High-Level Application Specific Performance Analysis using the G-PM Tool

(A Case Study)

Roland Wismüller

Operating Systems and Distributed Systems
University of Siegen, Germany
roland.wismueller@uni-siegen.de

1. Motivation

Performance analysis of interactive Grid applications

- Background: EU IST-project CrossGrid (03/2002 - 02/2005)
  - applications: medical, flooding, HEP, air pollution
  - tools, middleware, and testbed
- Development of G-PM performance analysis tool, based on OCM-G application monitoring system
- Requirements on performance analysis tools, e.g.:
  - measurement and breakdown of response times
  - convergence behavior of solvers (progress of computation)
  - performance numbers for individual interactions
  - performance measurements for specific program phases

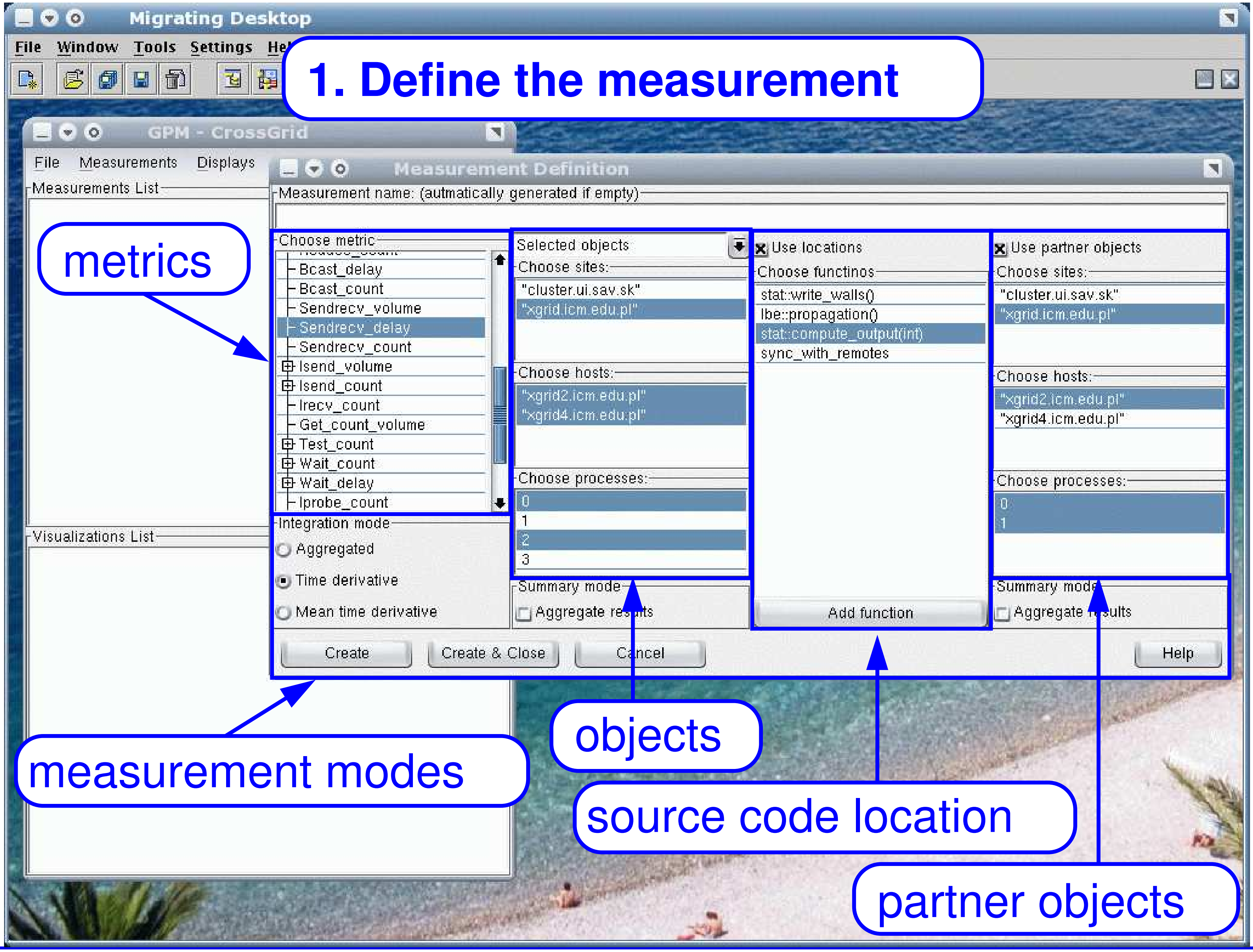
2. The G-PM tool: Goals

- **On-line analysis and visualization**
  - direct correlation between performance and interactions
- **Support for user-defined metrics**
  - e.g. response time, convergence rate, data per interaction
  - map performance data to user's conceptual model
  - with minimal effort for the user (application programmer)
- **Meaningful performance measurement in the Grid**
  - relate application performance to performance of Grid infrastructure
- **Efficient, distributed data evaluation**
  - no centralized processing of event traces

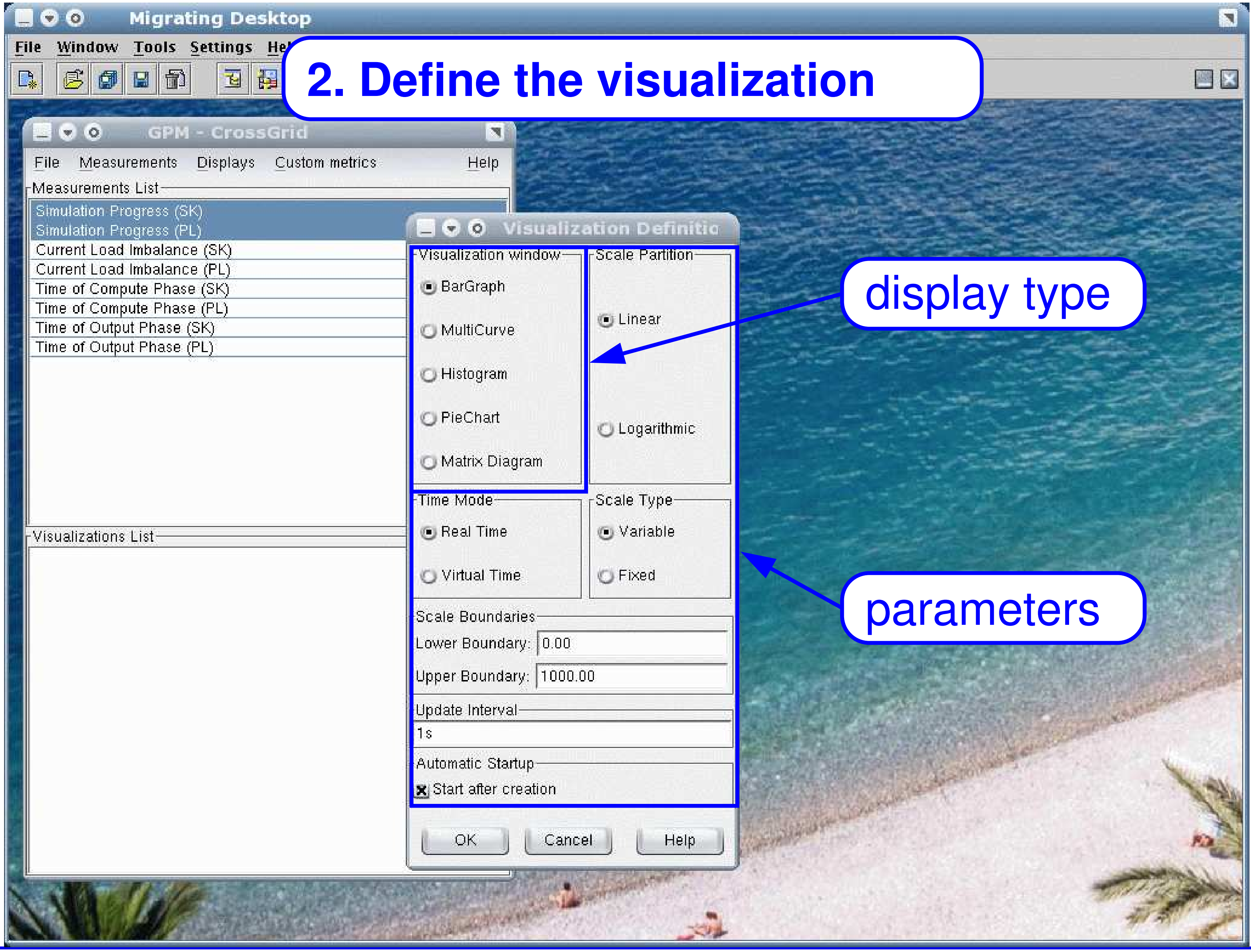
2. The G-PM tool: Basic Approach

- **G-PM tool offers a rich set of standard metrics**
  - related to applications, e.g. time and volume of communication and I/O, CPU and memory usage, ...
    - automatic instrumentation based on counters and timers, transparent to the user
  - related to Grid infrastructure, e.g. load of nodes, ...
- **User-defined metrics and probes**
  - probes: minimal manual source code instrumentation
    - important events, plus application data (optional)
  - user can define new metrics at run-time
    - basis: existing metrics and information from probes
    - flexibility, re-use of metrics and probes

