Linux Multicore Performance Analysis and Optimization in a Nutshell

Philip Mucci
mucci at eecs.utk.edu

NOTUR 2009
Trondheim, Norway

Schedule

- 10:15 Begin
- (1:00)
- 11:15 – 11:30 Coffee
- (1:30)
- 13:00 – 14:00 Lunch
- (2:00)
- 16:00 Finish
- 18:00 Beer

Outline

- Commentary
- HW/SW Overview
- Working With the Compiler
- Single Core Optimization
- Programming Models
- Multicore Optimization
- Performance Analysis

Initial Commentary

Optimization is an Art Form

- As such, this tutorial is partially subjective.
- Expect some contradictions to your experience(s).
  - Don't panic.
- Negative comments (on the tutorial) are welcomed (afterwards).
  - Please include “bug fixes”.
  - mucci at eecs.utk.edu

What's so special about Multicore?

- Parallel programming is somewhat easier...
  - Shared address space means direct access to data.
- Multicore is a great latency hiding mechanism.
  - Most computers are doing many different activities at once. (What about us?)
- We *really* can't get much faster without liquid cooling.
- Multicore appears to lower power and cooling requirements per FLOP.

What's so hard about Multicore?

- For 30+ years, we've been optimizing for cache.
  - Compilers are still limited by static analysis
  - Most developers technical computing are not (supposed to be) computer architects
  - Languages have further abstracted performance
  - DRAM ignores Moore's Law
  - Memory controllers are neither bigger or smarter
- But it's ″easy″ to put multiple cores on a die!

Evaluation of Multicore

- Lots of high GF cores with many shared resources means *more work for you*.
- Resource constraints must be examined system wide, with attention to per-core performance.
  - Size/speed of dedicated cache/TLB
  - Memory bandwidth and latency per-core
  - On/Off-chip communications per-core
    - PCI, I/O, Interprocessor, Interconnect

Multicore Performance

- Cores generally don't ″know″ about each other
  - They communicate only through cache and memory.
  - No dedicated instructions to do sync, comm, dispatch, must be done by (slow) software.
- External bandwidth limited by pins and power.
- Starvation is a very real issue at every shared resource. Some extreme examples:
  - Intel's Hyper Threading
  - IBM's POWER4 Turbo vs. HPC

Architecture Overview

Multicore Architecture Overview

- Hardware
  - Caches, Coherency and Prefetching
  - Translation Lookaside Buffers (TLB)
  - Hardware Multithreadeding (SMT/HT)
- Software
  - Threads vs. Processes

Multicore and Memory Bandwidth

- Biggest bottleneck is memory bandwidth and memory latency.
  - Multicore has made this (much) worse in order to claim increased peak performance.
- At least 3 major approaches:
  - Make cores as fast/slow as main memory (SiCortex, Tensilica)
  - Add faster/closer memory pipes (Opteron, Nehalem)
  - Streaming compute engines (NVIDIA,AMD), vectorized memory pipelines (Convey).

Multicore, SMP and NUMA

- Single socket multicores are SMP's.
  - Cost of memory access is uniform to every core
  - Less work for programmer, OS, etc.
  - Not possible to scale (well)
    - Crossbar works, but ultimately you have to slow nearest neighbors down.
- NUMA – Non Uniform Memory Access
  - All memory is not the same.
  - Problem: Memory can be “far” from the CPU

Caches

- Small high-speed memories to keep data ″close″ to the processor.
  - Memory is moved in and out of caches in blocks called ″lines″, usually 32 to 256 bytes.
- Multiple levels of cache, with at least one level being dedicated to a single core. i.e.
  - 32K Level 1 -> 1 core
  - 1MB Level 2 -> 2 cores, 1 die
  - 8MB Level 3 -> 4 cores, 1 package

Caches Exploit Locality

- Spatial – If I look at address M(n), it is likely that **M(n ± z) will be used**, where z is small.
- Temporal – If I look at address M(n) at time t, it is likely that **M(n) will be used again at time t + t'**, where t' is small.
- If true for one core, for us (technical-computing) true for multicore.
  - So how do we still make caches Correct (consistent/coherent) and Effective (fast)?

Cache Architecture

- Memory cannot live anywhere in a cache.
- Cache associativity – The number of unique places in a cache where any given memory item can reside.
  - Location is determined by some bits in the physical or virtual address..
  - Direct mapped means **only one** location.
    - But very, very fast.
  - Higher associativity is better, but costly in terms of gates and complexity (power and performance).

Why do we care?

- Tuning for cache yields **most** of your performance gain.
- On **multicore** true, but opportunity for creating **contention**.
  - It can happen: A cache unfriendly code may run **faster** than the same code highly tuned without thought to contention.
    - Data layout and algorithm design.
- And you thought multicore was free performance...

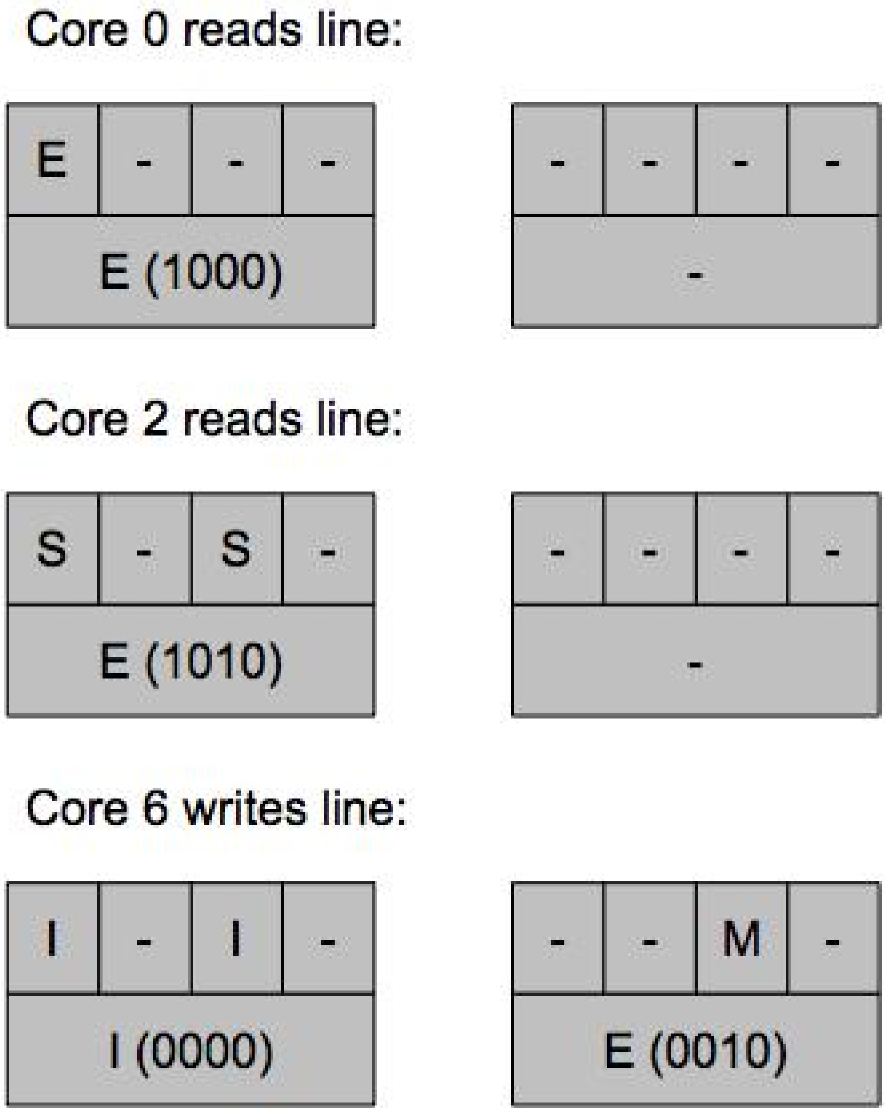
Cache Consistency

- For correctness, cores must see a consistent view of memory through the caches.
- Thus the caches communicate with a protocol that indicates the state of each cache line. Most common is MESI.
  - M – modified, E – exclusive
  - S – shared, I – invalid
- Method of communication may be different. (Snoop, directory, broadcast etc...)

Coherency and Cache Inclusivity/Exclusivity

- Most caches are inclusive.
  - Data kept in multiple levels at the same time
- With multiple cores, more than one level can keep MESI states.
  - In Nehalem, L3 keeps state per socket, L1 and L2 per core
- Transitions to E, I, S are often performance hits.
  - But they can be identified with the right tools and insight.

Coherency Example (2Sx4C)



Core 0 reads line:

E - - - | - - - -

E (1000) | -

Core 2 reads line:

S - S - | - - - -

E (1010) | -

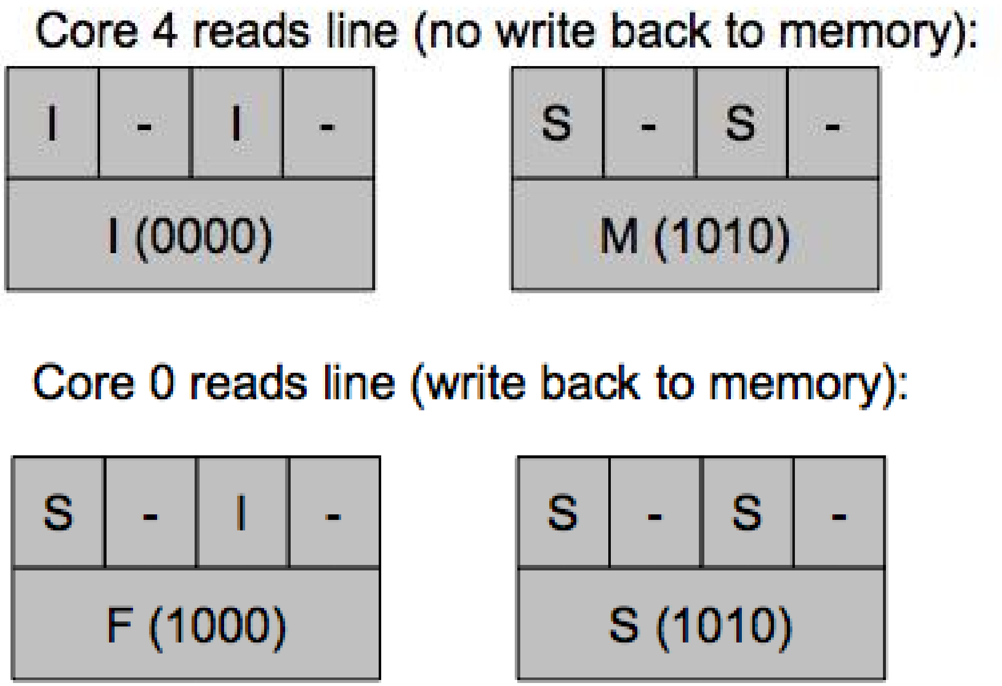
Core 6 writes line:

I - I - | - - M -

I (0000) | E (0010)

Courtesy Daniel Molka of T.U. Dresden

Coherency Example part 2



Core 4 reads line (no write back to memory):

I	-	I	-
I (0000)			

S	-	S	-
M (1010)			

Core 0 reads line (write back to memory):

S	-	I	-
F (1000)			

S	-	S	-
S (1010)			

Courtesy Daniel Molka of T.U. Dresden

Hardware Prefetching

- Automatic
  - Streams of reference automatically predicted by the hardware.
  - N consecutive misses trigger fetch of lines ahead.
  - Usually unit line stride and one or two streams.
    - On/Off can sometimes be controlled by the BIOS.
- Prefetch Instructions
  - Many variants, separate from the above, but may trigger it. Can even be invalid addresses.

Software Prefetching

- Compiler usually sprinkles in these instructions
  - To do a good job, it needs to predict misses or find slots to hide them.
- Pragmas and Intrinsics
  - High level manual placement
- Explicit
  - Programmer puts in assembly primitives

TLB

- Memory is divided up into **pages**.
  - Pages can be variable size. (4K,64K,4M)
- A page of logical (virtual) address space can have a physical address.
  - Computing this address is expensive!
- So we keep a cache of them around: the TLB.
  - It is usually fully associative and multi-level.
  - A TLB miss can be very expensive.

Memory Pages and the TLB

- Each TLB entry covers one page.
  - Big pages are good for lowering TLB misses.
  - But the OS moves data in pages!
- A miss in all levels of the TLB is also called a **page fault.**
  - If the data is in physical memory, then this is a **minor** page fault.
  - If the data is on disk, this is a **major** page fault.

SMT and Hyperthreading

- Simultaneous Multithreading
  - Hyperthreading is Intel's name for it
  - Share physical resources on a chip among multiple hardware threads except context (PC and regs)
  - Goal is to attempt to hide latency of instructions.
  - When one instruction stalls, try another thread, OS does not need to context switch.
  - To the OS, it's an SMP
- This really only works when you've got a rich and diverse instruction mix among threads.

Vector Instructions

- Instructions that operate on more than one operand.
  - More accurately called micro-vector instructions
  - Real vector machines do this on 1000's of items.
- Intel's SSE 2,3,4 are examples
  - A register contains a number of Byte, Int, Float, etc...
- Hardware is free to schedule and move data in larger chunks.
  - Restrictions on alignment, accuracy, etc...

Threads and Processes

- Operating systems support threads and processes.
  - A thread is an **execution context**; machine state scheduled by the operating system.
  - A process is a thread plus virtual memory, files, etc.
    - A process can contain multiple threads.
    - Each thread in a process shares everything except state (stack, registers and program counter)
- Both are managed by the operating system.

OS Scheduling and Threads

- On Linux, threads are free to bounce around.
  - Other threads can steal the CPU as can OS work like hard and soft interrupts.
  - OS's do this to provide "fairness".
  - Linux does understand a cost penalty when moving a thread from one core to another.
  - Rescheduling a new thread often involves a cache and TLB flushing.
- For technical computing (often SPMD), this is bad for performance.

OS and System Calls

- System calls are function calls that ask the OS to do something.
  - Going to the OS (crossing from user to kernel domain) is slow.
    - Argument checking
    - Data copies
    - Rescheduling points
- Function calls are cheap, just register bits.
- Many system calls contain locks, are serialized or are not scalable.

Architecture Summary

- To tune, you need to have *some* background.
  - Single core performance comes first!
- With multicore, we will tune:
  - To use all of the cache
  - To avoid cache conflicts
  - To minimize shared resource contention
    - Memory bandwidth, OS, I/O, Comm
  - Minimize NUMA effects

Working With the Compiler

Optimizing with the Compiler

- It can't read your mind, only your code.
- Correctness is always emphasized over performance.
- For *popular*" and "*simple*" constructs, the compiler will usually do a better job than you.
- But as code gets more abstract, it can't guess the things that matter!
  - Loop lengths, alignment, cache misses, etc...

Understanding Compilers

- The **best** things you can do to work with the compiler are:
  - Learn a compiler well and **stick with it.**
  - **Clearly** express your intentions to the compiler through:
    - Well structured code
    - Compiler directives
    - Compile time options
      - Extensive array to control different behaviors.

Correctness and Compilers

- We often talk about getting *"correct"* answers.
  - IEEE has a standard for correctness (IEEE754)
  - Applications relax that standard for performance and because correct is somewhat arbitrary.
- Consider the following:

```
sum = 0.0
do i = 1, n
  sum = sum + a(i)
enddo
```

```
sum1 = 0.0
sum2 = 0.0
do i = 1, n-1, 2
  sum1 = sum1 + a(i)
  sum2 = sum2 + a(i+1)
enddo
sum = sum1+sum2
```

Inlining

- Replacing a subroutine call with the code from the original function.
- Good because:
  - Function calls inside loops (often) inhibit vectorization.
  - Function calls are not free, they take cycles and cycles to set up and tear down.
- Has potential to bloat code and stack.

Vectorization

- Generate code that takes advantage of *vector* instructions.
  - Helped by inlining, unrolling, fusion, SWP, IPA, etc.
- The entire motivation behind using accelerators
  - GPGPUs and FPGAs
- x86, PPC, MIPS all have variants of vector instructions:
  - SSE, AltiVec, etc...

IPO/IPA

- Interprocedural Optimization/Analysis
  - Compiler can move, optimize, restructure and delete code between procedures and files.
- Generates intermediate code at compile time.
- Generates object code during final link.
  - As with SWP, exposes more opportunities to optimization passes.
- Stronger typing of pointers, arguments and data structures can vastly increase effectiveness.

Software Pipelining

- Consider more than one iteration of a loop.
  - Keep more intermediate results in registers and cache.
- To use it, the compiler must predict:
  - Loop count
  - Inter-iteration dependencies
  - Aliasing
- Optimization can be a trade off.
  - Loop set up and tear down can be costly.

Pointer Aliasing

- The most efficient optimization is **deletion**.
  - Especially loads and stores!
- Compilers must assume that memory (by pointers) has changed or overlaps.
  - Unless you help it to conclude otherwise.
- This is called the **pointer aliasing problem**. It is *really* bad in C and C++.
  - Can be controlled on command line and through keywords.

Types of Aliasing

- Strict
  - Pointers don't alias if they are different types.
- Typed
  - Pointers of the same type can alias and overlap.
- Restricted
  - Pointers of same type are assumed to not overlap.
- Disjointed
  - All pointer expressions result in no overlap.

Profile Directed Feedback

- a.k.a Feedback Directed Optimization
- Collect data about what the code really does and then adapt.
  - Old idea, but (still) not very well developed.
- Important for:
  - Branches (I-cache/ITLB misses, BP misprediction)
  - Loop bounds (unroll, SWP, jam, etc)
- Future will be to make most decisions based on real data.

Compiler Flags

- All compilers support the **`-O(n)`** flag.
  - This flag actually turns on lots of other optimizations.
- Better to start at **`-O(big)`** and disable optimizations rather than other way around.
  - Develop your knowledge of what to turn off.
  - Compiler documentation is usually clear about which ***n*** can result in wrong answers.

GNU Compiler Flags

- **`-O3 -ffast-math -funroll-all-loops -msse3 -fomit-frame-pointer -march=native -mtune=native`**
  - **`-Q --help=optimizers`**
- Sometimes you need **`-fno-strict-aliasing`** to get correct results.
  - **`-O2`** and higher assume strict aliasing.
- Feedback directed optimization:
  - First time use **`-fprofile-generate`**
  - Subsequent times use **`-fprofile-use`**

PathScale Compiler Flags

- **`-Ofast`** is equivalent to:
  - **`-O3 -ipa -OPT:Ofast -ffast-math -fno-math-errno -fomit-frame-pointer`**
- Takes most of the same flags as GCC.
- To find out what the compiler is doing:
  - **`-LNO:vintr_verbose=1`**
  - **`-LNO:simd_verbose=1`**
- Feedback directed optimization:
  - First time use **`-fb_create fbdata`**
  - Subsequent times use **`-fb_opt fbdata`**

Intel Compiler Flags

- **`-fast`** equals **`-O3 -ipo -xT -static -no-prec-div`**
  - **`-ip`** is subset of **`-ipo`** for single files
  - **`-shared-intel`** to allow tools to work
- To find out what the compiler is doing:
  - **`-opt-report [0123], -opt-report-file f`**
- Feedback directed optimization
  - First time use **`-prof-gen`**
  - Subsequent times use **`-prof-use`**

Intel Compiler Directives

- C (**`#pragma`**) or Fortran (**!DEC$**)
- Prefetching
  - **`[no]prefetch var1[,var2]`**
  - **`GCC: __builtin_prefetch()`**
- Software Pipelining (of Loop)
  - **`[no]swp`**

Intel Compiler Directives

- Loop Count
  - `loop count(n)`
- No Loop Interdepedencies (w/SWP)
  - `ivdep`
- Loop Unroll
  - `[no]unroll(n)`
- Loop Split
  - `distribute point`

Limiting Aliasing

- **`restrict`** keyword
  - Part of the C99 standard (**`-std=c99`** with GCC)
  - A pointer refers to unique memory.
    - Writes through this pointer will not affect anyone else.
  - Allows very good optimization!
- **`-fstrict-aliasing`** allows aliasing only for pointers of the same type.
  - For GCC and many compilers, auto when >= **`-O2`**

Aligning Data

- Specifying alignment eliminates manual padding.
- Intel says:
- **Align** 8-bit data at any address.
- **Align** 16-bit data to be contained within an aligned four-byte word.
- **Align** 32-bit data so that its base address is a multiple of four.
- **Align** 64-bit data so that its base address is a multiple of eight.
- **Align** 80-bit data so that its base address is a multiple of sixteen.
- **Align** 128-bit data so that its base address is a multiple of sixteen.

```
/* Intel, align to 16 bytes */
__declspec(align(16)) unsigned long lock;
/* GCC */
unsigned long lock __attribute__ ((aligned(16)));
```

Other Important C/C++ Keywords

- **`static`**
  - In global scope, used only in this file.
- **`const`**
  - Data or location never changes.
- **`volatile`**
  - Data may change from an alias outside of scope.
- **`inline`**
  - Inline all the time.

