



Innovative Computing Laboratory
UNIVERSITY OF TENNESSEE
COMPUTER SCIENCE DEPARTMENT

Performance Optimization for HPC Architectures

Shirley V. Moore

Philip J. Mucci

Innovative Computing Laboratory

University of Tennessee

shirley@cs.utk.edu

mucci@cs.utk.edu

Sameer Shende

University of Oregon

sameer@cs.uoregon.edu

NRL-Monterey

Dec 3-4, 2003

Course Outline

HPC Architectures

Performance Optimization Issues

Compiler Optimizations

Tuned Numerical Libraries

Hand Tuning

Communication Performance

OpenMP Performance

Performance Analysis Tools

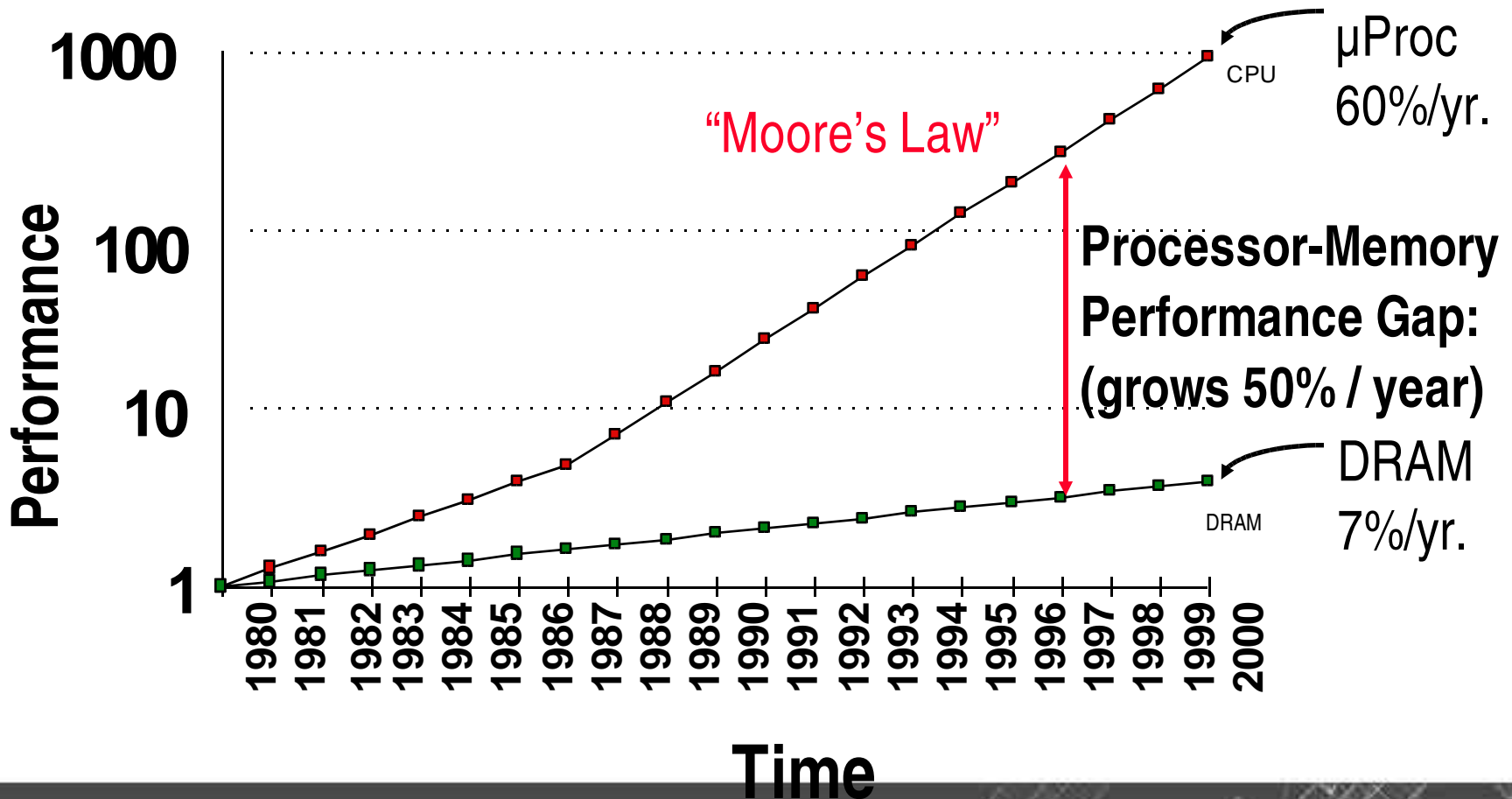
Performance Results

HPC Architectures

Architecture Evolution

- › Moore's Law: Microprocessor CPU performance doubles every 18 months.
- › Cost and size of storage have fallen along a similar exponential curve.
- › But decrease in time to access storage, called *latency*, has not kept up, thus leading to
 - › deeper and more complex memory hierarchies
 - › “load-store” architecture

Processor-DRAM Gap (latency)



Processor Families

- › Have high-level design features in common
- › Four broad families over the past 30 years
 - › CISC
 - › Vector
 - › RISC
 - › VLIW

CISC

- › Complex Instruction Set Computer
- › Designed in the 1970s
- › Goal: define a set of assembly instructions so that high-level language constructs could be translated into as few assembly language instructions as possible => many instructions access memory, many instruction types
- › CISC instructions are typically broken down into lower level instructions called *microcode*.
- › Difficult to pipeline instructions on CISC processors
- › Examples: VAX 11/780, Intel Pentium Pro

Vector Processors

- › Seymour Cray introduced the Cray 1 in 1976.
- › Dominated HPC in the 1980s
- › Perform operations on vectors of data
- › Vector pipelining (called *chaining*)
- › Examples: Cray T90, Convex C-4, Cray SV1, Cray SX-6, Cray X1, POWER5?

RISC

- › Reduced Instruction Set Computer
- › Designed in the 1980s
- › Goals
 - › Decrease the number of clocks per instruction (CPI)
 - › Pipeline instructions as much as possible
- › Features
 - › No microcode
 - › Relatively few instructions all the same length
 - › Only load and store instructions access memory
 - › Execution of branch delay slots
 - › More registers than CISC processors

RISC (cont.)

- › Additional features
 - › Branch prediction
 - › Superscalar processors
 - Static scheduling
 - Dynamic scheduling
 - › Out-of-order execution
 - › Speculative execution
- › Examples: MIPS R10K/12K/14K, Alpha21264, Sun UltraSparc-3, IBM Power3/Power4

VLIW

- › Very Long Instruction Word
- › Explicitly designed for instruction level parallelism (ILP)
- › Software determines which instructions can be performed in parallel, bundles this information and the instructions, and passes the bundle to the hardware.
- › Example: Intel-HP Itanium

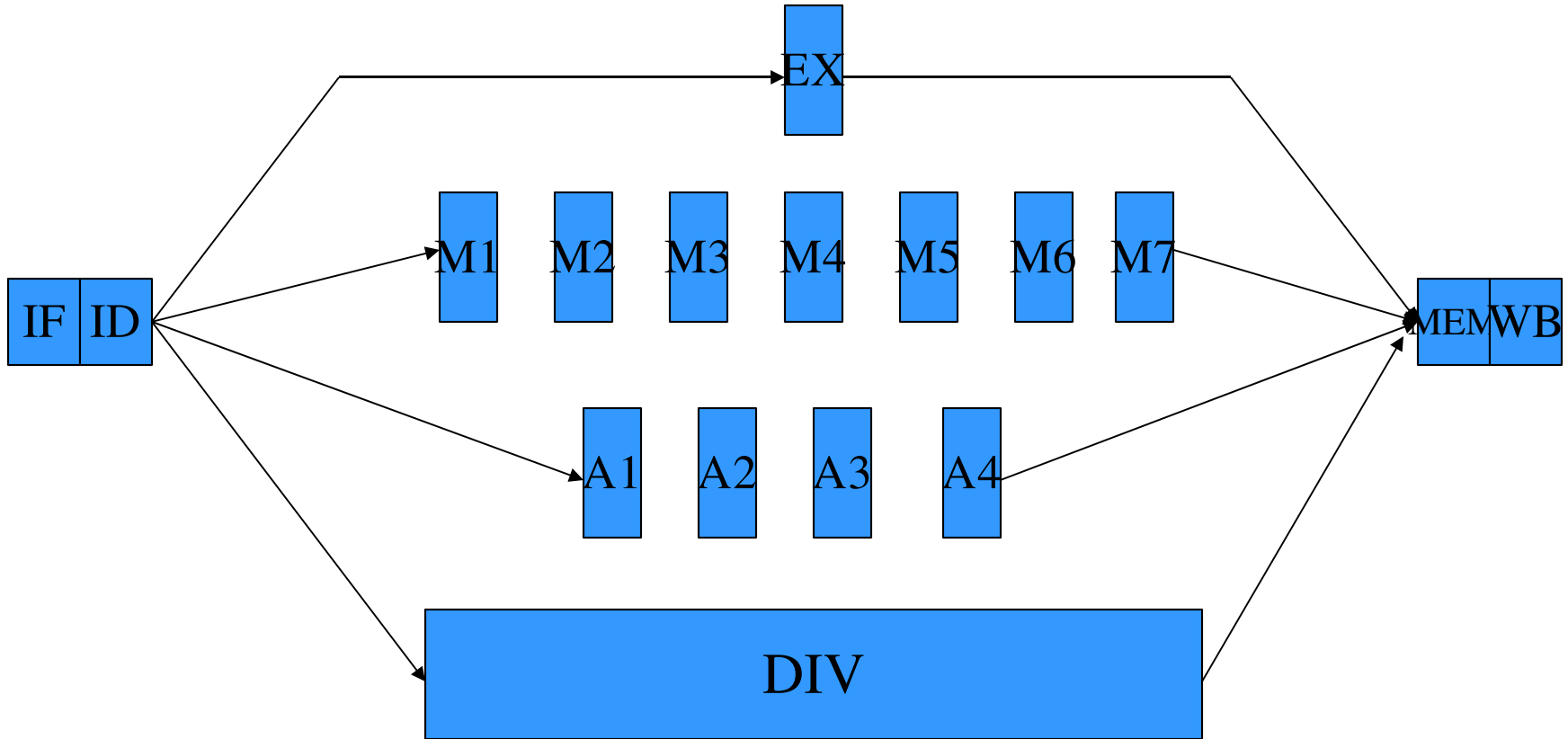
Architecture Changes in the 1990s

- › 64-bit addresses
- › Optimization of conditional branches via conditional execution (e.g., conditional move)
- › Optimization of cache performance via prefetch
- › Support for multimedia and DSP instructions
- › Faster integer and floating-point operations
- › Reducing branch costs with dynamic hardware prediction

Pipelining

- › Overlapping the execution of multiple instructions
- › Assembly line metaphor
- › Simple pipeline stages
 - › Instruction fetch cycle (IF)
 - › Instruction decode/register fetch cycle (ID)
 - › Execution/effective address cycle (EX)
 - › Memory access/branch completion cycle (MEM)
 - › Write-back cycle (WB)

Pipeline with Multicycle Operations



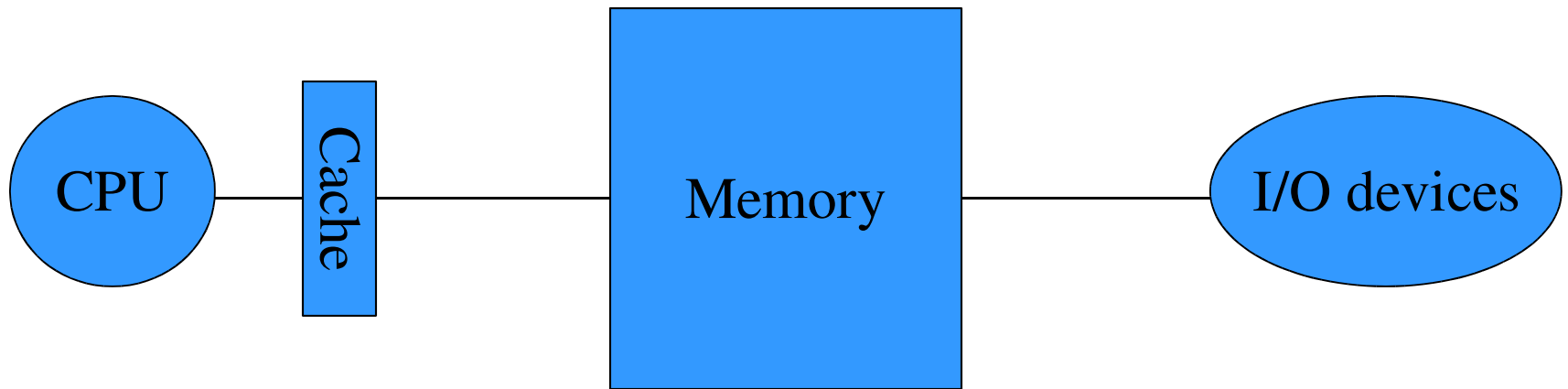
Pipeline Hazards

- › Situations that prevent the next instruction in the pipeline from executing during its designated clock cycle and thus cause *pipeline stalls*
- › Types of hazards
 - › *Structural hazard* – resource conflict when the hardware cannot support all instructions simultaneously
 - › *Data hazard* – when an instruction depends on the results of a previous instruction
 - › *Control hazard* – caused by branches and other instructions that change the PC

Memory Hierarchy Design

- › Exploits principle of *locality* – programs tend to reuse data and instructions they have used recently
 - › *Temporal locality* – recently accessed items like to be accessed in the near future
 - › *Spatial locality* – items whose addresses are near each other likely to accessed close together in time
- › Take advantage of cost-performance of memory technologies
- › Fast memory is more expensive.
- › Goal: Provide a memory system with cost almost as low as the cheapest level of memory and speed almost as fast as the fastest level.

Typical Memory Hierarchy



Register
reference

Cache
reference

Memory
reference

Disk
reference

Size: 500 bytes

64 KB

512 MB

100 GB

Speed: 0.25ns

1 ns

100 ns

5 ms

Memory Technologies

- › Main memory usually built from dynamic random access memory (DRAM) chips.
 - › DRAM must be “refreshed”
- › Caches usually built from faster but more expensive static random access memory (SRAM) chips.
- › *cycle time* – minimum time between requests to memory
- › Cycle time of SRAMs is 8 to 6 times faster than DRAMs, but they are also 8 to 16 times more expensive.

Memory Technologies (cont.)

- › Two times that are important in measuring memory performance:
 - › *Access time* is the time from when a read or write is requested until it arrives at its destination.
 - › *Cycle time* is the minimum time between requests to memory.
- › Since SRAM does not need to be refreshed, it has no difference between access time and cycle time.
- › Simple DRAM results in each memory transaction requiring the sum of access time plus cycle time.

Memory Interleaving

- › Multiple banks of memory organized so that sequential words are located in different banks
- › Multiple banks can be accessed simultaneously.
- › Reduces effective cycle time
- › *Bank stall or bank contention* – when the memory access pattern is such that the same banks are repeatedly accessed

Cache Characteristics

- › Number of caches
- › Cache sizes
- › Cache line size
- › Associativity
- › Replacement policy
- › Write strategy

Cache Characteristics (cont.)

- › A *cache line* is the smallest unit of memory that can be transferred to and from main memory.
 - › Usually between 32 and 128 bytes
- › In an *n-way associative cache*, any cache line from memory can map to any of the *n* locations in a set.
 - › 1-way set associative cache is called *direct mapped*
 - › A *fully associative cache* is one in which a cache line can be placed anywhere in cache.

Cache Hits and Misses

- › When the CPU finds a requested data item in the cache, a *cache hit* occurs.
- › If the CPU does not find the data item it needs in the cache, a *cache miss* occurs.
- › Upon a cache miss, a cache line is retrieved from main memory (or a higher level of cache) and placed in the cache.
- › The *cache miss rate* is the fraction of cache accesses that result in a miss.
- › The time required for a cache miss, called the *cache miss penalty*, depends on both the latency and bandwidth of the memory.
- › The cycles during which the CPU is stalled waiting for memory access are called *memory stall cycles*.

Types of Cache Misses

Types of cache misses can be classified as follows:

- 2) *compulsory* – the very first access to a cache line
- 3) *capacity* – when the cache cannot contain all the cache lines needed during execution of a program
- 4) *conflict* – In a (less than fully) set associative or direct mapped cache, a conflict miss occurs when a block must be discarded and later retrieved because too many cache lines mapped to the same set.

Multiple Levels of Cache

- › Using multiple levels of cache allows a small fast cache to keep pace with the CPU, while slower larger caches can be used to reduce the miss penalty since the next level cache can be used to capture many accesses that would go to main memory.
- › The local miss rate is large for higher level caches because the first level cache benefits the most from data locality. Thus a global miss rate that indicates what fraction of the memory access that leave the CPU go all the way to memory is a more useful measure. Let us define these terms as follows:
 - › *local miss rate* – the number of misses in a cache divided by the total number of memory access to this cache
 - › *global miss rate* – the number of misses in a cache divided by the total number of memory accesses generated by the CPU (Note: for the first level cache, this is the same as the local miss rate)

Nonblocking Caches

- › Pipelined computers that allow out-of-order execution can continue fetching instructions from the instruction cache while waiting on a data cache miss. A *non-blocking cache* design allows the data cache to continue to supply cache hits during a miss, called “hit under miss”, or “hit under multiple miss” if multiple misses can be overlapped. Hit under miss significantly increases the complexity of the cache controller, with the complexity increasing as the number of outstanding misses allowed increases. Out-of-order processors with hit under miss are generally capable of hiding the miss penalty of an L1 data cache miss that hits in the L2 cache, but are not capable of hiding a significant portion of the L2 miss penalty.

Cache Replacement Policy

- › Possible policies
 - › Least Recently Used (LRU)
 - › Random
 - › Round robin
- › LRU performs better than random or round robin but is more difficult to implement.

Cache Write Strategy

- › When a store instruction writes data into a cache-resident line, one of following policies is usually used:
 - › *Write through*: The data are written to both the cache line in the cache and to main memory.
 - › *Write back*: The data are written only to the cache line in the cache. The modified cache line is written to memory only when necessary (e.g., when it is replaced by another cache line).

Cache Contention on SMPs

- › When two or more CPUs alternately and repeatedly update the same cache line
 - › *memory contention*
 - when two or more CPUs update the same variable
 - correcting it involves an algorithm change
 - › *false sharing*
 - when CPUs update distinct variables that occupy the same cache line
 - correcting it involves modification of data structure layout

Virtual Memory

- › A *page* is the smallest contiguous block of memory that the operating systems allocates to your program.
- › The operating system creates a virtual address space for each process, keeping unneeded pages on disk and loading them into main memory as needed.

