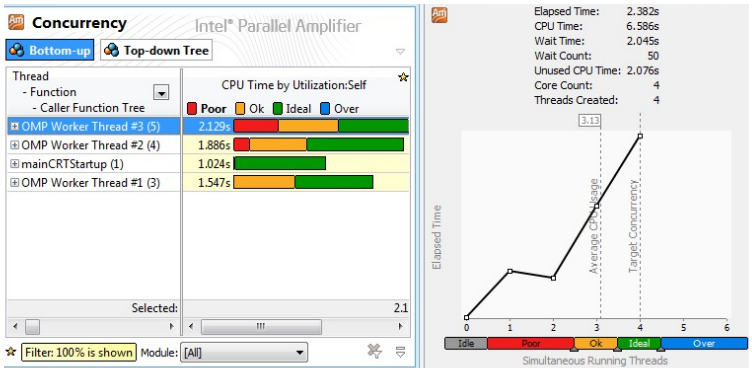
Load Balance Analysis

- Use Parallel Amplifier – Concurrency Analysis
- Select the “Thread –Function -Caller Function Tree”
- Observe that the 4 threads do unequal amounts of work



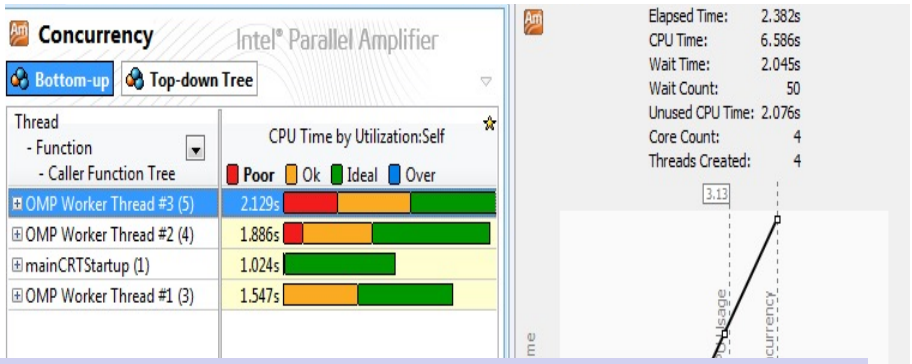
Fixing the Load Imbalance

- Distribute the work more evenly

```
void FindPrimes(int start, int end)
{
    // start is always odd
    int range = end - start + 1;

#pragma omp parallel for schedule(static,8)
    for( int i = start; i <= end; i += 2 )
    {
        if( TestForPrime(i) )
            globalPrimes[InterlockedIncrement(&gPrimesFound)] = i;
    }
}
```

Speedup achieved is 1.68X

Comparative Analysis



Threading applications require multiple iterations of going through the software development cycle

294

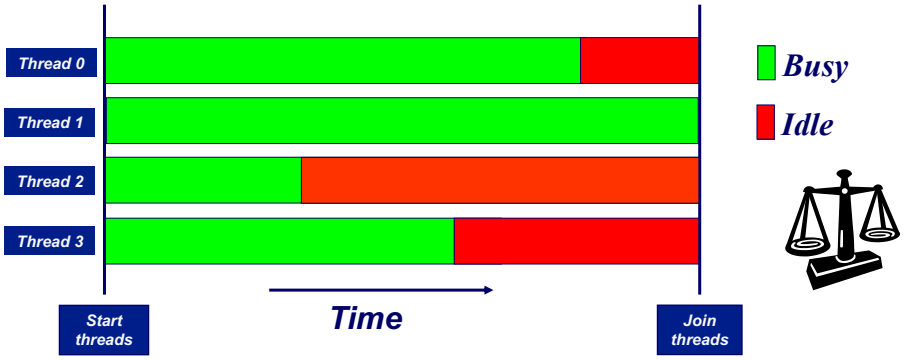
Common Performance Issues

- Load balance
  - Improper distribution of parallel work
- Synchronization
  - Excessive use of global data, contention for the same synchronization object
- Parallel Overhead
  - Due to thread creation, scheduling
- Granularity
  - Not sufficient parallel work