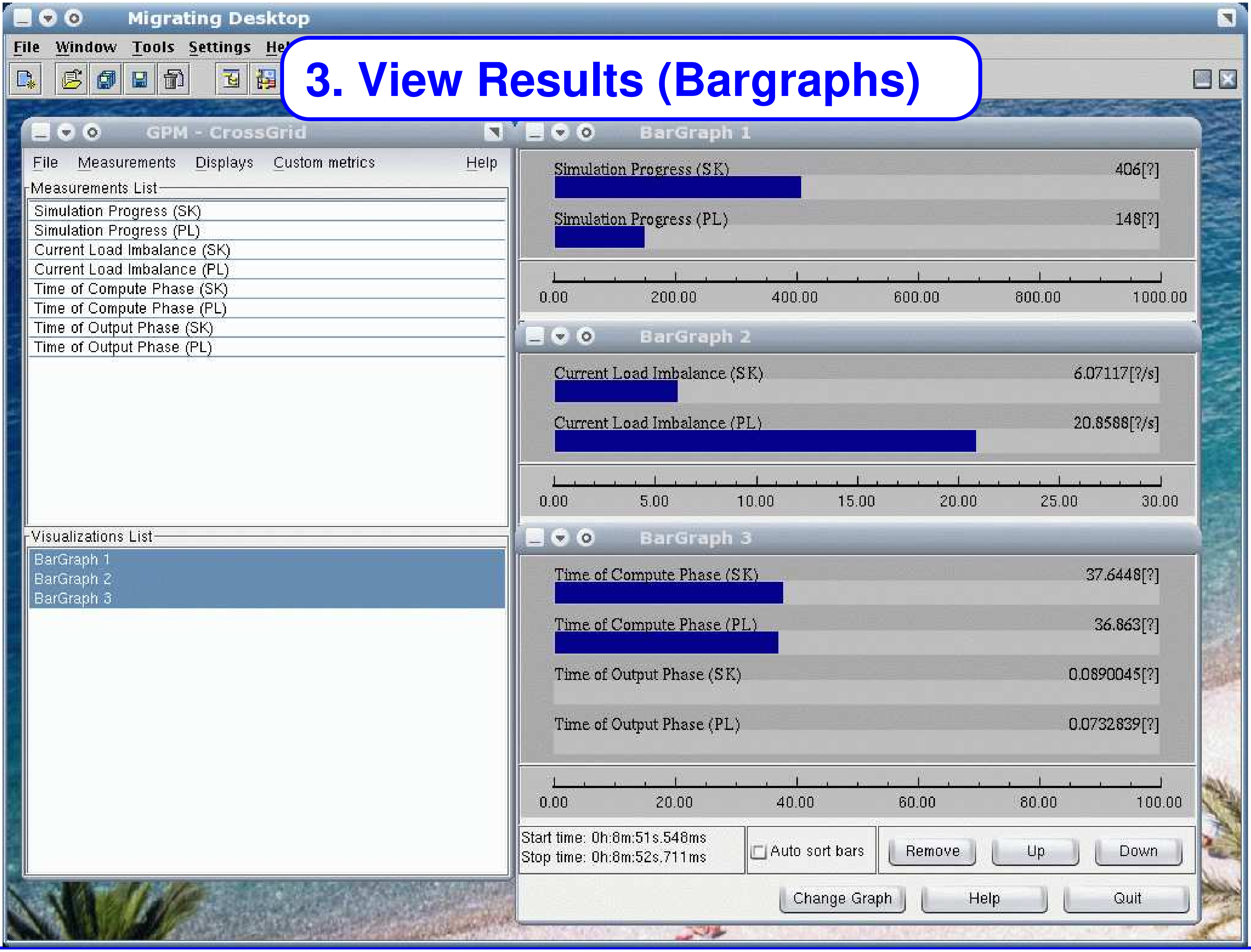
2. The G-PM tool: Definition of Measurements



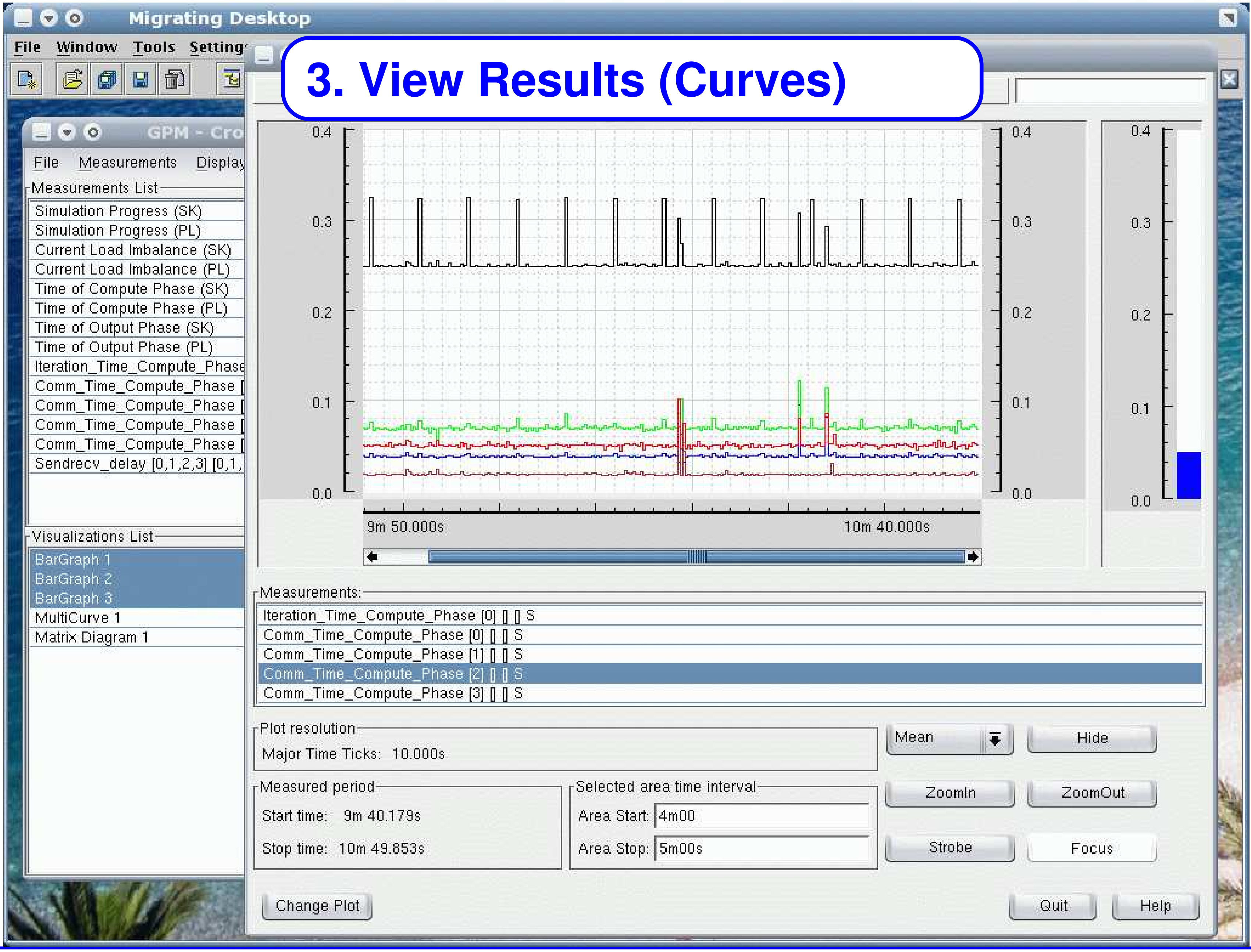
2. The G-PM tool: Definition of Measurements