Serial Code Optimization

*“The single most important impediment to good parallel performance is **still** single-node performance”*

William Gropp, Argonne National Lab.

s/parallel/multicore; s/node/core;

Guidelines for Performance

- Cache gets you all of your performance.
- Compilers like to optimize loops without.
  - Function calls
  - Side effects
  - Pointers that can overlap
  - Dependencies
- Function calls are not free
- System calls are slower
- I/O is even worse

Loop and Array Optimizations

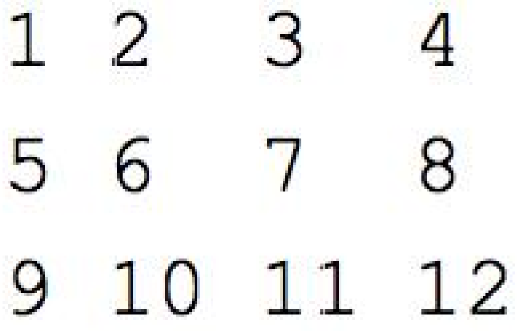
- Allocation
- Unit Stride Reference
- Initialization
- Padding
- Packing
- Stride Minimization
- Blocking
- Unrolling
- Fusion
- Defactorization
- Peeling
- Collapse
- Floating IF's
- Indirect Addressing
- Gather/Scatter

Code Examples

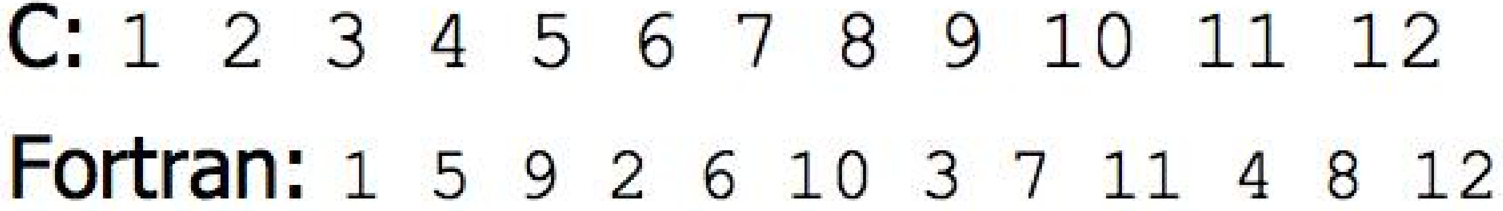
- All of the examples that follow are contrived.
  - Compilers can optimize them very well.
- In production codes, these patterns are harder to spot.
  - And thus poorly optimized.
- Write the simplest code first, make sure it's correct.
  - Debugging a highly optimized loop is terrible work.

Array Allocation

- As we know, arrays are allocated differently in C and Fortran.



1	2	3	4
5	6	7	8
9	10	11	12



C: 1 2 3 4 5 6 7 8 9 10 11 12

Fortran: 1 5 9 2 6 10 3 7 11 4 8 12

Unit Stride Access

- Unit stride is always best.
  - Small stride (< line size) is also ok.
- When data comes in, think about using as much of it as possible as soon as possible.
- When touching large amounts of memory, TLB misses faults can be a concern.

Array Referencing

- In C, outer most index should move the fastest.

[x,**Y**]

- In Fortran, inner-most should change the fastest.

(**X**,y)

Array Initialization

- No one really uses formal static initialization anymore. Waste space, restricts program, etc.
  - But static bounds were great for optimizers.
- C and Fortran now dialects allow:
  - Dynamic array allocation on the stack.
  - Run time specification of array bounds.
- Opinions vary on this.
  - Simpler and more expressive the code, the better.
  - Array addressing can waste a lot of cycles.

Array Padding

- Memory often needs to be padded to avoid cache line conflicts.
  - Fortran common block is a contiguous region of memory.
  - Lots of codes just use powers of 2. Yikes!
- Same can easily be true of dynamically allocated memory.
- Some elements on systems love aligned data.
  - I/O, Infiniband
  - But caches on multicore do not!.

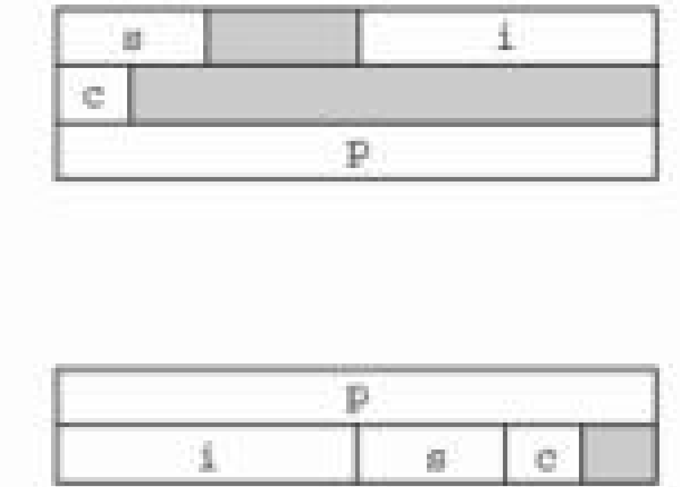
Intra-Array Padding

- Same problem can happen when accessing a single array.
  - Consider striding across each dimension as in a transpose.
- This can be avoided by allocating extra space.
  - C: Avoid leading dimension of power of 2
  - Fortran: Avoid trailing dimension of power of 2.
- As with previous item, depends on associativity of the cache.

Structure Packing

- Unaligned access to data is usually slower.
- So align items on word, double-word or bigger.
- Pack from smallest to largest, maybe add padding?
- But this is a multicore problem! (more later)

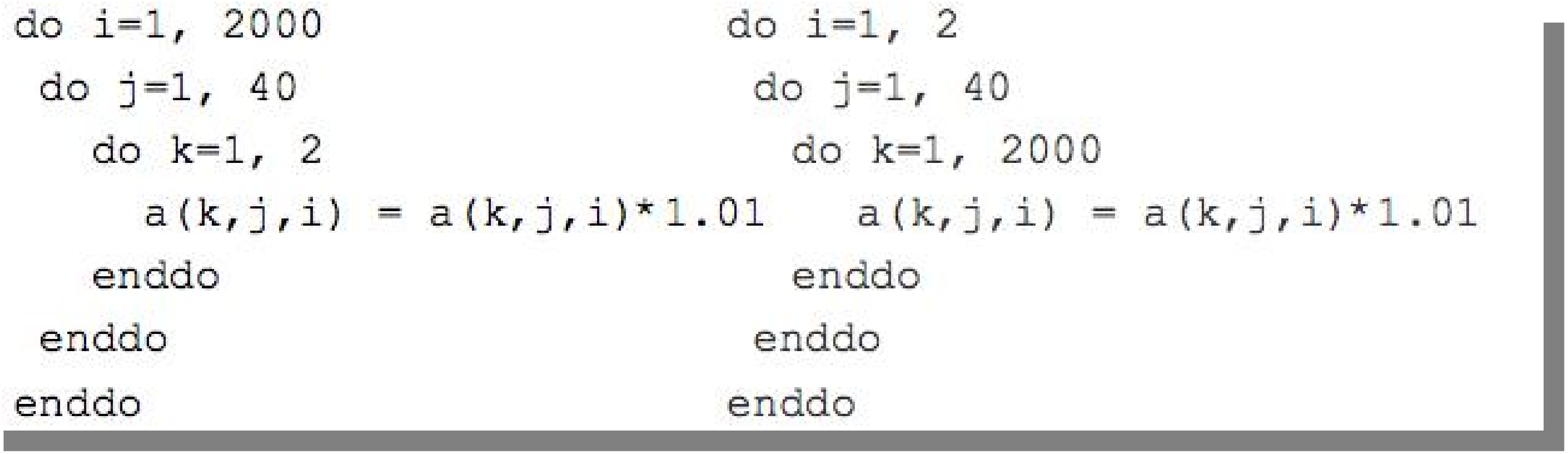
```
struct {
  short s;
  int i;
  char c;
  void *p;
}
```



```
struct {
  void *p;
  int i;
  short s;
  char c;
 }
```

Stride Minimization
Loop Interchange

- Always think about spatial and temporal locality.
- Often, this is just an oversight of the original implementor.
- With simple loops, compiler will interchange them for you.

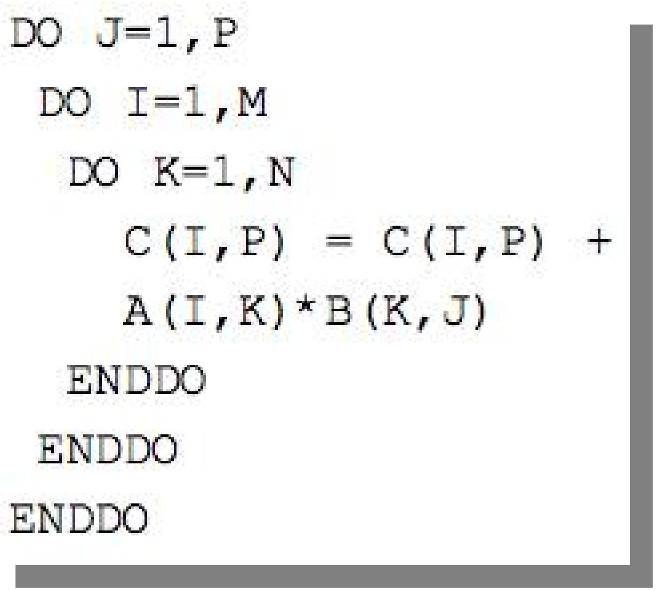


```
do i=1, 2000                      do i=1, 2
 do j=1, 40                        do j=1, 40
   do k=1, 2                         do k=1, 2000
     a(k,j,i) = a(k,j,i)*1.01          a(k,j,i) = a(k,j,i)*1.01
   enddo                             enddo
 enddo                             enddo
enddo                             enddo
```

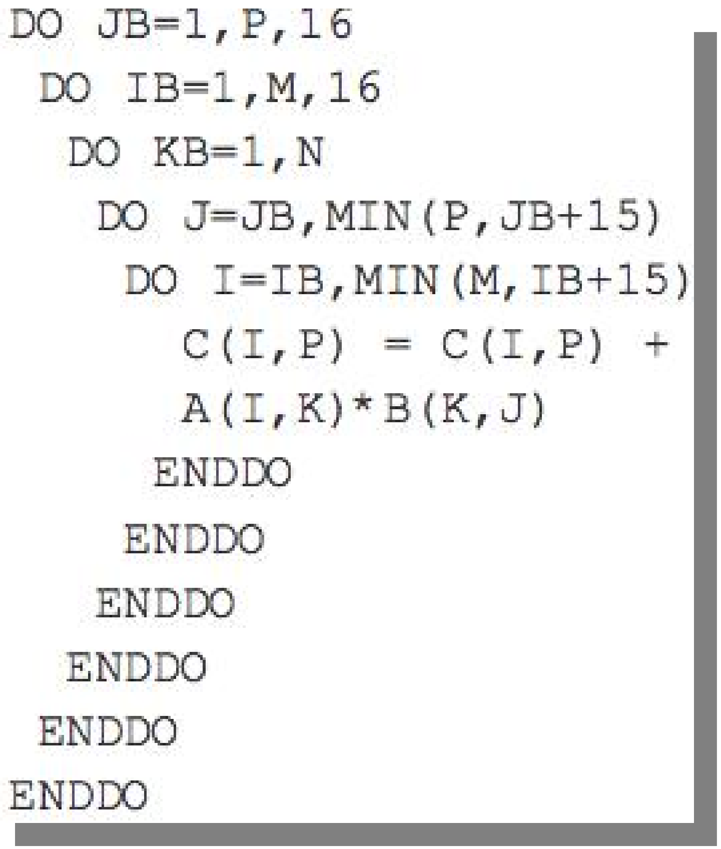
Cache Blocking

- Standard transformation
  - Most compilers are decent at it, if the loop is simple and has no subroutine calls or side-effects
- Goal is to reduce memory pressure by making use of the caches.
  - Helps when potential for re-use is high.
  - Naturally blends with sum reduction and unrolling.
- Good for multicore too but, some caches are shared! And which loop should we parallelize?

Cache Blocking



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

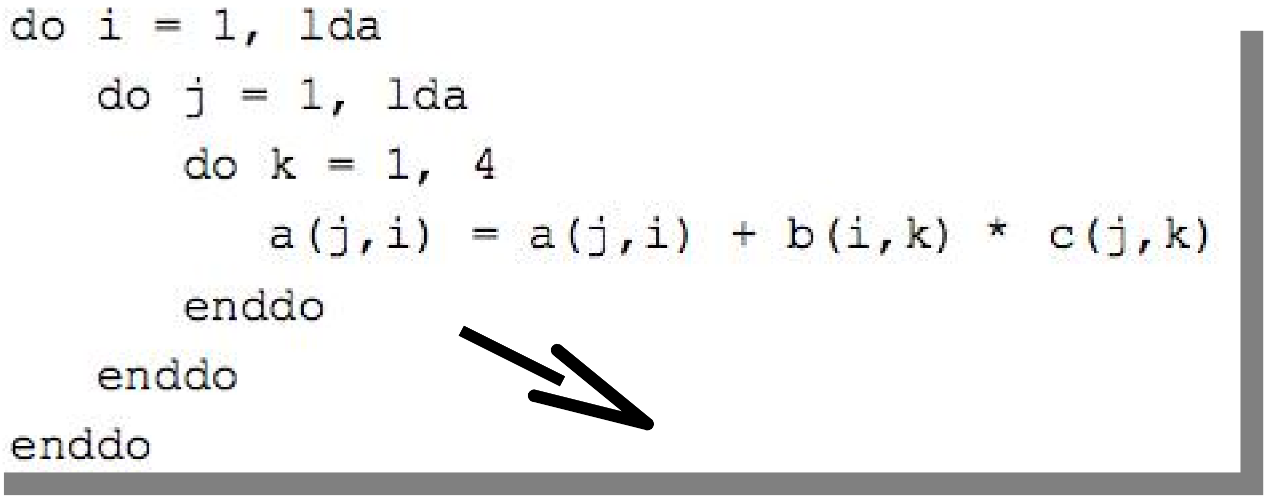


```
DO JB=1,P,16
 DO IB=1,M,16
  DO KB=1,N
   DO J=JB,MIN(P,JB+15)
    DO I=IB,MIN(M,IB+15)
      C(I,P) = C(I,P) +
      A(I,K)*B(K,J)
     ENDDO
    ENDDO
   ENDDO
  ENDDO
 ENDDO
ENDDO
```

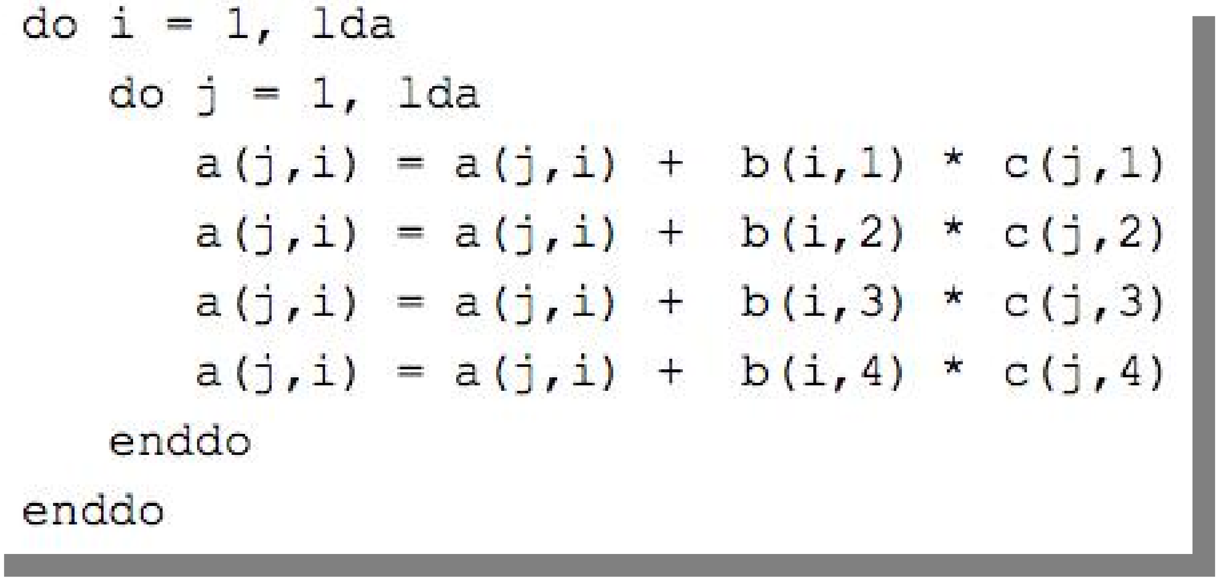
Loop Unrolling

- Standard transformation to improve processor pipeline utilitization and reduce loop overhead.
  - More work per iteration
- Compilers are very good except when
  - Function calls inside
  - Inter-iteration dependencies
  - Global variables
  - Pointer aliasing

Loop Unrolling



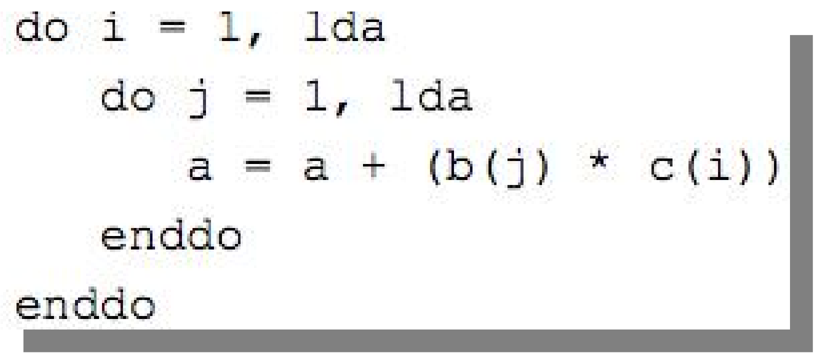
```
do i = 1, lda
   do j = 1, lda
      do k = 1, 4
         a(j,i) = a(j,i) + b(i,k) * c(j,k)
      enddo
   enddo
enddo
```



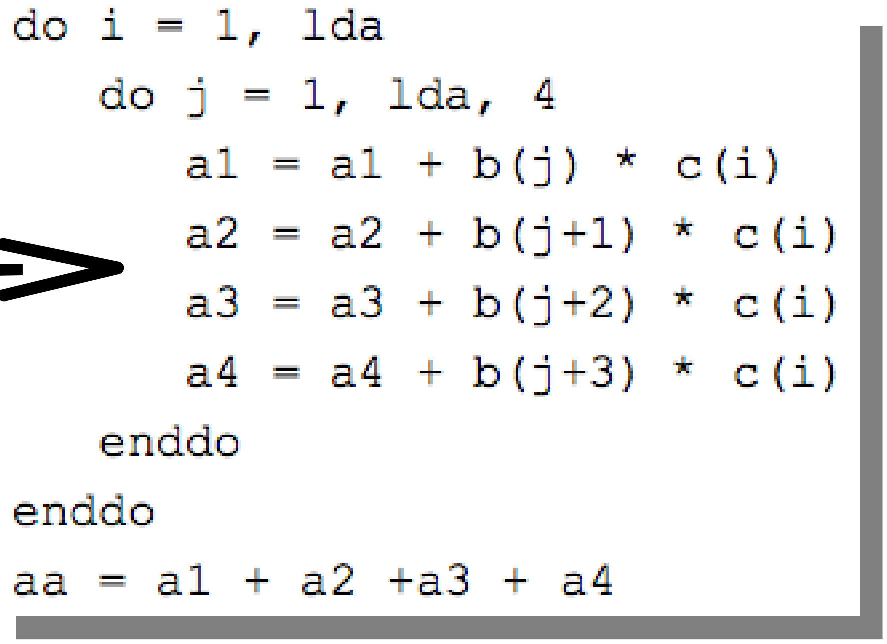
```
do i = 1, lda
   do j = 1, lda
      a(j,i) = a(j,i) +  b(i,1) * c(j,1)
      a(j,i) = a(j,i) +  b(i,2) * c(j,2)
      a(j,i) = a(j,i) +  b(i,3) * c(j,3)
      a(j,i) = a(j,i) +  b(i,4) * c(j,4)
   enddo
enddo
```

Loop Unrolling & Sum Reduction

- When an loop has a data dependency that introduces serialization.
- Solution is to unroll and introduce intermediate registers.



```
do i = 1, lda
   do j = 1, lda
      a = a + (b(j) * c(i))
   enddo
enddo
```



```
do i = 1, lda
   do j = 1, lda, 4
      a1 = a1 + b(j) * c(i)
      a2 = a2 + b(j+1) * c(i)
      a3 = a3 + b(j+2) * c(i)
      a4 = a4 + b(j+3) * c(i)
   enddo
enddo
aa = a1 + a2 +a3 + a4
```