295

Load Imbalance

- Unequal work loads lead to idle threads and wasted time



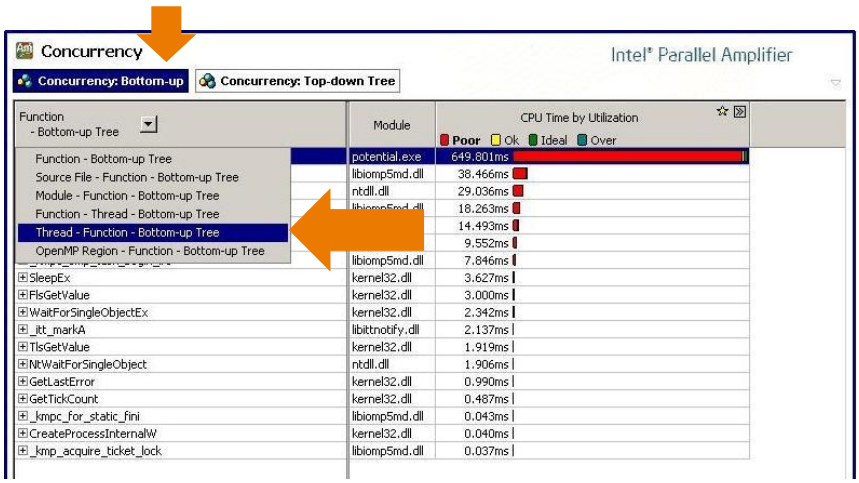
Redistribute Work to Threads

- Static assignment
  - Are the same number of tasks assigned to each thread?
  - Do tasks take different processing time?
    - Do tasks change in a predictable pattern?
      - Rearrange (static) order of assignment to threads
    - Use dynamic assignment of tasks

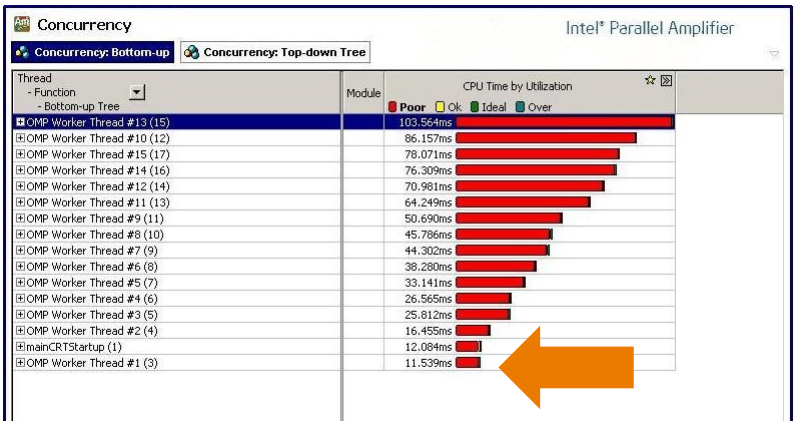
Redistribute Work to Threads

- Dynamic assignment
  - Is there one big task being assigned?
    - Break up large task to smaller parts
  - Are small computations agglomerated into larger task?
    - Adjust number of computations in a task
    - More small computations into single task?
    - Fewer small computations into single task?

Unbalanced Workloads



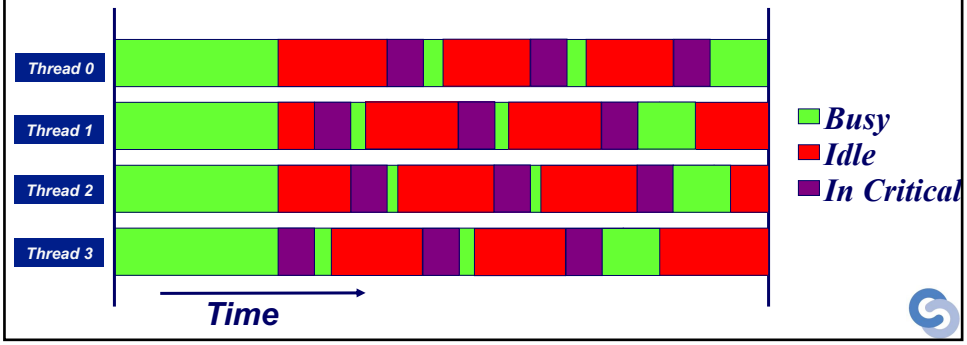
Unbalanced Workloads



300

Synchronization

- By definition, synchronization serializes execution
- Lock contention means more idle time for threads