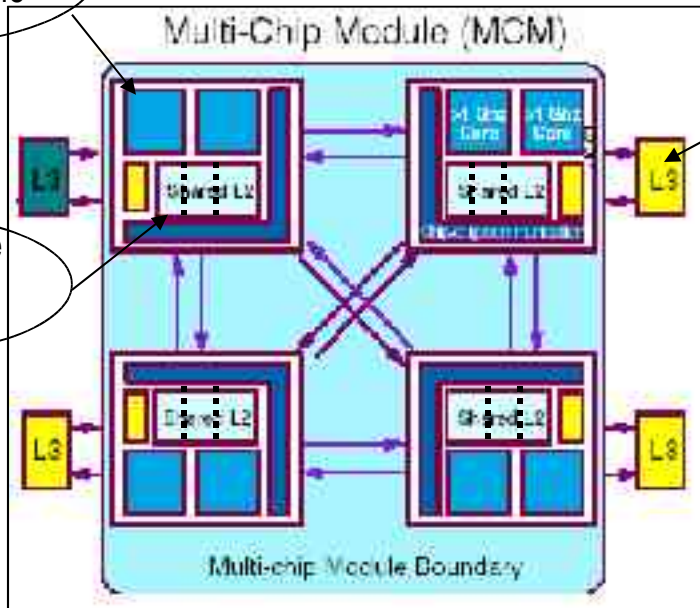
The TLB

- › For every memory access, the operating system translates the virtual address to a physical address.
- › To keep latency low, the most recently used address translations are cached in the Translation Lookaside Buffer (TLB).
- › When the program refers to a virtual address that is not cached in the TLB, a *TLB miss* occurs.
- › If a referenced page is not in physical memory, a *page fault* occurs.

Node Design

1.3 GHz processor
32 KB Level 1 cache

Level 2 cache
1440 KB total
480 KB ea.



Level 3 cache
512 MB total
128 MB ea.

Four MCMs comprise one node of a regatta system.

POWER4 L1 Cache

L1 cache

32 KB of data and 64 KB of instruction cache for each processor

Fetches 128 byte lines

Uses FIFO replacement policy rather than touch. Thus, blocking for cache reuse is not advisable.

Uses eight pre-fetching streams. Use the

`-qhot -qcache=auto -qarch=pwr4`

`-qtune=pwr4` compiler options to perform loop optimizations that improve cache use.

POWER4 L2 Cache

L2 cache

Total of 1440 KB shared between the two processors on a chip

Use “least recently touched” replacement policy

For applications with memory requirements >1GB/process, placement of processes on processors may impact cache performance.

If you are blocking for cache size, you should block for L2 cache. To leave it to the compiler use `-qarch=pwr4`, `-qtune=pwr4`, `-qcache=auto`, and `-qhot`. Implied with `-O4`

POWER4 L3 Cache

L3 cache

32MB 8-way set associative caches combined in pairs or quadruplets to create a 64MB or 128MB address-interleaved cache

Shared between the eight processors on an MCM

Access times vary depending on location of the data to be retrieved

POWER4 Cache: Performance

Cache	Line Size (Bytes)	Bandwidth (Bytes/cycle)	Latency (Cycles)
Level 1 (32 KB)	128	16	4
Level 2 (1440 KB)	128	32	13
Level 3 (32 MB)	512	5.33	125

POWER4 Hardware Data Prefetch

- › Hardware for prefetching data cache lines from memory, L3 cache, and L2 cache transparently in the L1 data cache
- › Prefetch *stream* is a sequence of loads from storage that references at least two or more contiguous data cache lines in either ascending or descending order.
- › Eight streams per processor are supported.
- › Triggered by data cache line misses and paced by loads to the stream
- › Streams must be reinitialized at page boundaries.

POWER4 vs. POWER3

- › 1.3 GHz POWER 4 clock rate compared to 375 MHz for POWER3
- › But performance of floating-point intensive applications is typically only two to three times faster.
- › More difficult to approach peak performance on POWER4 than on POWER3 because of
 - › Increased FPU pipeline depth
 - › Reduced L1 cache size
 - › Higher latency (in terms of cycles) of the higher level caches

Performance Optimization Issues

What to Time?

- › User time – amount of time spent performing the work of a program
- › System time – amount of time spent in the operating system supporting the execution of the program
- › CPU time – sum of user time and system time (also called *virtual time*)
- › Wall clock time – elapsed time from when execution starts until it stops

How to Time

- › Use the most accurate lowest overhead timers available on the system.
 - › PAPI timers attempt to do this
- › Time on a dedicated system or light loaded system or using a queueing system that gives you dedicated resources.
- › CPU time is important for non-dedicated systems.

Parallel Performance - Speedup and Scalability

- › *Speedup* is the ratio of the running time on a single processor to the parallel running time on N processors.
Speedup = $T(1)/T(N)$
- › An application is *scalable* if the speedup on N processors is close to N.
- › With *scaled speedup*, an application is said to be scalable if, when the number of processors and the problem size are increased by a factor of N, the running time remains the same.

Factors that Limit Scalability

- › Amdahl's Law
- › Communication Overhead
- › Load Imbalance

Amdahl's Law

- › Suppose an application has a portion S that must be run serially.
- › Assume T_s is the time required by the serial region, and T_p is the time required by the region that can be parallelized.
- › Then

$$\text{Speedup}(N) = (T_s + T_p) / (T_s + T_p/n) \leq T(1) / T_s$$

Communication Overhead

Suppose communication cost is logarithmic in N .
Then

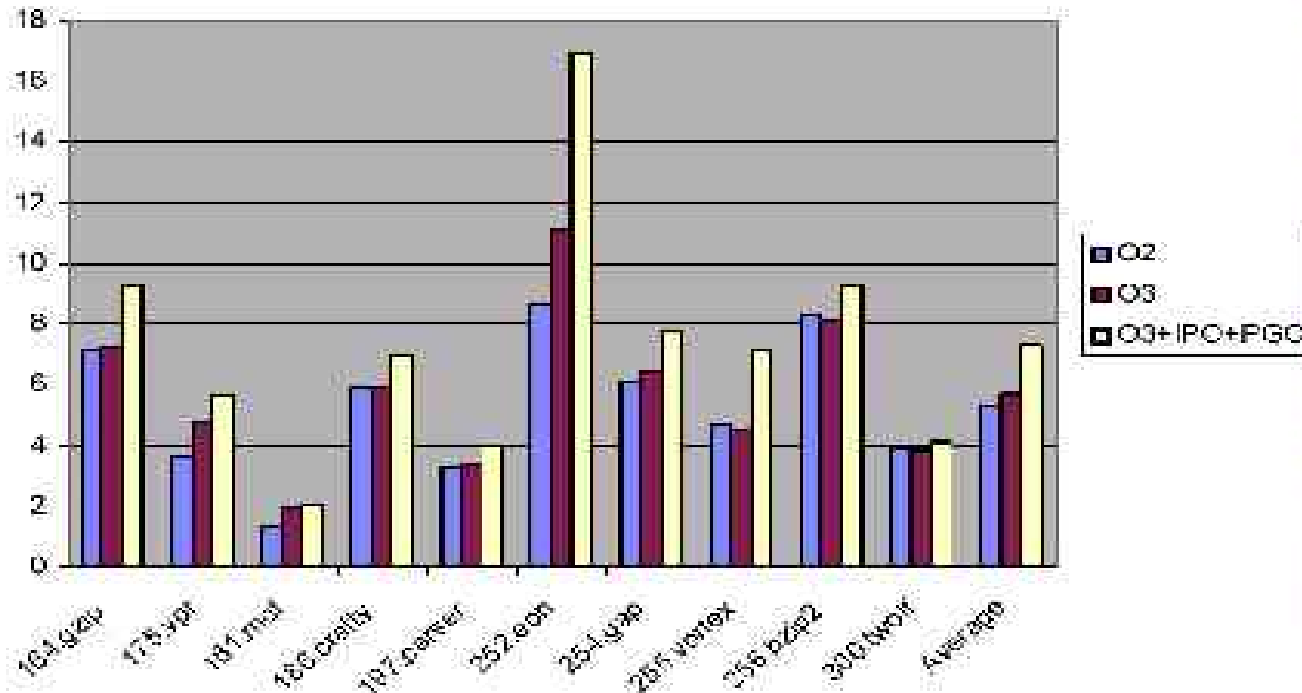
$$\begin{aligned}\text{Speedup}(N) &= T(1)/(T_s + T_p/N + c \lg N) \\ &= O(1/\lg N)\end{aligned}$$

Load Imbalance

- › Load imbalance is the time that some processors in the system are idle due to
 - › insufficient parallelism
 - › unequal size tasks
- › **Examples of the latter**
 - › Adapting to “interesting parts of a domain”
 - › Tree-structured computations
 - › Fundamentally unstructured problems
- › Algorithm needs to balance load

Importance of Optimization

Example: Speed up from Static Compiler Optimization on Itanium-1 in 2002 (SpecInt)



Steps in Optimizing Code

- › Optimize compiler switches
- › Integrate high-performance libraries
- › Profile code to determine where most time is being spent
- › Optimize blocks of code that dominate execution time by using performance data to determine why the bottlenecks exist
- › Always examine correctness at every stage!

Compiler Optimizations

Role of the Compiler

- › Transform higher-level abstract representation of a program into code for a particular instruction set architecture (ISA)
- › Goals
 - › Correct compiled program
 - › Efficient compiled program
 - › Fast compilation
 - › Debugging and performance analysis support

Compiler Structure

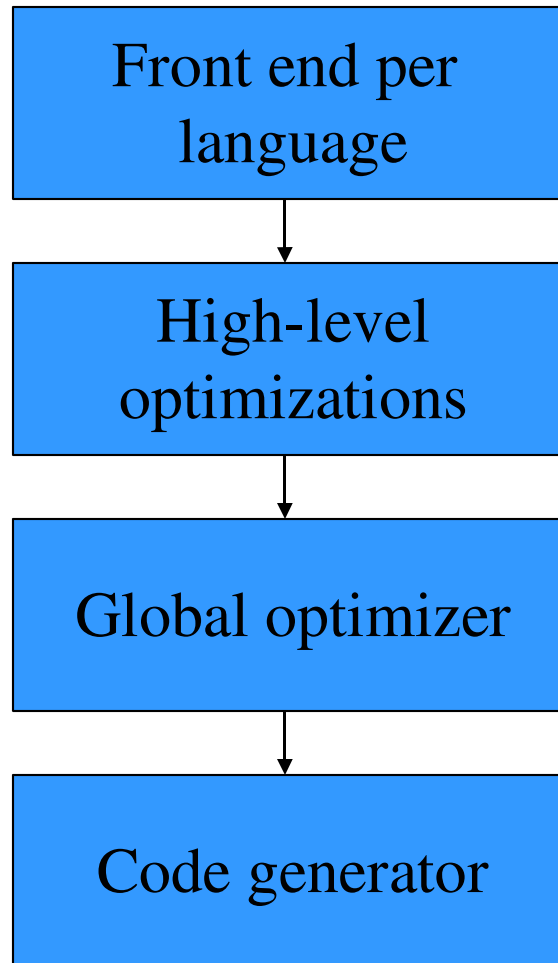
Dependencies

Language dependent,
machine independent

Somewhat language
dependent, largely
machine independent

Some machine
dependencies (e.g.,
register number and type)

Language independent,
highly machine dependent



Function

Transform language to
common intermediate
form

e.g., loop
transformations and
procedure inlining

Including global and
local optimizations
plus register
allocation

Detailed instruction
selection and machine
dependent
optimizations

Compiler Operation

- › Compilers typically consist of two to four passes, or phases, with higher optimization having more passes.
- › Transformations made in previous passes are not revisited.
- › Phase-ordering problem: compiler must order some transformation before others – e.g.,
 - › Choose which procedure calls to expand inline before the exact size of the procedure is known
 - › Eliminate common subexpressions before register allocation is known

Classes of Compiler Optimizations

- › High-level optimizations
- › Local optimizations
- › Global optimizations
- › Register allocation
- › Processor-dependent optimizations
- › Loop optimizations
- › Interprocedural optimization

High-level Optimizations

- › At or near the source level
- › Processor-independent
- › Example
 - › Procedure inlining: replace procedure call by procedure body

Local Optimizations

- › Optimize code only within a basic block
- › Examples
 - › *Common subexpression elimination* – replace multiple instances of the same computation with one computation with results stored in temporary
 - › *Constant propagation* – Replace all instances of a variable that is assigned a constant with the constant
 - › *Stack height reduction* – rearrange expression tree to minimize resources needed for expression evaluation

Global Optimizations

- › Extension of local optimizations across branches
- › Transformations aimed at optimizing loops
- › Examples
 - › *Global common subexpression elimination* – same as local but across branches
 - › *Copy propagation* – replace all instances of a variable A that is assigned X (i.e., $A=X$) with X
 - › *Code motion* – move code that computes the same value each iteration outside the loop
 - › *Induction variable elimination* – simplify/eliminate array addressing calculations within loops

Register Allocation

- › Associates registers with operands
- › Central role in speeding up code and making other optimization useful
- › NP-complete problem
- › Heuristics work best with at least 16 general-purpose registers and additional registers for floating-point.
- › More effective for stack-allocated objects than for global variables
- › Variables that are *aliased* (i.e., that have more than one way to refer to them) cannot be placed in registers.

Processor-dependent Optimizations

- › Take advantage of specific architectural features
- › Examples:
 - › *Strength reduction* – e.g., replace multiply by a constant with adds and shifts
 - › *Pipeline scheduling* – reorder instructions to improve pipeline performance

Single Loop Transformations

- › Induction variable optimization
- › Prefetching
- › Test promotion in loops
- › Loop peeling
- › Loop fusion
- › Loop fission
- › Copying
- › Block and copy
- › Loop unrolling
- › Software pipelining
- › Loop invariant code motion
- › Array padding
- › Optimizing reductions

Loop Unrolling

- › Generates multiple copies of the code for the loop body
- › Reduces number of branches and groups more instructions together to enable more efficient instruction pipelining
- › Best candidates are innermost loops with limited control flow.
- › Often combined with data prefetching

Software Pipelining

- › Applies instruction scheduling, allowing instructions within a loop to “wrap around” and execute in a different iteration of the loop
- › Reduces the impact of long-latency operations, resulting in faster loop execution
- › Enables prefetching of data to reduce the impact of cache misses
- › Often used with together with loop unrolling

Nested Loop Optimizations

- › Performed at higher levels of optimization and/or with special options
- › Tricky and time-consuming for the compiler to get right
- › Can significantly speed up performance but may slow down performance.
 - › Time code carefully to make sure these optimizations improve performance.

Nested Loop Optimizations (cont.)

- › Loop interchange
- › Outer loop unrolling
- › Unroll and jam
- › Blocking
- › Block and copy

Interprocedural Optimization (IPO)

- › Looks at all routines and tries to make optimizations across routine boundaries, including but not limited to inlining and cloning
- › *Inlining* – replacing a subprogram call with the replicated code of the subprogram
- › *Cloning* – optimizes logic in the copied routine for the particular call

Optimization Levels

-O0, -O1, -O2, -O3, -O4, etc.

- › No standardization across compilers
- › Increasing levels apply increasingly sophisticated optimizations but also increase compilation time.
- › Higher levels of optimization may cause the program to produce different results.

IBM XL Fortran Optimization Levels

The `-g` option tells the compiler to include information in the executable to enable effective debugging. It doesn't inhibit optimization at all, so we recommend that you include it during the program development phase.

The `-O` flag is the main compiler optimization flag and can be specified with several levels of optimization. `-O` and `-O2` are currently equivalent.

IBM XL Fortran Optimization Levels (cont.)

- At -O2, the XL Fortran compiler's optimization is highly reliable and usually improves performance, often quite dramatically. -O2 avoids optimization techniques that could alter program semantics.
- O3 provides an increased level of optimization but can result in the reordering of associative floating-point operations or operations that may cause runtime exceptions and thus can slightly alter numerical results. This can be prevented through the use of the -qstrict option together with -O3. -O3 is often used together with -qhot, -qarch, and -qtune.

IBM XL Fortran Optimization Levels (cont.)

-O4 provides more aggressive optimization and implies the following options:

- › -qhot
- › -qipa
- › -O3
- › -qarch=auto
- › -qtune=auto
- › -qcache=auto

-O5 implies the same optimizations as -O4 with the addition of -qipa=level=2.

IBM XL Fortran - Recommended Options

For production codes, we recommend

```
xlf_r -O3 -qstrict -qarch=pwr3 -qtune=pw3
```

when compiling for POWER3 systems, or

```
xlf_r -O3 -qstrict -qarch=pwr4 -qtune=pwr4
```

when compiling for POWER4 systems.

If compiling on the same machine on which the code will be run,
you can use

```
xlf_r -O3 -qstrict -qarch=auto -qtune=auto
```

By removing the `-qstrict` option, you may obtain more optimization,
but you should carefully validate the results of your program if
you compile without `-qstrict`.

-qhot

- › -qhot[=[no]vector | arraypad[=n]]
- › Performs higher-order transformations to maximize the efficiency of loops and array language
- › Supported for all languages
- › Can pad arrays for more efficient cache usage
- › Optimized handling of F90 array constructs
- › Can generate calls to vector intrinsic functions
- › Can see performance improvement if program spends substantial time in certain types of nested loop patterns
- › Use `-qreport=hotlist` to have compiler generate a report about loops it has transformed.

Tips for Getting the Most out of `-qhot`

- › Try using `-qhot` along with `-O2` or `-O3` for all of your code (should have neutral effect when no opportunities exist)
- › If you encounter unacceptably long compile times or if performance degrades, try using `-qhot=novector`, or `-qstrict` or `-qcompact` along with `-qhot`.
- › If necessary, deactivate `-qhot` selectively, allowing it to improve some of your code.
- › If your hot loops are not transformed as you expect, try using assertive directives such as `INDEPENDENT` or `CNCALL` or prescriptive directives such as `UNROLL` or `PREFETCH`.

-qipa

- › Performs interprocedural analysis (IPA)
- › Can be specified on the compile step only or on both compile and link steps
- › Whole program mode expands the scope of optimization to an entire program unit (executable or shared library)
- › -qipa[=level=n | inline= | fine tuning]
 - level=0: Simple interprocedural optimization
 - level=1: Inlining and global data mapping
 - level=2: Global alias analysis specialization, interprocedural data flow
 - inline=: Precise user control of inlining

-qarch

- › Specifies the type of processor on which the compiled code will be run and produces an executable that may contain machine instructions specific to that processor. This
- › Allows the compiler to take advantage of processor-specific instructions that can improve performance at the cost of generating code that may not run on other processor types The
- › Default for this option is `-qarch=comm`, which will produce an executable that is runnable on any POWER or POWERPC processor
- › `-qarch=auto` option tells the compiler to produce an executable with machine instructions specific to the processor on which it was compiled.

-qtune

- › Tells the compiler to perform optimizations for the specified processor
- › Processor-specific optimizations include instruction selection and scheduling, setting up pipelining, and taking into account the cache hierarchy to take advantage of the specified processor's hardware.
- › Unlike the -qarch option, the -qtune option does not produce processor-specific code, but the performance of the code on a different processor may be worse than if it were compiled with the appropriate -qtune argument or even with no -qtune option at all.
- › -qtune=auto tells the compiler to produce a program tuned for the processor on which it was compiled.

-qcache

- › Specifies the cache configuration of the processor on which the program will be run.
- › Only available with the Fortran compiler
- › Can only be used in combination with the -qhot option.
- › As with -qtune, a code compiled with this option will run correctly on a different processor but its performance may be worse than if it were compiled with an appropriate -qcache option.
- › -qcache=auto tells the compiler to produce an executable for the cache configuration of the processor on which it was compiled.

HP/Compaq Fortran Debugging Options

-g0

Prevents symbolic debugging information from appearing in the object file.

-g1

Produces traceback information (showing pc to line correlation) in the object file, substantially increasing its size. This is the default.

HP/Compaq Fortran Debugging Options (cont.)