(Outer) Loop Unroll and Jam

- Reduce register pressure
- Decrease loads and stores per iteration

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

```
DO I = 1, N, 2
  DO J = 1, N, 4
    DO K = 1, N
      A(I,J) = A(I,J) + B(I,K) * C(K,J)
      A(I+1,J) = A(I+1,J) + B(I+1,K) * C(K,J)
    ENDDO
    DO K = 1, N
      A(I,J+1) = A(I,J+1) + B(I,K) * C(K,J+1)
      A(I+1,J+1) = A(I+1,J+1) + B(I+1,K) * C(K,J+1)
    ENDDO
    DO K = 1, N
      A(I,J+2) = A(I,J+2) + B(I,K) * C(K,J+2)
      A(I+1,J+2) = A(I+1,J+2) + B(I+1,K) * C(K,J+2)
    ENDDO
    DO K = 1, N
      A(I,J+3) = A(I,J+3) + B(I,K) * C(K,J+3)
      A(I+1,J+3) = A(I+1,J+3) + B(I+1,K) * C(K,J+3)
    ENDDO
  ENDDO
ENDDO
```

(Outer) Loop Unroll and Jam

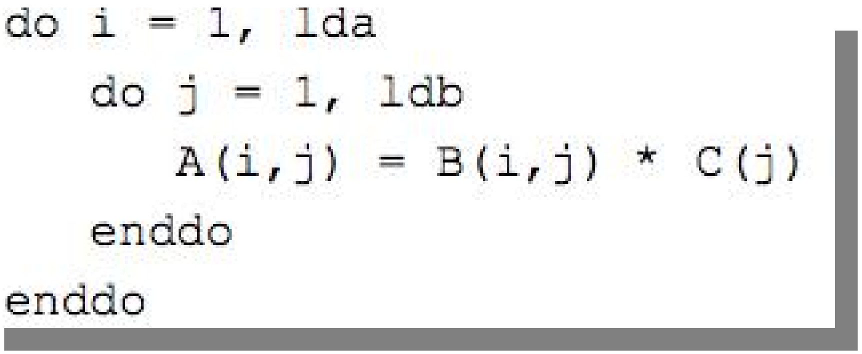
- Be careful loop body does not become too large.
  - Should have enough registers for int. results.

```
DO I = 1, N, 2
  DO J = 1, N, 4
    DO K = 1, N
      A(I,J) = A(I,J) + B(I,
      A(I+1,J) = A(I+1,J) +
    ENDDO
    DO K = 1, N
      A(I,J+1) = A(I,J+1) +
      A(I+1,J+1) = A(I+1,J+1
    ENDDO
    DO K = 1, N
      A(I,J+2) = A(I,J+2) +
      A(I+1,J+2) = A(I+1,J+2
    ENDDO
    DO K = 1, N
      A(I,J+3) = A(I,J+3) +
      A(I+1,J+3) = A(I+1,J+3
    ENDDO
  ENDDO
ENDDO
```

```
DO I = 1, N, 2
  DO J = 1, N, 4
    DO K = 1, N
      A(I,J) = A(I,J) + B(I,K) * C(K,J)
      A(I+1,J) = A(I+1,J) + B(I+1,K) * C(K,J)
      A(I,J+1) = A(I,J+1) + B(I,K) * C(K,J+1)
      A(I+1,J+1) = A(I+1,J+1) + B(I+1,K) * C(K,J+1)
      A(I,J+2) = A(I,J+2) + B(I,K) * C(K,J+2)
      A(I+1,J+2) = A(I+1,J+2) + B(I+1,K) * C(K,J+2)
      A(I,J+3) = A(I,J+3) + B(I,K) * C(K,J+3)
      A(I+1,J+3) = A(I+1,J+3) + B(I+1,K) * C(K,J+3)
    ENDDO
  ENDDO
ENDDO
```

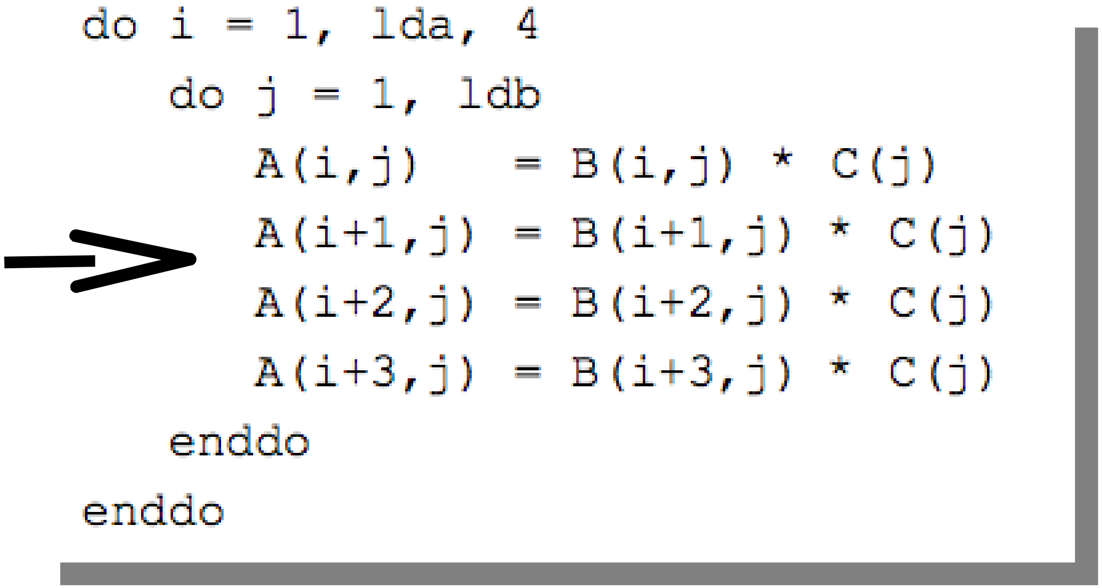
Outer Loop Unrolling

- Goal is to reduce number of loads and stores on inner loops with invariants
  - More results can be kept in registers or in cache
- Compilers not quite as good at this.



```
do i = 1, lda
   do j = 1, ldb
      A(i,j) = B(i,j) * C(j)
   enddo
enddo
```

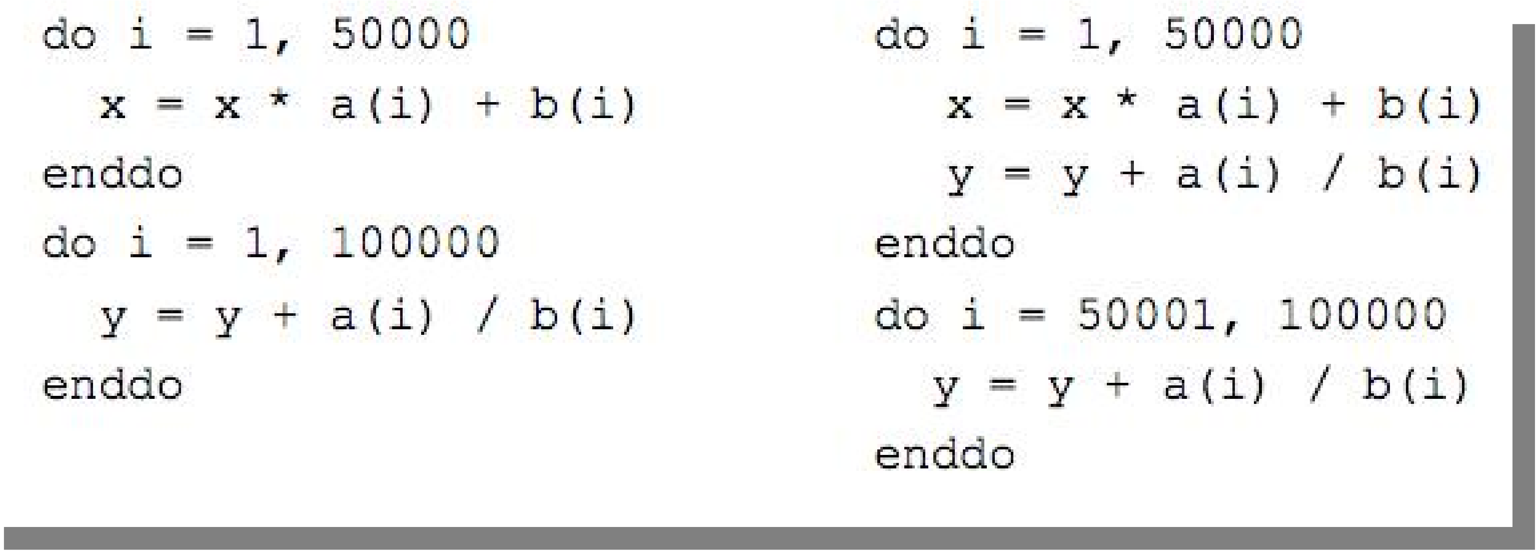
→



```
do i = 1, lda, 4
   do j = 1, ldb
      A(i,j)   = B(i,j) * C(j)
      A(i+1,j) = B(i+1,j) * C(j)
      A(i+2,j) = B(i+2,j) * C(j)
      A(i+3,j) = B(i+3,j) * C(j)
   enddo
enddo
```

Loop Jam/Fusion

- Merge two loops that access (some) similar data to:
  - Reduce loop overhead, Improve instruction mix, Lower cache misses
- Fusion can create associativity conflicts



```
do i = 1, 50000
  x = x * a(i) + b(i)
enddo
do i = 1, 100000
  y = y + a(i) / b(i)
enddo
```

```
do i = 1, 50000
   x = x * a(i) + b(i)
   y = y + a(i) / b(i)
enddo
do i = 50001, 100000
  y = y + a(i) / b(i)
enddo
```

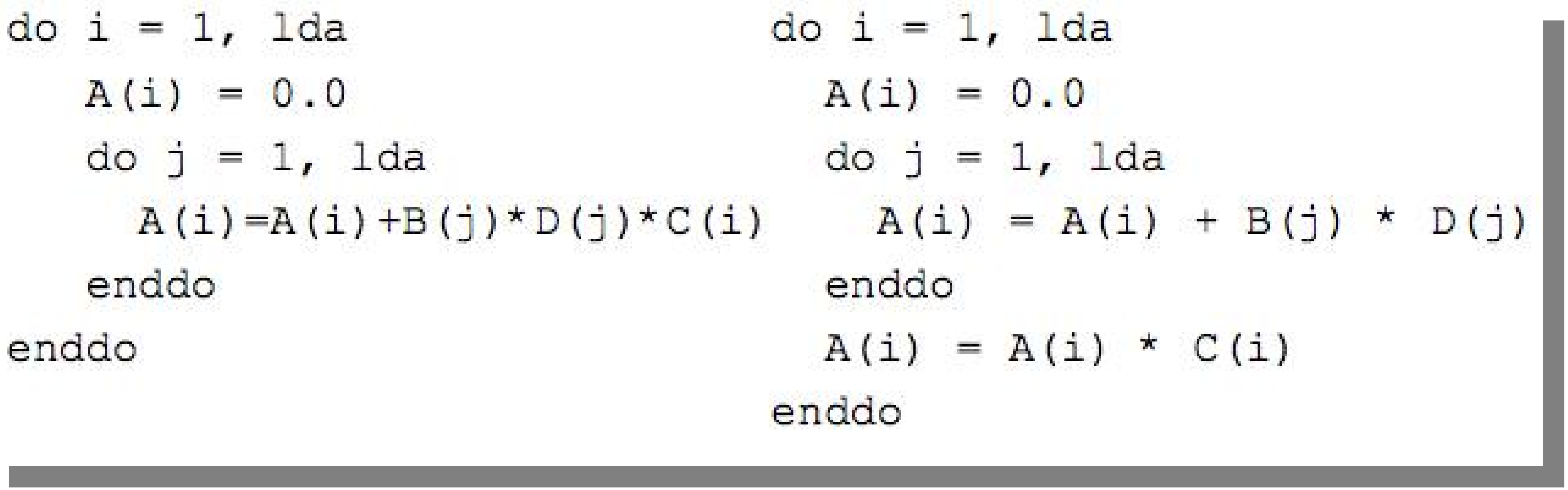
Loop Defactorization

- Reduce the number of array elements referenced, to reduce cache traffic.
- But floating point operations are not always associative.

$$(A + B) + C \quad != A + (B + C)$$

- Verify that your results are still “correct”

Loop Defactorization

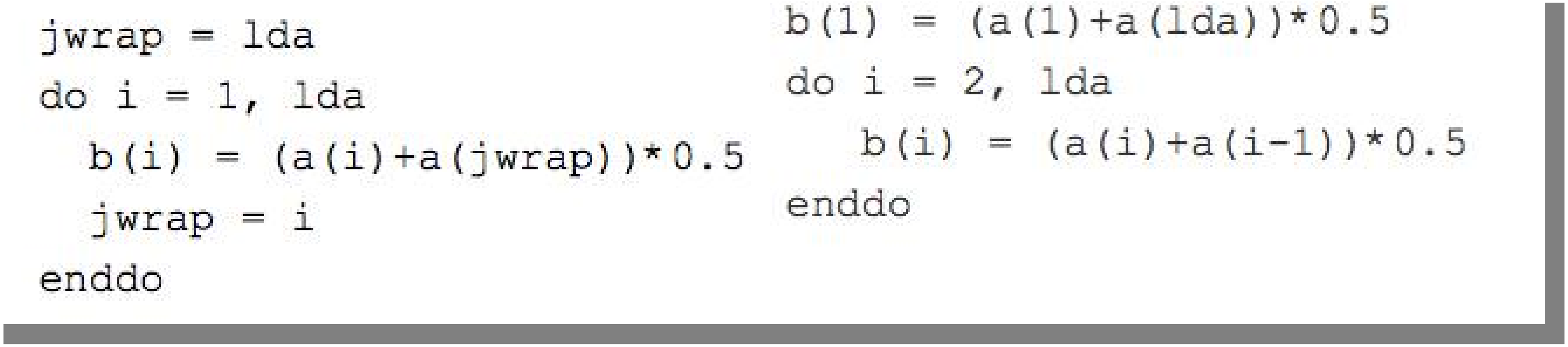


```
do i = 1, lda
   A(i) = 0.0
   do j = 1, lda
     A(i)=A(i)+B(j)*D(j)*C(i)
   enddo
enddo
```

```
do i = 1, lda
  A(i) = 0.0
  do j = 1, lda
    A(i) = A(i) + B(j) * D(j)
  enddo
  A(i) = A(i) * C(i)
enddo
```

Loop Peeling

- For loops which access previous elements in arrays.
- Compiler cannot determine that an item does not need to be reloaded on every iteration.

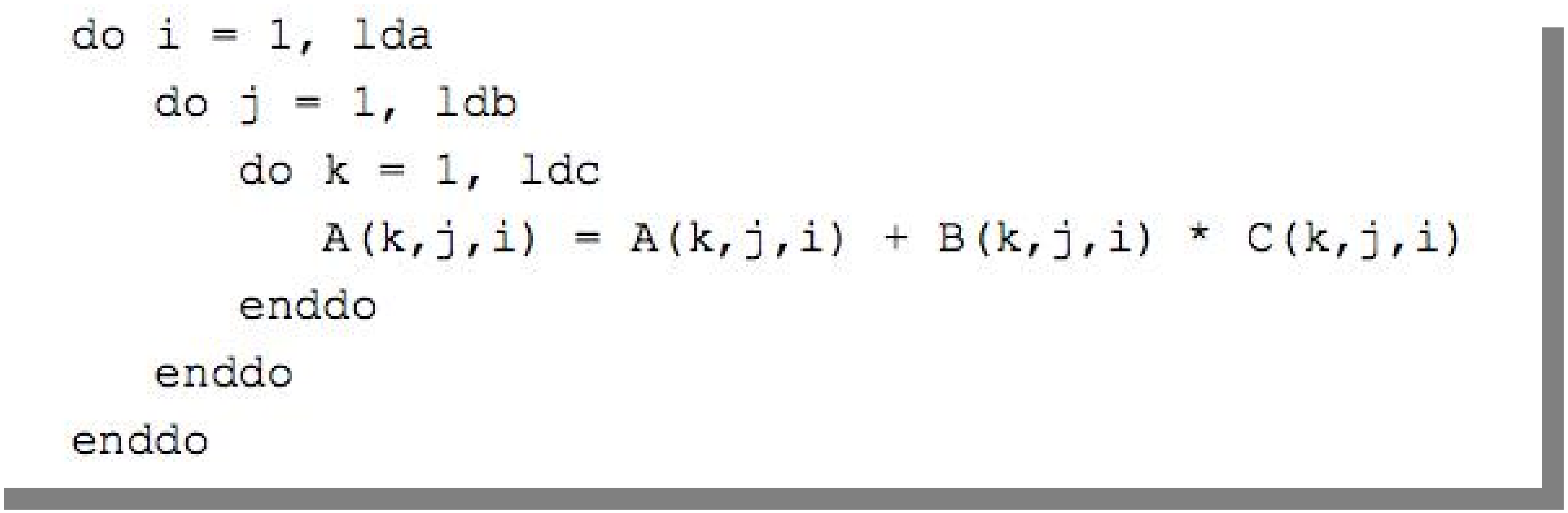


```
jwrap = lda
do i = 1, lda
  b(i) = (a(i)+a(jwrap))*0.5
  jwrap = i
enddo
```

```
b(1) = (a(1)+a(lda))*0.5
do i = 2, lda
   b(i) = (a(i)+a(i-1))*0.5
enddo
```

Loop Collapse

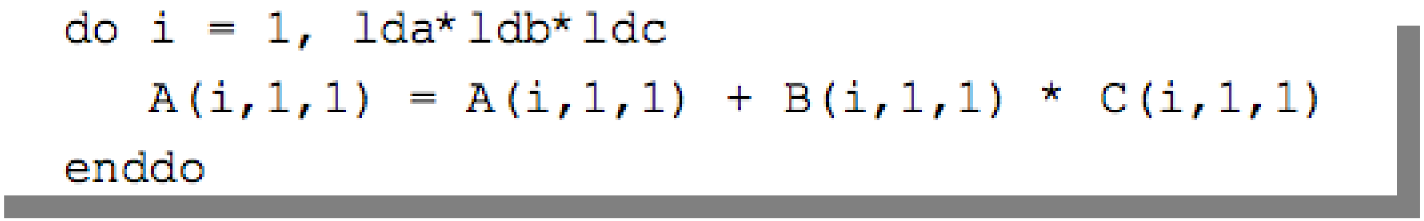
- Reduce address computation and loop nesting.
- Reduces loop overhead and increases chance of vectorization.



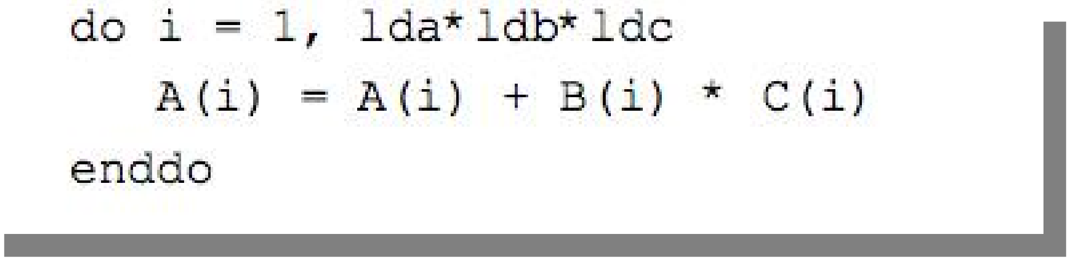
```
do i = 1, lda
   do j = 1, ldb
      do k = 1, ldc
         A(k,j,i) = A(k,j,i) + B(k,j,i) * C(k,j,i)
      enddo
   enddo
enddo
```

Loop Collapse

- This can be especially effective in C and C++, where often macros are used to compute multi-dimensional array offsets.



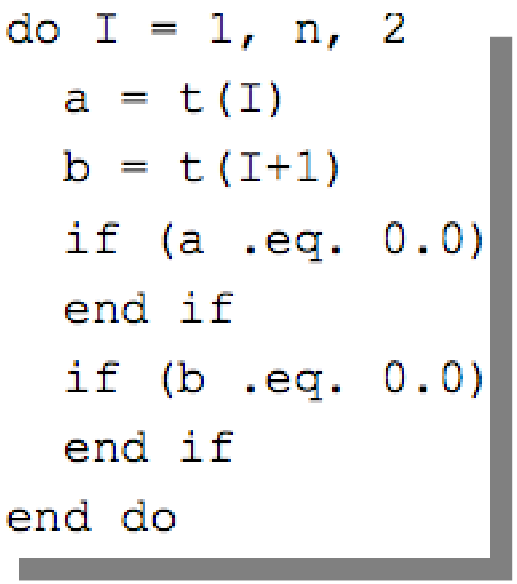
```
do i = 1, lda*ldb*ldc
   A(i,1,1) = A(i,1,1) + B(i,1,1) * C(i,1,1)
enddo
```



```
do i = 1, lda*ldb*ldc
   A(i) = A(i) + B(i) * C(i)
enddo
```

If statements in Loops

- We already know many optimizations that this inhibits.
- Unroll loop, move conditional elements into scalars early, test scalars at end of loop.