301

Synchronization Fixes

- Eliminate synchronization
  - Expensive but necessary “evil”
  - Use storage local to threads
    - Use local variable for partial results, update global after local computations
    - Allocate space on thread stack (`alloca`)
    - Use thread-local storage API (TlsAlloc)
  - Use atomic updates whenever possible
    - Some global data updates can use atomic operations (Interlocked API family)

General Optimizations

- Serial Optimizations
  - Serial optimizations along the critical path should affect execution time
- Parallel Optimizations
  - Reduce synchronization object contention
  - Balance workload
  - Functional parallelism
- Analyze benefit of increasing number of processors - scalability
- Analyze the effect of increasing the number of threads on scaling performance

Measurement: Profiling vs. Tracing

- Profiling
  - Summary statistics of performance metrics
    - Number of times a routine was invoked
    - Exclusive, inclusive time
    - Hardware performance counters
    - Number of child routines invoked, etc.
    - Structure of invocations (call-trees/call-graphs)
    - Memory, message communication sizes
- Tracing
  - When and where events took place along a global timeline
    - Time-stamped log of events
    - Message communication events (sends/receives) are tracked
    - Shows when and from/to where messages were sent
    - Large volume of performance data generated usually leads to more perturbation in the program

Measurement: Profiling

- Profiling
  - Helps to expose performance bottlenecks and hotspots
  - 80/20 – rule or *Pareto principle:* often 80% of the execution time in 20% of your application
  - Optimize what matters, don't waste time optimizing things that have negligible overall influence on performance
- Implementation
  - **Sampling**: periodic OS interrupts or hardware counter traps
    - Build a histogram of sampled program counter (PC) values
    - Hotspots will show up as regions with many hits
  - **Measurement**: direct insertion of measurement code
    - Measure at start and end of regions of interests, compute difference

Measurement: Tracing

- Tracing
  - Recording of information about significant points (events) during program execution
    - Entering/exiting code region (function, loop, block, …)
    - Thread/process interactions (e.g., send/receive message)
  - Save information in event record
    - Timestamp
    - CPU identifier, thread identifier
    - Event type and event-specific information
  - Event trace is a time-sequenced stream of event records
  - Can be used to reconstruct dynamic program behavior
  - Typically requires code instrumentation

Performance Data Analysis

- Draw conclusions from measured performance data
- Manual analysis
  - Visualization
  - Interactive exploration
  - Statistical analysis
  - Modeling
- Automated analysis
  - Try to cope with huge amounts of performance by automation
  - Examples: Paradyn, KOJAK, Scalasca, Periscope

Hardware Performance Counters

- **Specialized hardware registers** to measure the performance of various aspects of a microprocessor
- Originally used for hardware verification purposes
- Can provide insight into:
  - Cache behavior
  - Branching behavior
  - Memory and resource contention and access patterns
  - Pipeline stalls
  - Floating point efficiency
  - Instructions per cycle
- Counters vs. events
  - Usually a large number of countable events - hundreds
  - On a small number of counters (4-18)
  - PAPI handles multiplexing if required

What is PAPI

- **Middleware** that provides a consistent and efficient programming interface for the performance counter hardware found in most major microprocessors.
- Countable events are defined in two ways:
  - Platform-neutral **Preset Events** (e.g., PAPI_TOT_INS)
  - Platform-dependent **Native Events** (e.g., L3_CACHE_MISS)
- Preset Events can be **derived** from multiple Native Events (e.g. PAPI_L1_TCM might be the sum of L1 Data Misses and L1 Instruction Misses on a given platform)
- Preset events are defined in a best-effort way
  - No guarantees of semantics portably
  - Figuring out what a counter actually counts and if it does so correctly can be hairy