`-g2` or `-g`

Produces traceback and symbolic debugging information in the object file. Unless an explicit optimization level has been specified, these options set the `-O0` option, which turns off all compiler optimizations and makes debugging more accurate.

HP/Compaq Fortran Debugging Options (cont.)

-g3

Produces traceback and symbolic debugging information in the object file, but does not set an optimization level. This option can produce additional debugging information describing the effects of optimizations, but debugger inaccuracies can occur as a result of the optimizations that have been performed.

HP/Compaq Fortran Optimization Levels

-O0

Disables all optimizations.

-O1

Enables local optimizations and recognition of common subexpressions. Optimizations include integer multiplication and division expansion using shifts.

-O2

Enables global optimization and all -O1 optimizations. This includes code motion, strength reduction and test replacement, split lifetime analysis, code scheduling, and inlining of arithmetic statement functions.

HP/Compaq Fortran Optimization Levels (cont.)

-O3

Enables global optimizations that improve speed at the cost of increased code size, and all -O2 optimizations. Optimizations include prefetching, loop unrolling, and code replication to eliminate branches.

-O4 or -O

Enables inline expansion of small procedures, software pipelining, and all -O3 optimizations. This is the default.

HP/Compaq Fortran Optimization Levels (cont.)

-O5

Enables loop transformation optimizations, all -O4 optimizations, and other optimizations, including byte-vectorization, and insertion of additional NOPs (No Operations) for alignment of multi-issue sequences.

HP/Compaq Fortran – Recommended Options

- › For production codes, we recommend the following options if you are compiling on the same machine on which the code will be run:
`-arch=host -tune=host -O`
- › Avoid using the `-fast` option unless you understand the options that `-fast` sets. For example, the `-fast` option sets the `-assume noaccuracy_sensitive` and `-math_library fast` options, which can change the calculated results of a program.

-arch

The -arch option determines for which version of the Alpha architecture the compiler will generate instructions.

arch -generic

generates instructions that are appropriate for most Alpha processors. This is the default.

arch -host

generates instructions for machine the compiler is running on

arch -ev6

generates instructions for ev6 processors (21264 chips)

arch -ev67

generates instructions for ev67 processors (21264A chips)

arch -ev68

generates instructions for ev68 processors (21264C chips)

-tune

The -tune option selects processor-specific instruction tuning.

Regardless of the setting of the -tune option, the generated code will run correctly on all implementations of the Alpha architecture. Tuning for a specific implementation can improve run-time performance, but code tuned for a specific target may run more slowly than generic code on another target.

-tune generic

Selects instruction tuning that is appropriate for all implementations of the Alpha architecture. This is the default.

-tune host

Selects instruction tuning that is appropriate for the machine the compilation is occurring on.

-fast

- › The -fast option Sets the following command options that can improve run-time performance:
 - align dcommons
 - arch host
 - assume noaccuracy_sensitive
 - math_library fast
 - O4
 - tune host
- › For f90 and f95, -fast also sets -align sequence, -assume bigarrays, and -assume nozsize.

-assume noaccuracy_sensitive

- › Reorders floating-point operations, based on algebraic identities (inverses, associativity, and distribution) to improve performance.
- › The default is
-assume accuracy_sensitive

`-math_library fast`

- › Specifies that the compiler is to select the version of the math library routine which provides the highest execution performance for certain mathematical intrinsic functions, such as EXP and SQRT
- › For certain ranges of input values, the selected routine may not provide a result as accurate as `-math_library accurate` (the default setting) provides.

SGI MIPSpro Fortran Compiler Debug Options

-g0

No debugging information is provided.

-g2, -g

Information for symbolic debugging is provided and optimization is disabled.

-g3

Information for symbolic debugging of fully optimized code is produced. The debugging information produced may be inaccurate. This option can be used in conjunction with the -O, -O1, -O2, and -O3 options.

SGI MIPSpro Fortran Compiler Optimization Levels

-O0

No optimization. This is the default.

-O1

Local optimization

-O2, -O

Extensive optimization

-O3

Aggressive optimization. Optimizations performed at this level may generate results that differ from those obtained when -O2 is specified. Vector versions of certain single-precision and double-precision intrinsic procedures are used at this level.

-Ofast

- › Maximizes performance for the target platform ipxx processor type.
- › To determine your default platform ipxx designation, use the **hinv** command.
- › The optimizations performed may differ from release to release and among the supported platforms.
- › The optimizations always enable the full instruction set of the target platform.
- › Although the optimizations are generally safe, they may affect floating-point accuracy due to operator reassociation.

Profile Guided Optimization (PGO)

- › Also called *profile-directed feedback* optimization
- › Uses feedback file(s) to allow the compiler to use profiling data from execution(s) to improve optimizations
- › IBM
 - fdpr command and `-fdpr` compiler option
 - `-qpdf1` and `-qpdf2` compiler options
- › HP/Compaq Fortran options
 - `-gen_feedback`
 - `-feedback`
 - `-cord`
- › SGI MIPSpro Fortran 90 compiler
 - `-fb_create`, `-fb`, and `-fb_opt` options

Tuned Numerical Libraries

Tuned Numerical Libraries

- › BLAS
- › LAPACK
- › ScaLAPACK
- › IBM ESSL, PESSL, and MASS
- › HP/Compaq CXML
- › SGI SCSL
- › SuperLU
- › PETSc
- › ARPACK

BLAS

- › Basic Linear Algebra Subroutines
- › Basic vector and matrix operations
 - › Level 1 – vector operations
 - › Level 2 – matrix-vector operations
 - › Level 3 – matrix-matrix operations
- › Almost all vendors provide highly tuned BLAS libraries.
- › More information – <http://www.netlib.org/blas/>

LAPACK

- › Linear Algebra PACKage
- › Dense linear algebra routines for shared-memory parallel computers
 - › Systems of linear equations
 - › Linear least squares
 - › Eigenvalue problems
 - › Singular value problems
- › Call Level 2 and Level3 wherever possible
- › Efficient use of the memory hierarchy
- › Most vendors provide at least a subset of LAPACK routines in their math libraries.
- › More information – <http://www.netlib.org/lapack/>

ScaLAPACK

- › Scalable Linear Algebra PACKage
- › Dense linear algebra routines for distributed memory parallel computers
- › Uses its own communication package called BLACS (Basic Linear Algebra Communication Subprograms)
- › BLACS can be built with MPI or PVM
- › Subset of ScaLAPACK included in some vendor math libraries
- › More information – <http://www.netlib.org/scalapack/>

IBM Engineering and Scientific Subroutine Library (ESSL)

- › ESSL provides mathematical subroutines for the following areas:
 - › Linear algebra and matrix operations (including BLAS and LAPACK)
 - › Eigensystem analysis (including LAPACK routines)
 - › Fourier transforms, convolutions and correlations, and related computations
 - › Sorting and searching
 - › Interpolation
 - › Numerical quadrature
 - › Random number generation
- › Significant optimizations have been done in ESSL to effectively use the cache and memory hierarchy and minimize memory bandwidth requirements.

IBM ESSL (cont.)

- › ESSL routines are callable from application programs written in Fortran, C, and C++. In order to use ESSL, you must link to one of the ESSL runtime libraries. For example,
`xlf -o executable_name myprog.f -lessl`
- › ESSL includes both serial and SMP versions of the library. If the `-lesslsmp` argument is used, the multithreaded versions will be loaded if they are available. use either the XL Fortran `XLSMPOPTS` or the `OMP_NUM_THREADS` environment variable to specify the number of threads.

IBM ESSL (cont.)

- › The SMP version of ESSL is thread-safe. A thread-safe serial version is also provided. To use the thread-safe serial version with threaded programs, link using `-lessl_r`.

IBM Parallel ESSL (PESSL)

Distributed memory version of ESSL that provides subroutines for the following areas:

- › Subset of Level 2 and Level 3 Parallel BLAS (PBLAS)
- › Dense and sparse linear systems
- › Subset of ScaLAPACK (dense and banded)
- › Sparse systems
- › Subset of ScaLAPACK eigensystem analysis and singular value analysis
- › Fourier transforms
- › Random number generation

IBM PESSL (cont.)

- › PESSL routines are callable from application programs written in Fortran, C, and C++.
- › PESSL is not thread safe; however, PESSL is thread tolerant and can therefore be called from a single thread of a multithreaded application. Multiple simultaneous calls to PESSL from different threads of a single process can cause unpredictable results.

IBM PESSL (cont.)

PESSL provides two run-time libraries:

- › The PESSL SMP library is provided for use with the MPI threads library on POWER SMP processors. To use this library, link using
-lblacs -lesslsmp -lpesslsmp
- › The PESSL serial library is provided for use with the MPI signal handling library on all types of nodes. To use this library, link using
-lblacs -lessl -lpessl.

IBM PESSL (cont.)

- › Sample programs using PESSL can be found in the `/usr/lpp/pessl.rte.common` directory on your IBM POWER3/4.

IBM Mathematical Acceleration SubSystem (MASS)

- › Provides high-performance versions of a subset of Fortran intrinsic functions.
- › Sacrifices a small amount of accuracy to allow for faster execution. Compared to the standard mathematical library, libm.a, the MASS library results differ at most only in the last bit.
- › There are two basic types of functions available for each operation:
 - › A single instance function
 - › A vector function

IBM MASS (cont.)

- › To use the scalar library, link using `-lmass` ahead of `-lm`, e.g.,

```
xlf progf.f -o progf -lmass
```

```
cc progc.c -o progf -lmass -lm
```

- › To use the vector library, link using `-lmassv`, e.g.,

```
xlf progf.f -o progf -lmassv
```

```
cc progc.c -o progf -lmassv -lm
```

(`-lmassv` is the POWER3/POWER4 library. For the POWER3-specific library, use `-lmassv3`. For the POWER4-specific library, use `-lmassv4`.)

IBM MASS (cont.)

- › The Fortran source library libmassv.f has been provided for use on non-IBM systems where the MASS libraries are not available. The recommended procedure for writing a portable code that is vectorized for using the fast MASS vector libraries, is to write in ANSI standard language and use the vector functions defined by libmassv.f. Then, to prepare to run on a system other than an IBM system, compile the application source code together with the libmassv.f source.

HP/Compaq Extended Math Library (CXML)

- › Provides a comprehensive set of mathematical library routines callable from Fortran and other languages.
- › Included with Compaq Fortran for Tru64 UNIX systems
- › To specify the CXML routines library when linking, use the `-lcxml` option.
- › On Tru64 UNIX systems, selected key CXML routines have been parallelized using OpenMP. To link with the parallel version of the CXML library, use `-lcxmlp`.

SGI Scientific Computing Software Library (SCSL)

- › Comprehensive collection of scientific and mathematical functions that have been optimized for SGI systems, both IRIX/MIPS and Linux/IA64.
- › Callable from Fortran, C, and C++
- › Includes BLAS and LAPACK
- › Also includes
 - › Sparse direct solvers
 - › Sparse iterative solvers
 - › Signal processing routines
- › Provides shared memory parallelism

SuperLU

- › General purpose library for the direct solution of large, sparse, nonsymmetric systems of linear equations on high performance machines
- › Routines perform an LU decomposition with partial pivoting and triangular system solves through forward and back substitution.
- › Routines are also provided to equilibrate the system, estimate the condition number, calculate the relative backward error, and estimate error bounds for the refined solutions.
- › Serial, shared memory, and distributed memory versions
- › <http://www.cs.berkeley.edu/~demmel/SuperLU.html>
- › Being installed and supported on DoD HPC Center systems as part of PET CE project

PETSc

- › Large suite of data structures and routines for both uni- and parallel-processor scientific computing
- › Intended especially for the numerical solution of large-scale problems modeled by partial differential equations
- › Includes scalable parallel preconditions and Krylov subspace methods for solving sparse linear systems, as well as parallel nonlinear equation solvers and parallel ODE solvers
- › <http://www-unix.mcs.anl.gov/petsc/petsc-2/>
- › Being installed on DoD HPC Center systems as part of PET CE project

ARPACK

- › Collection of Fortran77 subroutines designed to solve large-scale eigenvalue problems
- › Designed to compute a few eigenvalues and corresponding eigenvectors of a general n by n matrix A
- › <http://www.caam.rice.edu/software/ARPACK>
- › Being installed on DoD HPC Center systems as part of PET CE project

Hand Tuning

General Tuning Guidelines

- › Use local variables, preferably automatic variables, as much as possible
- › Tuning loops
 - › Keep the size of do loops manageable
 - › Access data sequentially (i.e, with unit stride)
 - › Move loop invariant IF statements outside loops
 - › Avoid subroutine and function calls in loops
 - › Simplify array subscripts
 - › Use local INTEGER loop variables
 - › Avoid use of I/O statements in loops

General Tuning Guidelines (cont.)

- › Avoid mixed data type arithmetic expressions
- › Use the smallest floating point precision possible for your code (not necessarily on IBM systems)
- › Avoid using EQUIVALENCE statements
- › Use module variables rather than common blocks for global storage
- › Avoid unnecessary use of pointers
- › Do not excessively hand optimize your code
- › Avoid using many small functions or use interprocedural optimization

Tuning for the Cache and Memory Subsystem

- › Stride minimization
- › Encouragement of data prefetch streaming (on IBM POWER3/POWER4 systems)
- › Avoidance of cache set associativity constraints
- › Data cache blocking

Tuning to Maximize the Efficiency of Computational Units

- › Unrolling inner loops to increase the number of independent computations in each iteration to keep the pipelines full
- › Unrolling outer loops to increase the ratio of computation to load and store instructions so that loop performance is limited by computation rather than data movement

Encouragement of Data Prefetch Streaming (POWER4)

- › Situations where tuning can more fully exploit the hardware prefetch engine:
 - › There are too few or too many streams in a performance-critical loop.
 - › The length of the streams in a performance-critical loop is too short.
- › Loop fusion and fission
- › Midpoint bisection of a loop doubles the number of streams but halves its vector length.
- › Increasing the number of streams from one to eight can improve data bandwidth out of L3cache and memory by up to 70 percent.

Avoidance of Cache Associativity Constraints

- › Avoid leading array dimensions that are multiples of large powers of 2.

- › Example:

```
real*8 a(2048,75)
```

```
...
```

```
do i = 1,75
```

```
  a(100,i) = a(100,i)*1.15
```

```
enddo
```

Assuming a 2-way set associative L1 data cache, how should a be re-dimensioned to avoid cache thrashing?

Data Cache Blocking

If arrays are too big to fit into cache, process them in blocks that do fit in cache.

Four scenarios:

3. All arrays are stride 1 and no data reuse =>no benefit to blocking
4. Some arrays are not stride 1 and no data reuse =>moderate benefit
5. All arrays are stride 1 and much data reuse => moderate benefit
6. Some arrays are not stride 1 and much data reuse (e.g., matrix multiply) => blocking essential

Matrix Multiply

```
DO I = 1,N
  DO J = 1,N
    DO K = 1,N
      C(I,J) = C(I,J)+A(I,K)*B(K,J)
    ENDDO
  ENDDO
ENDDO
```

Matrix Multiply with Blocking

```
DO II = 1,N,NB
  DO JJ = 1,N,NB
    DO KK = 1,N,NB
      DO I = II,MIN(N,II+NB-1)
        DO J = JJ,MIN(N,JJ+NB-1)
          DO K = KK,MIN(N,KK+NB-1)
            C(I,J) = C(I,J)+A(I,K)*B(K,J)
          ENDDO
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

Matrix Multiply with Blocking (cont.)

- › Size of the blocks is $NB \times NB$
- › Three such blocks should fit in L2 cache (on IBM POWER4)
- › Try different values of NB and measure the performance, or observe hardware counter data, to determine the optimal value

Outer Loop Unrolling

- › Goal: Increase the F:M (floating-point to memory operation) ratio
- › Example:

```
DO J = 1,N
```

```
    DO I = 1,N
```

```
        A(I,J) = A(I,J) + X(I) * Y(J)
```

```
    ENDDO
```

```
ENDDO
```

Unrolling by two will require half as many loads of X.

Outer Loop Unrolling (etc.)

```
DO J = 1,N,2
  T0 = Y(J)
  T1 = Y(J+1)
  DO I = 1,N
    A(I,J) = A(I,J) + X(I) * T0
    A(I,J+1) = A(I,J+1) + X(I) * T1
  ENDO
ENDDO
```

Data Alignment

- › For optimal performance, make sure your data are aligned naturally.
- › A *natural bounday* is a memory address that is a multiple of the data item's size.
 - › e.g., a REAL (KIND=8) data item is *naturally aligned* if its starting address is a multiple of 8.
- › Most compilers naturally align individual data items when they can, but
 - › EQUIVALENCE statements can cause data items to become unaligned, and
 - › Compilers may have trouble aligning data within common blocks and structures.

Data Alignment (cont.)

- › Within each common block, derived type, or record structure, carefully specify the order and sizes of data declarations to ensure naturally aligned data.
- › Start with the largest size numeric items first, followed by smaller size numeric items, followed by nonnumeric (character) data.
- › HP/Compaq Fortran *-align* option
- › IBM xlf *-qalign* option
- › Pay attention to compiler and runtime warnings about unaligned data.

Accessing Arrays Efficiently

- › The fastest array access occurs when contiguous access to the entire array or most of an array occurs.
- › Rather than explicit loops for array access, use Fortran 90/95 array syntax.
- › Access multidimensional arrays in *natural storage order*, which is *column-major order* (where the leftmost subscript varies most rapidly) for Fortran.
- › Use padding when necessary to avoid leftmost array dimensions that are a power of 2.
- › Whenever possible, use Fortran 90/95 array intrinsic procedures rather than creating your own routines.

Passing Array Arguments Efficiently

- › Passing an assumed-shape array or an array pointer to an explicit-shape array can slow performance because the compiler needs to create an array temporary for the entire array (because the passed array may not be contiguous and the receiving explicit-shape array must be contiguous).
- › Amount of slowdown depends on the array size.

Test Promotion in Loops (if-do interchange)

- › Branches in code can greatly reduce performance since they interfere with pipelining.
- › Consider the following example:

```
DO I = 1,N
  IF (A. GT. 0) THEN
    X(I) = X(I) + 1
  ELSE
    X(I) = 0.0
  ENDIF
ENDDO
```

Test Promotion in Loops (cont.)

- › Exchanging the if and do constructs causes the if test to be evaluated only once:

```
IF (A. GT. 0.0) THEN
  DO I = 1,N
    X(I) = X(I) + 1.0
  ENDDO
ELSE
  DO I = 1,N
    X(I) = 0.0
  ENDDO
ENDIF
```

Loop Peeling

- › Consider the if tests in the following loop to handle the boundary conditions:

```
DO I = 1,N
  IF (I .EQ. 1) THEN
    X(I) = 0
  ELSEIF (I .EQ. N) THEN
    X(I) = N
  ELSE
    X(I) = X(I) + Y(I)
  ENDIF
ENDDO
```

Loop Peeling (cont.)

- › The if tests can be eliminated by “peeling off” the edge values:

$X(1) = 0$

DO I = 2, N-1

$X(I) = X(I) + Y(I)$

ENDDO

$X(N) = N$

Loop Fusion and Fission

- › Which is better, assuming n is large and that the temp array is not used following these two loops?