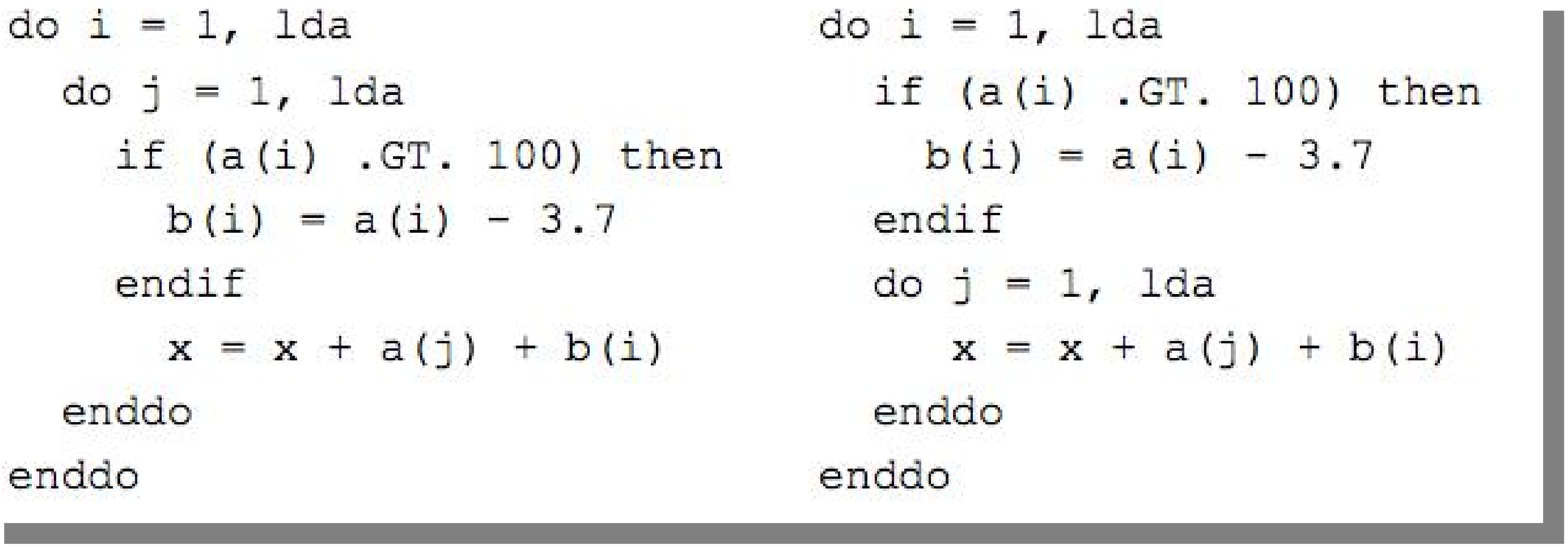


```
do I = 1, n, 2
  a = t(I)
  b = t(I+1)
  if (a .eq. 0.0)
  end if
  if (b .eq. 0.0)
  end if
end do
```

Floating IF's

- IF statements that do not change from iteration to iteration can be hoisted.
- Compilers are usually good at this except when:
  - Loops contain calls to procedures
  - Loops have variable bounds
  - Loops reference global variables that may be aliased to data in the IF statement.

Floating IF's



```
do i = 1, lda
  do j = 1, lda
    if (a(i) .GT. 100) then
      b(i) = a(i) - 3.7
    endif
      x = x + a(j) + b(i)
  enddo
enddo
```

```
do i = 1, lda
  if (a(i) .GT. 100) then
   b(i) = a(i) - 3.7
  endif
  do j = 1, lda
     x = x + a(j) + b(i)
  enddo
enddo
```

Some Results

- Taken years ago on 3 different architectures with the **best** compilation technology at that time.
- Percent is of -O3 version but untuned.

	Arch A	Arch B	Arch C
Stride Minimization	35%	9%	100%
Fusion	69%	80%	81%
Interchange	75%	100%	100%
Floating IF's	46%	100%	101%
Loop Defactor	66%	76%	94%
Loop Peeling	97%	64%	81%
Loop Unrolling	97%	89%	67%
Loop Unroll + SumR	77%	100%	39%
Outer Loop Unrolling	83%	26%	46%

Indirect Addressing

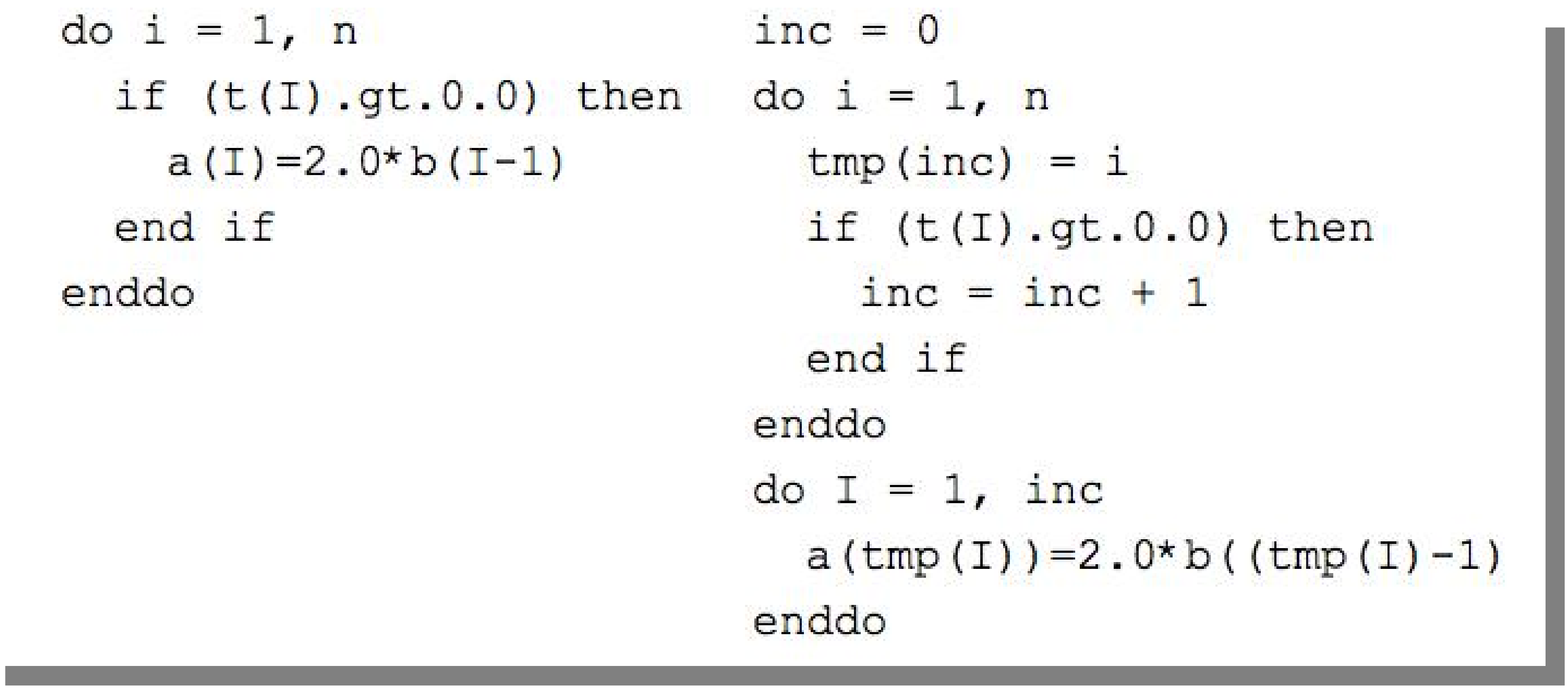
$$X(I) = X(I) * Y(A(I))$$

- Very hard for a compiler to optimize.
- Very difficult for “normal” memory subsystems.
  - Most memory subsystems are just bad at pseudo-random accesses.
  - Hardware prefetch can mitigate, but can also hurt
- When you have this construct, either:
  - Consider using a sparse solver package.
  - Block your data into small cache-line sized chunks and do some redundant computation.

Gather-Scatter Optimization

- For loops with conditional work.
- Split loop to gather indirect array where work needs to be done.
- Can increase pipelining, effectiveness of prefetching and enable other loop optimizations.
  - Depends on amount of work per iteration and locality of reference.

Gather-Scatter Optimization



```
do i = 1, n
  if (t(I).gt.0.0) then
    a(I)=2.0*b(I-1)
  end if
enddo
```

```
inc = 0
do i = 1, n
  tmp(inc) = i
  if (t(I).gt.0.0) then
    inc = inc + 1
  end if
enddo
do I = 1, inc
  a(tmp(I))=2.0*b((tmp(I)-1)
enddo
```

OOC and C++ Considerations

- Extensive use creates much greater memory pressure, lots and lots of pointers.
- Dynamic typing and polymorphism is not free.
- Make use of **inline**, **const** and **restrict** keywords
- Use STL, Boost and other support libraries
  - Expresses more of author's intent to compiler to increase performance.
  - But be careful with multicore of the above.

Fortran Considerations

- WHERE statements
- ALLOCATE alignment
- Array shapes, sizes, slices etc.

Fortran 90 Arrays

- The (:) syntax is very useful.
- But this can hide significant amount of data movement, often repeatedly.
  - Pollutes the caches
  - Creates temporaries that may have pathological alignment, especially with 'assumed shapes'
- Consider creating an explicit temporary if you need to pass slices around.

Fortran 90 WHERE statements

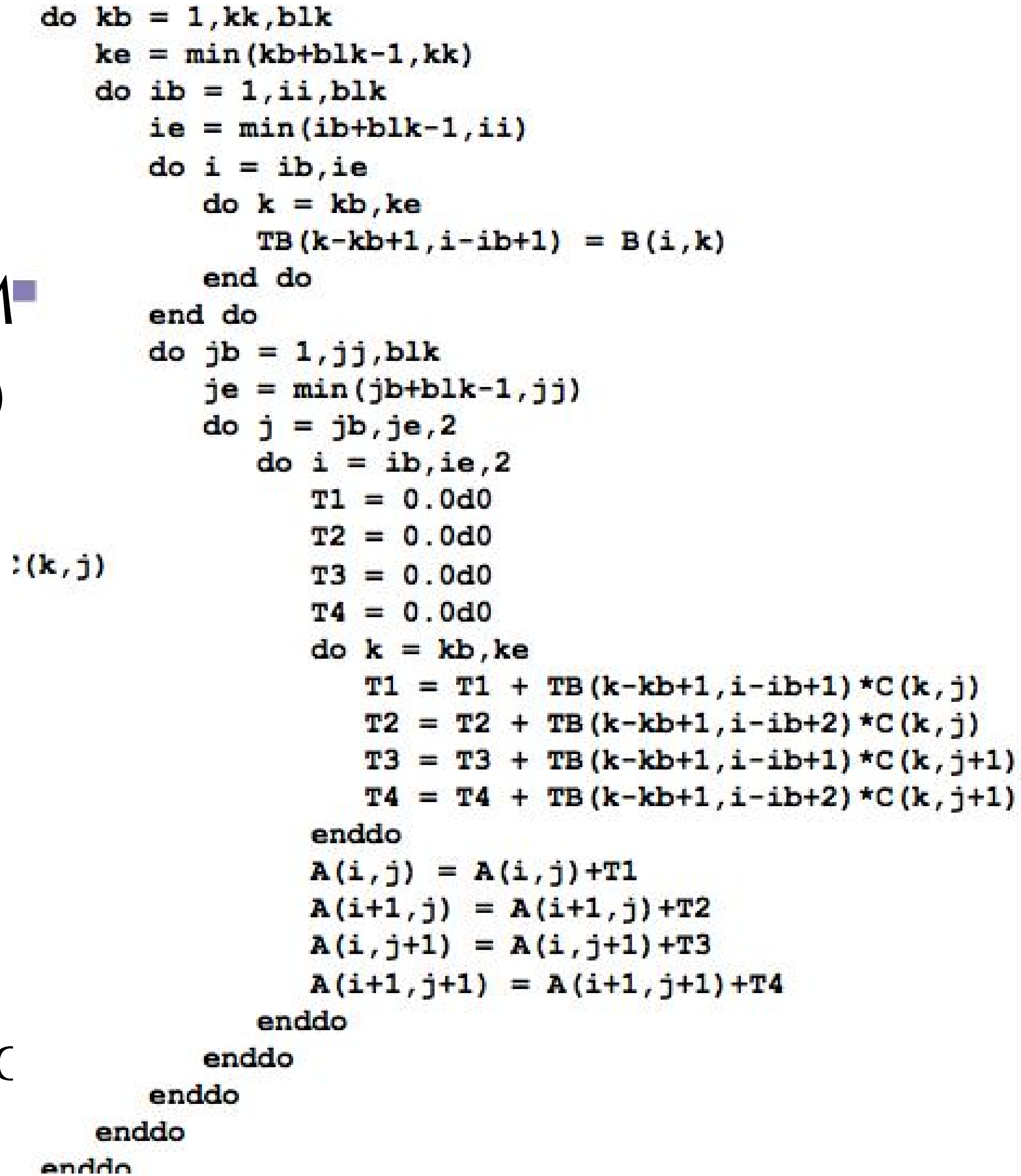
- A construct for masking array operations
- Generated code is often required to be a loop containing an if statement.
  - Highly inefficient
- Consider multiplying by a 0 or 1 mask array with the same shape into a temporary.

Optimized Arithmetic Libraries

- Usually, it's best NOT to write your own code.
  - Many good programmers are focussed on multicore development
- Advantages:
  - Performance, Portability, Prototyping
  - Let someone else solve the hard problems.
- Disadvantages:
  - Extensive use can lead to vertical code structure.
  - May make performance debugging difficult.

- Sample DGEMM
  - (old Pentium IV)
- Naïve
  - 200 MF
- Advanced ->
  - 1 GF
- Optimal
  - 2.5GF
- Think you can dc

:(k,j)



```
do kb = 1,kk,blk
   ke = min(kb+blk-1,kk)
   do ib = 1,ii,blk
      ie = min(ib+blk-1,ii)
      do i = ib,ie
         do k = kb,ke
            TB(k-kb+1,i-ib+1) = B(i,k)
         end do
      end do
      do jb = 1,jj,blk
         je = min(jb+blk-1,jj)
         do j = jb,je,2
            do i = ib,ie,2
               T1 = 0.0d0
               T2 = 0.0d0
               T3 = 0.0d0
               T4 = 0.0d0
               do k = kb,ke
                  T1 = T1 + TB(k-kb+1,i-ib+1)*C(k,j)
                  T2 = T2 + TB(k-kb+1,i-ib+2)*C(k,j)
                  T3 = T3 + TB(k-kb+1,i-ib+1)*C(k,j+1)
                  T4 = T4 + TB(k-kb+1,i-ib+2)*C(k,j+1)
               enddo
               A(i,j) = A(i,j)+T1
               A(i+1,j) = A(i+1,j)+T2
               A(i,j+1) = A(i,j+1)+T3
               A(i+1,j+1) = A(i+1,j+1)+T4
            enddo
         enddo
      enddo
   enddo
enddo
```

Multicore Programming

Multithreaded Programming

- Here we will cover three popular models:
  - MPI
  - Pthreads (C and C++)
  - OpenMP
- We will talk a bit about
  - PGAS languages

Expressing Parallelism

- Data parallelism
  - Programmer specifies chunk of work to be done in parallel.
    - Same operation on every thread, using different data
    - OpenMP, UPC, Co-Array Fortran, etc...
- Functional (or task) parallelism
  - Programmer partitions work by thread or function.
    - MPI, Pthreads, Cilk, etc...

Message Passing

- Program explicitly exchanges data.
- Semantics are send/receive (identified by tag) or get/put (direct to address).
- Ordering and consistency are somewhat implicit.
  - Synchronization usually not needed
- Designed for distinct address spaces.
  - Nothing really shared other than task ID's
- MPI, PVM, SHMEM, Sockets

Shared Memory

- Data is exchanged implicitly as part of an expression.
  - Load/store or language feature.
- No guarantee of ordering or consistency
  - Synchronization is needed.
- Programs share everything
  - Or in higher level models, data that is declared **shared**.
- One can be used to implement the other...

MPI and Multicore

- MPI was originally designed for distributed memory machines.
  - Receiver is not expected to be in the same address space of the sender.
  - Data was expected to be copied, packed, sent, received, unpacked, copied, etc...
  - Much work has been done to "eliminate the copies".
    - You can get a 4 byte message across a wire in 1us these days (if you do it 1000's of times and average)
  - But that's still way more expensive than 2ns.

MPI and Multicore 2

- MPI-2 introduced some get/put primitives to introduce more direct access to remote memory.
  - Not nearly as lightweight or flexible as they should have been, thus limited acceptance.
  - Require synchronization.
- Most MPI's were not previously safe for threads.
  - You had to run multiple processes on a multicore machine.
  - Things are different now.

MPI and Multicore 3

- Many MPI's are now both thread safe and tuned for on-node, shared memory operation.
- This means you can easily use MPI for multicore programming.
  - Advantages to this are:
    - Explicit coding of data exchange and synchronization. Code may be easier to read and tune.
  - Disadvantages are:
    - You can lose a substantial amount of performance. Granularity of parallelism must be coarse. Programming model is limited.

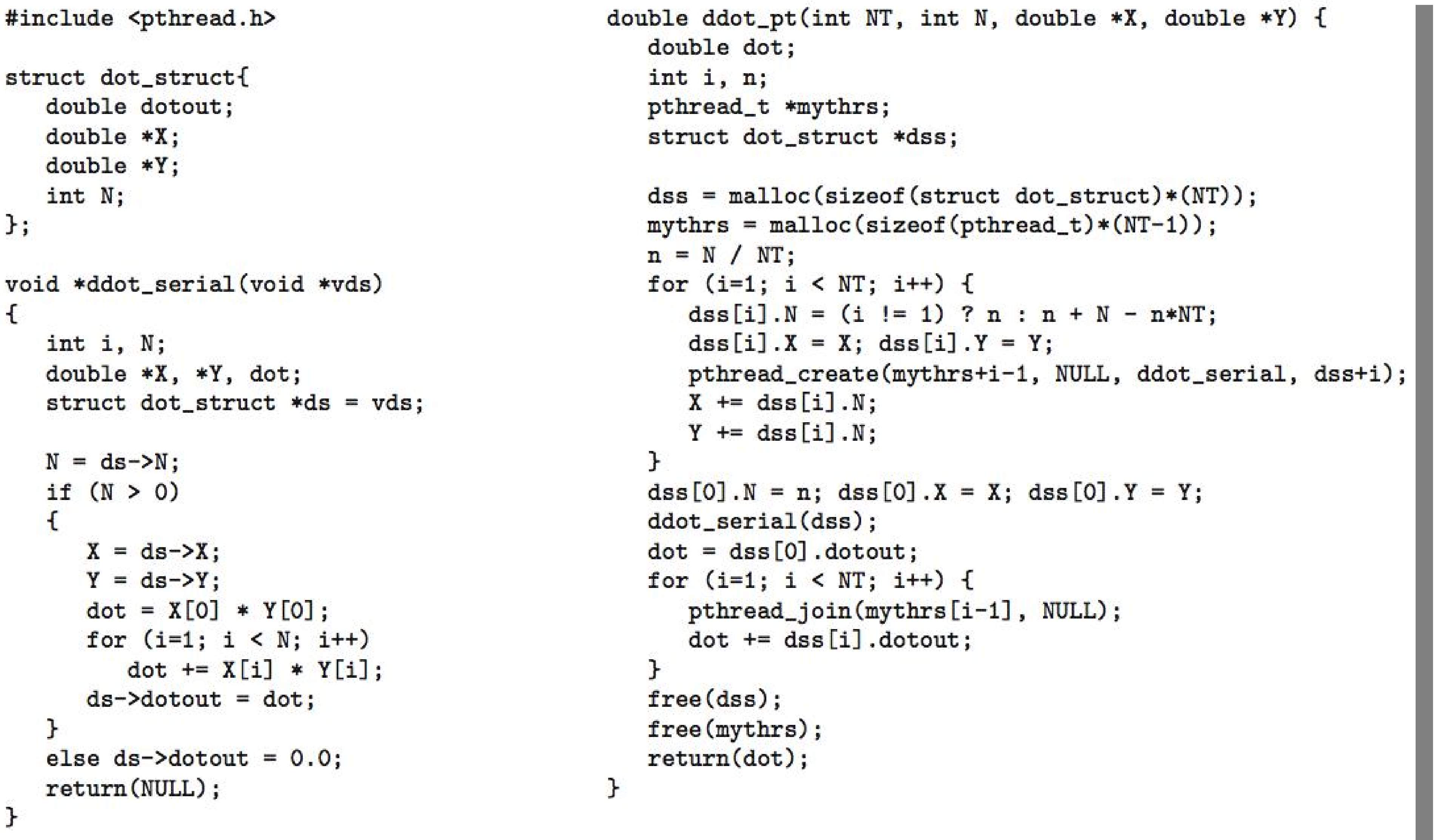
Pthreads

- Assembly language of thread programming.
  - A Pthread is an OS thread
  - Not for you Fortran programmers.
- Basic primitives are Create, Join and Mutex
- Used in combination with “messages” to create different models.
  - master/worker model
  - gang model (master is a worker)
  - pipeline (dataflow)

Pthreads

- With pthreads, everything is shared except variables declared on the stack.
  - Extreme care must be used to coordinate access to global data structures.
  - Reads and writes need to be consistent.
  - Different cores should be working with different cache lines.
- 3 types of synchronization
  - Mutex, condition variables and rwlocks.