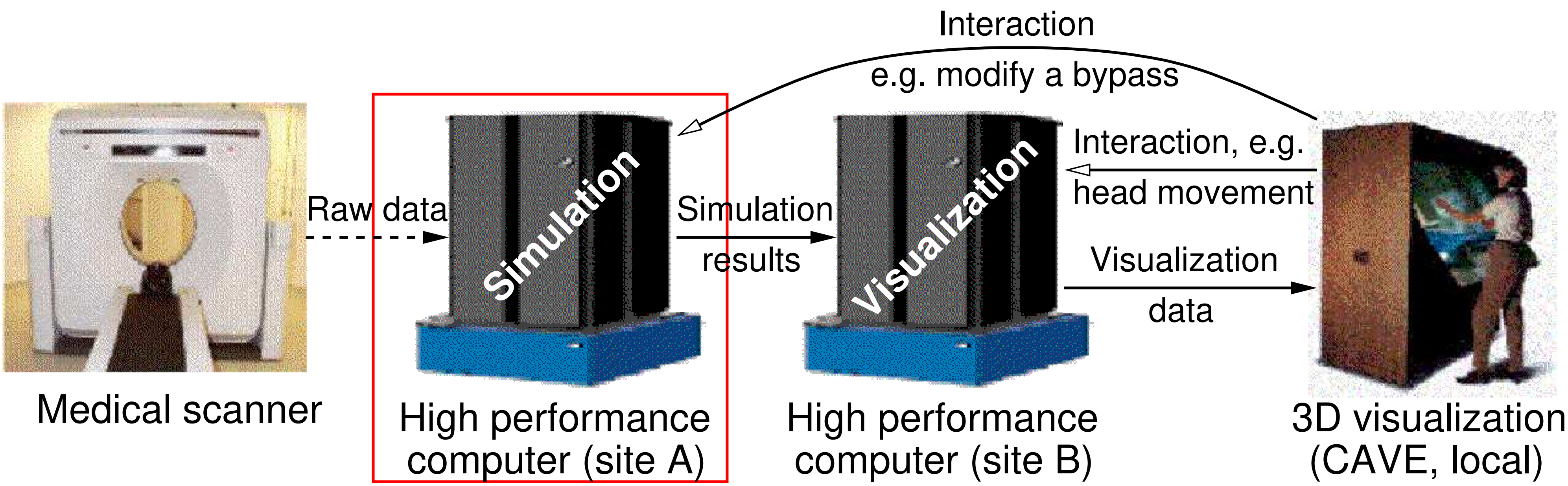
2. The G-PM tool: Definition of Measurements



2. The G-PM tool: Definition of Measurements



3. Use-Case: Blood Flow Simulation



- Pretreatment planning in vascular surgery
  - simulation / interactive optimization of bypasses
- Developed by Univ. of Amsterdam (Sloot, Tirado-Ramos, Hoekstra)
  - Grid-enabled within the EU CrossGrid project
- Simulation based on iterative Lattice-Boltzmann method

3. Structure of Blood Flow Solver

- Iterative solver: main loop iterates over simulated time
- Parallelization based on domain decomposition
  - one dimension of simulation volume (geometry of arteries) is distributed evenly across the processors
  - in each iteration: synchronous exchange of border values
    - linear communication topology
- Structure of main loop:

```
for (i=0; i<niter; i++) {
  // local computation of one time step
  // exchange data with neighbors (MPI_Sendrecv)
  // write data to output file
}
```

3. Instrumentation with Probes in G-PM

- Insert probes at relevant points in the source code
  - probes mark events in the program execution (e.g., interactions, program phases, ...)
  - virtual time parameter for identifying corresponding events
  - additional parameters can pass application specific data
- Set up specification file for the probes
- Recompile modified files and link with OCM-G link wrapper:

```
cg-ocmg-cc -probes probes.c gcc -o myapp ... -lmpich
```

- Probes do not yet define any metrics
  - metrics can be defined at run-time
    - without re-instrumenting the application
  - probes can be shared (re-used) by many metrics

3. Instrumentation of Blood-Flow Solver

Main loop of blood flow solver

```
for (i=0; i<niter; i++) {

  // local computation of one time step
  // exchange data with neighbors (MPI_Sendrecv)

  // write data to output file

}
```

3. Instrumentation of Blood-Flow Solver

Main loop of blood flow solver (instrumented)

```
for (i=0; i<niter; i++) {
   iterBegin(i);
   // local computation of one time step
   // exchange data with neighbors (MPI_Sendrecv)
   outputBegin(i);
   // write data to output file
   iterEnd(i);
}
```

3. Instrumentation of Blood-Flow Solver

Main loop of blood flow solver (instrumented)

```
for (i=0; i<niter; i++) {
   iterBegin(i);
   // local computation of one time step
   // exchange data with neighbors (MPI_Sendrecv)
   outputBegin(i);
   // write data to output file
   iterEnd(i);
}
```

➥ Loop counter `i` naturally serves as the virtual time

3. Instrumentation of Blood-Flow Solver

Main loop of blood flow solver (instrumented)

```
for (i=0; i<niter; i++) {
   iterBegin(i);
   // local computation of one time step
   // exchange data with neighbors (MPI_Sendrecv)
   outputBegin(i);
   // write data to
   iterEnd(i);
}
```

- Probe specification file:

```
void iterBegin(int vt){}
void outputBegin(int vt){}
void iterEnd(int vt){}
```

3.1 Experiment 1

Overview measurements

- Program progress (number of iterations executed so far)
- Performance loss due to load imbalance

- Comparing measurements of two different program runs
  - on a cluster in Krakow, Poland
  - on a cluster in Bratislava, Slovakia
    - with improved data partitioning

3.1 PMSL Metrics for Number of Loop Iterations

```
LoopExecutions(Process p, TimeInterval ti)
    : Unit("iterations")
{
    PROBE iterEnd(Process, VirtualTime);
    VirtualTime vt;
    Value[] val;
    val[vt] = 1 AT iterEnd(p,vt);
    return SUM(val[vt] WHERE val[vt].time IN ti);
}
```

- The `iterEnd` probe events create a stream of 1's
- The result is the some over all stream values produced in the measured time interval

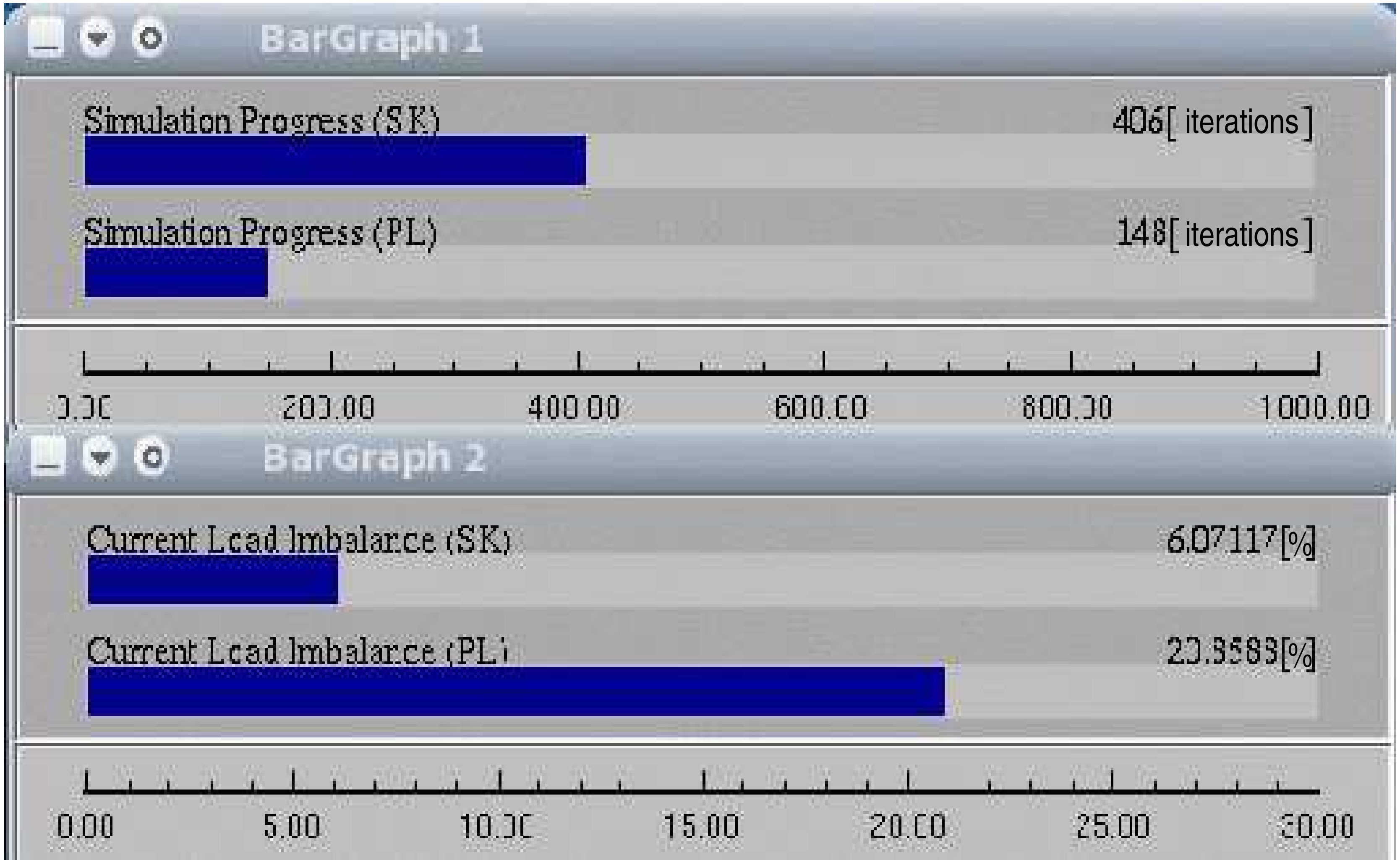
3.1 PMSL Metrics for load imbalance

```
WaitTimeInIter(Process p, VirtualTime vt)
  : Unit("s")
{
  PROBE iterBegin(Process, VirtualTime);
  PROBE iterEnd(Process, VirtualTime);
  return Sendrecv_delay(p,[START,NOW]) AT iterEnd(p, vt)
       - Sendrecv_delay(p,[START,NOW]) AT iterBegin(p, vt);
}

LoadImbalance(Process[] procs, TimeInterval t)
  : Unit("s")
{
  Process p; Value[] imbal; VirtualTime vt;
  imbal[vt] = MAX(WaitTimeInIter(p, vt) WHERE p IN procs)
            - MIN(WaitTimeInIter(p, vt) WHERE p IN procs);
  return SUM(imbal[vt] WHERE imbal[vt].time IN t);
}
```