```
DO I = 1,N
  TEMP(I) = X(I) + Y(I)
ENDDO
DO I = 1,N
  Z(I) = W(I) + TEMP(I)
ENDDO
```

or

```
DO I = 1,N
  Z(I) = W(I) + X(I)*Y(I)
ENDDO
```

Efficient I/O

- › Use unformatted files whenever possible
- › Write whole arrays or strings instead of individual elements
 - › Each item in an I/O list generates its own calling sequence.
 - › Use Fortran 90/95 array syntax rather than implied DO loops
- › Write array data in natural storage order
 - › If you cannot use natural order, it may be more efficient to reorder the data in memory before performing the I/O
- › Use buffered I/O if possible
- › For MPI programs, use MPI-IO if possible.

Large Page Sizes

- › Increasing the page size (if permitted by the operating system) may reduce TLB misses.
- › SGI IRIX
 - › MIPSpro –bigp_on option
 - › PAGESIZE_DATA, PAGESIZE_STACK, PAGESIZE_TEXT environment variables
 - › Choices – 16, 256, 1024, 4096, 16384 KB
- › IBM POWER4 – 4KB (default), 16MB
 - › http://www-1.ibm.com/servers/aix/whitepapers/large_page.html

Communication Performance

Factors Affecting Communication Performance

- › Platform/architecture related
- › Network related
- › Application related
- › MPI implementation related
- › IBM specific

Platform/Architecture Related Factors

- › Network adapters – type, latency, bandwidth
- › Operating system characteristics
 - › Multithreading
 - › Interrupt handling

Network Related Factors

- › Hardware - ethernet, FDDI, switch, intermediate hardware (routers)
- › Protocols - TCP/IP, UDP/IP, other
- › Configuration, routing, etc
- › Network tuning options ("no" command)
- › Network contention / saturation

Application Related Factors

- › Algorithm efficiency and scalability
- › Communication to computation ratios
- › Load balance
- › Memory usage patterns
- › I/O
- › Message size used
- › Types of MPI routines used - blocking, non-blocking, point-to-point, collective communications

MPI Implementation Related Factors

- › Message buffering
- › Message passing protocols - eager, rendezvous, other
- › Sender-Receiver synchronization - polling, interrupt
- › Routine internals - efficiency of algorithm used to implement a given routine

IBM Specific Factors

- › Type of SP switch and switch adapter
- › Communications network used
- › Number of MPI tasks on an SMP node
 - › Per-task communications bandwidth decreases when more than one MPI task is running on an SMP node.
 - › Aggregate bandwidth increases with the number of MPI tasks until saturation is reached, and then stays steady or may decrease
- › Use of the `MP_SHARED_MEMORY` environment variable
 - › Shared memory mechanism minimizes the degradation in per task communication bandwidth as the number of tasks on a node increases.

Message Buffering

- › Storage of data between the time a send operation begins and when the matching receive operation completes
- › What should be done with a send operation when the matching receive is not posted
- › Buffer space can be provided by the system, or by the user.
 - › System buffering occurs transparently to the user and is determined by the implementation.
 - › User provided buffer space is explicitly declared and managed by the program developer.

Message Buffering (cont.)

- › MPI standard is purposely vague.
- › Implementations differ.
- › E.g., Standard send operation can be implemented in any of the following ways:
 - › Buffer at the sending side
 - › Buffer at the receiving side
 - › Not buffer at all
 - › Buffer under some conditions and not others – e.g., eager vs. rendezvous protocols

System Buffering

- › Advantages
 - › Offers the possibility for improving performance by permitting communications to be asynchronous
 - › E.g., a system buffered send operation can complete even though a matching receive operation has not been posted. The data which are to be sent can be copied and held by the system until the receive occurs, allowing the sending process to do work instead of waiting.

System Buffering (cont.)

- › Main disadvantage - robustness
 - › Buffer space is always a finite resource. Buffer exhaustion/overflow can cause program failure or stalling.
 - › It is not always obvious to the programmer just exactly how (or even whether) an MPI implementation uses buffering. This can make it easy to write programs that fail due to buffer depletion.
 - › Implementations differ on how buffering is performed. A program that runs successfully under one set of conditions may fail under another set.
 - › A correct MPI program does not rely upon system buffer space. Programs that do are called ***unsafe***, even though they may run and produce correct results under most conditions.

MPI Message Passing Protocols

- › An MPI message passing protocol describes the internal methods and policies an MPI implementation employs to accomplish message delivery.
- › Message passing protocols are not defined by the MPI standard, but are left up to implementors, and will vary.
- › MPI implementations can use a combination of protocols for the same MPI routine. For example, a standard send might use eager protocol for a small message, and rendezvous protocol for larger messages.
- › MPI message passing protocols often work in conjunction with message buffering.

Two Common Message Passing Protocols

- › **Eager** - asynchronous protocol that allows a send operation to complete without acknowledgement from a matching receive
- › **Rendezvous** - synchronous protocol that requires an acknowledgement

Eager Protocol

- › Assumption by the sending process that the receiving process can store the message if it is sent
- › Responsibility of the receiving process to buffer the message upon its arrival, especially if the receive operation has not been posted
- › Assumption may be based upon the implementation's guarantee of a certain amount of available buffer space on the receive process
- › Generally used for smaller message size (message size may be limited by the number of MPI tasks)

Eager Protocol - Advantages

- › Reduces synchronization delays - send process does not need acknowledgement from receive process that it's OK to send message.
- › Simplifies programming - only need to use MPI_Send

Eager Protocol - Disadvantages

- › Not scalable - significant buffering may be required to provide space for messages from an arbitrary number of senders
- › Can cause memory exhaustion and program termination when receive process buffer is exceeded
- › Buffer "wastage" - allocated even if only a few messages are sent
- › May consume CPU cycles by the receive process side to pull messages from the network and/or copy the data into buffer space

Rendezvous Protocol

- › Used when assumptions about the receiving process buffer space can't be made, or when the limits of the eager protocol are exceeded
- › Requires some type of "handshaking" between the sender and the receiver processes. For example:
 1. Sender process sends message envelope to destination process
 2. Envelope received and stored by destination process
 3. When buffer space is available, destination process replies to sender that requested data can be sent
 4. Sender process receives reply from destination process and then sends data
 5. Destination process receives data

Rendezvous Protocol - Advantages

- › Scalable compared to eager protocol
- › Robust - prevents memory exhaustion and termination on receive process
- › Only required to buffer small message envelopes
- › Possibility for eliminating a data copy - user space to user space direct

Rendezvous Protocol - Disadvantages

- › Inherent synchronization delays due to necessary handshaking between sender and receiver
- › More programming complexity - may need to use non-blocking sends with waits/tests to prevent program from blocking while waiting on the OK from receive process

Sender-Receiver Synchronization

- › Synchronous MPI communication operations, such as those using a rendezvous protocol, require synchronization of the sending task and the receiving task.
- › The MPI standard does not define how this synchronization should be accomplished. Some questions left up to MPI implementors:
 - › How does a receive task know if a task is requesting to send?
 - › How often should a receive task check for incoming messages?
 - › After issuing a non-blocking send, how should a CPU bound process yield its cycle time to complete the send operation?
- › MPI implementations typically rely upon two different modes to accomplish sender-receiver synchronization: ***polling*** or ***interrupt***.

Polling vs. Interrupt

- › Polling Mode
 - › The user MPI task will be interrupted by the system to check for and service communication events at regular (implementation defined) intervals. If a communication event occurs while the user task is busy doing other work, it must wait.
- › Interrupt Mode
 - › The user MPI task will be interrupted by the system for communication events when they occur.
 - › Usually a cost associated with interrupts

Polling vs. Interrupt - Performance

- › Applications that have the following characteristics may see performance improvements when using with interrupt mode:
 - › Applications that use nonblocking send or receive operations for communication.
 - › Applications that have non-synchronized sets of send or receive pairs. In other words, the send from node0 is issued at a different point in time with respect to the matching receive in node1.
 - › Applications that do not issue waits for nonblocking send or receive operations immediately after the send or receive, but rather do some computation prior to issuing the waits.

Message Size

- › Can be a very significant contributor to MPI application performance
- › In most cases, increasing the message size will yield better performance.
- › For communication intensive applications, algorithm modifications that take advantage of message size "economies of scale" may be worth the effort.
- › Performance can often improve significantly within a relatively small range of message sizes.

Point-to-point Communications

- › MPI provides many ways to send and receive messages.
- › **Send routines** (match any receive, probe; non-blocking can match any completion/testing)
 - › Blocking - standard, buffered, ready, synchronous
 - › Non-blocking - standard, buffered, ready, synchronous
 - › Persistent - standard, buffered, ready, synchronous

Point-to-point Communications (cont.)

- › **Receive routines** (match any send)
 - › Blocking
 - › Non-blocking
 - › Persistent
- › **Probe routines**(match any send)
 - › Blocking
 - › Non-blocking
- › **Completion / Testing routines** (match any non-blocking send/receive)
 - › Blocking - one, some, any, all
 - › Non-blocking - one, some, any, all

Point-to-point Communications - Performance

- › Performance can vary depending on which routines are used and how they are implemented.
- › General observations
 - › Non-blocking operations perform best.
 - › Greatest performance gains occur within the small to mid-size message range.

Persistent Communications

- › Can be used to reduce communications overhead in applications that repeatedly call the same point-to-point message passing routines with the same arguments
 - › Example of application that might benefit is an iterative, data decomposition algorithm that exchanges border elements with its neighbors
- › Minimize the software overhead associated with redundant message setup
- › Persistent communication routines are non-blocking.

Collective Communications

- › Require the participation of all tasks within the communicator
- › Inter-task synchronization and load balance are critical to performance.
- › The same operations can alternately be accomplished by ordinary send-receive operations.
- › The MPI-1 standard specifies blocking collective communication routines only.
- › MPI-2 defines corresponding non-blocking routines, which may provide better performance for some applications.

Collective Communications - Implementation

- › The MPI standard does not define how collective communication operations should be implemented.
- › Implementations will vary in how efficiently their collective communication routines are implemented:
 - › The algorithms used to implement collective communication routines are generally hidden from the user.
 - › For critical applications, it may be useful to compare hand-coded methods against vendor collective communication routines.
 - › For critical applications, it may also be useful to compare different implementations on the same platform, or implementations on different platforms.

Derived Datatypes

- › Allow the programmer to create arbitrary, contiguous and non-contiguous structures from the MPI primitive datatypes
- › Useful for constructing messages that contain values with different datatypes (e.g., an integer count followed by a sequence of real numbers) and for sending noncontiguous data (e.g., a sub-block of a matrix)
- › Can eliminate the overhead of sending and receiving multiple small messages

Derived Datatypes - Performance

- › For non-contiguous data, the MPI standard specifies that the "gaps" in the data structure should not be sent. It leaves it up to implementors however, whether or not the actual data to be sent are:
 - › first copied into a buffer and then sent as a contiguous message, which must be "unpacked" on the receiving side
 - › sent directly from the application buffer to the receive process, where it may be buffered or not.
- › Using non-contiguous datatypes may actually result in performance degradation. In these cases, the programmer may consider packing and unpacking the data "by hand".

Network Contention

- › Occurs when the volume of data being communicated between MPI tasks saturates the bandwidth of the network
- › Results in an overall decrease of communications performance for all tasks
- › Not much a user can do about this situation except be aware of it, especially if running on a system with other users running communication intensive applications.

SGI MPI Performance Tips

- › Sometimes desirable to avoid buffering (see next slide)
- › Avoid derived data types
 - › May disable unbuffered or single copy data transfer optimizations
- › Avoid use of wild cards for large process counts
- › Consider replacing MPI send/recv calls with MPI-2 MPI_Put/MPI_Get calls in latency sensitive sections of an application


SGI MPI – Avoiding Message Buffering – Single Copy Method

To use the unbuffered pathway

- › Link using `-64`
- › The MPI data type on the sender side must be a contiguous type.
- › Sender and receiver MPI processes must reside on the same host.
- › The sender data must be globally accessible.
- › Set `MPI_BUFFER_MAX` to the message length in bytes beyond which the single copy method should be tried – e.g., 2048

SGI MPI – Buffered vs. Unbuffered

MPI_Send/MPI_Recv bandwidth improvement using unbuffered instead of buffered (from the SGI MPI Programmer's Manual)

Message length	O2000	O3000
8KB	1.2	1.1
1MB	1.2	1.2
10MB	1,8 	1.6

SGI MPI Collective Operations

- › MPI_Alltoall and MPI_Barrier optimized to avoid message buffering when using shared memory
- › MPI_Allreduce and MPI_Bcast not optimized for shared memory

SGI MPI Environment Variables

- › Use of the following environment variables may improve MPI performance in some situations (type “man mpi” for more information):
 - › MPI_BAR_DISSEM
 - › MPI_DSM_PLACEMENT
 - › MPI_DSM_PPM
 - › MPI_BUFFER_MAX
 - › MPI_XPMEM_ON
 - › MPI_DSM_MUSTRUN
 - › MPI_DSM_PLACEMENT
 - › MPI_DSM_PPM
 - › MPI_DSM_TOPOLOGY

OpenMP Performance

What is OpenMP?

Three components:

- › Set of *compiler directives* for
 - creating teams of threads
 - sharing the work among threads
 - synchronizing the threads
- › *Library routines* for setting and querying thread attributes
- › *Environment variables* for controlling run-time behavior of the parallel program

Parallelism in OpenMP

- › The *parallel region* is the construct for creating multiple threads in an OpenMP program.
- › A *team of threads* is created at run time for a parallel region.

Specifying Parallel Regions

Fortran

```
PARALLEL [clause [clause...]]
```

```
! $OMP
```

```
! Block of
```

```
code executed by all threads
```

```
! $OMP END PARALLEL
```

C and C++

```
#pragma omp parallel [clause [clause...]] {
```

```
/*
```

```
Block executed by all threads */
```

```
}
```

Compiling and Linking

MIPSpro: compile & link with `-mp` option

Fortran:

```
f90 [options]-mp -O3 prog.f  
-freeform needed for free form source  
-cpp needed when using #ifdef s
```

C/C++:

```
cc -mp -O3 prog.c  
CC -mp -O3 prog.C
```

AIX/ XL Fortran: use `-qsmp` option

```
xlf_r -qsmp -O3 prog.f
```

Work sharing in OpenMP

Two ways to specify parallel work:

- Explicitly coded in parallel regions
- Work-sharing constructs
 - » DO and for constructs: parallel loops
 - » sections
 - » single

SPMD type of parallelism supported

Work and Data Partitioning

Loop parallelization

- › distribute the work among the threads, without explicitly distributing the data
- › scheduling determines which thread accesses which data
- › communication between threads is implicit, through data sharing
- › synchronization via parallel constructs or explicitly inserted into the code

Work Sharing Constructs

DO and for : parallelizes a loop, dividing the iterations among the threads

sections : defines a sequence of contiguous blocks, the beginning of each block being marked by a section directive. The block within each section is assigned to one thread

single: assigns a block of a parallel region to a single thread

Data Scoping

- › Work-sharing and `parallel` directives accept *data scoping* clauses.
- › Scope clauses apply to the static extent of the directive and to variables passed as actual arguments.
- › The `shared` clause applied to a variable means that all threads will access the single copy of that variable created in the master thread.

Data Scoping (cont.)

- › The `private` clause applied to a variable means that a volatile copy of the variable is cloned for each thread.

OpenMP Scheduling

Static

- › threads are statically assigned chunks of size `chunk` in a round-robin fashion. The default for `chunk` is $\text{ceiling}(N/p)$ where N is the number of iterations and p is the number of processors

Dynamic

- › threads are dynamically assigned chunks of size `chunk`, i.e., when a thread is ready to receive new work, it is assigned the next pending chunk. Default value for `chunk` is 1.

OpenMP Scheduling (cont.)

Guided

- › a variant of dynamic scheduling in which the size of the chunk decreases exponentially from chunk to 1. Default value for `chunk` is *ceiling(N/p)*

Runtime

- › indicates that the schedule type and chunk are specified by the environment variable `OMP_SCHEDULE`. A `chunk` cannot be specified with `runtime`.
- › Example of run-time specified scheduling
- ›

```
setenv OMP_SCHEDULE "dynamic, 2"
```

Message Passing versus Shared Memory

- › Process versus thread address space
 - › Threads have shared address space, but the thread stack holds thread-private data.
 - › Processes have separate address spaces.
- › For message passing, e.g., MPI, all data are explicitly communicated, no data are shared.
- › For OpenMP, threads in a parallel region reference both private and shared data.
- › Synchronization: explicit or embedded in communication

OpenMP Loop Parallelization

Identify the loops that are bottleneck to performance

Parallelize the loops, and ensure that

- no data races are created
- cache friendliness is preserved
- page locality is achieved
- synchronization and scheduling overheads are minimized

Obstacles to Loop Parallelization

Data dependencies among iterations caused by shared variables

Input/Output operations inside the loop

Calls to thread-unsafe code, e.g., the intrinsic function `rtc`

Branches out of the loop

Insufficient work in the loop body

The auto-parallelizer can help in identifying these obstacles.

Automatic Parallelization

Some compilers have an option that will automatically parallelize your code.

The auto-parallelizer will insert OpenMP directives into your code if a loop can be parallelized. If not, it will tell you why not.

“Safe” parallelization implies there are no dependencies.

Only loops can be parallelized automatically.

Should be considered, at best, as a first step toward parallelizing your code.

The next steps should be inserting your own directives and tuning the parallel sections.

Strategies for Using Auto-parallelization

Run the auto-parallelizer on source files and example the listing.

For loops that don't automatically parallelize, try to eliminate inhibiting dependencies by modifying the source code.

Use the listing to implement parallelization by hand using OpenMP directives.

MIPSPRO Auto-parallelizer (APO)

The MIPSPRO auto-parallelizer (APO) can be used both for automatically parallelizing loops and for determining the reasons which prevent a loop from being parallelized.

The auto-parallelizer is activated using command line option `apo` to the `f90`, `f77`, `cc`, and `CC` compilers.

Other auto-parallelizer options: `list` and `mplist`

MIPSpro APO Usage

Example:

```
f90 -apo list -mplist myprog.f
```

`apo list` enables APO and generates the file `myprog.list` which describes which loops have been parallelized, which have not, and why not

`mplist` generates the parallelized source program `myprog.w2f.f` (`myprog.w2c.c` for C) equivalent to the original code `myprog.f`

MIPSpro OpenMP Example

cvpav analyzes files created by compiling
with the option `-apo keep`

Try out the tutorial examples:

```
cp -r /  
usr/demos/ProMP/omp_tutorial  
make  
cvpav -f omp_demo.f
```

Data Races

- › Parallelizing a loop with data dependencies causes *data races*: unordered or interfering accesses by multiple threads to shared variables, which make the values of these variables different from the values assumed in a serial execution.
- › A program with data races produces unpredictable results, which depend on thread scheduling and speed.

Data Dependencies Example

Carried dependence on a shared array, e.g., recurrence:

```
const int n = 4096;  
int a[n], i;  
for (i = 0; i < n-1; i++) {  
    a[i] = a[i+1];  
}
```

Non-trivial to eliminate, the auto-parallelizer cannot do it

Parallelizing the Recurrence

Idea: Segregate the even and odd indices

```
#define N 16384
int a[N], work[N+1];

// Save border element
work[N]= a[0];

// Save & shift even
indices
#pragma omp parallel for
for ( i = 2; i < N; i+=2)
{
    work[i-1] = a[i];
}

