: Example (cont.)
Distribution of selected metric across call tree

When expanding, value changes from inclusive to exclusive

Box plot view shows distribution across processes/threads

Selection updates columns to the right
Split base metrics into more specific metrics
Score-P: Advanced Features

- Measurement can be extensively configured via environment variables
  - Check output of “scorep-info config-vars” for details
- Allows for targeted measurements:
  - Selective recording
  - Phase profiling
  - Parameter-based profiling
  - ...
- Please ask us or see the user manual for details
Why is the Bottleneck There?

- This is **highly** application dependent!
- Might require additional measurements
  - Hardware-counter analysis
    - CPU utilization
    - Cache behavior
  - Selective instrumentation
  - Manual/automatic event trace analysis
HW Counter Measurements w/ Score-P

- Score-P supports both PAPI and native counters
- Available counters:

```bash
module load UNITE papi/5.0.1
less $PAPI_ROOT/doc/papi-5.0.1_avail.txt
less $PAPI_ROOT/doc/papi-5.0.1-native_avail.txt
less $PAPI_ROOT/doc/papi-5.0.1_avail-detail.txt
```

- Specify using “SCOREP_METRIC_PAPI” environment variable

```bash
###
## In the job script:  ##
###

module load UNITE scalasca
export SCOREP_METRIC_PAPI="PAPI_FP_OPS,PAPI_TOT_CYC"
scalasca -analyze -f filter.txt \
runjob --ranks-per-node P --np n [...] --exe ./myprog
```
Automatic Trace Analysis w/ Scalasca

- **Idea:** Automatic search for patterns of inefficient behavior
  - Identification of wait states and their root causes
  - Classification of behavior & quantification of significance
  - Scalable identification of the critical execution path

- **Advantages**
  - Guaranteed to cover the entire event trace
  - Quicker than manual/visual trace analysis
  - Helps to identify hot-spots for in-depth manual analysis
Trace Generation & Analysis w/ Scalasca

- Enable trace collection & analysis using “-t” option of “scalasca -analyze”:

```
#############################
## In the job script:  ##
#############################
module load UNITE scalasca
export SCOREP_TOTAL_MEMORY=120MB   # Consult score report
scalasca -analyze -f filter.txt -t \
    runjob --ranks-per-node P --np n [...] --exe ./myprog
```

- **ATTENTION:**
  - Traces can quickly become extremely large!
  - Remember to use proper filtering, selective instrumentation, and Score-P memory specification
  - Before flooding the file system, **ask us for assistance!**
Scalasca Trace Analysis Example

Additional wait-state metrics from the trace analysis

Delay / root-cause metrics

Critical-path profile
Vampir Event Trace Visualizer

- **Offline** trace visualization for Score-P’s OTF2 trace files
- **Visualization of MPI, OpenMP and application events:**
  - All diagrams highly customizable (through context menus)
  - Large variety of displays for **ANY** part of the trace
- [http://www.vampir.eu](http://www.vampir.eu)

- Advantage:
  - Detailed view of dynamic application behavior
- Disadvantage:
  - Requires event traces (huge amount of data)
  - Completely manual analysis
Vampir Displays
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 through clicking on the representation
Vampir: Process and Counter Timelines

- Process timeline show call stack nesting

- Counter timelines for hardware or software counters
Vampir: Execution Statistics

- Aggregated profiling information: execution time, number of calls, inclusive/exclusive

- Available for all / any group (activity) or all routines (symbols)

- Available for any part of the trace ⇒ selectable through time line diagram
Vampir: Process Summary

- Execution statistics over all processes for comparison
- Clustering mode available for large process counts
Vampir: Communication Statistics

- Byte and message count, min/max/avg message length and min/max/avg bandwidth for each process pair
- Message length statistics

- Available for any part of the trace
Vampir: Recipe (JUQUEEN)

1. module load UNITE vampirserver

2. Start Vampir server component (on frontend) using “vampirserver start smp”
   - Check output for port and pid

3. Connect to server from remote machine (see next slide) and analyze the trace

4. vampirserver stop <pid>
   - See above (2.)
Vampir: Recipe (local system)

1. Open SSH tunnel to JUQUEEN using
   
   “ssh -L30000:localhost:<port> juqueen

2. Start Vampir client component
   
   ▪ For example: “/usr/local/zam/unite/bin/vampir”

3. Select
   
   1. “Open other…”
   2. “Remote file”
   3. “Connect” (keep defaults)
   4. File “traces.otf2” from Score-P trace measurement directory
TAU

- Very portable tool set for instrumentation, measurement and analysis of parallel multi-threaded applications
- http://tau.uoregon.edu/

- Supports
  - Various profiling modes and tracing
  - Various forms of code instrumentation
  - C, C++, Fortran, Java, Python
  - MPI, multi-threading (OpenMP, Pthreads, …)
TAU: Instrumentation