310

PAPI Hardware Events

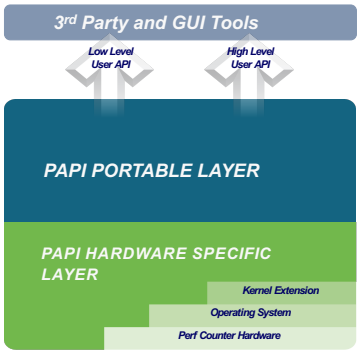
- Preset Events
  - Standard set of over 100 events for application performance tuning
  - No standardization of the exact definitions
  - Mapped to either single or linear combinations of native events on each platform
  - Use **papi_avail** to see what preset events are available on a given platform
- Native Events
  - Any event countable by the CPU
  - Same interface as for preset events
  - Use **papi_native_avail** utility to see all available native events
- Use **papi_event_chooser** utility to select a compatible set of events

310

311

PAPI Counter Interfaces

- PAPI provides 3 interfaces to the underlying counter hardware:
  - A **low level API** manages hardware events in user defined groups called EventSets. Meant for experienced application programmers wanting fine-grained measurements.
  - A **high level API** provides the ability to start, stop and read the counters for a specified list of events.
  - **Graphical** and end-user tools provide facile data collection and visualization.



311

PAPI Example Low Level API Usage

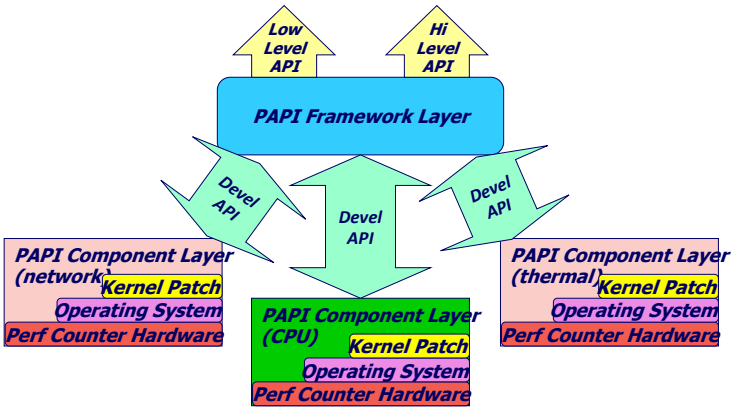
```
#include "papi.h"
#define NUM_EVENTS 2
int Events[NUM_EVENTS]={PAPI_FP_OPS,PAPI_TOT_CYC},
int EventSet;
long long values[NUM_EVENTS];
/* Initialize the Library */
retval = PAPI_library_init (PAPI_VER_CURRENT);
/* Allocate space for the new eventset and do setup */
retval = PAPI_create_eventset (&EventSet);
/* Add Flops and total cycles to the eventset */
retval = PAPI_add_events (&EventSet,Events,NUM_EVENTS);
/* Start the counters */
retval = PAPI_start (EventSet);

do_work();  /* What we want to monitor*/

/*Stop counters and store results in values */
retval = PAPI_stop (EventSet,values);
```

Using PAPI through tools

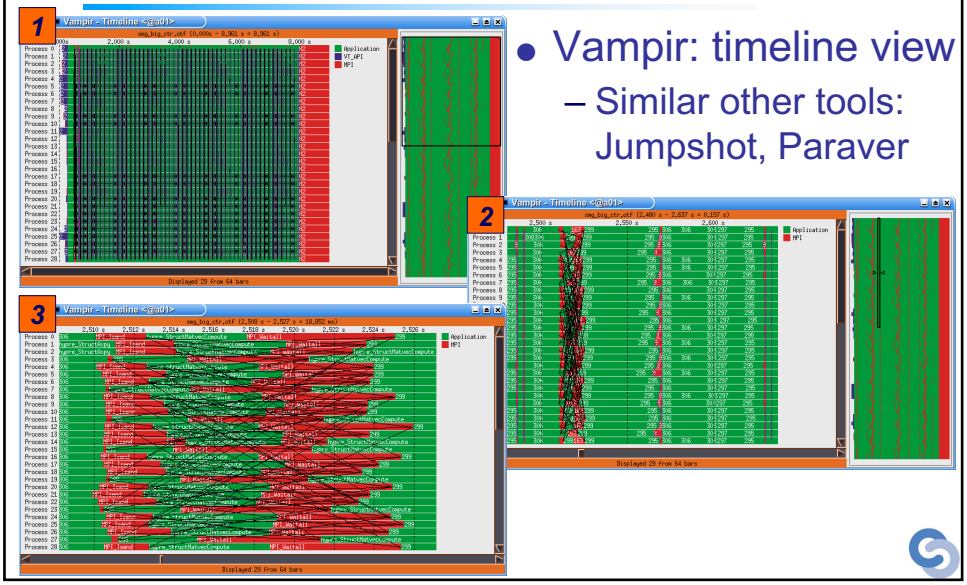
- You can use PAPI directly in your application, but most people use it through tools
- Tool might have a predfined set of counters or lets you select counters through a configuration file/environment variable, etc.
- Tools using PAPI
  - TAU (UO)
  - PerfSuite (NCSA)
  - HPCToolkit (Rice)
  - KOJAK, Scalasca (FZ Juelich, UTK)
  - Open|Speedshop (SGI)
  - ompP (UCB)
  - IPM (LBNL)