Pthread Dot Product Example

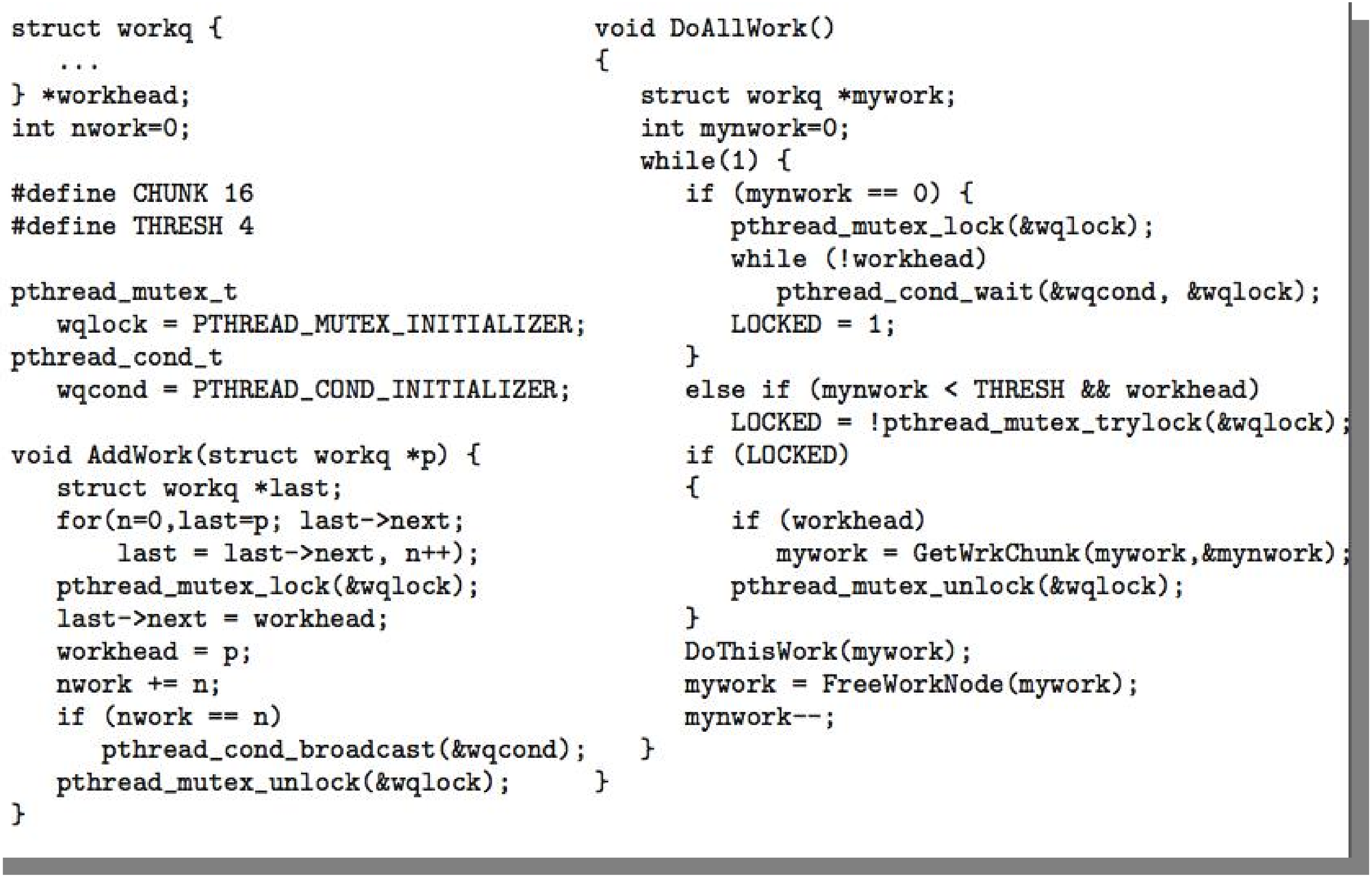


```
#include <pthread.h>

struct dot_struct{
   double dotout;
   double *X;
   double *Y;
   int N;
};

void *ddot_serial(void *vds)
{
   int i, N;
   double *X, *Y, dot;
   struct dot_struct *ds = vds;

   N = ds->N;
   if (N > 0)
   {
      X = ds->X;
      Y = ds->Y;
      dot = X[0] * Y[0];
      for (i=1; i < N; i++)
         dot += X[i] * Y[i];
      ds->dotout = dot;
   }
   else ds->dotout = 0.0;
   return(NULL);
}
```

```
double ddot_pt(int NT, int N, double *X, double *Y) {
   double dot;
   int i, n;
   pthread_t *mythrs;
   struct dot_struct *dss;

   dss = malloc(sizeof(struct dot_struct)*(NT));
   mythrs = malloc(sizeof(pthread_t)*(NT-1));
   n = N / NT;
   for (i=1; i < NT; i++) {
      dss[i].N = (i != 1) ? n : n + N - n*NT;
      dss[i].X = X; dss[i].Y = Y;
      pthread_create(mythrs+i-1, NULL, ddot_serial, dss+i);
      X += dss[i].N;
      Y += dss[i].N;
   }
   dss[0].N = n; dss[0].X = X; dss[0].Y = Y;
   ddot_serial(dss);
   dot = dss[0].dotout;
   for (i=1; i < NT; i++) {
      pthread_join(mythrs[i-1], NULL);
      dot += dss[i].dotout;
   }
   free(dss);
   free(mythrs);
   return(dot);
}
```

Thanks to Clint Whaley

Pthread Work Queue Example



```
struct workq {
   ...
} *workhead;
int nwork=0;

#define CHUNK 16
#define THRESH 4

pthread_mutex_t
   wqlock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t
   wqcond = PTHREAD_COND_INITIALIZER;

void AddWork(struct workq *p) {
   struct workq *last;
   for(n=0,last=p; last->next;
       last = last->next, n++);
   pthread_mutex_lock(&wqlock);
   last->next = workhead;
   workhead = p;
   nwork += n;
   if (nwork == n)
      pthread_cond_broadcast(&wqcond);
   pthread_mutex_unlock(&wqlock);
}
```

```
void DoAllWork()
{
   struct workq *mywork;
   int mynwork=0;
   while(1) {
      if (mynwork == 0) {
         pthread_mutex_lock(&wqlock);
         while (!workhead)
            pthread_cond_wait(&wqcond, &wqlock);
         LOCKED = 1;
      }
      else if (mynwork < THRESH && workhead)
         LOCKED = !pthread_mutex_trylock(&wqlock);
      if (LOCKED)
      {
         if (workhead)
            mywork = GetWrkChunk(mywork,&mynwork);
         pthread_mutex_unlock(&wqlock);
      }
      DoThisWork(mywork);
      mywork = FreeWorkNode(mywork);
      mynwork--;
   }
}
```

Thanks to Clint Whaley

Logical Parallelism

- Separate processors from programmers view of threads.
  - Make chunks or work and threads separate.
- Make queues of work for each thread.
  - Send work to threads in chunks.
  - If a thread finishes, get more work.
- Ideally, the programmer should not have to think about processors, just think in parallel!

OpenMP

- Designed for quick and easy parallel programming of shared memory machines.
- Works by inserting compiler directives in code, usually around loops.
- Threads are started implicitly and “fed” work.

OpenMP Directives

- Parallelization
  - parallel, for, do, workshare, section, sections, task
  - single, master
- Data placement and handling
  - shared, private, threadprivate, copyprivate, firstprivate, lastprivate, reduction
- Synchronization
  - barrier, ordered, critical, atomic, flush, nowait

OpenMP Data Parallelism

```
      PROGRAM WORKSHARE

      INTEGER N, I, J
      PARAMETER (N=100)
      REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N), FIRST, LAST
!     Some initializations
      DO I = 1, N
        DO J = 1, N
          AA(J,I) = I * 1.0
          BB(J,I) = J + 1.0
        ENDDO
      ENDDO
!$OMP PARALLEL SHARED(AA,BB,CC,DD,FIRST,LAST)
!$OMP WORKSHARE
      CC = AA * BB
      DD = AA + BB
      FIRST = CC(1,1) + DD(1,1)
      LAST = CC(N,N) + DD(N,N)
!$OMP END WORKSHARE
!$OMP END PARALLEL
      END
```

OpenMP Data Parallelism

```
#include <omp.h>
#define CHUNKSIZE 100
#define N     1000

main ()
{
int i, chunk;
float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;

#pragma omp parallel shared(a,b,c,chunk) private(i)
  {
  #pragma omp for schedule(dynamic,chunk)
  for (i=0; i < N; i++)
    c[i] = a[i] + b[i];

  }  /* end of parallel section */
}
```

OpenMP Task Parallelism

```
      PROGRAM VEC_ADD_SECTIONS
      INTEGER N, I
      PARAMETER (N=1000)
      REAL A(N), B(N), C(N), D(N)

!     Some initializations
      DO I = 1, N
        A(I) = I * 1.5
        B(I) = I + 22.35
      ENDDO

!$OMP PARALLEL SHARED(A,B,C,D), PRIVATE(I)
!$OMP SECTIONS
!$OMP SECTION
      DO I = 1, N
         C(I) = A(I) + B(I)
      ENDDO
!$OMP SECTION
      DO I = 1, N
         D(I) = A(I) * B(I)
      ENDDO
!$OMP END SECTIONS
!$OMP END PARALLEL
      END
```

CILK

- “Logical” task parallelism in a ANSI C
  - Handful of new keywords, **spawn** and **join**.
  - Work-stealing scheduler: programmer just thinks in parallel, scheduler does the work
- Mostly Open Source
  - http://supertech.csail.mit.edu/cilk
- Commercial compilers also available (Cilk++)
  - C++, parallel loops
  - http://www.cilk.com

Cilk Example

```
int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = fib(n-1);
    y = fib(n-2);
    return (x+y);
  }
}
```

```
cilk int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x+y);
   }
}
```

© Charles E. Leiserson
http://supertech.csail.mit.edu/cilk/lecture-1.pdf

Unified Parallel C

- Shared memory parallel extension to C
  - **shared**, **relaxed** and **strict** keywords
  - Intrinsinc functions for sync, get/put, collectives, worksharing, I/O
- Easy to program, but for performance...
  - Remote references on on single data items when they are used (not before)
  - Compiler must hoist and aggregate comm.
- OS and Commerical: http://upc.lbl.gov/

UPC Example

```
shared int our_hits[THREADS];  /* single writer */
main(int argc, char **argv) {
  int i, hits, trials = 1000000;
  double pi;

  seed48 ((MYTHREAD+1) * THREADS);
  for (i=0; i<trials; i++)
    hits += hit();
  our_hits[MYTHREAD] = hits;
  upc_barrier;

  for (i=0,hits=0; i<THREADS; i++)
    hits += our_hits[i];
  pi = 4.0*hits/trials;

  printf(”Thread %d estimates pi = %g”, MYTHREAD, pi);
}
```

Co-Array Fortran

- Small set of extensions to Fortran 95 standard for SPMD parallelism.
  - Similar to UPC but much simpler.
- [n|:] notation after arrays to denote processor.
- Performance issues are more serious.
  - Not as nearly expressive as UPC.
- Part of Fortran 2008. G95 has some support. Cray has commercial product.
- http://www.co-array.org/

Co-Array Fortran Example

```
REAL, DIMENSION(N)[*] :: X,Y ! declare X,Y as parallel
X(:) = Y(:)[Q]   ! collect from Q
X       = Y[PE]  ! get from Y[PE]
Y[PE]   = X      ! put into Y[PE]
Y[:]    = X      ! broadcast X
Y[LIST] = X      ! broadcast X over subset of LIST PE's
Z(:)    = Y[:]   ! collect all Y
S = MINVAL(Y[:]) ! min (reduce) all Y
```

Other PGAS languages

- Titanium (Java-like)
- Fortress (ML/Haskell-like)
- Chapel (C/Java-like)
- X10 (Java-like)
- Ok, but let's remember what happened to HPF, ZPL, Split-C, etc...

Question

- ***"When should I be trying to use CUDA over (my) multicore CPU? My experience with CUDA is that I am very good at writing very slow CUDA code."***

Answer

- When you've:
  - Had entirely too many *great days* in a row and need a change.
  - Don't want to go to church but still wish to be punished for your all your sins.
  - Voted for or (ever) supported George Bush.
- Save it for the platform on which it was made for.

Multicore Optimization

What is Good Parallel Performance?

- Single core performance is consistently high.
  - But how high is up?
- The code exhibits decent scaling.
  - Strong scaling: Total problem size is fixed.
  - Weak scaling: Problem size per processor is fixed.
- Interprocessor, Sync, Comm, I/O are not the bottlenecks.
- It all starts with a good parallel algorithm.

Reported Linear Scalability(?)

- When you see linear scaling graphs, be (somewhat) suspicious.
- Linear scalability is easy(ier) when per-core performance is low!
- The faster a single core computes, the more vulnerable it is to other bottlenecks.
  - Memory, Sync, Comm, I/O
- So producing a linear graph, does not make your program efficient.

Multicore Optimization

- Use multicore-tuned libraries.
- Reduce memory bandwidth requirements of algorithm
  - Make it cache friendly
  - Help the compilers tune the loops
- Reduce synchronization
  - Bigger chunks of work to each thread
- Reduce cache contention
  - Alignment, blocking, locks, etc..

Expressing Parallelism

- Important things to remember:
  - Granularity must always be as large as possible.
  - Synchronization and communication are expensive.
- Initiating parallel work is not “free”.

Libraries

- Do you really need to write the solver yourself?
- No!
  - Single core is hard enough.
  - You have to be willing to change your storage format.
- Vendor math libraries are probably best at this.
- ScalaPack, PetSC, SuperLU, FFTW, EISPACK, VSIPL, SPRNG, HYPRE etc etc.
- But if it's not your bottleneck, then it doesn't matter.

Optimal Usage of O.S. Libraries

- Ideally, you should compile the libraries with the same compiler and flags you use on your binary.
  - Vendor compilers are best, but may be fussy.
- Specialize as much as possible for your platform.
  - Unless you know it needs to run in many places.
- Many optimization flags (especially IPO and the like), need to be set for every stage.
- How can you check?
  - You need a good sys-admin or do it yourself.

What is a Multicore Library?

- They come in two forms, yet both can be called “multithreaded”.
- Monolithic
  - Your (serial) program calls a (library) routine which uses all the cores to solve the problem.
    - The library spawns/joins threads as necessary.
  - Good: It can't get much easier to use.
  - Bad: Memory contention and parallel overhead.

What is a Multicore Library?

- Really parallel
  - Your (parallel) program creates it's own threads and each calls the library on the relevant portion of data.
    - You control the degree of parallelism.
  - Good: The library can dictate alignment, placement, etc through allocators. Overhead can be amortized.
  - Bad: Increase in complexity and plenty of room for error.

Multicore Libraries

- Your code can be a hybrid.
  - MPI program running on every node, linked against a Intel MKL that spawns it's own threads.
- How should one use them?
  - HPL (Linpack): MPI between nodes and multi-threaded BLAS on node is usually slower than MPI.
    - But HPL runs at > 75% of peak, DGEMM sometimes 90%!
  - Your real code won't get anywhere near that.
    - So go with what's simple.

VSIPL

- Vector Signal Image Processing Library
- Filters
- Stencils
- Convolutions
- Wavelets
- Serial and Parallel versions

LAPACK/ScaLAPACK

- Comprehensive solver package for dense systems of linear equations
  - Eigenvalue problems
  - Factorizations
  - Reordering/Conditioning
  - Parallel and serial versions
  - Some out of core solvers and packaged storage routines
- ATLAS/BLAS/etc...

PETSc

- Generalized sparse solver package for solution of PDE's
- Contains different preconditions, explicit and implicit methods
- Storage format is highly optimized for performance
- Serial, Parallel and threaded

FFTW

- Multidimensional FFTs
  - Serial, threaded and parallel
  - Variety of radix sizes and data types

SuperLU

- LU factorization of sparse matrices
  - Highly optimized
  - Compressed block storage formats
  - Serial, parallel and threaded

The Cost of Threading

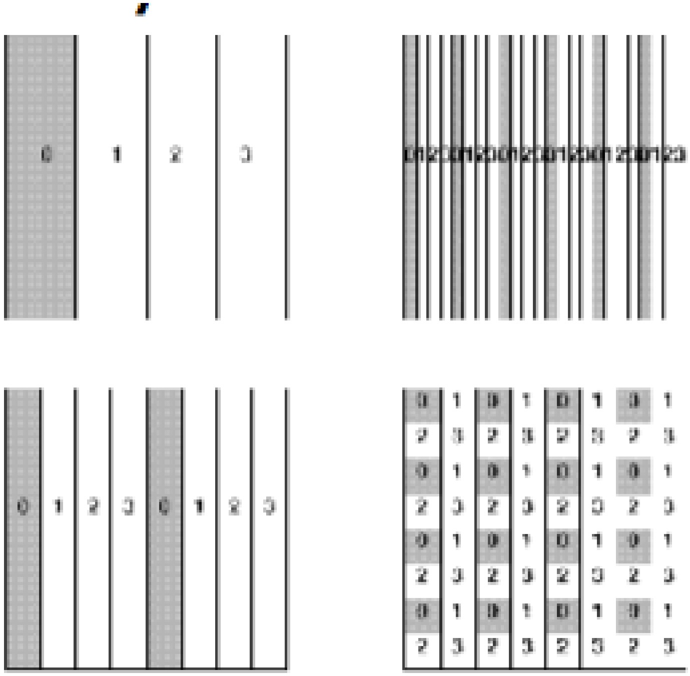
- Starting, stopping and scheduling them requires the OS to do *expensive work*.
  - TLB/Cache flushing
- Most multicore paradigms create and destroy real OS threads.
  - So do not use them as function calls!
  - Keep threads around and send them work.
- Case example: FDTD from Oil patch

Load Balance

- Balancing the work between cores can be an issue.
  - OpenMP and CILK can provide dynamic scheduling of iterations
  - Pthreads you are on your own
- Ok, but we still must consider **cache line contention** when choosing a data layout.