- Flexible instrumentation mechanisms at multiple levels
  - Source code
    - manual
    - automatic
    - C, C++, F77/90/95 (Program Database Toolkit (PDT))
    - OpenMP (directive rewriting with Opari)
  - Object code
    - pre-instrumented libraries (e.g., MPI using PMPI)
    - statically-linked and dynamically-loaded (e.g., Python)
  - Executable code
    - dynamic instrumentation (pre-execution) (DynInst)
    - virtual machine instrumentation (e.g., Java using JVMPI)
- Support for performance mapping
- Support for object-oriented and generic programming
TAU: Recipe

1. `module load UNITE tau`  # once per session
2. Specify programming model by setting `TAU_MAKEFILE` to one of `$$TAU_MF_DIR/Makefile.tau-*`
   - **MPI:** `Makefile.tau-bgqtimers-papi-mpi-pdt`
   - **OpenMP/MPI:** `Makefile.tau-bgqtimers-papi-mpi-pdt-openmp-opari`
3. Compile and link with
   - `tau_cc.sh file.c ...`
   - `tau_cxx.sh file.cxx ...`
   - `tau_f90.sh file.f90 ...`
4. Execute with real input data
   Environment variables control measurement mode
   - `TAU_PROFILE`, `TAU_TRACE`, `TAU_CALLPATH`, ...
5. Examine results with `paraprof`
TAU: Basic Profile View

![TAU Profile Image]

**COUNTER NAME: P_WALL_CLOCK_TIME (seconds)**

- **MPI_Allreduce()**
- **116.485**
  - algs::HyperbolicLevelIntegrator3::advance_bdry_fill_create
- **103.256**
  - algs::HyperbolicLevelIntegrator3::advanceLevel()
- **59.009**
  - algs::HyperbolicLevelIntegrator3::fill_new_level_create
- **37.448**
  - mesh::GriddingAlgorithm3::load_balance_boxes
- **32.854**
  - algs::HyperbolicLevelIntegrator3::advance_bdry_fill_comm
- **21.408**
  - mesh::GriddingAlgorithm3::findRefinementBoxes()
- **13.492**
  - algs::HyperbolicLevelIntegrator3::coarsen_fluxsum_create
- **12.657**
  - algs::HyperbolicLevelIntegrator3::coarsen_sync_create
- **10.440**
  - mesh::GriddingAlgorithm3::find_boxes_containing_tags
- **8.921**
  - **MPI_Init()**
- **8.689**
  - mesh::GriddingAlgorithm3::bdry_fill_tags_create
- **7.271**
  - **MPI_Bcast()**
- **7.132**
  - **MPI_Wait()**
- **4.083**
  - algs::HyperbolicLevelIntegrator3::error_bdry_fill_comm
- **3.677**
  - **MPI_Finalize()**
- **3.140**
  - **MPI_Isend()**
- **3.050**
  - **MPI_Waitall()**
- **2.345**
  - mesh::GriddingAlgorithm3::remove_intersections_regrid_all
- **1.727**
  - **MPI_Test()**
- **1.651**
  - algs::HyperbolicLevelIntegrator3::fill_new_level_comm
- **1.391**
  - **MPI_Comm_rank()**
TAU: Callgraph Profile View

Box width and color indicate different metrics.
TAU: 3D Profile View

Height and color indicate different metrics.
- Multi-platform sampling-based call-path profiler
- Works on unmodified, optimized executables
- http://hpctoolkit.org

- Advantages:
  - Overhead can be easily controlled via sampling interval
  - Advantageous for complex C++ codes with many small functions
  - Loop-level analysis (sometimes even individual source lines)
  - Supports POSIX threads

- Disadvantages:
  - Statistical approach that might miss details
  - MPI/OpenMP time displayed as low-level system calls
HPCToolkit: Recipe

1. Compile your code with “-g -qnoipa”
   ▪ For MPI, also make sure your application calls MPI_Comm_rank first on MPI_COMM_WORLD
2. Prefix your link command with “hpclink”
   ▪ Ignore potential linker warnings ;-)  
3. Run your application as usual, specifying requested metrics with sampling intervals in environment variable “HPCRUN_EVENT_LIST”
4. Perform static binary analysis with “hpcstruct --loop-fwd-subst=no <app>”
5. Combine measurements with “hpcprof -S <struct file> \ -I “<path_to_src>/∗” <measurement_dir>”
6. View results with “hpcviewer <hpct_database>”
HPCToolkit: Metric Specification

- General format:
  “name@interval [;name@interval …]”

- Possible sample sources:
  - WALLCLOCK
  - PAPI counters
  - IO (use w/o interval spec)
  - MEMLEAK (use w/o interval spec)

- Interval: given in microseconds
  - E.g., 10000 → 100 samples per second
Example: hpcviewer

```c
for (i=1+mod; i<=nxl; i+=2)
    delta = omega*(fkj[i+1] + fkj[i-1] +fkj[i] + fkj[i])
             -4.0*fkj[i] - rkj[i]);
    tmpres += fabs(delta);
    fkj[i] = tmp[i] + delta;
    tmperr += fabs(fkj[i] - akj[i]);
```