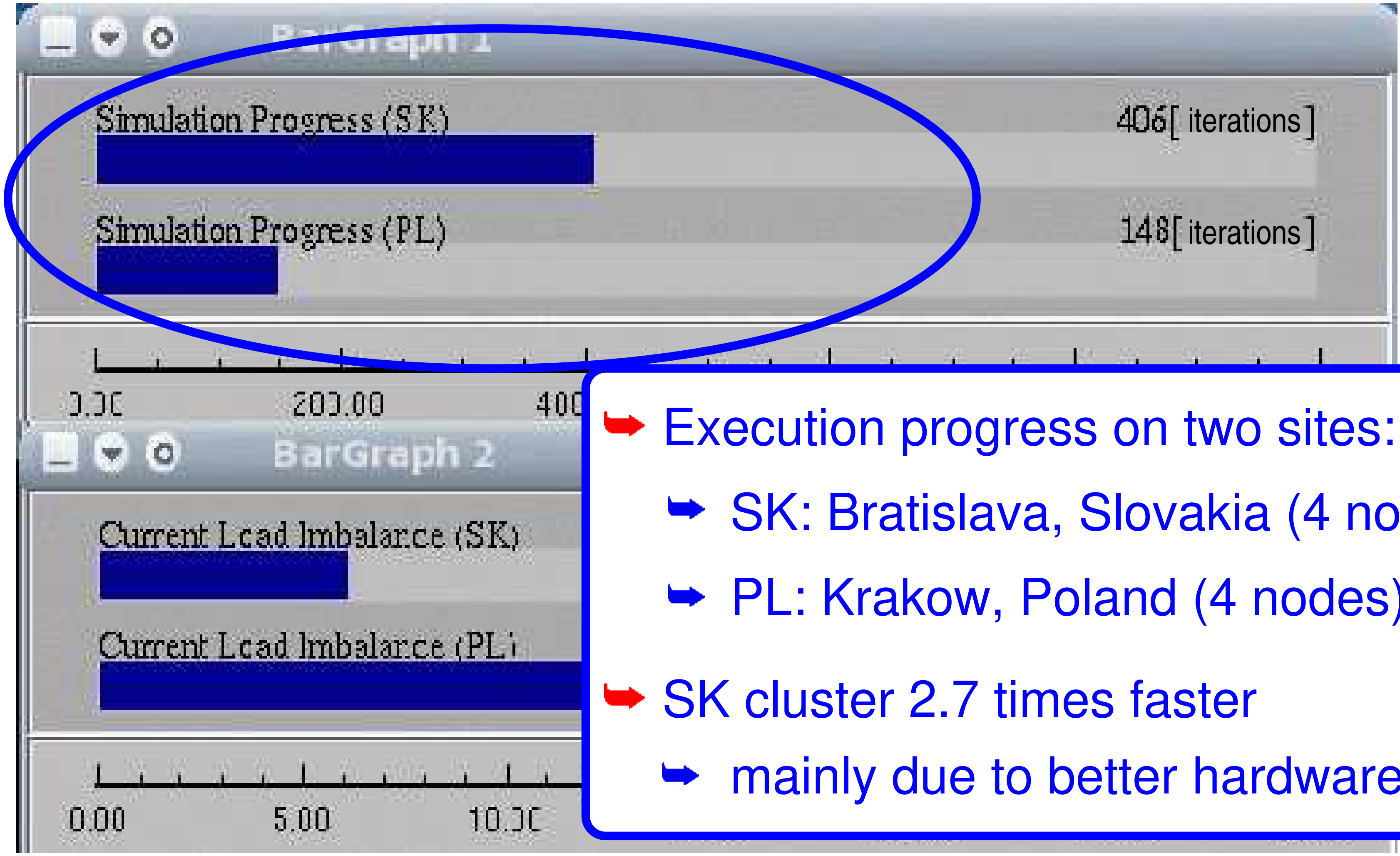
3.1 Overview Measurements

Progress of computation / load imbalance



3.1 Overview Measurements

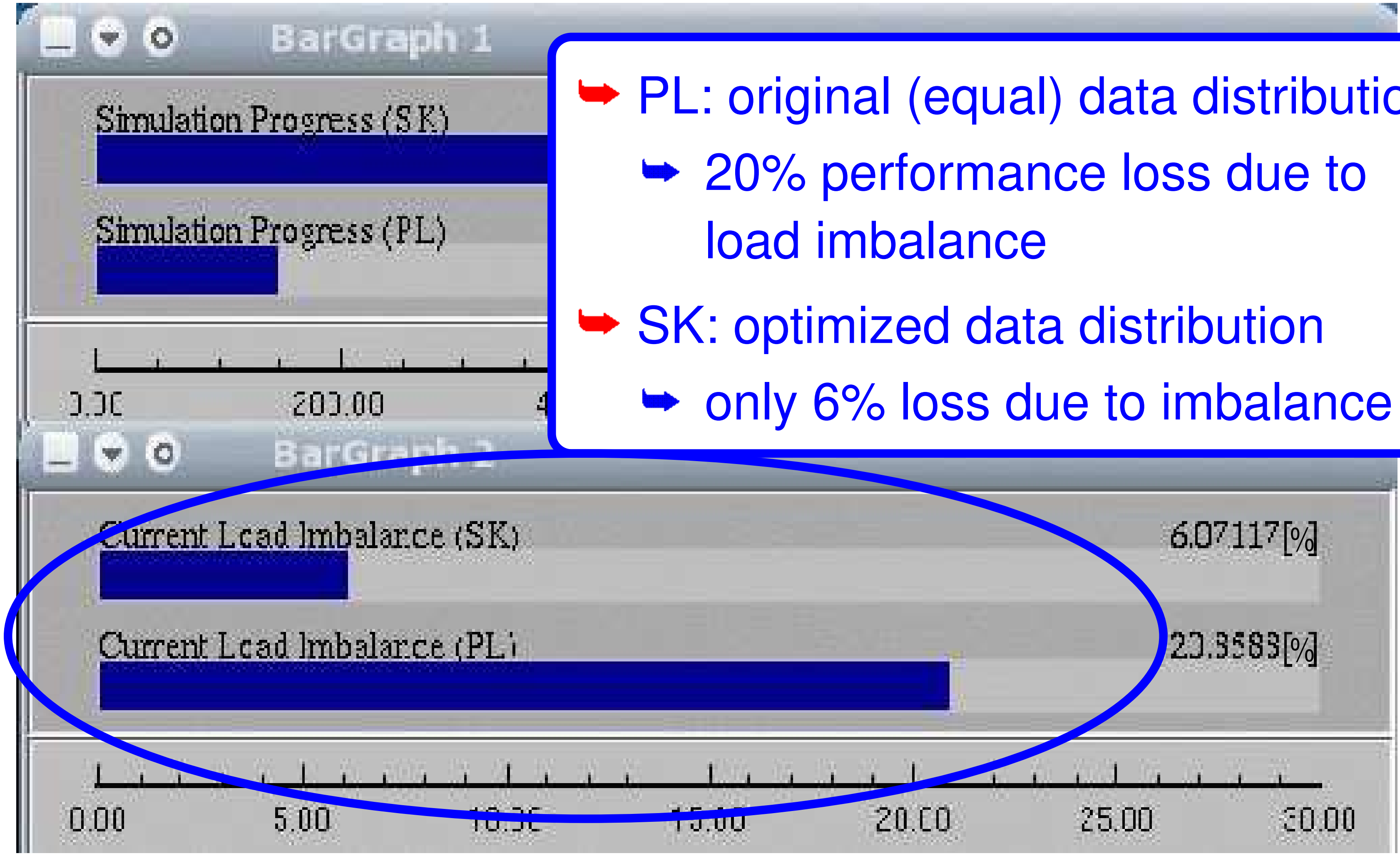
Progress of computation / load imbalance