Component PAPI Design

Re-Implementation of PAPI w/ support for multiple monitoring domains

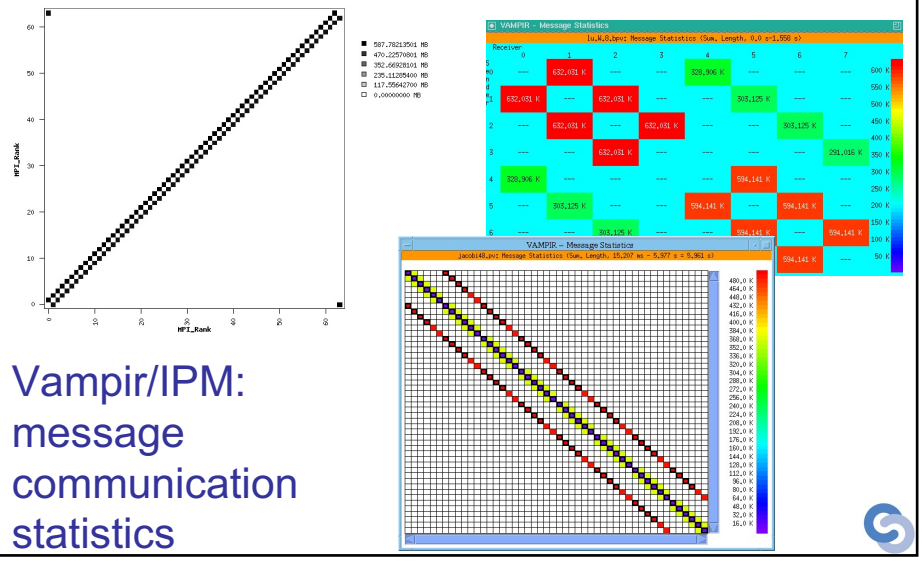


Trace File Visualization



- Vampir: timeline view
  - Similar other tools: Jumpshot, Paraver

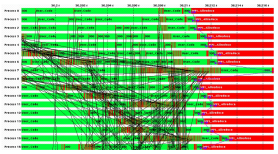
Trace File Visualization



- Vampir/IPM: message communication statistics

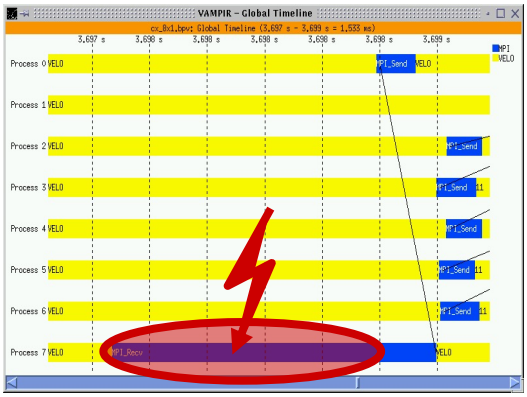
Automated Performance Analysis

- Reason for Automation
  - Size of systems: several tens of thousand of processors
  - LLNL Sequoia: 1.6 million cores
  - Trend to multi-core