// Update even indices from odd
#pragma omp parallel for
for ( i = 0; i < N-1; i+=2)
{
    a[i] = a[i+1];
}

// Update odd indices with even
#pragma omp parallel for
for ( i = 1; i < N-1; i+=2)
{
    a[i] = work[i];
}

// Set border element
a[N-1] = work[N];
```

Common OpenMP Performance Problems

- › Parallel startup costs
- › Small loops
- › Load imbalances
- › Many references to shared variables
- › Low cache affinity
- › Costly remote memory references in NUMA machines
- › Unnecessary synchronization

Parallelization Tradeoffs

- › To increase parallel fraction of work when parallelizing loops, it is best to *parallelize the outermost loop* of a nested loop
- › However, doing so may require loop transformations such as *loop interchanges*, which can destroy cache friendliness, e.g., defeat *cache blocking*

Scheduling Tradeoffs

- › Static loop scheduling in large chunks per thread promotes cache and page locality but may not achieve load balancing.
- › Dynamic and interleaved scheduling achieve good load balancing but may cause poor locality of data references.

Loop Fusion

- › Increases the work in the loop body
- › Better serial programs: fusion promotes software pipelining and reduces the frequency of branches
- › Better OpenMP programs:
 - › fusion reduces synchronization and scheduling overhead
 - › fewer parallel regions and work-sharing constructs

Promoting Loop Fusion

- › Loop fusion inhibited by statements between loops that may have dependencies with data accessed by the loops
- › Promote fusion: rearrange the code to get loops that are not separated by statements creating data dependencies
- › Use one **parallel do** construct for several adjacent loops; may leave it to the compiler to actually perform fusion

Performance Tuning – Data Locality

- › On NUMA platforms, it may be important to know
 - › where threads are running
 - › what data are in their local memories
 - › the cost of remote memory references
- › OpenMP itself provides no mechanisms for controlling
 - › the binding of threads to particular processors
 - › the placement of data in particular memories
- › Often system-specific mechanisms for addressing these problems
 - › additional directives for data placement
 - › ways to control where individual threads are running

Cache Friendliness

For both serial loops and parallel loops

- › locality of references
 - spatial locality: use adjacent cache lines and all items in a cache line
 - temporal locality: reuse same cache line; may employ techniques such as *cache blocking*

For parallel loops

- › low cache contention
 - avoid the sharing of cache lines among different objects; may resort to *array padding* or *increasing the rank* of an array

Cache Friendliness for Parallel Programs

Contention is an issue specific to parallel loops, e.g., *false sharing of cache lines*

cache friendliness =
high locality of references
+
low contention

Page Level Locality

An ideal application has full *page locality*: pages accessed by a processor are on the same node as the processor, and no page is accessed by more than one processor (no page sharing)

Twofold benefit:

- » low memory latency
- » scalability of memory bandwidth

Page Level Locality (cont.)

The benefits brought about by page locality are more important for programs that are *not* cache friendly

We look at several data placement strategies for improving page locality

- » system based placement
- » data initialization and directives
- » combination of system and program directed data placement

IRIX Page Placement

IRIX has two page placement policies:

first-touch: the process which first references a virtual address causes that address to be mapped to a page on the node where the process runs

round-robin: pages allocated to a job are selected from nodes traversed in round-robin order

In addition, IRIX supports fixed page placement

IRIX Page Placement (cont.)

Fixed page placement: pages are placed in memory by the program using compiler directives or system calls

IRIX uses first-touch, unless fixed placement is specified or

```
setenv __DSM_PLACEMENT ROUND_ROBIN
```

or

```
setenv __DSM_ROUND_ROBIN
```


IRIX Page Migration

IRIX allows to *migrate* pages between nodes, to adjust the page placement

A page is migrated based on the *affinity of data accesses* to that page, which is derived at run-time from the per-process cache-miss pattern.

Page migration follows the page affinity with a delay whose magnitude depends on the *aggressiveness* of migration.

IRIX Page Migration (cont.)

Page Migration:

- » makes initial data placement less important, e.g., allows sequential data initialization
- » improves locality of a computation whose data access pattern changes during the computations
- » is useful for programs that have stable affinity for long time intervals

IRIX Page Migration (cont.)

To enable data migration, except for explicitly placed data

```
setenv __DSM_MIGRATION ON
```

To enable migration of *all* data

```
setenv __DSM_MIGRATION ALL_ON
```

To set the aggressiveness of migration

```
setenv __DSM_MIGRATION_LEVEL
```

n

where *n* is an integer between 0 (least aggressive, disables migration) and 100 (most aggressive, the default)

Avoid Synchronization and Scheduling Overheads

- › Partition in few parallel regions
- › Make the code loop fusion friendly
- › Use the **nowait** clause where possible
- › Avoid single and critical sections
- › Use static scheduling unless dynamic load balancing is needed

Performance Analysis Tools

Definitions – Profiling

Profiling

Recording of summary information during execution

inclusive, exclusive time, # calls, hardware statistics, ...

Reflects performance behavior of program entities

functions, loops, basic blocks

user-defined “semantic” entities

Very good for low-cost performance assessment

Helps to expose performance bottlenecks and hotspots

Implemented through

sampling: periodic OS interrupts or hardware counter traps

instrumentation: direct insertion of measurement code

Definitions – 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

Available Profiling Tools

prof, gprof

PAPI (profiling based on timers or on any PAPI hardware counter metric)

Dynaprof (requires dyninst or DPCL)

GuideView (OpenMP) (being phased out)

Vampir (MPI)

TAU (OpenMP, MPI, MPI/OpenMP)

SvPablo

HPCView

Vprof

Vendor specific tools (e.g., SGI IRIX perfex and ssrun, IBM tprof and trace, Cray PAT)

prof, gprof

Available on many Unix platforms (e.g., IBM AIX, Sun Solaris, HP/Compaq Tru64)

Produces “flat” profile (prof) or “call graph” profile (gprof)

Compile with special flag or set environment variable or rewrite executable to enable instrumentation and produce profile data file

Collect profile data by periodically sampling the program counter during execution and/or calling monitoring routines

See

man prof

man gprof

prof Profile of FSPX Benchmark

%time	cumulative	self	calls	ms/call	tot/call	name
21.71	18.93	18.93	6080	3.11	3.11	flux_
19.99	36.36	17.43	9124	1.91	3.91	proflux_
8.26	43.56	7.20	6080	1.18	1.18	pde_
8.11	50.63	7.07	6080	1.16	4.17	phase_
7.96	57.57	6.94	100061386	0.00	0.00	cplintg_
7.46	64.08	6.51	100061388	0.00	0.00	cpsintg_
6.05	69.36	5.28	49807360	0.00	0.00	tsofx_
5.60	74.24	4.88	49807362	0.00	0.00	tlofx_
4.07	77.79	3.55	62202877	0.00	0.00	cpl_
2.44	79.92	2.13	37371906	0.00	0.00	cps_
1.67	81.38	1.46	37371904	0.00	0.00	hl_
1.43	82.63	1.25	37371904	0.00	0.00	hs_
1.07	83.56	0.93	24903681	0.00	0.00	elqds_
0.89	84.34	0.78	37371904	0.00	0.00	aks_

IBM Profiling Utilities

In addition to man pages, see the AIX 5
Performance Management Guide

prof

gprof

tprof

SGI Profiling Utilities

ssrun

Collects Speedshop and Workshop performance data

Types of experiments: totaltime, usertime, pcsamp, ideal, hardware counter

prof

Analyzes and displays SpeedShop performance data generated by ssrun

perfex

Runs program and collects hardware counter data

HP/Compaq Profiling Utilities

prof

Displays profiling data from `-p` or `pixie`

pixie

Instruction-counting profiler

gprof

Displays profiling data from `-pg` or `hiprof`

hiprof

Creates instrumented version of program for call-graph profiling

uprofile

Profiles a program with Alpha on-chip performance counters



Overview of PAPI

Performance **A**pplication **P**rogramming Interface

The purpose of the PAPI project is to design, standardize and implement a portable and efficient API to access the hardware performance monitor counters found on most modern microprocessors.

Parallel Tools Consortium project

Being installed and supported at the DoD HPC Centers as part of PET CE004

PAPI Counter Interfaces



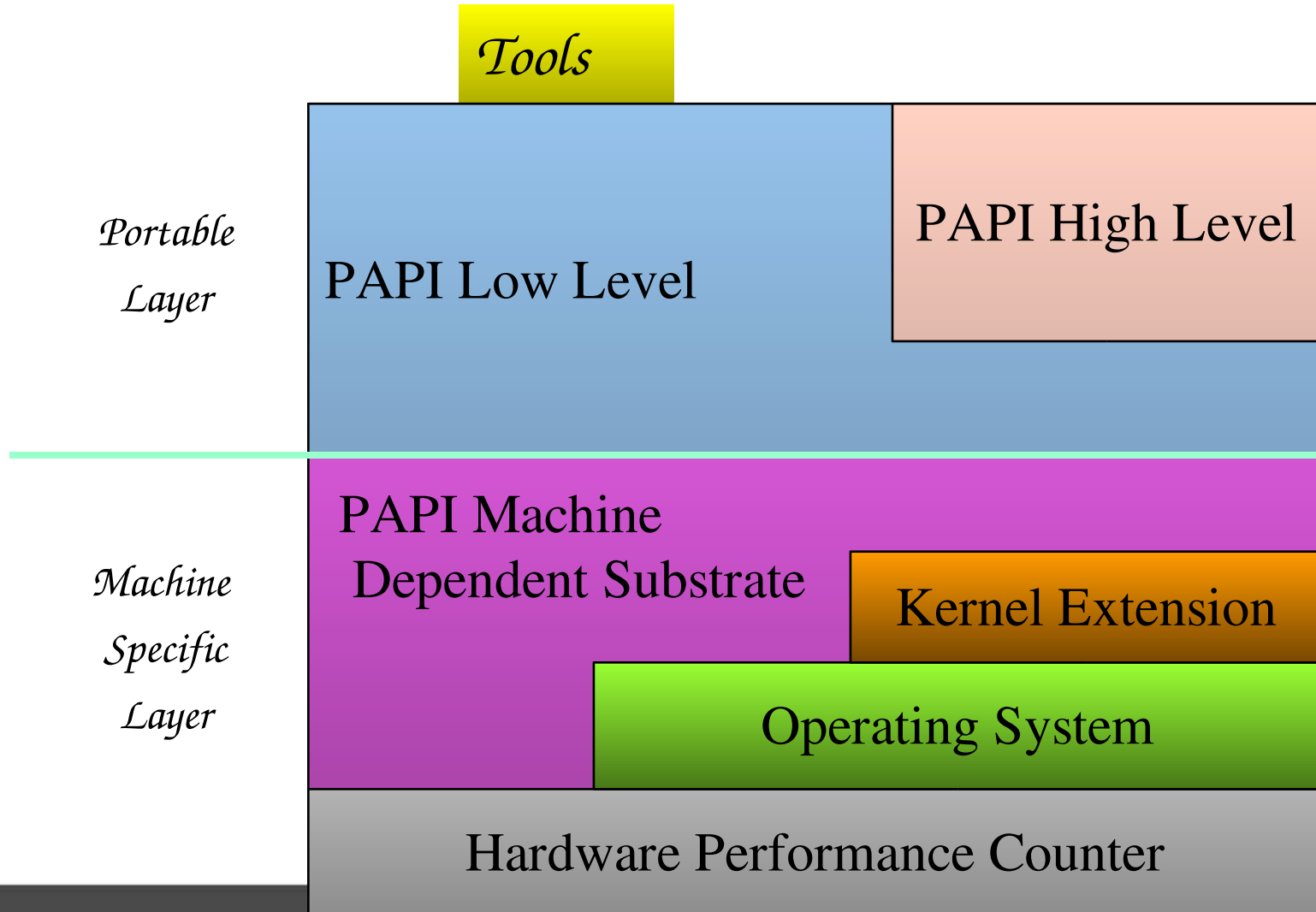
PAPI provides three interfaces to the underlying counter hardware:

The low level interface manages hardware events in user defined groups called *EventSets*.

The high level interface simply provides the ability to start, stop and read the counters for a specified list of events.

Graphical tools to visualize information.

PAPI Implementation



PAPI Preset Events

Proposed standard set of events deemed most relevant for application performance tuning

Defined in `papiStdEventDefs.h`

Mapped to native events on a given platform

Run tests/avail to see list of PAPI preset events available on a platform

High-level Interface

Meant for application programmers wanting coarse-grained measurements

Not thread safe

Calls the lower level API

Allows only PAPI preset events

Easier to use and less setup (additional code) than low-level

High-level API

C interface

PAPI_start_counters
PAPI_read_counters
PAPI_stop_counters
PAPI_accum_counters
PAPI_num_counters
PAPI_flips
PAPI_ipc

Fortran interface

PAPIF_start_counters
PAPIF_read_counters
PAPIF_stop_counters
PAPIF_accum_counters
PAPIF_num_counters
PAPIF_flips
PAPIF_ipc

PAPI_flips

```
int PAPI_flips(float *real_time, float *proc_time, long_long *flpins,  
float *mflops)
```

Only two calls needed, PAPI_flips before and after the code you want to monitor

real_time is the wall-clocktime between the two calls

proc_time is the “virtual” time or time the process was actually executing between the two calls (not as fine grained as real_time but better for longer measurements)

flpins is the total floating point instructions executed between the two calls

mflops is the Mflop/s rating between the two calls

Low-level Interface

Increased efficiency and functionality over the high level PAPI interface

About 40 functions

Obtain information about the executable and the hardware

Thread-safe

Fully programmable

Callbacks on counter overflow

Callbacks on Counter Overflow

PAPI provides the ability to call user-defined handlers when a specified event exceeds a specified threshold.

For systems that do not support interrupt on counter overflow at the operating system level, PAPI sets up a high resolution interval timer and installs a timer interrupt handler.

Statistical Profiling

PAPI provides support for execution profiling based on any counter event.

The `PAPI_profil()` call creates a histogram of overflow counts for a specified region of the application code.

PAPI_profil