- Execution progress on two sites:
  - SK: Bratislava, Slovakia (4 nodes)
  - PL: Krakow, Poland (4 nodes)
- SK cluster 2.7 times faster
  - mainly due to better hardware

3.1 Overview Measurements

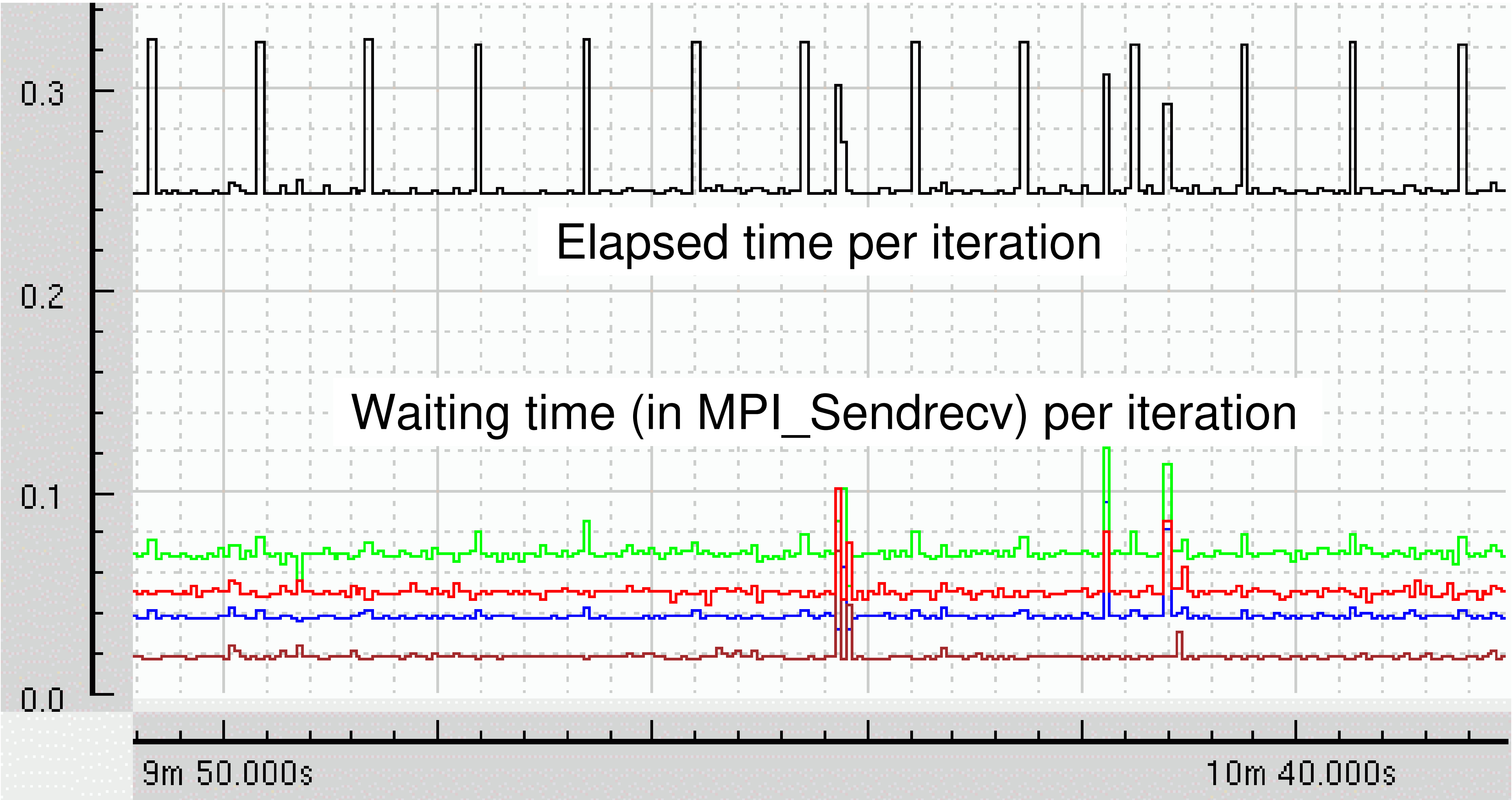
Progress of computation / load imbalance



- PL: original (equal) data distribution
  - 20% performance loss due to load imbalance
- SK: optimized data distribution
  - only 6% loss due to imbalance