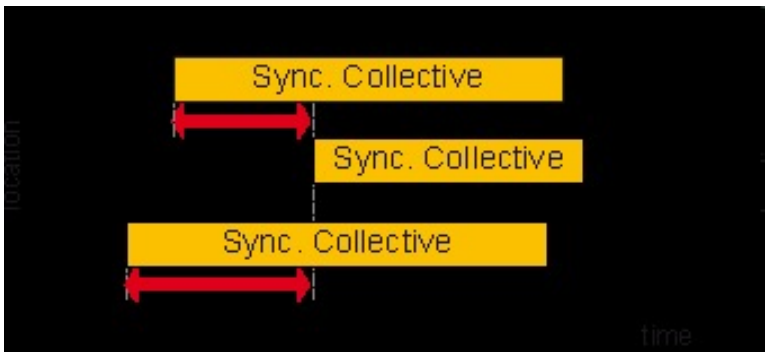
- Large amounts of performance data when tracing
  - Several gigabytes or even terabytes
- Not all programmers are performance experts
  - Scientists want to focus on their domain
  - Need to keep up with new machines
- Automation can solve some of these issues

Automation Example



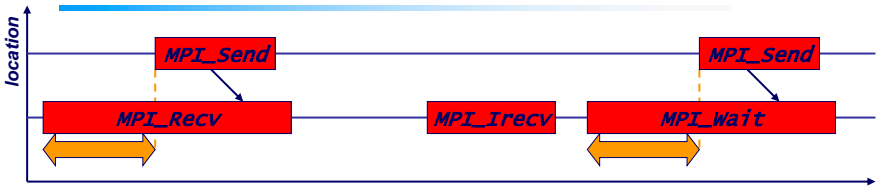
- „Late sender“ pattern
- This pattern can be detected automatically by analyzing the trace

MPI-1 Pattern: Wait at Barrier

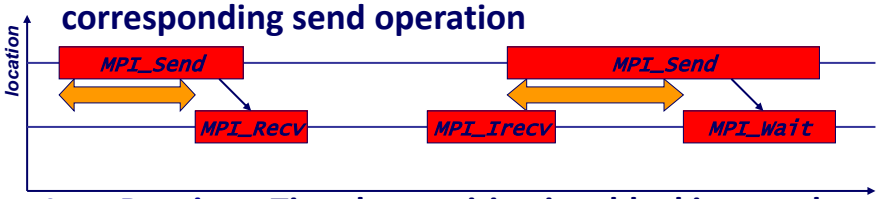


- **Time spent in front of MPI synchronizing operation such as barriers**

MPI-1 Pattern: Late Sender / Receiver

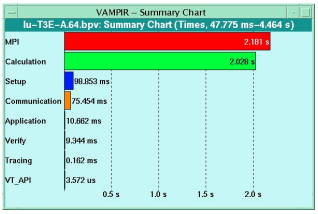


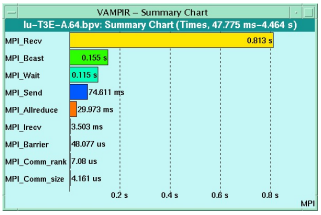
- **Late Sender: Time lost waiting caused by a blocking receive operation posted earlier than the corresponding send operation**



- **Late Receiver: Time lost waiting in a blocking send operation until the corresponding receive operation is called**