4 Sample Data Layouts

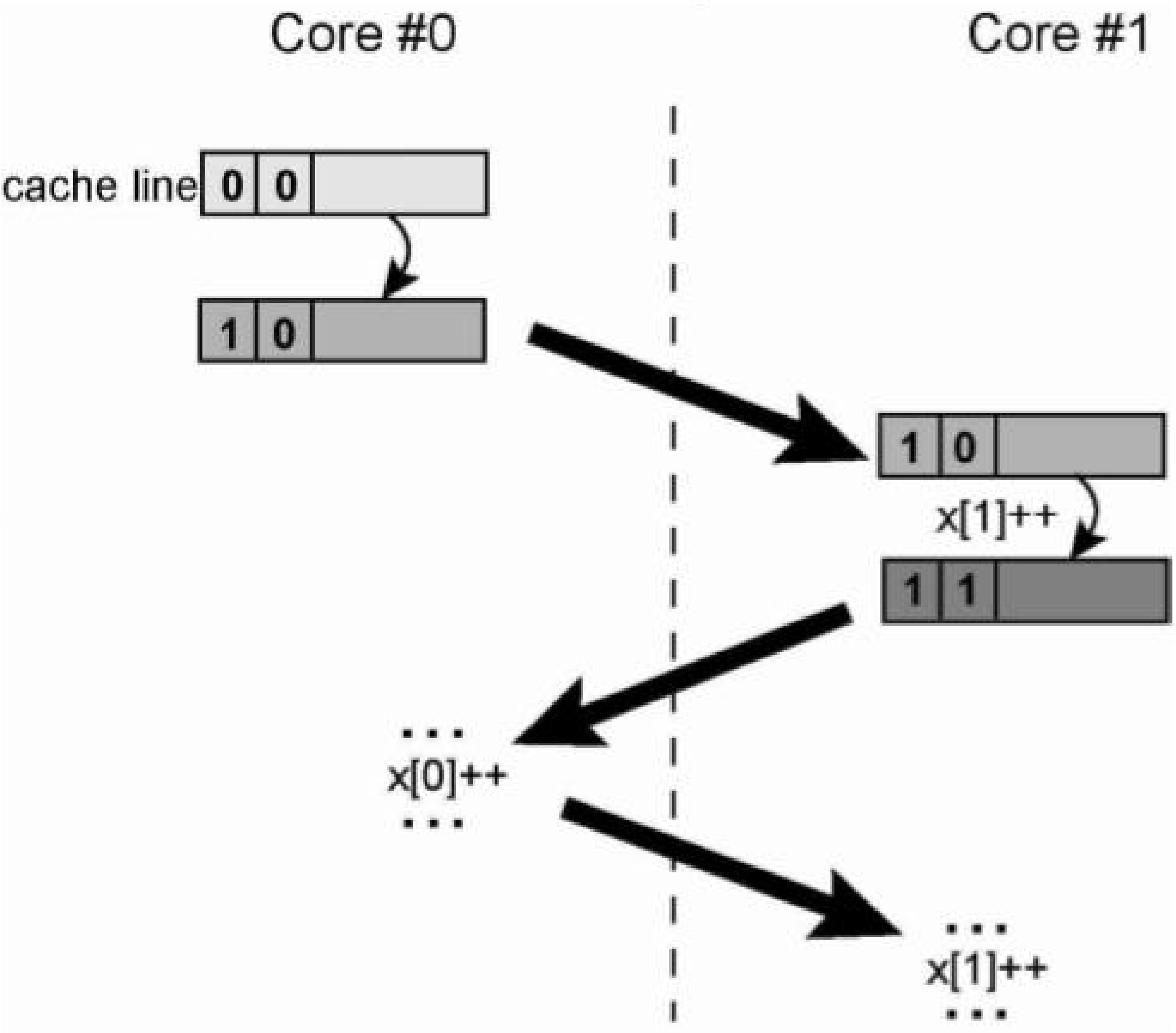
- 1D Block
- 1D Cyclic Column
- 1D Block-Cyclic
- 2D Block-Cyclic
- Which one creates contention?



Multithreading “Gotchas”

- False Sharing
  - Data moves back and forth between different core's caches.
    - Associativity conflicts
    - Improper alignment
- Invalidations
  - Two threads writing the same location causing the value to be flushed.
- Synchronization
  - Locking, barriers, etc.

False Sharing



http://isdlibrary.intel-dispatch.com/isd/1588/MC_Excerpt.pdf

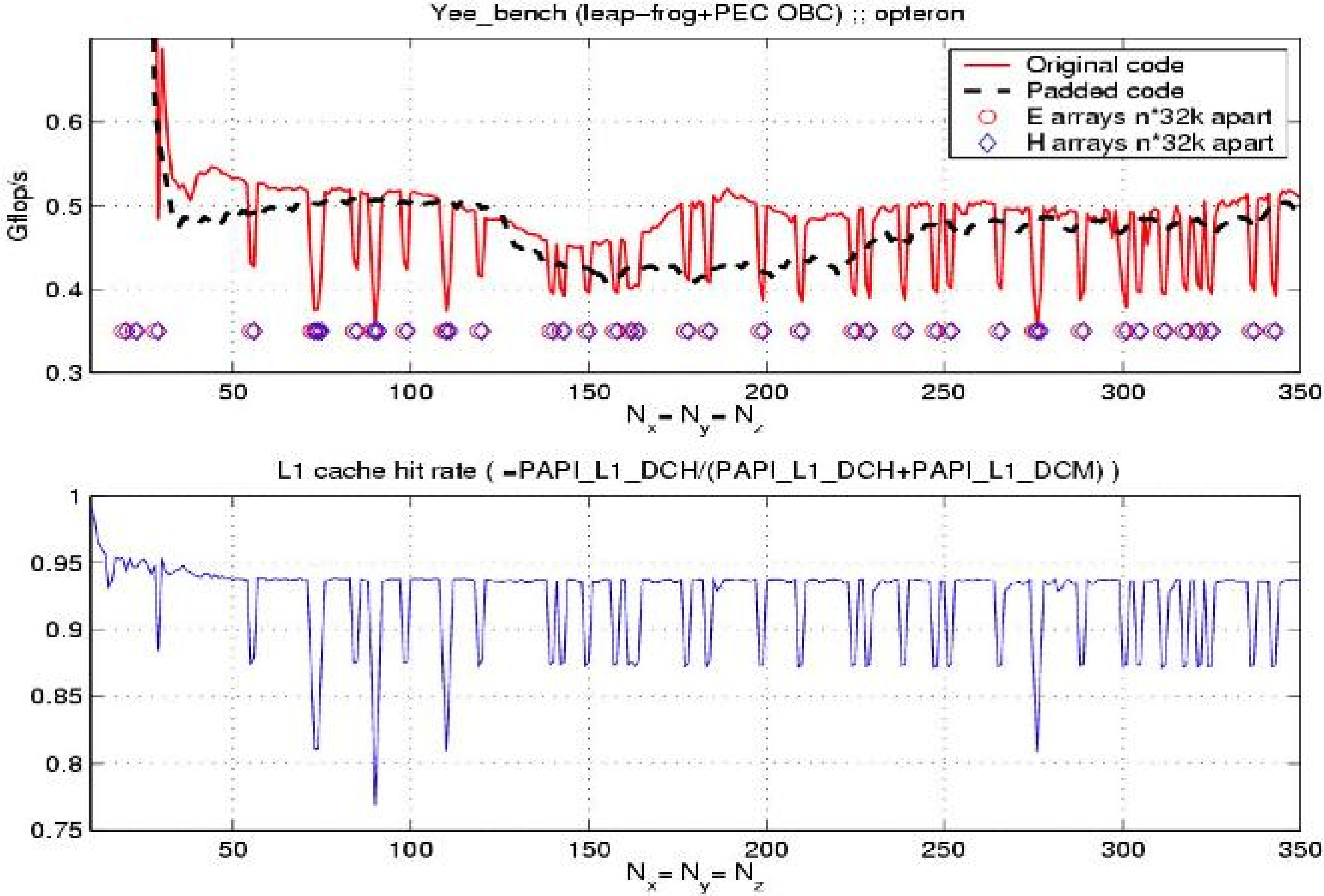
Types of False Sharing

- Read-Write contention
  - One core writes cache line, another one reads it
- Write-Write contention
  - Many cores writing to same cache line
- Next example has both types.
- Read-Read is perfectly OK!

Loop Structure

```
do k=1,nz            ! The magnetic field update
  do j=1,ny          ! Electric field update is very similar.
    do i=1,nx
      Hx(i,j,k) = Hx(i,j,k) +                               &
                (  (Ey(i,j,k+1)-Ey(i,j  ,k))*Cbdz +         &
                   (Ez(i,j,k  )-Ez(i,j+1,k))*Cbdy  )
      Hy(i,j,k) = Hy(i,j,k) +                               &
                (  (Ez(i+1,j,k)-Ez(i,j,k  ))*Cbdx +         &
                   (Ex(i  ,j,k)-Ex(i,j,k+1))*Cbdz  )
      Hz(i,j,k) = Hz(i,j,k) +                               &
                (  (Ex(i,j+1,k)-Ex(i  ,j,k))*Cbdy +         &
                   (Ey(i,j  ,k)-Ey(i+1,j,k))*Cbdx  )
    end do
  end do
end do
```

Memory Contention & OpenMP



Improved Padding

```
Hx(1:nx   +padHx(1),1:ny   +padHx(2),1:nz   +padHx(3))
Hy(1:nx   +padHy(1),1:ny   +padHy(2),1:nz   +padHy(3))
Hz(1:nx   +padHz(1),1:ny   +padHz(2),1:nz   +padHz(3))
Ex(1:nx+1 +padEx(1),1:ny+1 +padEx(2),1:nz+1 +padEx(3))
Ey(1:nx+1 +padEy(1),1:ny+1 +padEy(2),1:nz+1 +padEy(3))
Ez(1:nx+1 +padEz(1),1:ny+1 +padEz(2),1:nz+1 +padEz(3))
```

Managing Memory Contention

- Make sure shared (even read only) data is cacheline aligned.
- Use thread private variables to compute results, then merge to shared arrays.
- With OpenMP: use **`default(none)`** in your parallel clauses
  - Shared is default type, could cause contention!

Cache Blocking for Multicore

- Generally, block for the largest **non-shared** cache.
  - L2 on Nehalem.
- Depending on the speed difference and amount of work per iteration, L1 may be better.
- Never block for the shared cache size.

MultiCore and Locking

- Access to shared data structures & critical sections must be protected (and ordered).
  - Sometimes even if access is atomic.
- Numerous ways to accomplish this.
  - Unix has (horribly slow) semaphores.
  - Pthreads has rwlocks, mutexes and condition variables.
  - OpenMP has explicit locks and directives.

Locking

- Make locked regions as “small” as possible.
- Time
  - Locks should not be taken around any primitive that does not execute deterministically.
- Space
  - Instructions - do minimal work while performing the lock.
  - Data - lock items, not entire structures.
- Remember that locks always ping-pong in the cache.

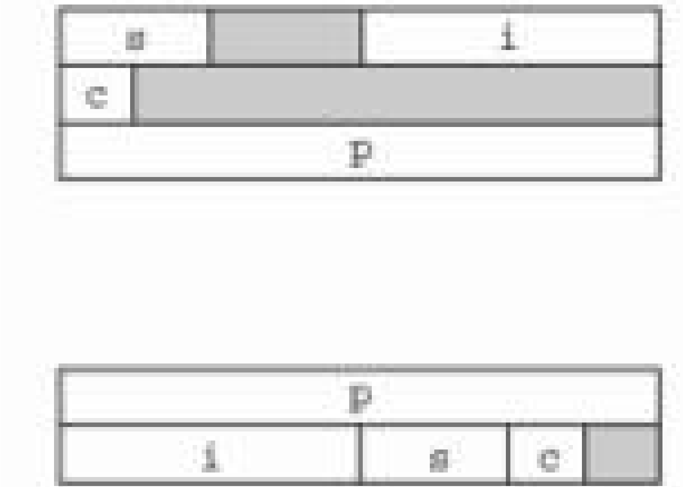
Locking Example

- Consider a multithreaded server where all threads read from the same socket into a shared FIFO for processing.
  - Lock the FIFO, read into it, increment and unlock.
- Trade memory for performance here.
  - We could lock each buffer in FIFO, but that would cause gaps.
  - Instead, make temporary message buffers which we copy into the FIFO when full. We only lock FIFO when the data is ready!.

Structure Packing Again

- Our single core optimization can be terrible for multicore.
  - Because we have increased our memory bandwidth!
- So here, pack from largest to smallest.
  - Some compilers have **`#pragma pack`**

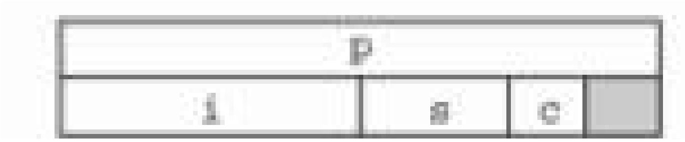
```
struct {
  short s;
  int i;
  char c;
  void *p;
}
```



```
struct {
  void *p;
  int i;
  short s;
  char c;
 }
```

Structure Packing, Padding and Locking

- What if we are locking each structure?
- What happens after lock is acquired?
- What if structures are allocated together?
- Usage dictates method, what we will be accessing and when.



```
struct {
  unsigned long lock;
  void *next;
  void *p;
  int i;
  short s;
  char c;
  unsigned long pad[5];
}
```

Global Data and Threads

- We know there is nothing wrong with **shared read-only** data.
- Unless it happens to be in the same cache line as something that gets written.
  - That line gets invalidated and must be reloaded.
- Solution is to pad, align or use a **thread specific variable.**

```
unsigned long read_write_a;
unsigned long read_only__b;
unsigned long read_write_c;
```

Thread Specific Data

- a.k.a Thread Local Storage: give each thread a private copy.
  - Great way to reduce contention.
  - Only most systems, this is very fast.
  - Variants exist in C and C++: the **__thread** keyword.
- When a thread dies (join, exit), it's gone!

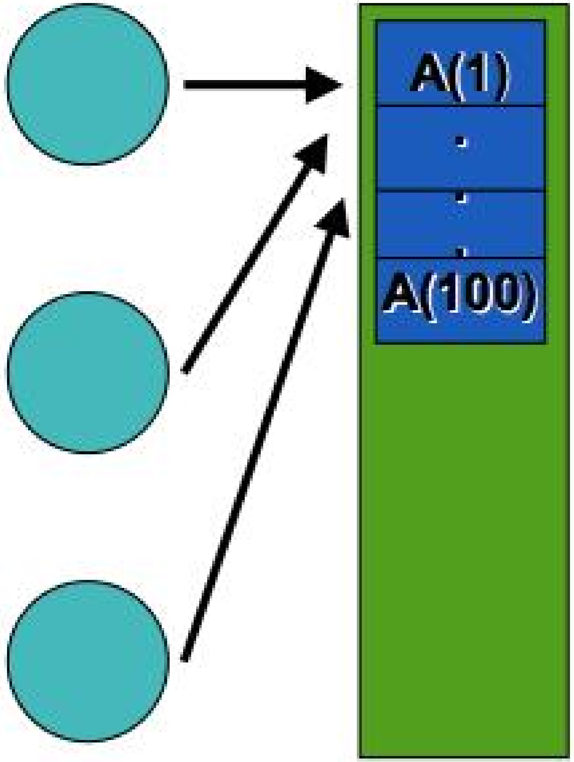
```
int i_first_val = 101;
__thread int i = i_first_val;
extern __thread struct state s;
static __thread char *p;
```

NUMA, Threading and First Touch Placement

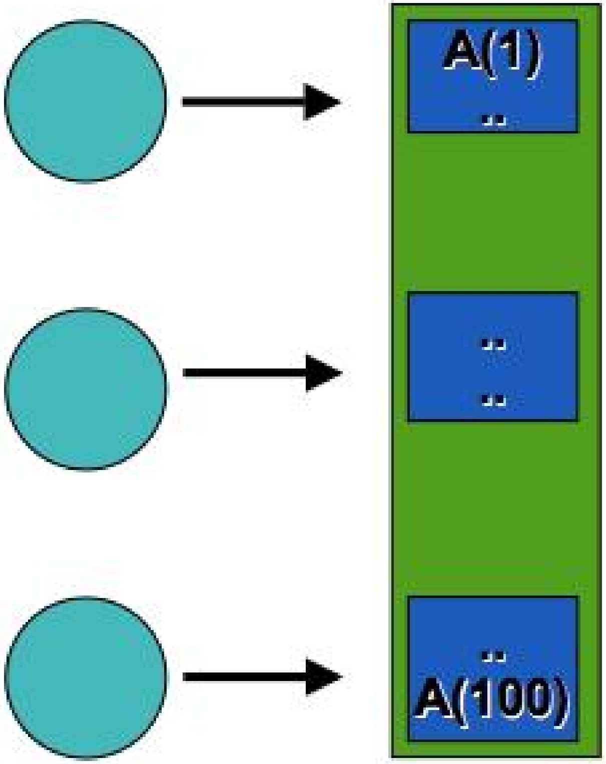
- The OS uses a **first touch** policy to place physical pages.
  - The first time it is written, it is placed.
- This means you want to parallelize your initialization!

NUMA, Threading and First Touch Placement

```
DO I = 1, N
   A(I) = 0
ENDDO
```



```
$!OMP DO
DO I = 1, N
   A(I) = 0
ENDDO
```



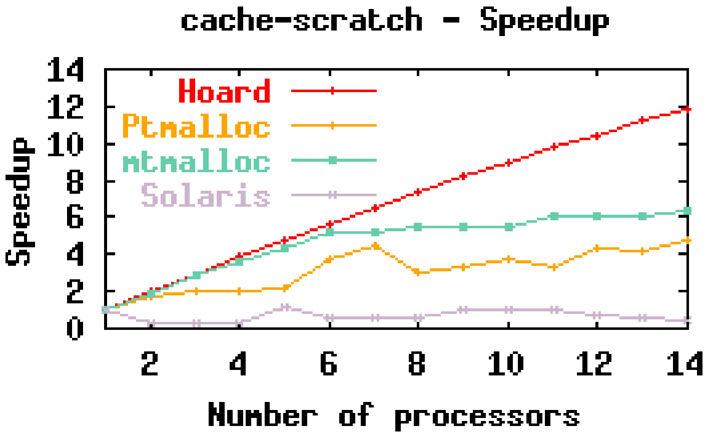
Thanks to Matthias Müller and HLRS

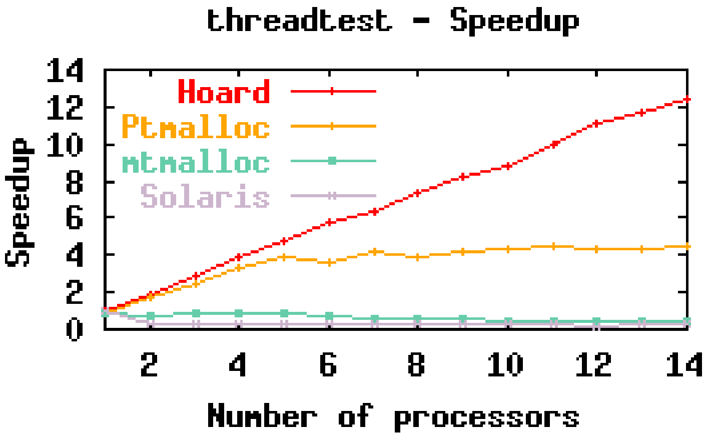
Multicore and Memory Allocation

- Many memory allocators do their best to align buffers to page boundaries.
  - This can be very bad for multicore due to false sharing, especially for caches with low associativity.
  - Be wary of your F90 allocate or your malloc/new.
  - 3rd party OS replacements are available
- Many malloc/new/free implementations are not often scalable for many-core.

The Hoard Memory Allocator

- A fast, scalable, drop-in replacement memory allocator that addresses:
  - Contention
  - False sharing
  - Per-CPU overhead





Mapping Threads to Processors

- How should you run your code?
  - It depends on what the code does.
- There is generally a sweet spot for M threads on N cores of a single socket. (M<N)
  - Usually depends on:
    - How tightly synchronized and balanced computation is
    - Memory bandwidth requirements
    - I/O and Comm traffic
- Oversubscription (more threads than cores) is usually never a good thing unless...

OS Scheduling and Threads

- Threads can bounce from core to core.
- You do have some control over this.
  - Run-time on the command line
  - Or directly inside the code
- But you cannot prevent Linux from scheduling something else onto your CPU.
  - Unless you boot the kernel with special options (isolcpus) or the massage system a bit.
- On a NUMA system, this can be really bad.

OS Scheduling and Threads

- For serial and threaded codes...

Print affinity mask of process PID 24732

```
> taskset -p 24732
pid 24732's current affinity mask: f
```

Print CPU list of process PID 24732

```
> taskset -c -p 4695
pid 24732's current affinity mask: 0-3
```

Set running process to only use CPU's 1 and 2

```
> taskset -c -p 1,2 4695
pid 4695's current affinity list: 0-3
pid 4695's new affinity list: 1,2
```

Launch bash shell with all CPU's to choose from

```
> taskset 0xffffffff /bin/bash
```

Launch bash shell with CPU's to choose from

```
> taskset -c 0-3 /bin/bash
```

OS Scheduling and Threads

- Even MPI supports this now...

Tell OpenMPI to bind each process

```
> mpirun –mca mpi_paffinity_alone 1 -np ...
```

Tell SLURM to bind each task to a core/socket

```
> srun --ntasks-per-core=N --ntasks-per-socket=M ...
```

More advanced SLURM binding 8 ranks, 4 nodes, 2 per socket, 1 per core (-B S[:C[:T]])

```
> srun -n 8 -N 4 -B 2:1 ...
```

Even more advanced SLURM binding

```
> srun --cpu_bind=cores –cpu_bind=verbose ...
> srun --cpu_bind=map_cpu:0,2,3 –cpu_bind=verbose ...
> srun --cpu_bind=help –cpu_bind=verbose ...
```

Types of Load Balancing

- Static
  - Data/tasks are split amongst processors for duration of execution.
  - Problem: How do we choose an efficient mapping?
- Dynamic
  - Work is performed when resources become available
    - How much work and when?
  - Problem: Requires periodic synchronization and data exchange

Measuring OpenMP Overhead

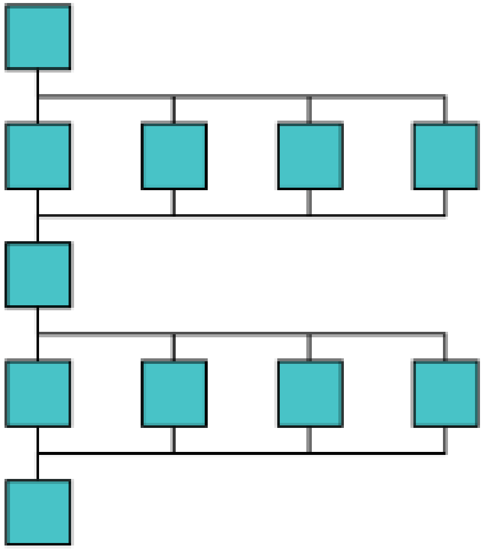
- OMP_NUM_THREADS sets the number of threads to use.
  - If not set, it defaults to the number of cores in a system. (As reported by /proc/cpuinfo on Linux, Hyperthreaders beware...)
- Set this to 1,2,etc. and time regions of your code.
- Time without OpenMP as well.