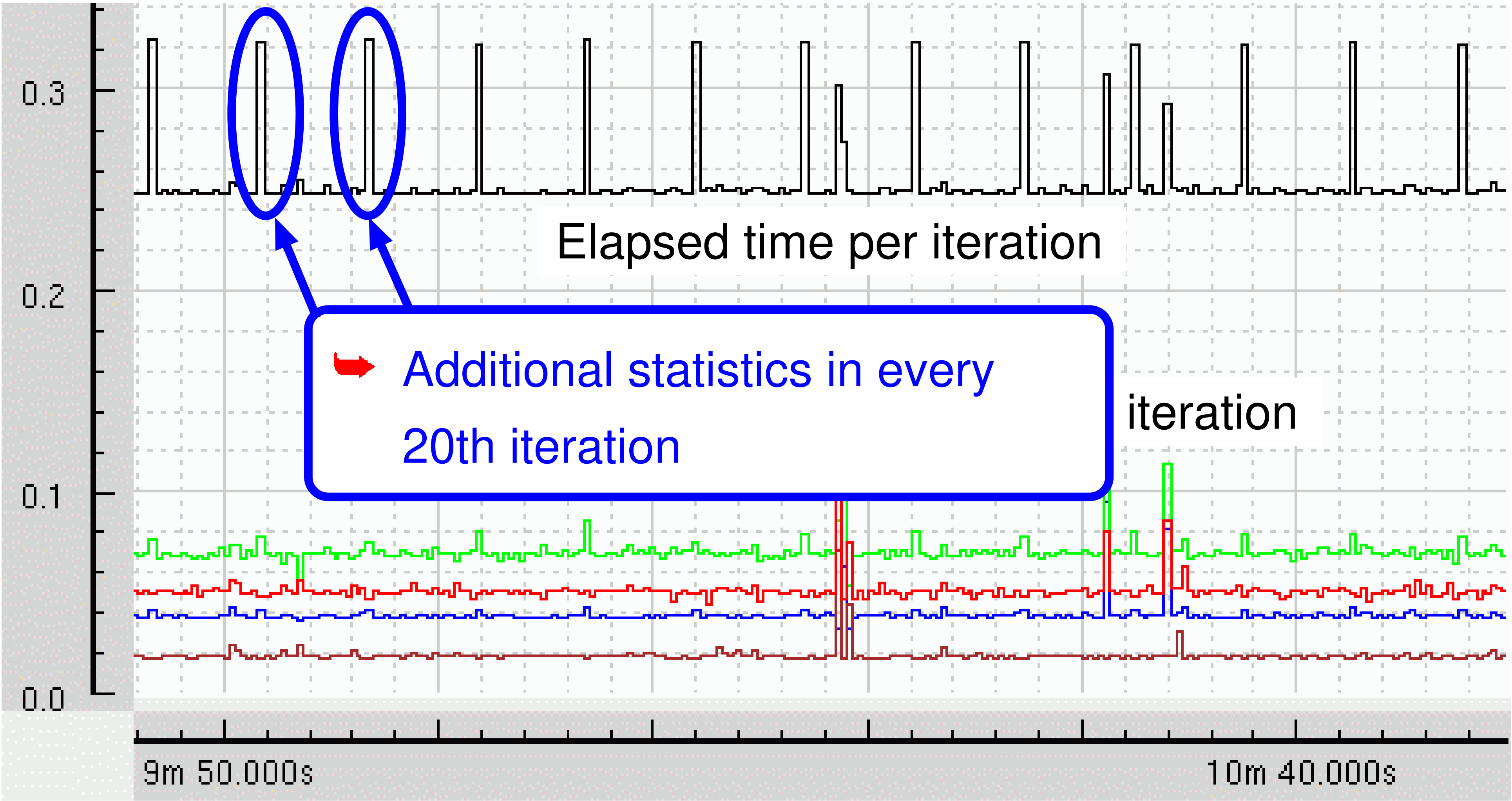
3.1 More Detailed Measurements

Measurements for single iterations



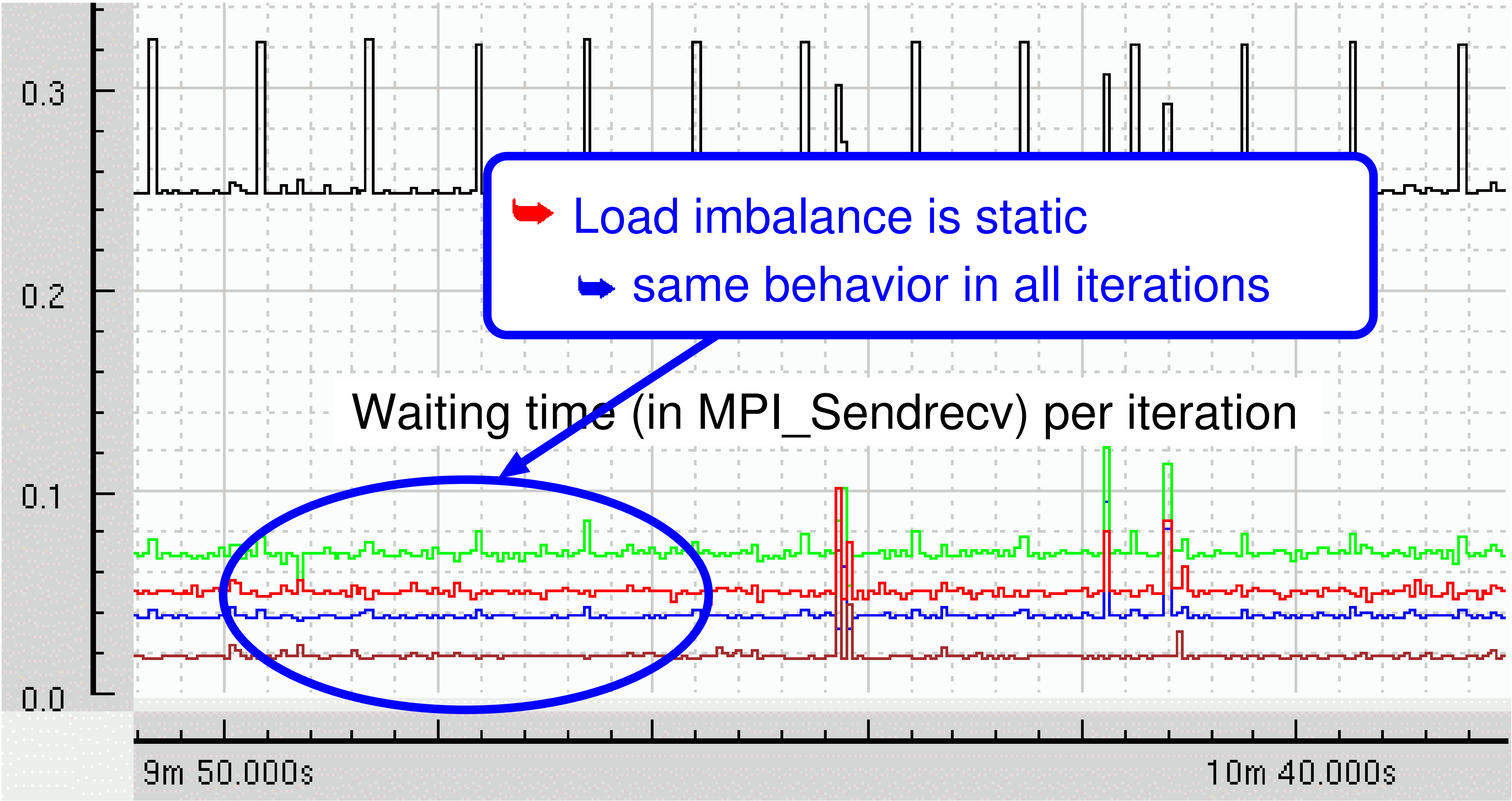
3.1 More Detailed Measurements

Measurements for single iterations



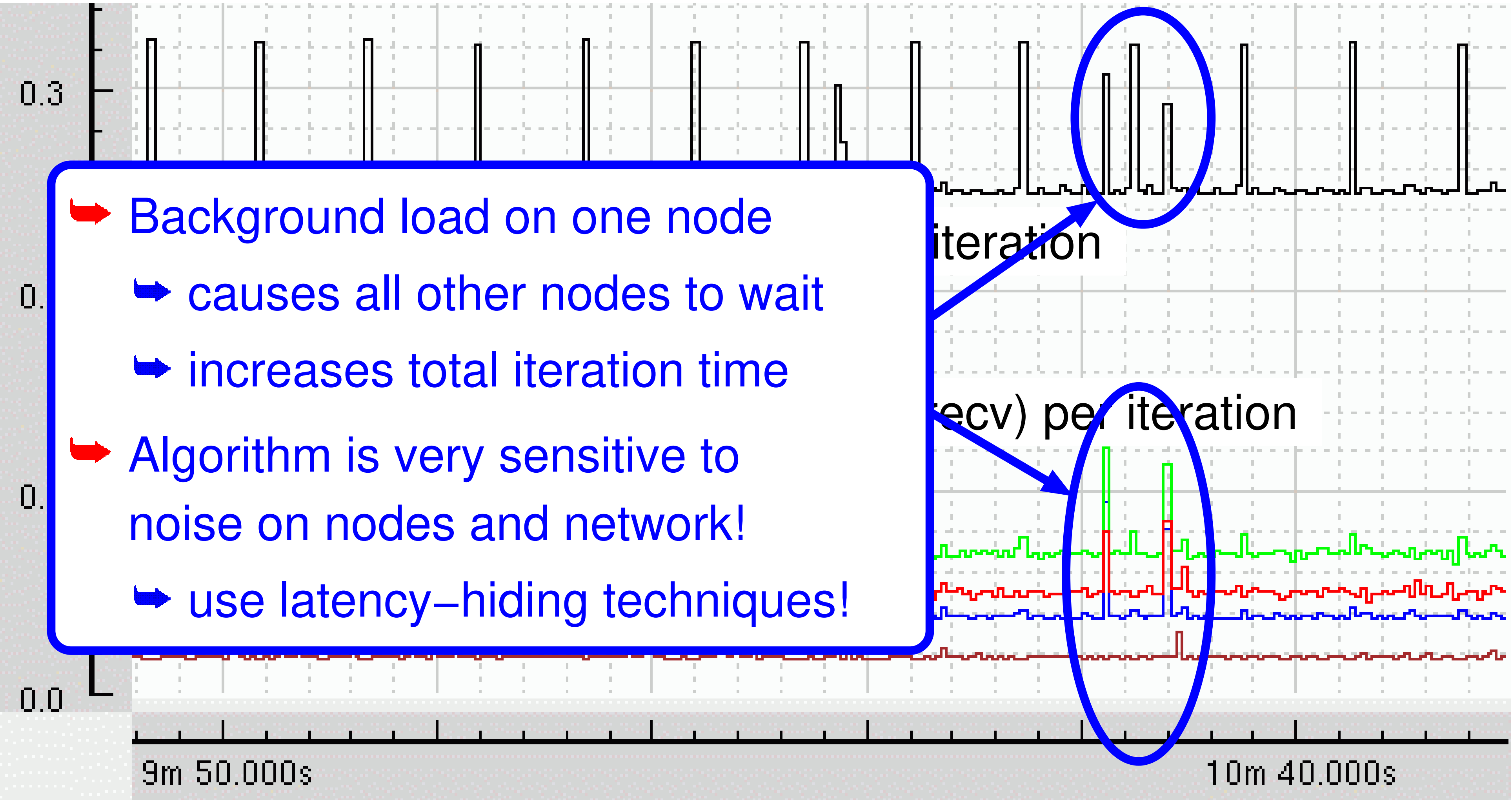
3.1 More Detailed Measurements

Measurements for single iterations



3.1 More Detailed Measurements

Measurements for single iterations



- Background load on one node
  - causes all other nodes to wait
  - increases total iteration time
- Algorithm is very sensitive to noise on nodes and network!
  - use latency–hiding techniques!