```
int PAPI_profil(unsigned short *buf, unsigned int  
bufsiz, unsigned long offset, unsigned scale, int  
EventSet, int EventCode, int threshold, int flags)
```

- buf – buffer of bufsiz bytes in which the histogram counts are stored
- offset – start address of the region to be profiled
- scale – contraction factor that indicates how much smaller the histogram buffer is than the region to be profiled

What is DynaProf?

A portable tool to instrument a running executable with *Probes* that monitor application performance.

Simple command line interface.

Open Source Software

A work in progress...

No source code required

DynaProf Methodology

Make collection of run-time performance data easy by:

- Avoiding instrumentation and recompilation

- Using the same tool with different probes

- Providing useful and meaningful probe data

- Providing different kinds of probes

- Allowing custom probes

No source code required!

Why the “Dyna”?

Instrumentation is selectively inserted directly into the program’s address space.

Why is this a better way?

- No perturbation of compiler optimizations

- Complete language independence

- Multiple Insert/Remove instrumentation cycles

DynaProf Design

GUI, command line & script driven user interface

Uses GNU readline for command line editing and command completion.

Instrumentation is done using:

Dyninst on Linux, Solaris and IRIX

DPCL on AIX

DynaProf Commands

load <executable>

list [module pattern]

use <probe> [probe args]

instr module <module> [probe args]

instr function <module> <function> [probe args]

stop

continue

run [args]

Info

unload

Dynaprof Probes

papiprobe

wallclockprobe

perfometerprobe

DynaProf Probe Design

Can be written in any compiled language

Probes export 3 functions with a standardized interface.

Easy to roll your own (<1day)

Supports separate probes for
MPI/OpenMP/Pthreads

Future development

GUI development

Additional probes

- Perfex probe

- Vprof probe

- TAU probe

Better support for parallel applications

KAP/Pro Toolset

<http://www.kai.com/>

Set of tools for developing parallel scientific software using
OpenMP

Components

- Guide OpenMP compiler

- Assure OpenMP debugger

- GuideView performance analyzer

- Utilities for translating older directives to OpenMP

Licensed at ERDC MSRC and ARL MSRC

Being phased out by Intel/KSL

GuideView

Intuitive, color-coded display of parallel performance bottlenecks

Regions of code are identified where improving local performance will have the greatest impact on overall performance.

Configuration file Gvproperties.txt controls GUI features (fonts, colors, window sizes and locations, etc.)

Online help system under Help menu

Use `kmp_set_parallel_name` to name parallel regions so that name will be displayed by GuideView

Guide Instrumentation Options

-WGnoopenmp

Enable profiling but no OpenMP

-WGprof

Activates profiling for Vampir and GuideView; implies `-WGstats`

-WGprof_leafprune=<integer>

Sets the minimum size of procedures to retain in Vampir or GuideView profiles to <integer> lines

Use to reduce instrumentation overhead and tracefile size

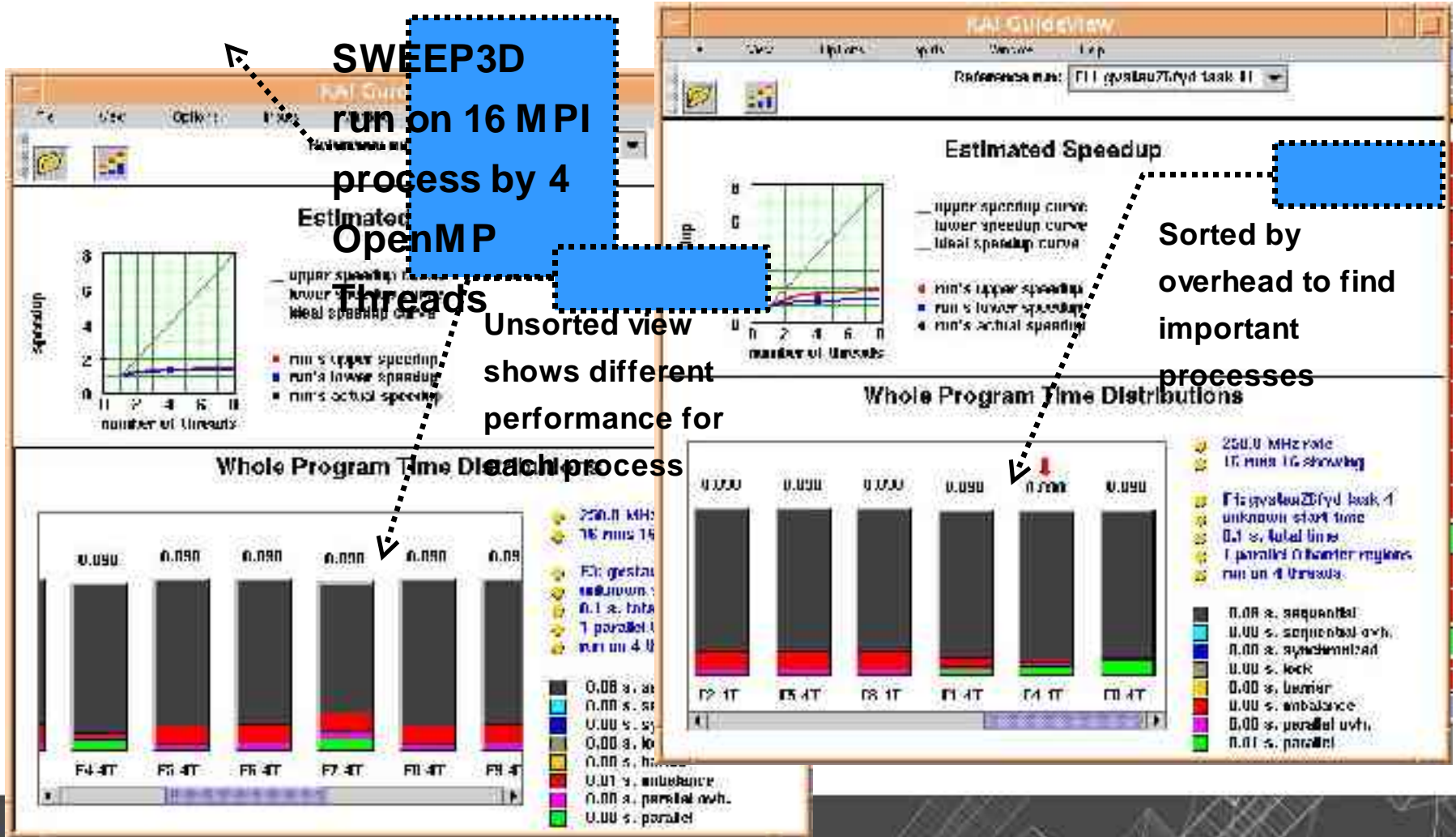
GuideView – OpenMP Profile

SWEEP3D
run on 16 MPI
process by 4
OpenMP

Threads

Unsorted view
shows different
performance for
each process

Sorted by
overhead to find
important
processes



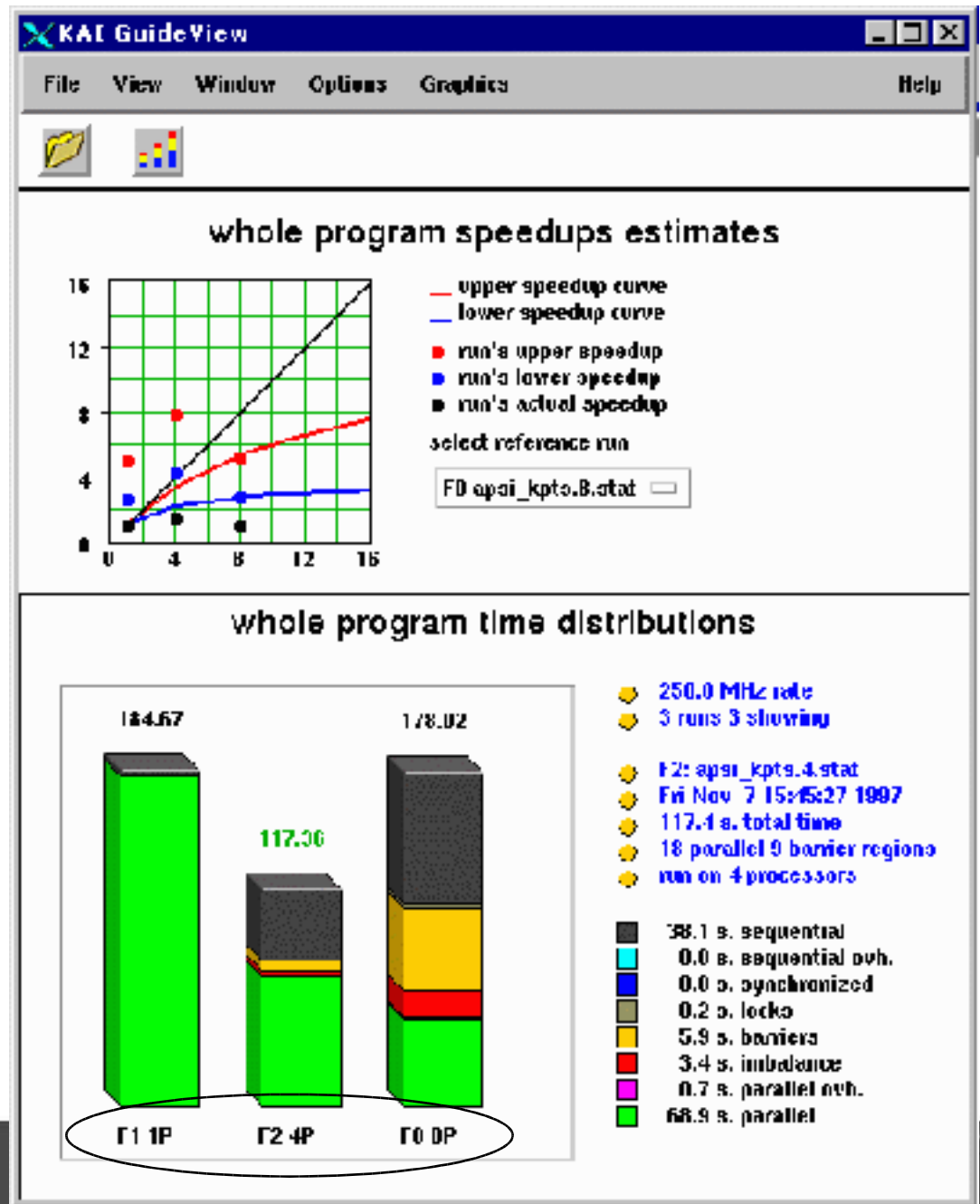
GuideView

Compare
achieved to ideal
Performance

Identify parallel
bottlenecks such
as Barriers,
Locks, and

Sequential time

Compare
multiple runs



Vampir

<http://www.pallas.com/e/products/vampir/index.htm>

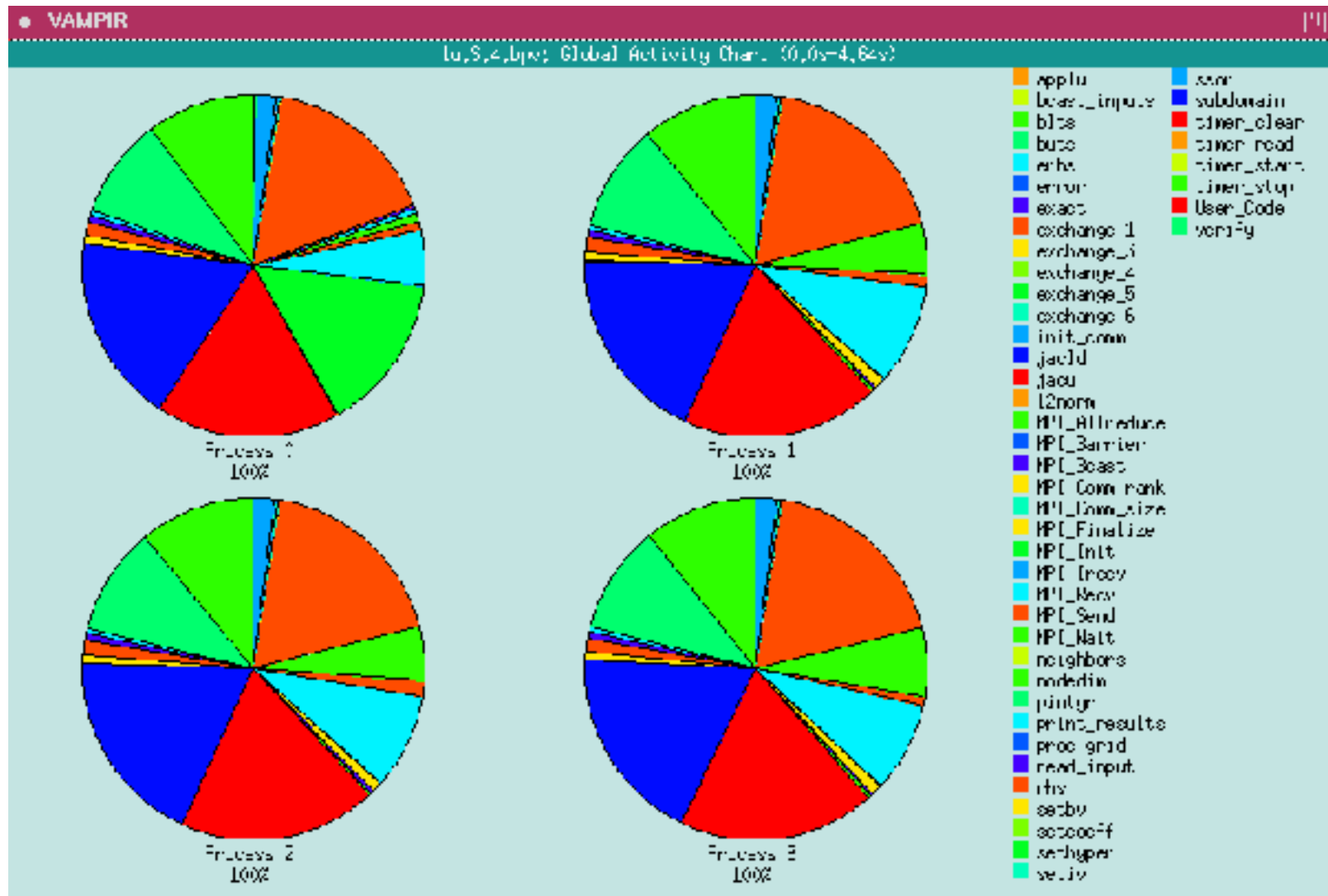
Primarily a tracing tool but also generates and displays profiling statistics

Version 3 will support OpenMP and mixed MPI/OpenMP