Managing Parallel Overhead in OpenMP

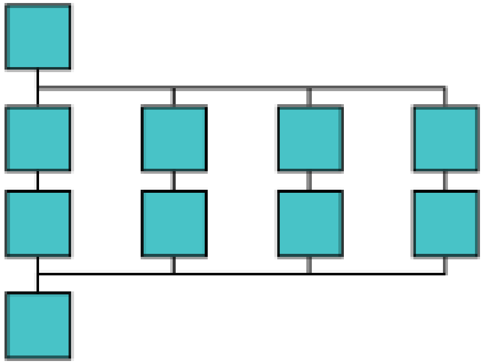
- Don't parallelize all your loops.
  - Just the ones that matter.
- Use conditional parallelism.
- Specify chunk size to each loop.
  - As big as possible.
- Make sure the compiler can unroll the loop.
- Merge parallel regions.
- Avoid barriers with NOWAIT.

OpenMP Overhead and Parallel Regions

```
#pragma omp parallel for
for () { ... }
#pragma omp parallel for
for () { ... }
```



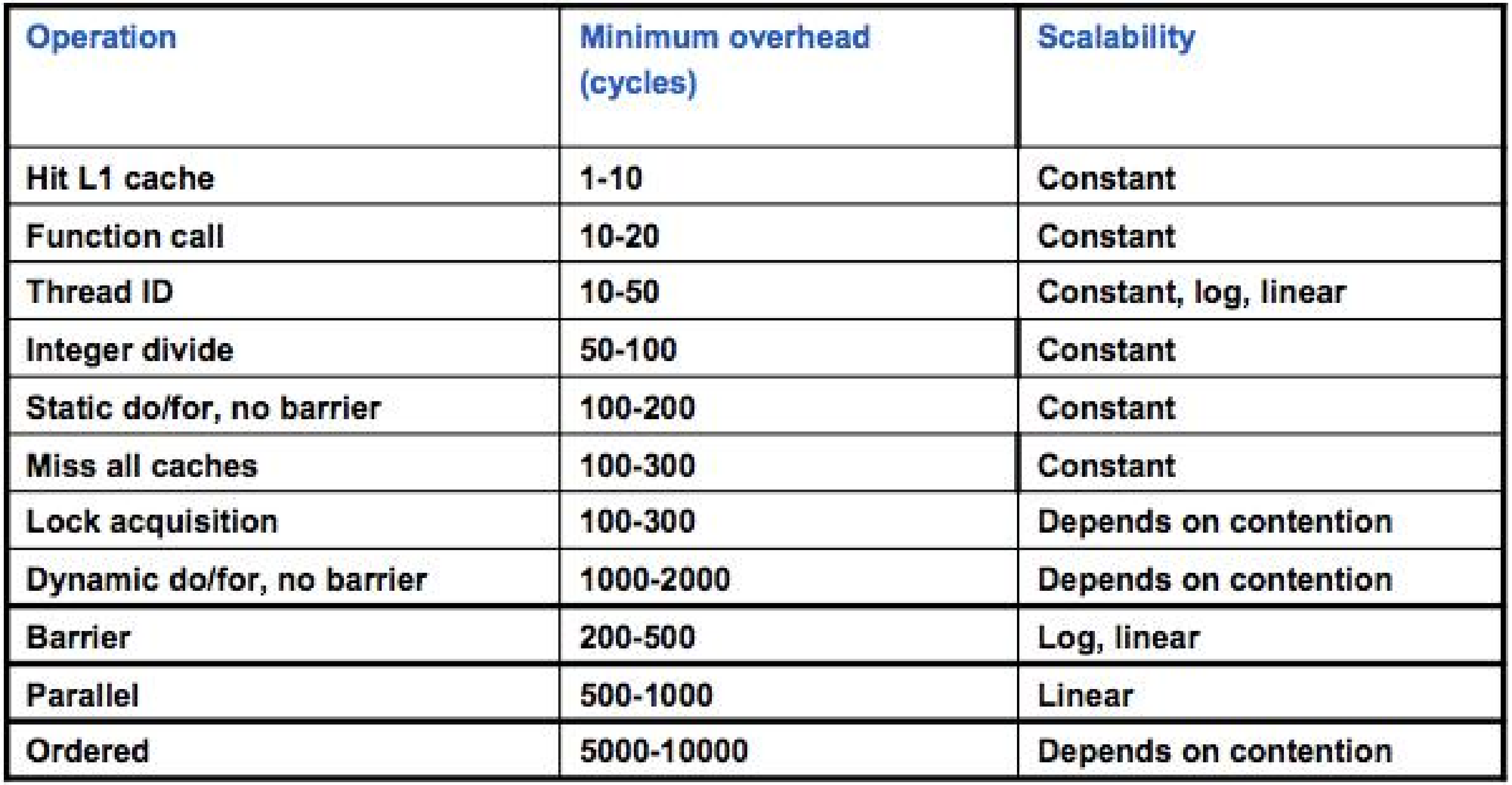
```
#pragma omp parallel
{
#pragma omp for
for () { ... }
#pragma omp for
for () { ... }
}
```



Thanks to Matthias Müller and HLRS

Rough Overheads of OpenMP

- These are **very** approximate.



Operation	Minimum overhead (cycles)	Scalability
Hit L1 cache	1-10	Constant
Function call	10-20	Constant
Thread ID	10-50	Constant, log, linear
Integer divide	50-100	Constant
Static do/for, no barrier	100-200	Constant
Miss all caches	100-300	Constant
Lock acquisition	100-300	Depends on contention
Dynamic do/for, no barrier	1000-2000	Depends on contention
Barrier	200-500	Log, linear
Parallel	500-1000	Linear
Ordered	5000-10000	Depends on contention

Thanks to Matthias Müller and HLRS

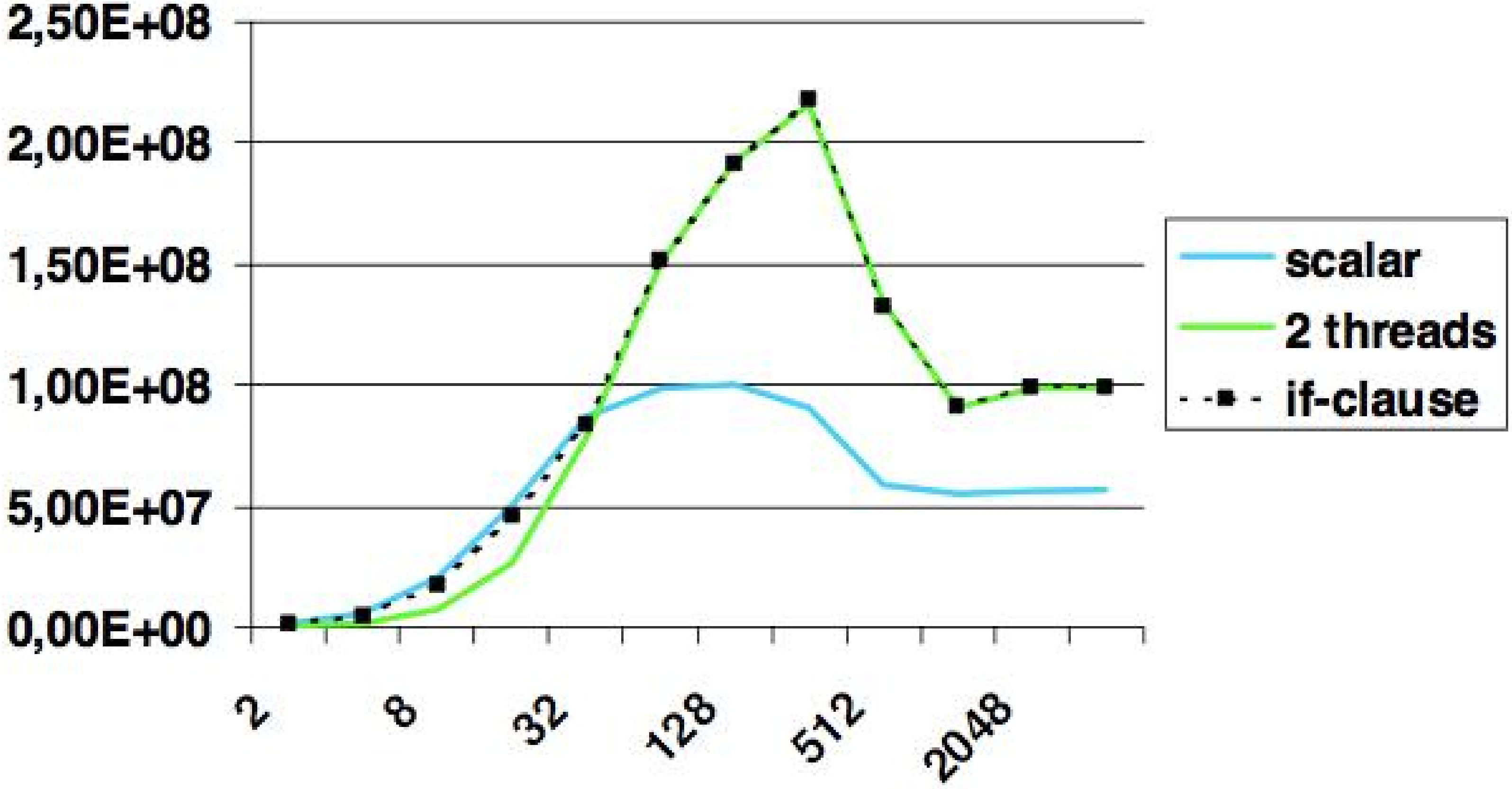
OpenMP Conditional Parallelism

- Execute in parallel if expression evaluates to true.
- Very powerful technique for mitigating parallel overhead.
  - `parallel if(expression)`
  - `parallel for if(expression)`
- Expression should evaluate to 1/yes or 0/no.

OpenMP Conditional Parallelism

```
for( i=0; i<n; i++ )
#pragma omp parallel for if ( n-i > 100 )
  for( j=i+1; j<n; j++ )
    for( k=i+1; k<n; k++ )
      a[j][k] = a[j][k] -a[i][k]*a[i][j] / a[j][j]
```

Performance of Conditional Parallelism



Thanks to Matthias Müller and HLRS

OpenMP Thread Specific

- **`firstprivate(list)`**
  - All copies get value in master at beginning.
- **`lastprivate(list)`**
  - All copies get value in last iteration/section.
- **`threadprivate(list)`**
  - Data is global data, but private in parallel regions.
    - common blocks etc. Use COPYIN or undefined.

OpenMP and Barriers

- Most constructs have an implicit barrier and flush at the end.
  - **`do, for, sections, workshare, single`**
  - We must work to limit when this happens.
- The **`NOWAIT`** clause eliminates the barrier, then insert a **`barrier`**barrier and/or **`flush`** youself.
- Also, you can use **`master`** instead of **`single`**.
  - But then thread 0 will do the work, so it better be ready.

OpenMP and Critical Sections

- If you can't use a **`reduction`** to update a **`shared`** variable and you need to use **`critical`**:
  - Only thread at a time executing the code.
- But it's better to use **`atomic`**
  - This will take advantage of special instructions instead of using locking.

Barrier Removal Exercise

- What's could be wrong with the below advice?

Replace

```
#pragma omp for
for(i=0; i<size; i++)
  a[i] = 1.0/a[i];
#pragma omp for
for(i=0; i<size; i++)
  b[i] = b[i]*2.0
```

with

```
#pragma omp for
for(i=0; i<size; i++) {
  a[i] = 1.0/a[i];
  b[i] = b[i]*2.0;
}
```

- Good: But we've reduced overhead and increased work per iteration.
- Bad: We're increasing memory bandwidth and cache pollution. (No data reused)
- This is better for multicore:

```
#pragma omp for nowait
for(i=0; i<size; i++)
  a[i] = 1.0/a[i];
#pragma omp for
for(i=0; i<size; i++)
  b[i] = b[i]*2.0
```

Thanks to Matthias Müller and HLRS

OpenMP Reduction

- OpenMP has special knowledge of **`reduction`** operations.
  - A **`shared`** variable that is updated by all threads must be updated atomically.
- OpenMP has a shortcut: **`reduction(op:var)`**
- You tell OpenMP how to combine the data.

Reduction By Hand

```
#pragma omp parallel for private( privIndx, privDbl )
for ( i = 0; i < arraySize; i++ )
{
  for ( privIndx = 0; privIndx < 16; privIndx++ )
    {
     privDbl = ( (double)privIndx ) / 16;
     y[i] = sin(exp(cos(-exp(sin(x[i]))))) +
            cos(privDbl);

          /* Here, each thread reads globalCount
             add 1 to the value, and write the
             new value back to globalCount.   */
#pragma omp critical
     { globalCount = globalCount + 1; }
    }
 }
```

Reduction

```
#pragma omp parallel for private( privIndx, privDbl ) \
reduction( + : globalCount )
for ( i = 0; i < arraySize; i++ )
{
  for ( privIndx = 0; privIndx < 16; privIndx++ )
    {
     privDbl = ( (double)privIndx ) / 16;
     y[i] = sin(exp(cos(-exp(sin(x[i]))))) +
            cos(privDbl);

          /* Here, each thread reads globalCount
             add 1 to the value, and write the
             new value back to globalCount.   */
     globalCount = globalCount + 1;
    }
 }
```

When Approaching a (nasty) Loop Nest with OpenMP

- If F90, rewrite with loops instead of (:).
  - Make everything explicit.
- Rewrite a few versions, unrolling each level individually. Look for opportunities to:
  - Re-use data (cache)
  - Reduce memory bandwidth.
  - Move temps into variables (register).
    - Stage shared data in per thread privates or to use reductions.
  - Make work per iteration as large as possible.

OpenMP Scheduling

- Controls the allocation of work to threads.
  - A form of load balancing.
- By default, OpenMP will allocate a small fixed number of iterations to each thread.
- This can be changed at compile time or run-time.
  - **`SCHEDULE(type)`** clause
  - **`runtime`** means refer to **`OMP_SCHEDULE`** env. var.

```
OMP_SCHEDULE=dynamic ./a.out
```

OpenMP Scheduling

- **`$OMP PARALLEL DO SCHEDULE(type)`**
- **`#pragma parallel for schedule(type)`**
  - **`STATIC[,size]`** – default
  - **`DYNAMIC[,size]`** – allocate iterations at runtime
  - **`GUIDED[,size]`** – start with big chunks, end with small chunks
  - **`RUNTIME`**
- For **`DYNAMIC`** and **`GUIDED`**, default size is 1!

MPI Tips

- Overlap comm. and compute
  - Ideally a background thread can send the data while this thread can continue.
  - **`MPI_ISEND, MPI_IRECV, MPI_ISENDRECV, MPI_IRSEND, MPI_WAITxxx, MPI_TESTxxxx`**
- Use native data types.
- Send big messages not small ones.
- Make sure receiver arrives early.
- Minimize collectives.

MPI Tips 2

- Avoid wildcard receives
- Attempt to align application buffers to (at least) 8 bytes
- Avoid data translation and derived data types.
- Always think about overlapping comm and compute

Performance Analysis Tools

Performance Analysis

- What's really meaningful?
  - Wall Clock time
- MFLOPS, MIPS, etc are useless.
  - What are comparing it to? Peak? Ask your vendor to send you a code that performs at peak.
- For purposes of optimization, we need data over a range of data sets, problem sizes and number of nodes.

Comparisons

- For the purposes of comparing performance data, time is the best place to start.
  - Unless you are completely aware of architecture, compiler, run-time systems, etc...
- Hennessey and Patterson: Fallacies of Performance
  - Synthetic benchmarks predict performance of real programs
  - Peak performance tracks observed performance

Performance Measurement Methods

- Instrumentation
  - Tracing
  - Aggregate
- Sampling
  - IP Profiling, stack-walking
- Simulation
  - Instruction cracking and emulation

The Problem with Tracing

- Tracing generates a record with a timestamp for every event, say function invocation. This presents numerous problems.
  - Measurement pollution
  - Data management
  - Visualization
- Cure is worse than the disease.
- Tracing often reserved for the worst and most intermittent problems.

Aggregated Profiling

- By using simple start, stop and accumulate points in the code, a relatively complete picture of the overall execution can be obtained.
- This loses temporal performance information.
  - i.e. problem X started at time Y
- However, significant problems still 'bubble' to the top of the overall profile.
  - If it doesn't show there, it's not important.

Statistical Profiling

- Upon defined periodic events, record where in the program the CPU is.
- Gather data into a histogram, the shape of which approaches the actual profile over time.
- Periodic events can be clock ticks or other events based on hardware performance counters, like cache misses.

Understanding Timers

- Real time, Wall Clock time: A measure of time that doesn't stop, as when using a stop watch.
- User time: Time when the CPU is executing your process and is executing your code (not OS code)
- System time: Time when the CPU is executing your process and is executing OS code on your behalf.
- CPU utilization is usually (U + S)/R

Timing Utilities

- Linux **`/usr/bin/time`**
  - Wall time
  - User time
  - System time
    - Above two are added up for all threads
  - Minor/major page faults.
- This is different than 'time' from tcsh.

Wallclock Time

- Usually accurate to a few microseconds.
- C
  - **`gettimeofday()`**
  - **`clock_gettime()`**
- Fortran
  - **`second()`**
  - **`etime()`**
- Both
  - **`MPI_Wtime()`**
  - **`OMP_GET_WTIME()`**

CPU Time

- Can be system, user or both.
  - Usually summed over all threads.
  - Not nearly as accurate as wallclock time.
- C
  - **`clock()`**
  - **`getrusage()`**
  - **`clock_gettime()`**
  - **`times()`**
- Fortran
  - **`dtime()`**

Hardware Performance Analysis

- No longer can we easily understand the performance of a code segment.
  - Out of order execution
  - Branch prediction
  - Prefetching
  - Register renaming
- A measure of wallclock is not enough to point to the culprit. We need to know what's happening “under the hood”.