3.2 Experiment 2

Behavior of compute and output phases of main loop

- Time spent in these phases in each iteration
- Communication requirements of these phases in each iteration

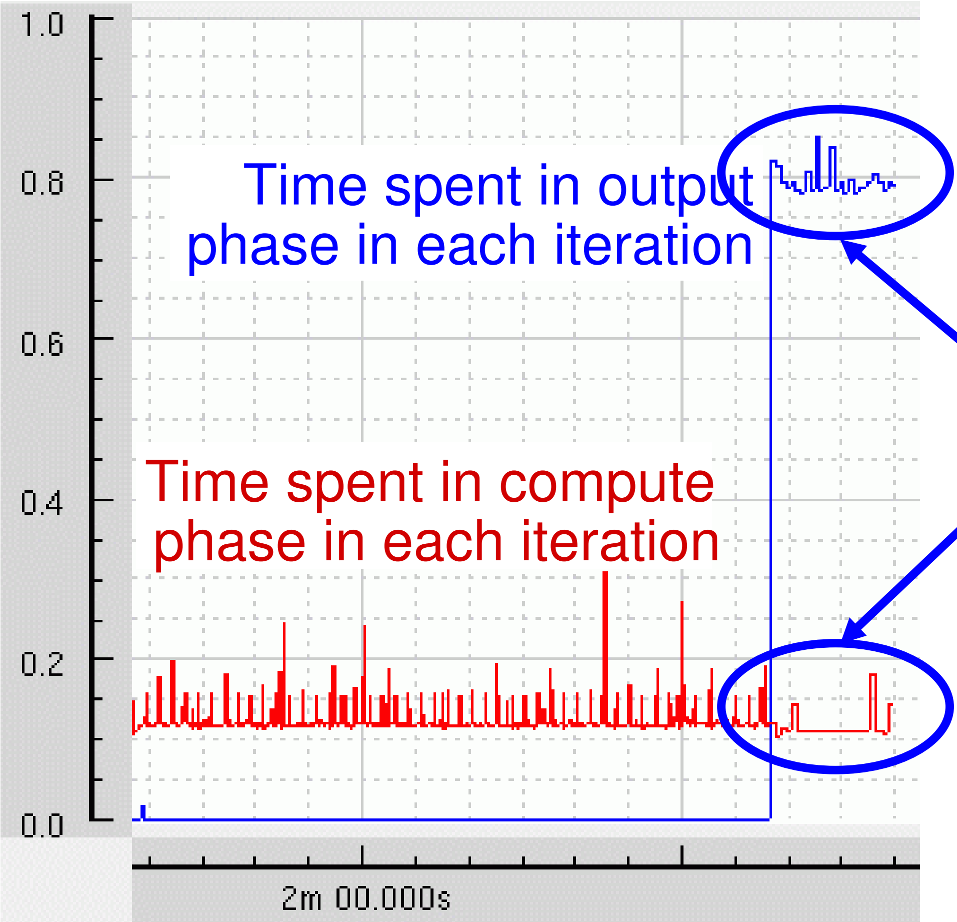
3.2 PMSL Metrics: Comm. Time in Output Phase

```
CommTime(Process p, TimeInterval ti)
     : Unit("s")
{
     return   Send_delay(p, ti) + Receive_delay(p, ti)
            + Barrier_delay(p, ti) + Reduce_delay(p, ti)
            + Gather_delay(p, ti) + Bcast_delay(p, ti)
            + Sendrecv_delay(p, ti);
}

CommTimeOutputPhase(Process p, VirtualTime vt)
     : Unit("s")
{
     PROBE outputBegin(Process, VirtualTime);
     PROBE iterEnd(Process, VirtualTime);
     return CommTime(p,[START,NOW]) AT iterEnd(p, vt)
          - CommTime(p,[START,NOW]) AT outputBegin(p, vt);
}
```

3.2 Measurement Resuls

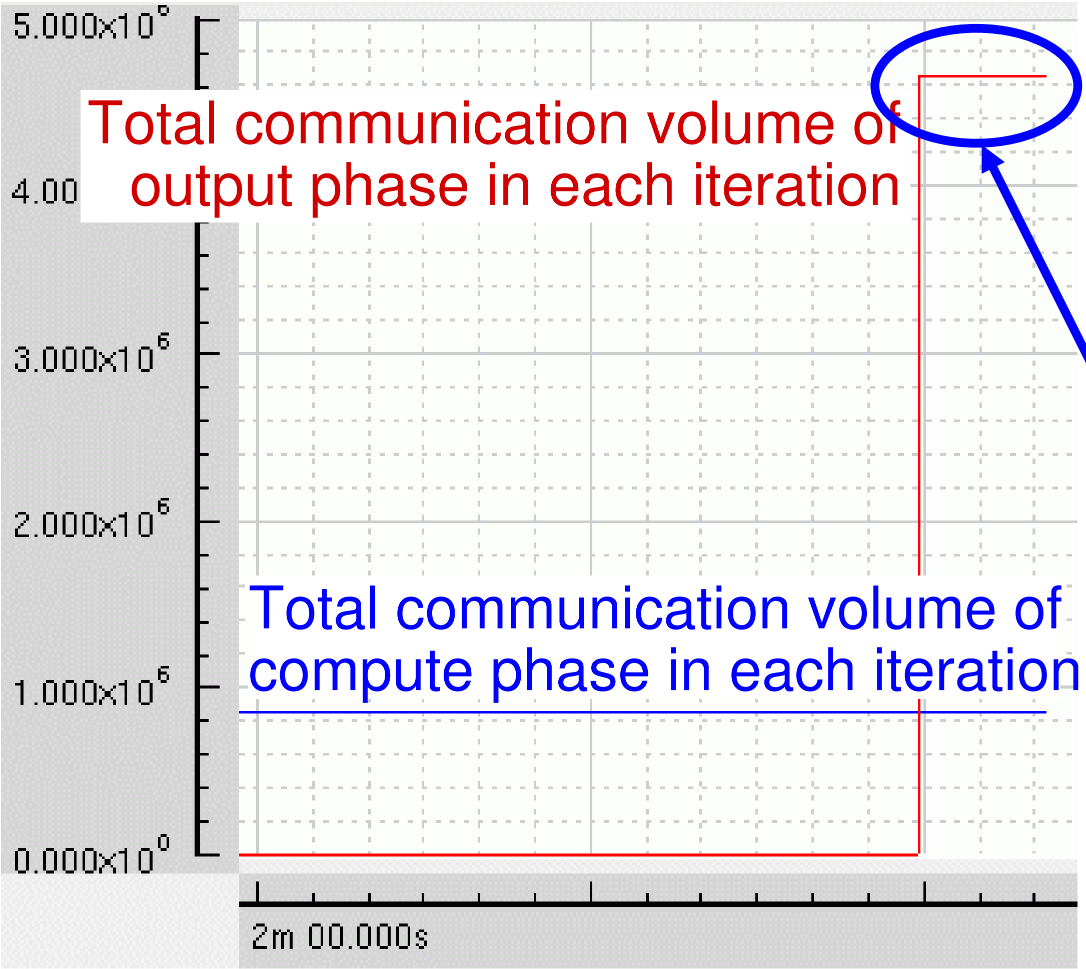
Time in compute and output phase



- Output phase is active only in the last 25 iterations
- Writing the output needs about 20% of the total run–time
- Output phase is time–consuming
  - 0.8s / iteration for output
  - 0.12s / iteration for computation
- Reason: ASCII–formatting
  - use binary output format!

3.2 Measurement Resuls

Communication in compute and output phase



- Ouptut phase has huge communication requirements
- Reason: all output is written by process 0
  - use parallel I/O (MPI–IO)!