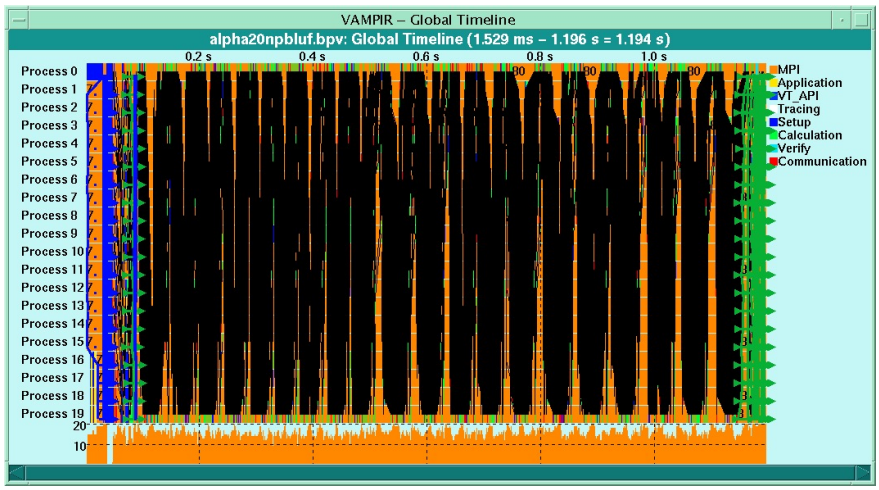
Vampir overview statistics





- Aggregated profiling information
  - Execution time
  - Number of calls
- This profiling information is computed from the trace
  - Change the selection in main timeline window
- Inclusive or exclusive of called routines

Timeline display

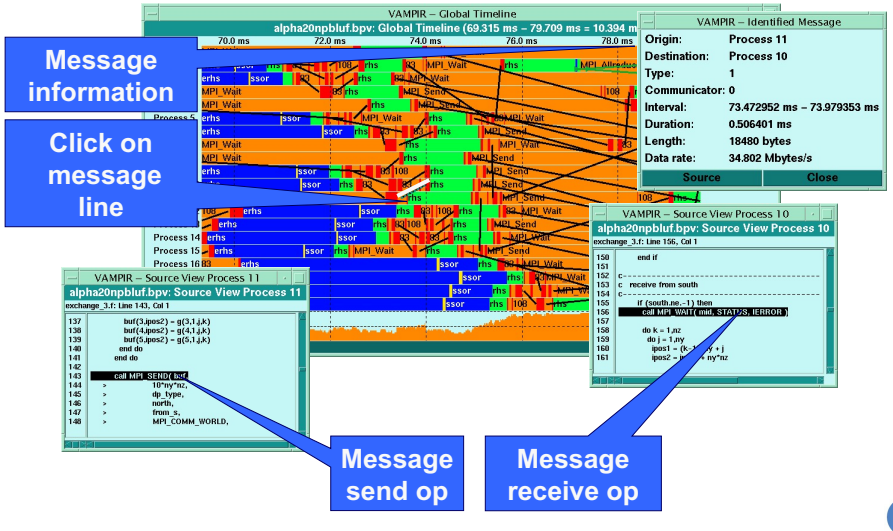


- To zoom, mark region with the mouse

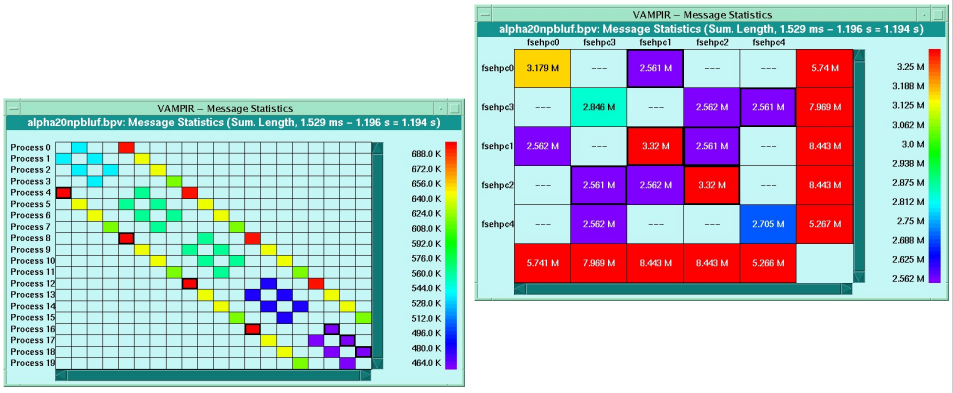
Timeline display – contents

- Shows all selected processes
- Shows state changes (activity color)
- Shows messages, collective and MPI–IO operations
- Can show parallelism display at the bottom