Version 3.x will use PAPI to access hardware counter data

Licensed at ERDC MSRC, ARL MSRC, and ARSC

Vampir Statistics Display (also available in text form)



Vampir: Timeline Diagram

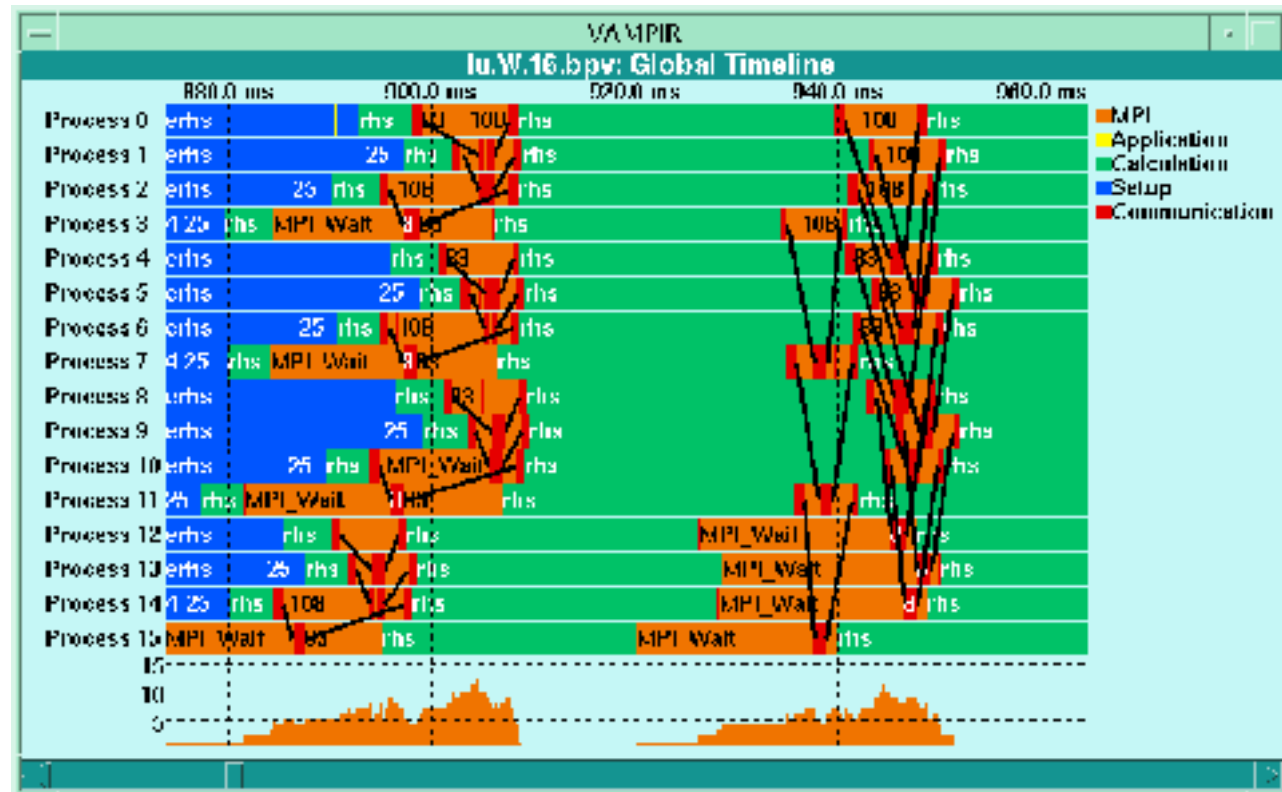
Functions

organized into
groups

Coloring by group

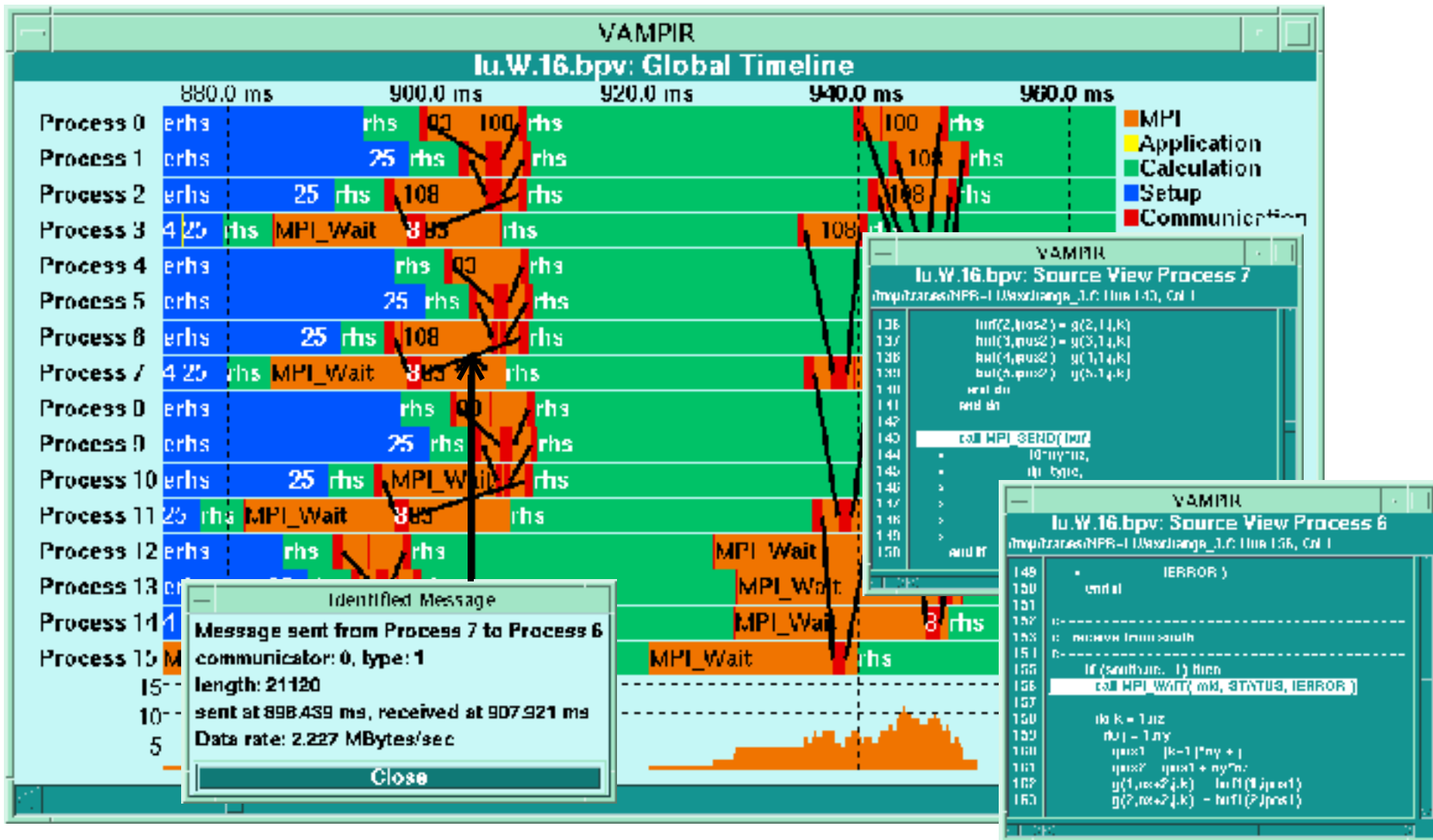
Message lines

can be colored
by tag or size

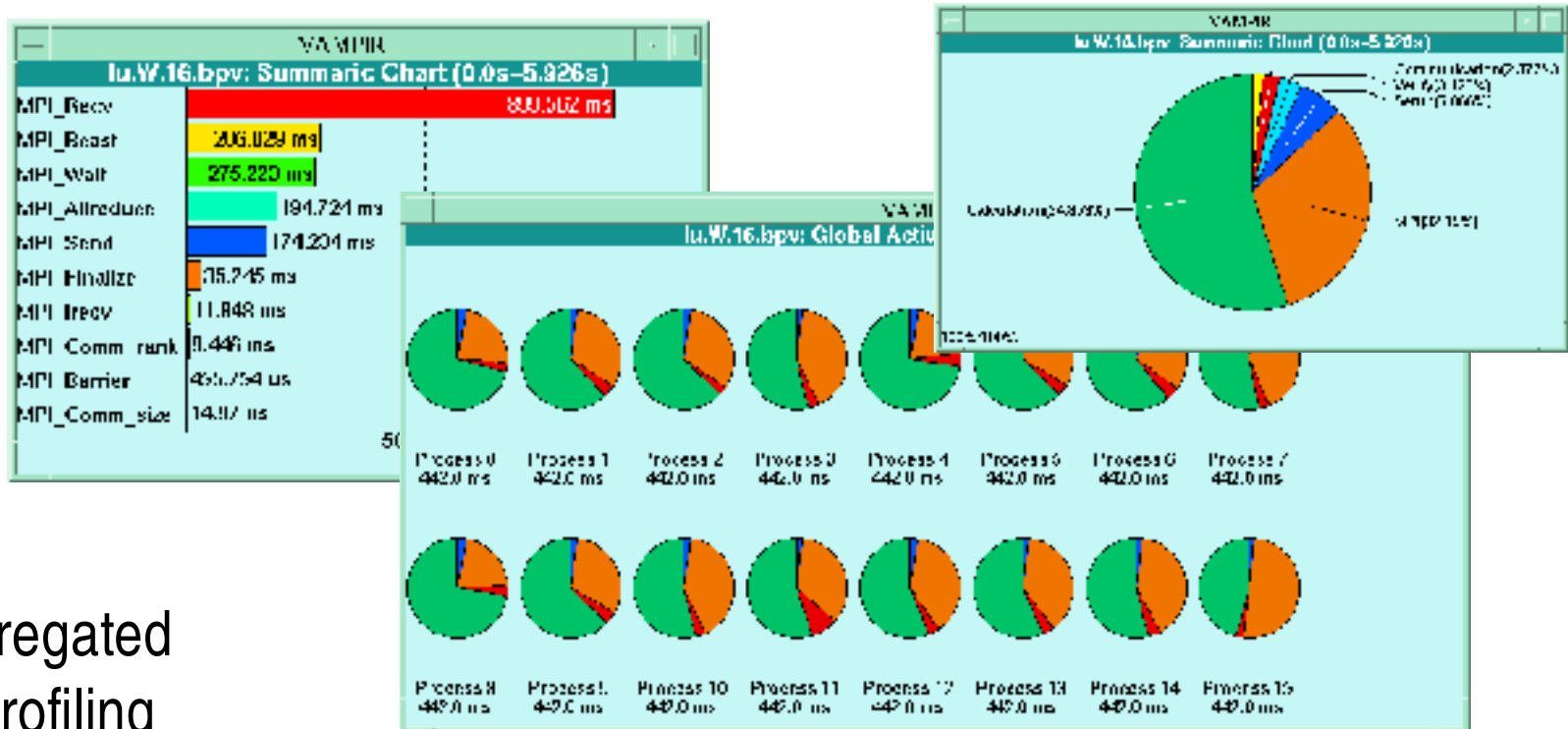


Information about states, messages, collective, and I/O operations available by clicking on the representation

Vampir: Timeline Diagram (Message Info)



Vampir: Profile Statistics Displays



Aggregated
profiling

information: execution time, # calls, inclusive/exclusive

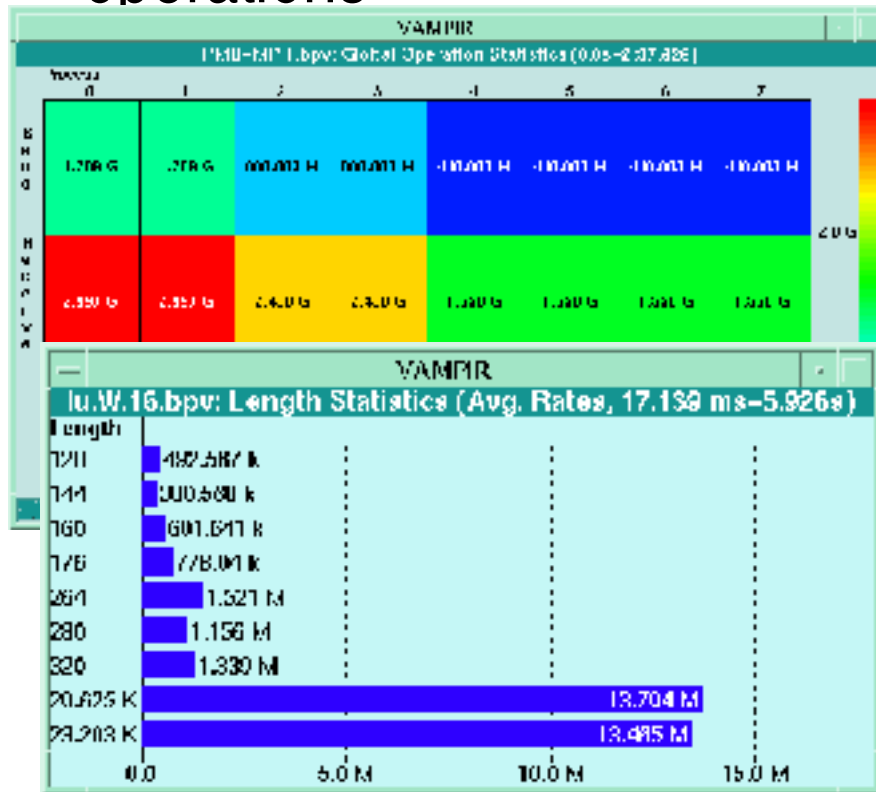
Available for all/any group (activity)

Available for all routines (symbols)

Available for any trace part (select in timeline diagram)

Vampir: Communication Statistics Displays

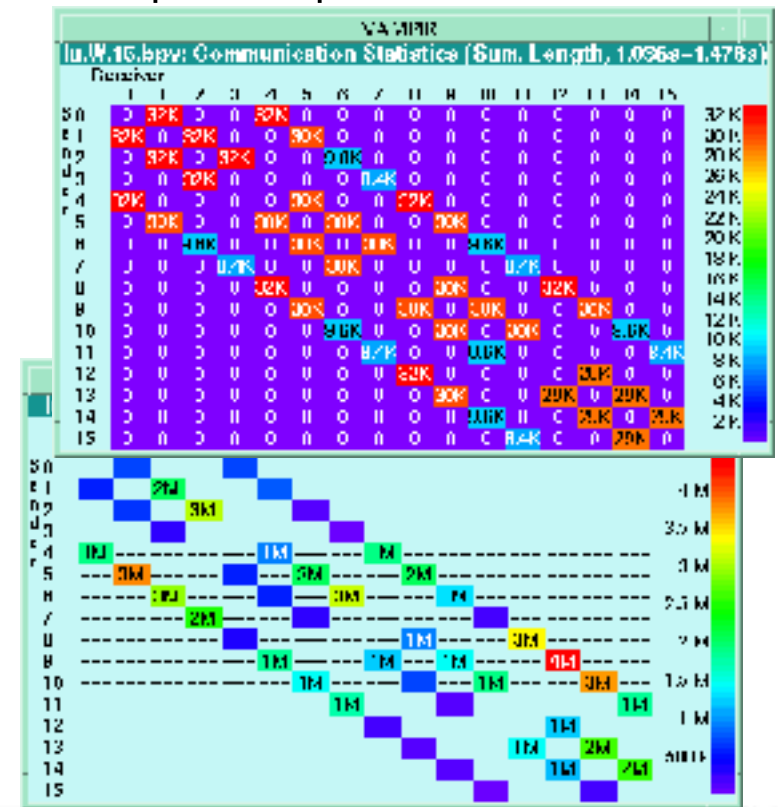
Bytes sent/received for collective operations



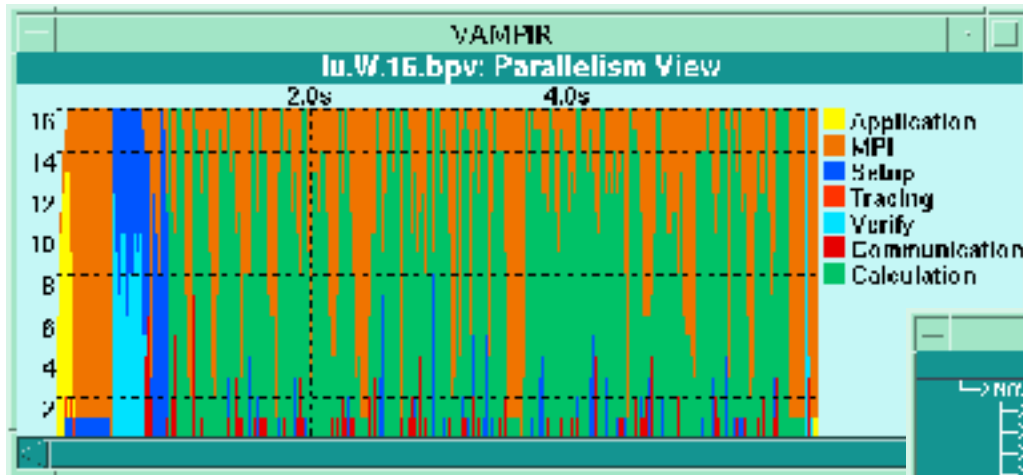
Message length statistics

Available for any trace part

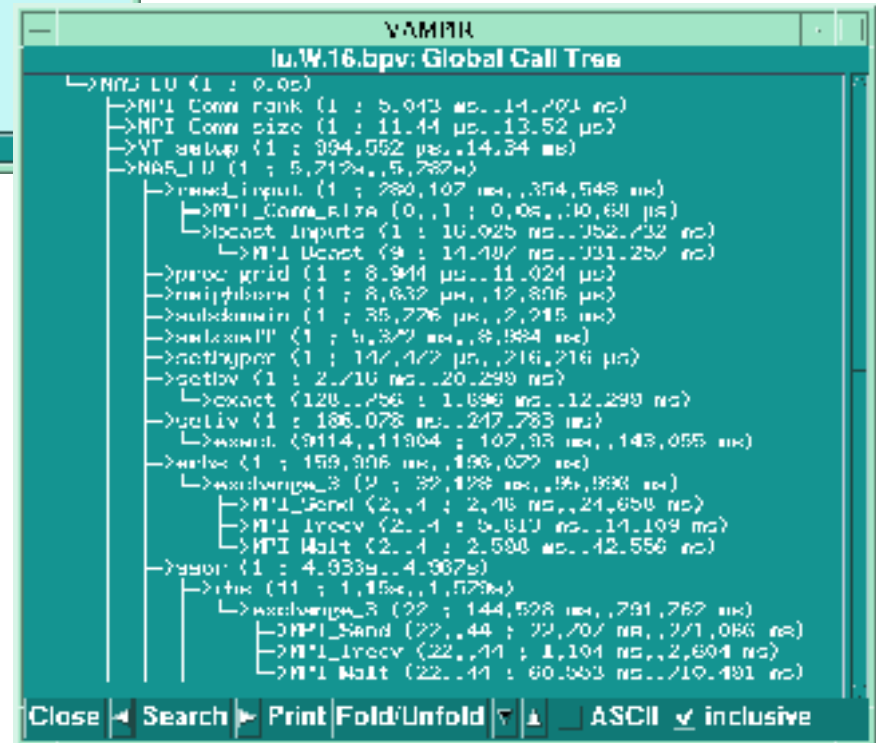
Byte and message count, min/max/avg message length and min/max/avg bandwidth for each process pair



Vampir: Other Features



Dynamic global call graph tree

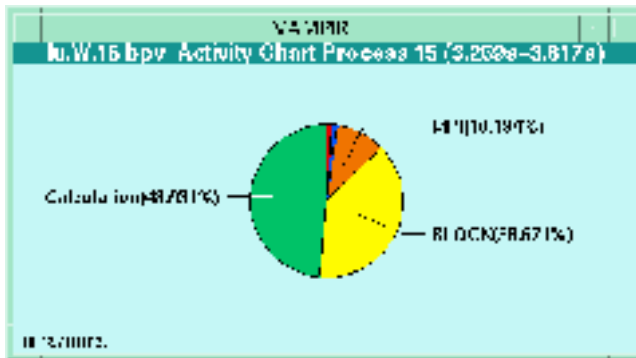


Parallelism display

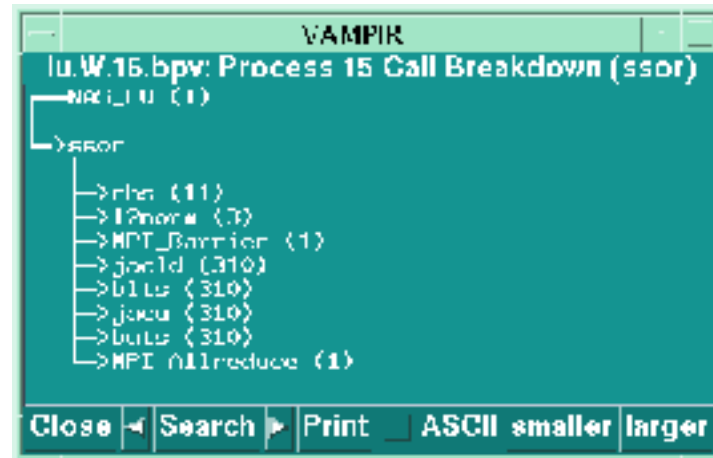
Powerful filtering and trace comparison features

All diagrams highly customizable (through context menus)

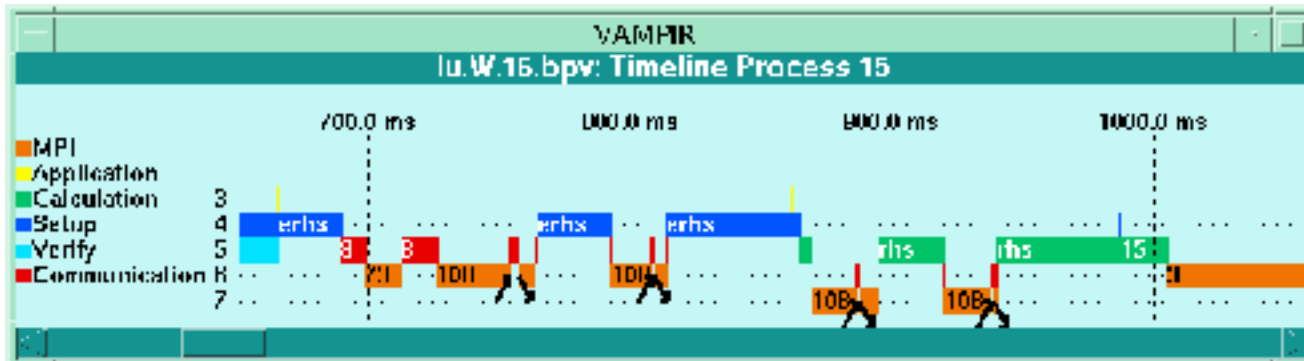
Vampir: Process Displays



Activity chart



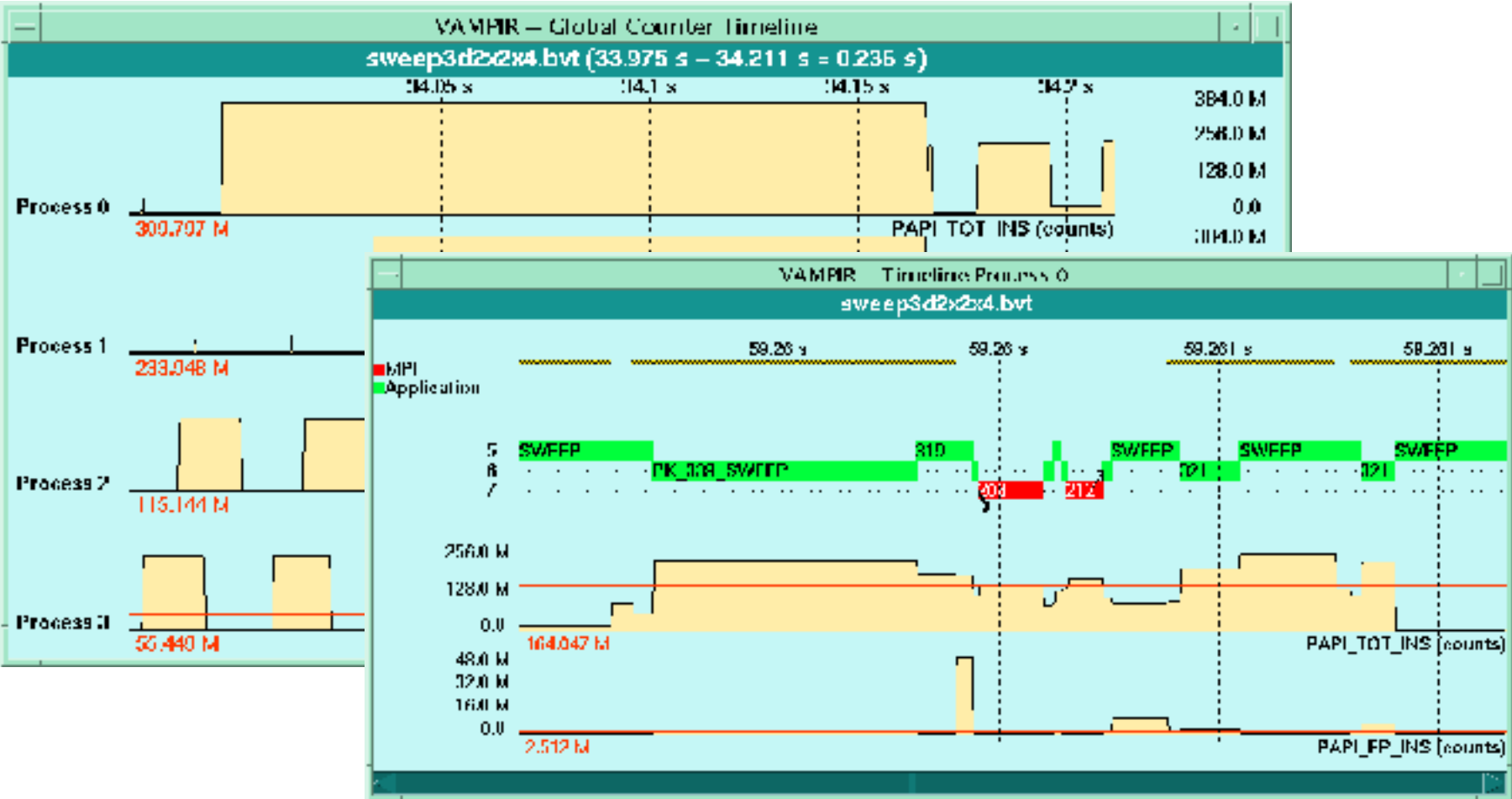
Call tree



Timeline

For all selected processes in the global displays

Vampir v3.x: Hardware Counter Data



VProf

TAU

Tuning and Analysis Utilities

<http://www.cs.uoregon.edu/research/paracomp/tau/>

Portable profiling and tracing toolkit for performance analysis of parallel programs

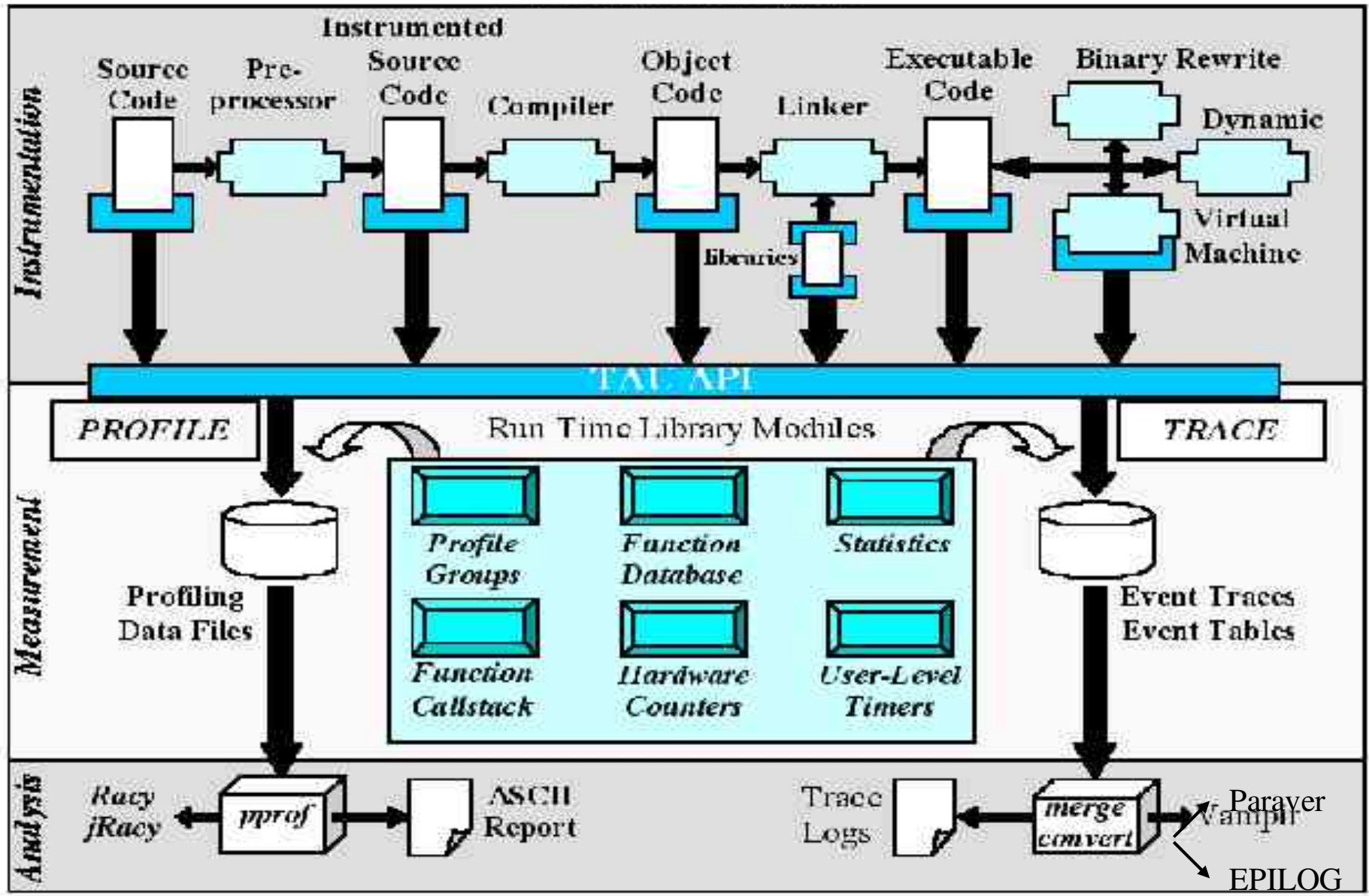
Fortran 77/90, C, C++, Java

OpenMP, Pthreads, MPI, mixed mode

In use at DOE ASCI Labs and at NCSA

Being installed and supported at DoD HPC Centers as part of PET CE project

TAU Performance System Architecture



TAU Instrumentation

Manually using TAU instrumentation API

Automatically using

- Program Database Toolkit (PDT)

- MPI profiling library

- Opari OpenMP rewriting tool

Uses PAPI to access hardware counter data

Program Database Toolkit (PDT)

Program code analysis framework for developing source-based tools

High-level interface to source code information

Integrated toolkit for source code parsing, database creation, and database query

- commercial grade front end parsers

- portable IL analyzer, database format, and access API

- open software approach for tool development

Target and integrate multiple source languages

Use in TAU to build automated performance instrumentation tools

PDT Components

Language front end

Edison Design Group (EDG): C, C++

Mutek Solutions Ltd.: F77, F90

creates an intermediate-language (IL) tree

IL Analyzer

processes the intermediate language (IL) tree

creates “program database” (PDB) formatted file

DUCTAPE (Bernd Mohr, ZAM, Germany)

C++ program Database Utilities and Conversion Tools Appl_ication
Environment

processes and merges PDB files

C++ library to access the PDB for PDT applications

PDT Status

Program Database Toolkit (Version 2.2, web download)

EDG C++ front end (Version 2.45.2)

Mutek Fortran 90 front end (Version 2.4.1)

C++ and Fortran 90 IL Analyzer

DUCTAPE library

Standard C++ system header files (KCC Version 4.0f)

PDT-constructed tools

TAU instrumentor (C/C++/F90)

Program analysis support for SILOON and CHASM

Platforms

SGI, IBM, Compaq, SUN, HP, Linux (IA32/IA64),
Apple, Windows, Cray T3E, Hitachi

OPARI: Basic Usage (f90)

Reset **OPARI** state information