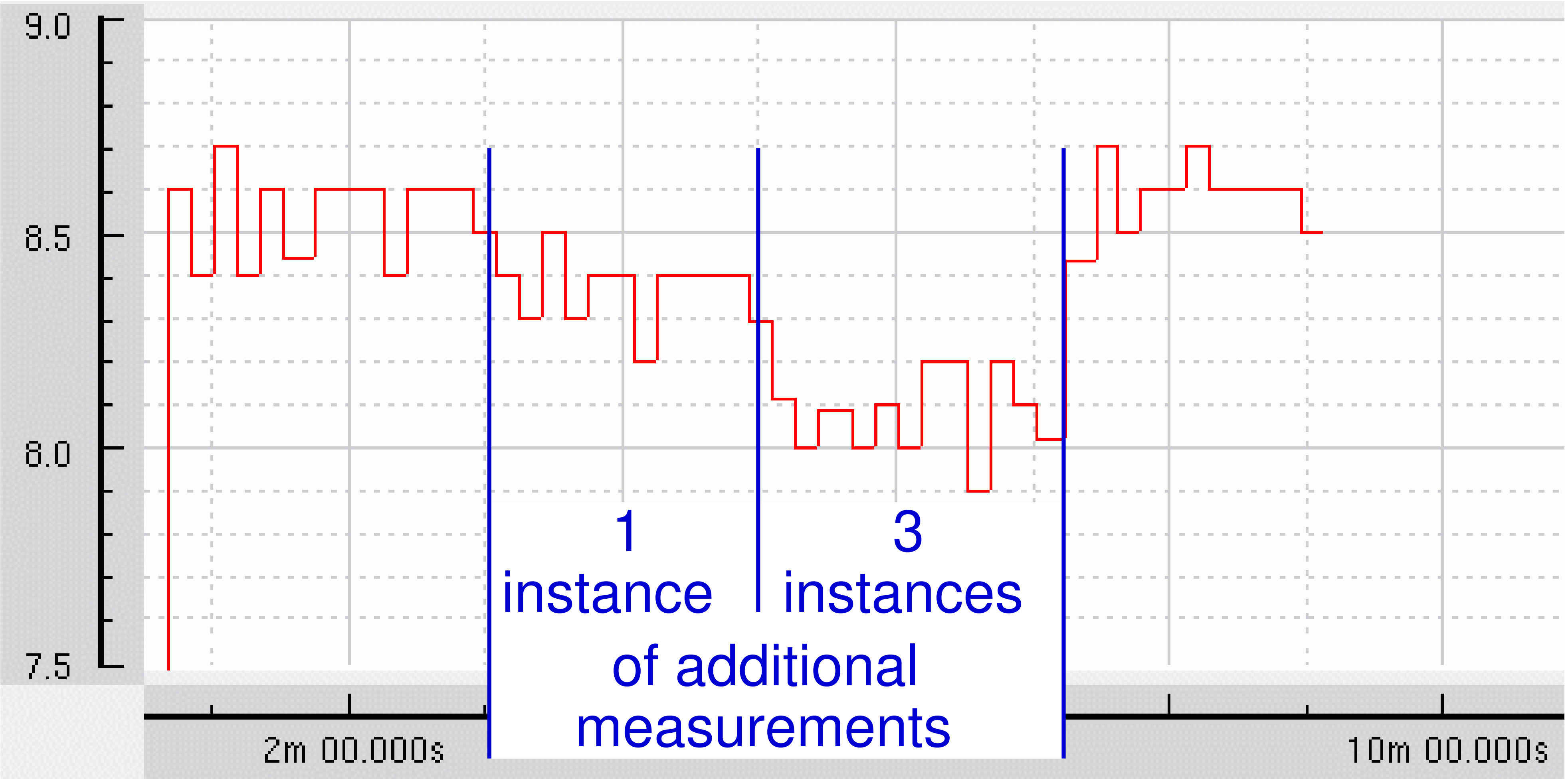
3.3 Experiment 3

Assessing the overhead of measurements

- Measure the computational speed with and without (additional) measurements
- Computational speed measured with G-PM: iterations / second
  - low overhead due to distributed evaluation
    - just a local counter, periodically read by G-PM
- Additional measurements:
  - communication time in each iteration (from experiment 1)
  - no distributed evaluation yet
    - G-PM gets raw data for each iteration

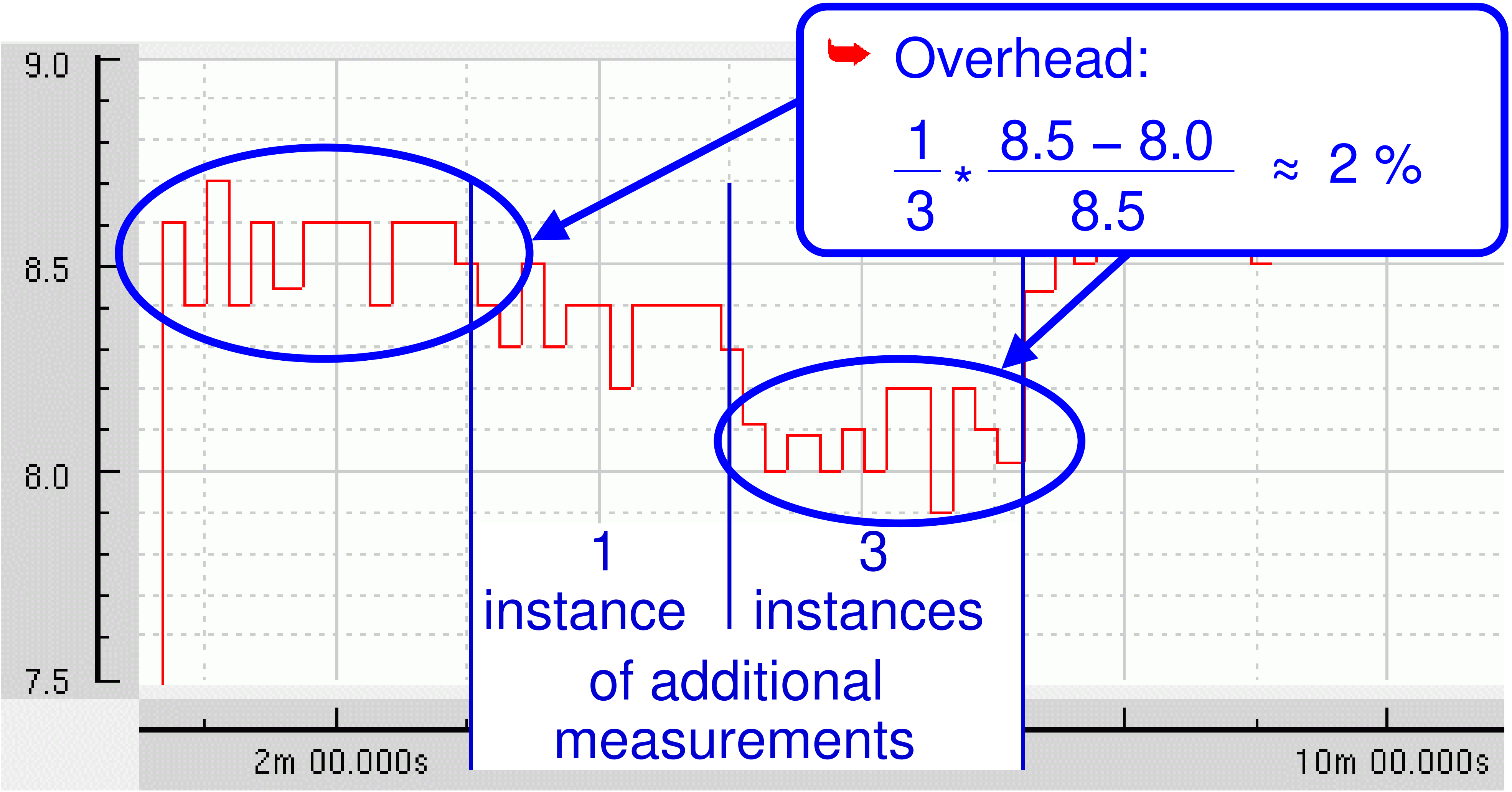
3.3 Measurement Resuls

Measurement overhead



3.3 Measurement Resuls

Measurement overhead



4. Conclusions

- G-PM's concepts of probes and user-defined metrics proved useful for application oriented performance analysis
- Performance of blood-flow solver could be improved by 14% (on 4 nodes)
  - optimized data distribution
  - binary output format
- Further improvements possible:
  - latency hiding
  - parallel I/O (MPI-IO)
- Overhead of G-PM is reasonable (few percent), but noticeable
  - distributed evaluation of metrics is needed

5. Status and Future Work

- G-PM tool and underlying OCM-G monitoring system fully usable for MPI applications on the Grid (MPICH, Globus 2.x)
  - documentation, source code and binary packages downloadable at `http://www.crossgrid.org`
  - GNU Public License

Ongoing work

- Distributed evaluation of user-defined metrics
- Back-port to MPI applications on clusters
- Enhancement of user interface
  - concurrent analysis of multiple applications
  - pre-configured measurements
  - enhanced display windows