Timeline display – message details

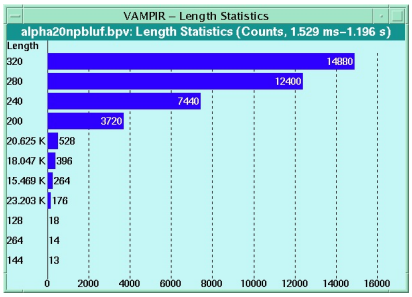


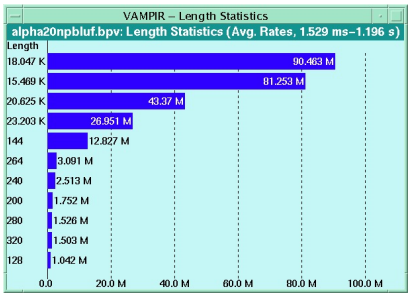
Communication statistics



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

Message histograms

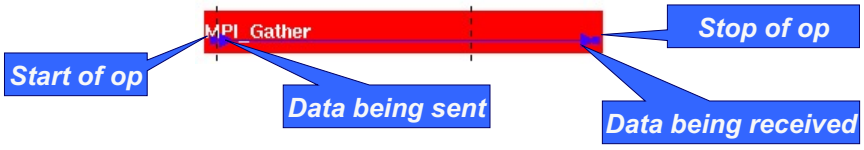


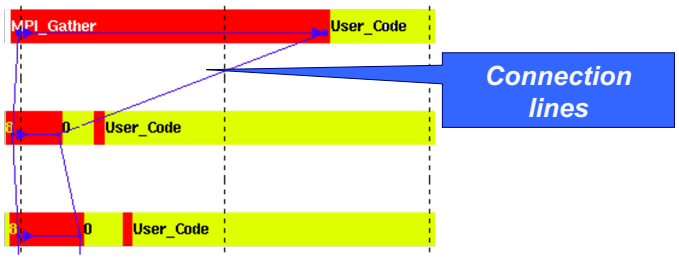


- Message statistics by length, tag or communicator
  - Byte and message count
  - Min/max/avg bandwidth

Collective operations

- For each process: mark operation locally

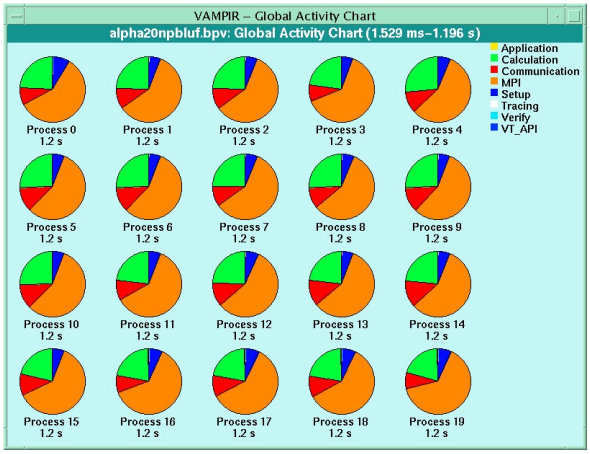




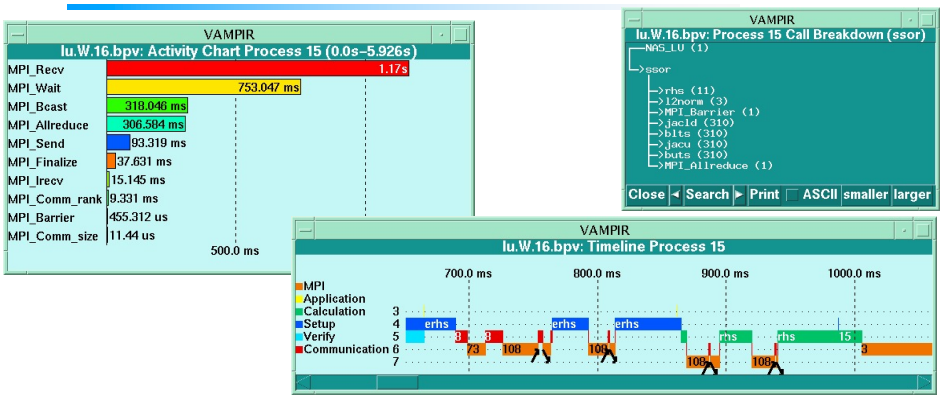
- Connect start/stop points by lines

Activity chart

- Profiling information for all processes



Process–local displays



- Timeline (showing calling levels)
- Activity chart
- Calling tree (showing number of calls)

Effects of zooming