```
rm -f opari.rc
```

Call **OPARI** for each input source file

```
opari file1.f90
```

```
...
```

```
opari fileN.f90
```

Generate **OPARI** runtime table, compile it with ANSI C

```
opari -table opari.tab.c
```

```
cc -c opari.tab.c
```

Compile modified files `*.mod.f90` using OpenMP

Link the resulting object files, the **OPARI** runtime table

```
opari.tab.o and the TAU POMP RTL
```

TAU Analysis

Profile analysis

pprof

parallel profiler with text-based display

racy

graphical interface to pprof (Tcl/Tk)

jracy

Java implementation of Racy

Trace analysis and visualization

Trace merging and clock adjustment (if necessary)

Trace format conversion (ALOG, SDDF, Vampir)

Vampir (Pallas) trace visualization

Paraver (CEPBA) trace visualization

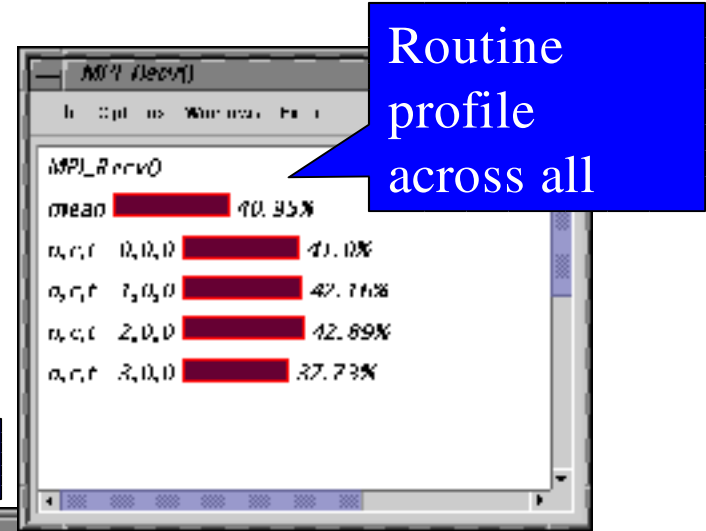
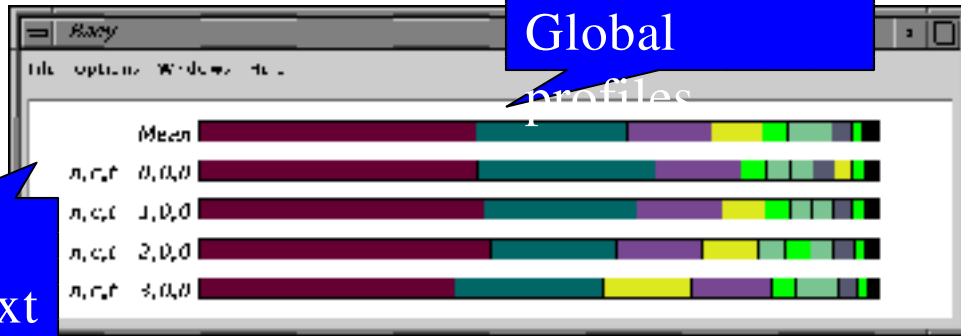
TAU Pprof Display

```

-----
mpirun@farsite01: /usr/local/mpi/bin
-----
Buffers Files Tools Edit Search Misc Help
-----
Reading Profile Files in profile.0:
NULL 0;COUNT 0; READ 0:
-----
%Time   Exclusive   Inclusive   #Call     #Subr=   Inclusive   Name
         msec       and 1/ma    -----
-----
100.0   0.000       11.203      1         12      101233260  applo
99.6    3.667       10.467      37517     37517     67487925  accept_fmpi --
67.1    291        08.322      37200     37200     3450      exchange_1
44.5    0.461       25.153      9300      10600     9157      puts
41.0    118.436     18.433      18600     0         4217      MPI_Recv()
39.5    6.778      6.407      4500      18600     6068      calls
26.2    20.142      20.142      19204     0         2611      MPI_Send()
16.2    24.452      31.031      301       602      10309E    rhs
3.0     7.501       7.501      9300      0         607      jac1c
3.2     838        6,951      601       1817     10418     worker_g12
3.4     0.590       0.590      9300      0         709      jac1c
2.6     4.080      4.080      600       0         2206      MPI_Exit()
0.2     0.174       1.000      4         4         100081    init_comm
0.2     398        398        0         28      29634    MPI_Init()
0.1     140        247        57252     47610     24700E    setiv
0.1     131        131        0         0         2      exact
0.1     89         107        0         0         103168    rhs
0.1     0.966       0.966     0         0         30450    read_input
0.0     98         98         0         0         10000    MPI_Finalize()
0.0     26         44         7037     0         44678    error
0.0     21         21         608      0         40      MPI_Init()
0.0     15         15         0         0         15030    MPI_Finalize()
0.0     4          12         1700     0         12335    setbv
0.0     7          8          0         4         2897     12nnr
0.0     1         1         0         0         491      MPI_Comm_dup()
0.0     1         1         0         0         3074     ointegr
0.0     1         1         0         0         1007     MPI_Error()
0.0     0.116      0.837     4         4         837      worker_g12
0.0     0.512      0.512     0         0         512      MPI_Keyval_create()
0.0     0.121      0.353     0         0         353      exchange_5
0.0     0.024      0.101     0         0         101      exchange_6
0.0     0.107      0.107     0         0         17      MPI_Finalize()
-----
--:-- MPD L:0:0 (Uncments) --L0--Top-----

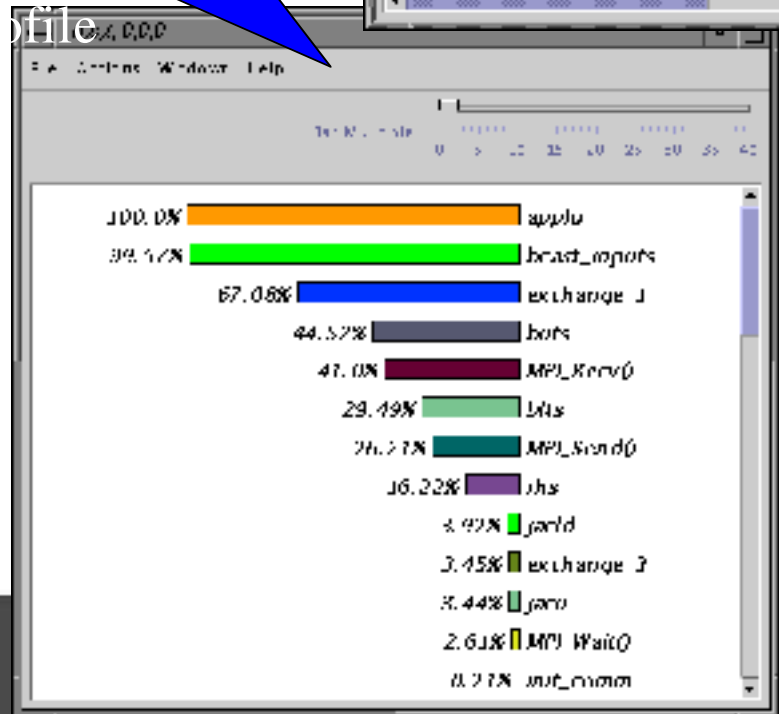
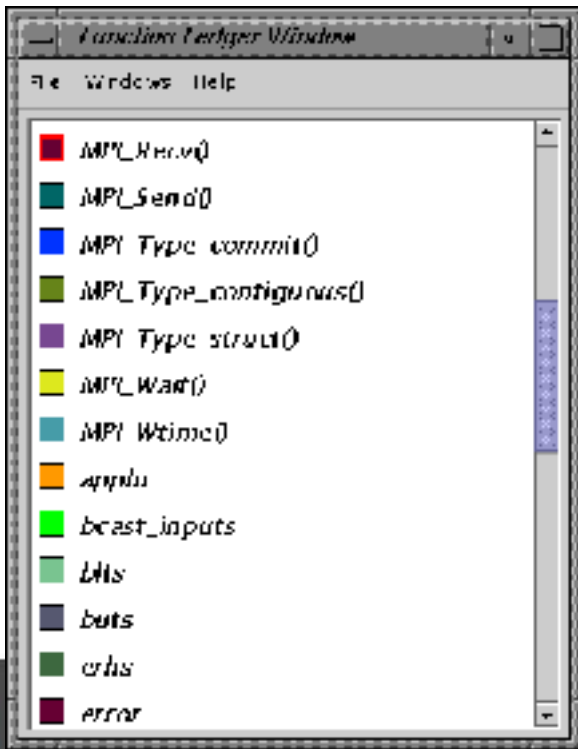
```

Proficiency (NAS Parallel Benchmark – LU)



n: node
c: context
t: thread

Individual profile



1-Level Callpath Implementation in TAU

TAU maintains a performance event (routine) callstack

Profiled routine (child) looks in callstack for parent

Previous profiled performance event is the parent

A *callpath profile structure* created first time parent calls

TAU records parent in a *callgraph map* for child

String representing 1-level callpath used as its key

“a()=>b()” : name for time spent in “b” when called by “a”

Map returns pointer to callpath profile structure

1-level callpath is profiled using this profiling data

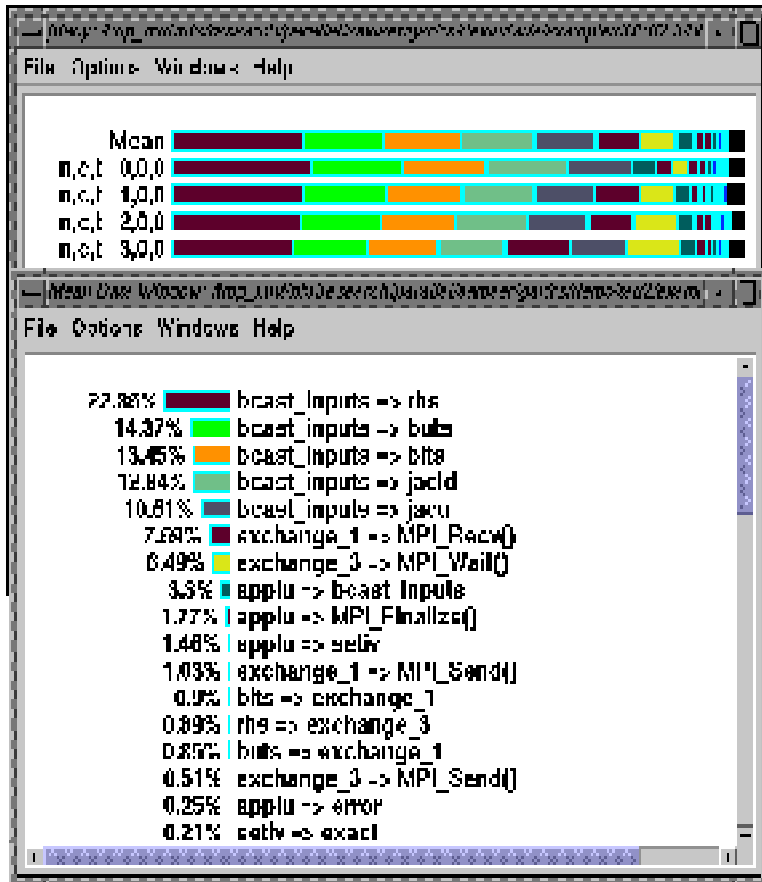
Build upon TAU’s performance mapping technology

Measurement is independent of instrumentation

Use *–PROFILECALLPATH* to configure TAU

Tracy (Callpath Profiles)

Callpath profile across all



Name	used	total used	Count	Number	used/count	zero
36.7	10,357	13,252	361	632	5812	broadcast_inputs => rhs
13.5	8,195	7,029	9303	10830	1029	broadcast_inputs => bits
13.0	7,564	11,994	9501	2800	281	bits => applu => error
12.8	7,326	7,323	9303	0	787	broadcast_inputs => jacld
10.6	6,347	6,040	9303	0	621	broadcast_inputs => jacu
7.7	4,485	4,584	14601	0	0	exchange_1 => MPI_Recv()
6.5	2,700	2,703	603	0	212	exchange_3 => MPI_Wait()
55.7	1,300	58,608	1,29	3730	436908	applu => broadcast_inputs
1.2	1,315	1,012	1	0	101219	applu => MPI_Finalize()
1.7	332	353	1	48506.5	50340	applu => setlv
1.0	488	484	14601	0	0	exchange_1 => MPI_Recv()
1.0	315	4,329	19003	18030	233	bits => exchange_1
1.0	303	4,429	602	1836	7405	rhs => exchange_3
0.9	384	1,643	14601	2800	14	bits => applu => error
0.5	292	252	603	0	164	exchange_3 => MPI_Wait()

SGI Performance Tools

ProDev workshop (formerly CASE Vision) :

`cvd, cvperf, cvstatic`

ProMP: parallel analyzer: `cvpav`

SpeedShop: profiling execution and reporting
profile data

Perfex: per process event count statistics

dprof: address space profiling

Perfex

Provides event statistics at the process level

Reports the number of occurrences of the events captured by the R1X000 hardware counters in each process of a parallel program

In addition, reports information derived from the event counts, e.g. MFLOPS, memory bandwidth

Perfex Usage

```
perfex -mp [other options] a.out
```

To count secondary cache misses in the data cache (event 26) and instruction cache (event 10):

```
perfex -mp -e 26 -e 10 a.out
```

To multiplex all 32 events (-a) , get time estimates (-y) and trace exceptions (-x)

```
perfex -a -x -y a.out
```

Type “man perfex” for more information

Speedshop

Speedshop is a set of tools that support profiling at the function and source line level.

Uses several methods for collecting information

- PC and call stack sampling
- basic block counting
- exception tracing

Speedshop (cont.)

Data Collection

- `ssrun` main data collection tool. Running it on `a.out` creates the files `a.out.experiment.mPID` and `a.out.experiment.pPID`
- `ssusage` summary of resources used, similar to the `time` commands
- `ssapi` API for caliper points

Data Analysis

- `prof`

ssrun Sampling Experiments

Statistical sampling, triggered by a preset time base or by overflow of hardware counters

- `pcsamp` PC sampling gives user CPU time
- `usertime` call stack sampling, gives user and system CPU time
- `totaltime` call stack sampling, gives walltime

ssrun Sampling Options

Sampling triggered by overflow of R1X000 hardware counters

- gi_hwc Graduated instructions
- gfp_hwc Floating point instructions
- ic_hwc Misses in L1 I-cache
- dc_hwc Misses in L1 D-cache
- dsc_hwc Data misses in L2 cache
- tlb_hwc TLB misses
- prof_hwc User selected event

ssrun Usage

Select a hardware counter, say secondary cache misses (26), and an overflow value

```
setenv _SPEEDSHOP_HWC_COUNTER_NUMBER 26
```

```
setenv _SPEEDSHOP_HWC_COUNTER_OVERFLOW 99
```

Run the experiment

```
ssrun -prof_hwc a.out
```

Default counter is L1 I-cache misses (9) and default overflow is 2,053

ssrun Ideal and Tracing Experiments

Ideal Experiment: basic block counting

`-ideal` counts the number of times each basic block is executed and estimates the time.

Tracing

- `-fpe` floating point exceptions
- `-io` file open, read, write, close
- `-heap` malloc and free

prof

Display event counts or time in routines sorted in descending order of the counts

Source line granularity with command line option `-h` or `-l`

For ideal and usertime experiments get call hierarchy with `-butterfly` option

For ideal experiment can get architecture information with the `-archinfo` option

Cut off report at top 100-p% with `-quit p%`

Address Space Profiling: dprof

Gives per process histograms of page accesses

Sampling with a specified time base

- the current instruction is interrupted
- the address of the operand referenced by the interrupted instruction is recorded

Time base is either the interval timer or an R1X000 hardware counter overflow

man dprof

R1X000 counters: man r10k_counters

dprof Usage

Syntax

```
dprof [-hwpc [-cntr n] [-ovfl m]]  
[-itimer [-ms t]] [-out profile_file]  
a.out
```

Default is interval timer (`-itimer`) with *t=100 ms*

Can select hardware counter (`-hwpc`) which has the defaults

n = 0 is the R1X000 cycle counter

m=10000 is the counter's overflow value

Performance Results

(separate presentation by
Patrick Worley, ORNL)