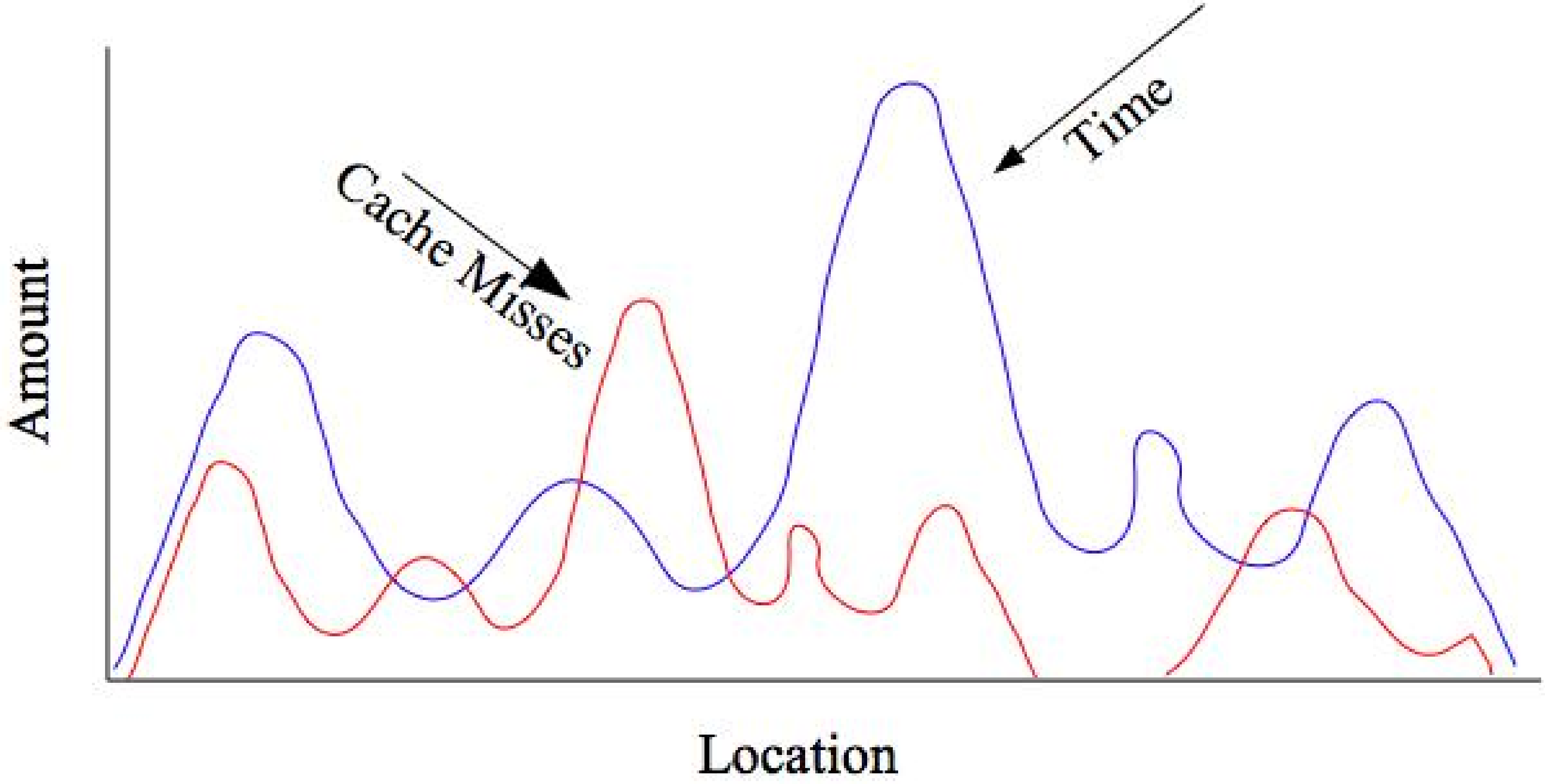
Hardware Performance Counters

- On/off chip registers that count hardware events
  - Often 100's of different events, specialized to the processor, usually just a few registers to count on.
- OS support accumulates counts into 64 bit quantities that run only when process is running.
  - User, kernel and interrupt modes can be measured separately
  - Can count aggregate or use them as sampling triggers

Sample Performance Counter Events

- Cycles
- Instructions
- Floating point ops
- Branches mispredicted
- Cycles stalled on memory
- Cache lines invalidated
- Loads, Stores
- Ratios of these counters are indicative of performance problems.

Statistical Profiling 2



Hardware Metrics for Multicore

- Absolutely! But the metrics are different for each processor.
  - Load/store to Cache miss ratio
    - On a loop that should not miss, misses mean contention.
- Cache state transitions
  - You can actually count transitions to E and I on some platforms.
- Interprocessor traffic
  - Can isolate offending processor/thread.

PAPI

- Performance Application Programming Interface
- A standardized, portable and efficient API to access the hardware performance counters.
- Goal is to facilitate the development of cross-platform optimization tools.
- Patching kernel is required.
  - Stable and supported patches. (perfctr & perfmon)
  - Many HPC systems have already been patched.

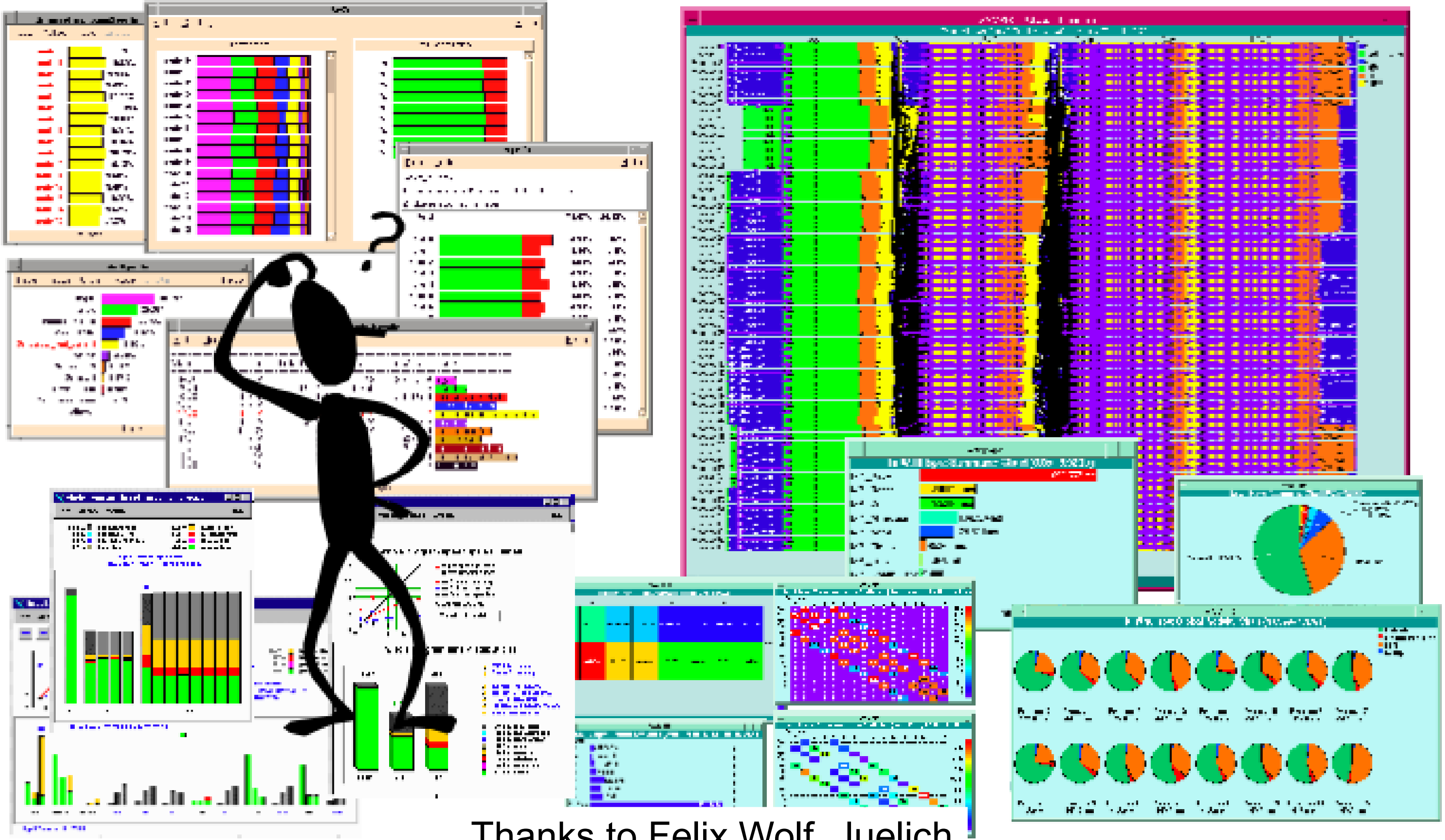
PAPI Events

- Performance counters are measured in terms of events
  - Symbol names for something to count
  - Events have different names/meanings for different vendors/processors/revisions etc
  - Some native events are mapped to general names in PAPI
    - And all the problems associated with such abstractions
- PAPI supports derived events

O.S. Linux Performance Tools

- From the desktop world, most are familiar with:
  - gprof, valgrind, oprofile
- Linux performance tools are actually well established:
  - Most are not 'production' quality, lacking proper
    - Testing, Documentation, Integration
  - But some are better than others, all can be useful in the proper situations

The Right Tool for the Job



Thanks to Felix Wolf, Juelich

Issues to Consider

- Usage
  - GUI
  - ASCII
  - Simplicity vs…
- Collection
  - Instrumentation
  - Direct vs Indirect
  - Tracing
- Performance Data
  - MPI, Pthreads, OpenMP
  - Libraries
  - Processor
  - I/O
- Experiment management
- Visualization

Tools

- ompP
- mpiP
- HPCToolkit
- PerfSuite
- PapiEx
- GPTL
- pfmon
- TAU
- Scalasca
- valgrind
- gprof
- Non-OS
  - Vampir
  - SlowSpotter

ompP: The OpenMP Profiler

- Provides easy to read reports at end of execution.
  - Based on source code instrumentation
- Report on each OpenMP primitive
  - Flat Profiles
  - Callgraph Profiles
  - Hardware counter values
- Overhead Analysis
- Scalability Analysis

ompP: Usage

- Recompile code with wrapper.
  - Works on all compilers: source to source.
  - Optional: hand-instrument user regions.
- Set environment variables if necessary.
- Run and read report!

ompP: Flat Region Profile

```
#pragma omp parallel
{
 #pragma omp critical
 {
   sleep(1)
 }
}
```

```
R00002 main.c (34-37) (default) CRITICAL
 TID     execT     execC     bodyT    enterT     exitT  PAPI_TOT_INS
   0      3.00         1      1.00      2.00      0.00          1595
   1      1.00         1      1.00      0.00      0.00          6347
   2      2.00         1      1.00      1.00      0.00          1595
   3      4.00         1      1.00      3.00      0.00          1595
 SUM     10.01         4      4.00      6.00      0.00         11132
```

- Components:
  - Region number
  - Source code location and region type
  - Timing data and execution counts, **depending on the particular construct**
  - One line per thread, last line sums over all threads
  - Hardware counter data (if PAPI is available and HW counters are selected)
  - Data is exact (measured, not based on sampling)

ompP: Call Graphs

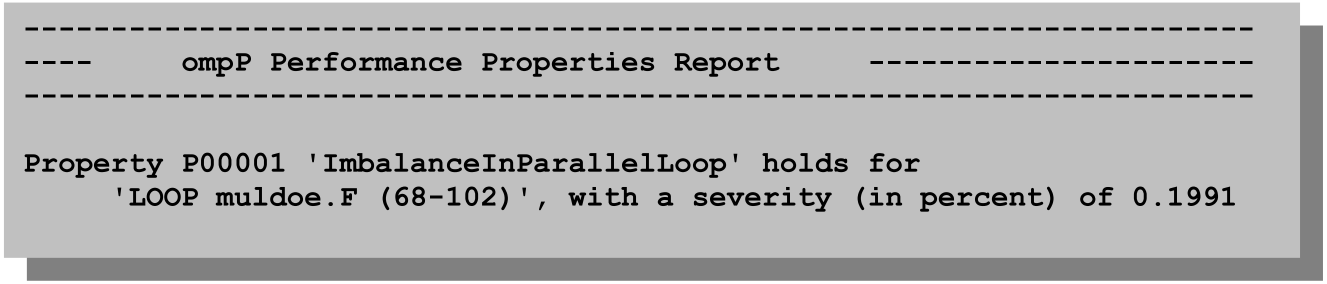
```
   Incl. CPU time
 32.22 (100.0%)              [APP 4 threads]
 32.06 (99.50%)  PARALLEL  +-R00004 main.c (42-46)
 10.02 (31.10%)   USERREG     |-R00001 main.c (19-21) ('foo1')
 10.02 (31.10%)  CRITICAL     |  +-R00003 main.c (33-36) (unnamed)
 16.03 (49.74%)   USERREG     +-R00002 main.c (26-28) ('foo2')
 16.03 (49.74%)  CRITICAL        +-R00003 main.c (33-36) (unnamed)
```

```
[*00] critical.ia64.ompp
[+01] R00004 main.c (42-46) PARALLEL
[+02] R00001 main.c (19-21) ('foo1') USER REGION
 TID    execT/I    execT/E      execC
   0       1.00       0.00          1
   1       3.00       0.00          1
   2       2.00       0.00          1
   3       4.00       0.00          1
 SUM      10.01       0.00          4

[*00] critical.ia64.ompp
[+01] R00004 main.c (42-46) PARALLEL
[+02] R00001 main.c (19-21) ('foo1') USER REGION
[=03] R00003 main.c (33-36) (unnamed) CRITICAL
 TID      execT      execC    bodyT/I    bodyT/E     enterT      exitT
   0       1.00          1       1.00       1.00       0.00       0.00
   1       3.00          1       1.00       1.00       2.00       0.00
   2       2.00          1       1.00       1.00       1.00       0.00
   3       4.00          1       1.00       1.00       3.00       0.00
 SUM      10.01          4       4.00       4.00       6.00       0.00
```

ompP: Overhead Analysis

```
Total runtime (wallclock)   : 172.64 sec [32 threads]
Number of parallel regions  : 12
Parallel coverage           : 134.83 sec (78.10%)

Parallel regions sorted by wallclock time:
          Type                            Location      Wallclock (%)
R00011  PARALL                   mgrid.F (360-384)      55.75 (32.29)
R00019  PARALL                   mgrid.F (403-427)      23.02 (13.34)
R00009  PARALL                   mgrid.F (204-217)      11.94 ( 6.92)
...
                                               SUM     134.83 (78.10)

Overheads wrt. each individual parallel region:
          Total        Ovhds (%)  =   Synch  (%)  +  Imbal   (%)  +   Limpar (%)   +    Mgmt (%)
R00011  1783.95   337.26 (18.91)    0.00 ( 0.00)  305.75 (17.14)    0.00 ( 0.00)   31.51 ( 1.77)
R00019   736.80   129.95 (17.64)    0.00 ( 0.00)  104.28 (14.15)    0.00 ( 0.00)   25.66 ( 3.48)
R00009   382.15   183.14 (47.92)    0.00 ( 0.00)   96.47 (25.24)    0.00 ( 0.00)   86.67 (22.68)
R00015   276.11    68.85 (24.94)    0.00 ( 0.00)   51.15 (18.52)    0.00 ( 0.00)   17.70 ( 6.41)
...


Overheads wrt. whole program:
          Total        Ovhds (%)  =   Synch  (%)  +  Imbal   (%)  +   Limpar (%)   +    Mgmt (%)
R00011  1783.95   337.26 ( 6.10)    0.00 ( 0.00)  305.75 ( 5.53)    0.00 ( 0.00)   31.51 ( 0.57)
R00009   382.15   183.14 ( 3.32)    0.00 ( 0.00)   96.47 ( 1.75)    0.00 ( 0.00)   86.67 ( 1.57)
R00005   264.16   164.90 ( 2.98)    0.00 ( 0.00)   63.92 ( 1.16)    0.00 ( 0.00)  100.98 ( 1.83)
R00007   230.63   151.91 ( 2.75)    0.00 ( 0.00)   68.58 ( 1.24)    0.00 ( 0.00)   83.33 ( 1.51)
...
   SUM  4314.62  1277.89 (23.13)    0.00 ( 0.00)  872.92 (15.80)    0.00 ( 0.00)  404.97 ( 7.33)
```

ompP: Performance Properties



```
----------------------------------------------------------------------
----     ompP Performance Properties Report     ---------------------
----------------------------------------------------------------------

Property P00001 'ImbalanceInParallelLoop' holds for
     'LOOP muldoe.F (68-102)', with a severity (in percent) of 0.1991
```

Deductions by ompP about what the problem is.

WaitAtBarrier

ImbalanceInParallel[Region/Loop/Workshare/Sections]

ImbalanceDueToNotEnoughSections

InbalanceDueToUnevenSectionDistribution

CriticalSectionContention

LockContention

FrequentAtomic

InsufficienWorkInParallelLoop

UnparallelizedIn[Master/Single]Region

Valgrind

- A tool infrastructure for debugging and performance evaluation.
- Works by instruction emulation and tracing.
  - Code can run up to 100x slower.
  - But can catch errors that other tools can't.
- Many tools
  - memcheck, cachegrind, callgrind, massif, helgrind, drd
  - cachegrind is based on simulated machine model (not real hardware)

Valgrind: Helgrind

- Detects
  - Pthreads API errors
  - Deadlocks and Data races
  - Broken for GNU OpenMP
- **`valgrind -tool=helgrind <app>`**

```
Thread #1 unlocked a not-locked lock at 0x7FEFFFA90
   at 0x4C2408D: pthread_mutex_unlock (hg_intercepts.c:492)
   by 0x40073A: nearly_main (tc09_bad_unlock.c:27)
   by 0x40079B: main (tc09_bad_unlock.c:50)
  Lock at 0x7FEFFFA90 was first observed
   at 0x4C25D01: pthread_mutex_init (hg_intercepts.c:326)
   by 0x40071F: nearly_main (tc09_bad_unlock.c:23)
   by 0x40079B: main (tc09_bad_unlock.c:50)
```

Valgrind: DRD

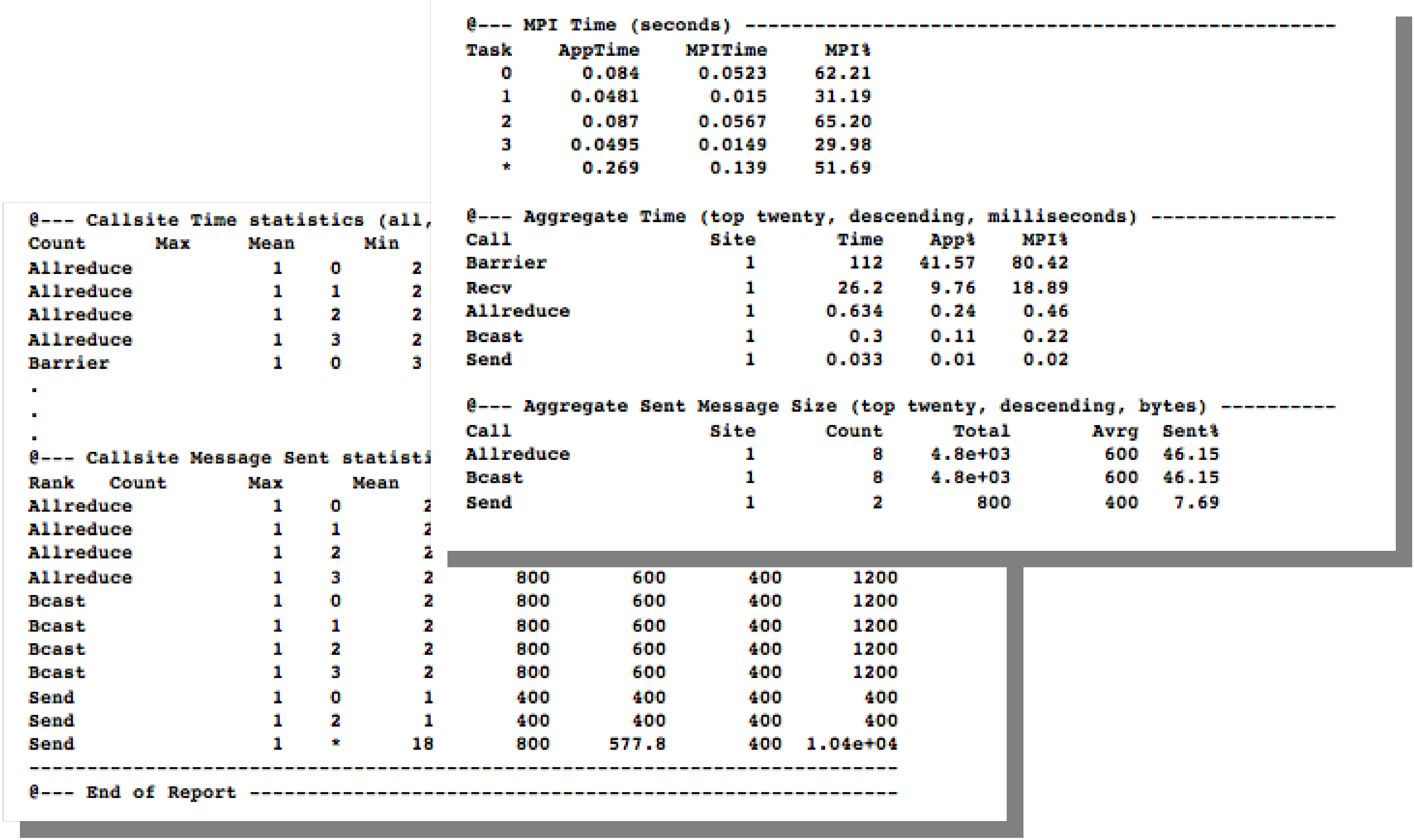
- Detects
  - Pthreads API errors
  - Deadlocks, Data races and Lock contention
  - Broken for GNU OpenMP
- **`valgrind –tool=drd –var-info=yes <app>`**

```
==10668== Acquired at:
==10668==    at 0x4C267C8: pthread_mutex_lock
(drd_pthread_intercepts.c:395)
==10668==    by 0x400D92: main (hold_lock.c:51)
==10668== Lock on mutex 0x7fefffd50 was held during 503 ms (threshold:
10 ms).
==10668==    at 0x4C26ADA: pthread_mutex_unlock
(drd_pthread_intercepts.c:441)
==10668==    by 0x400DB5: main (hold_lock.c:55)
...
```

mpiP: The MPI Profiler

- Easy to use, easy to interpret performance reports.
- mpiP performances only trace reduction and summarization.
- Compatible with all MPI's.
- No recompilation required.
  - Just relink or run with environment variable.

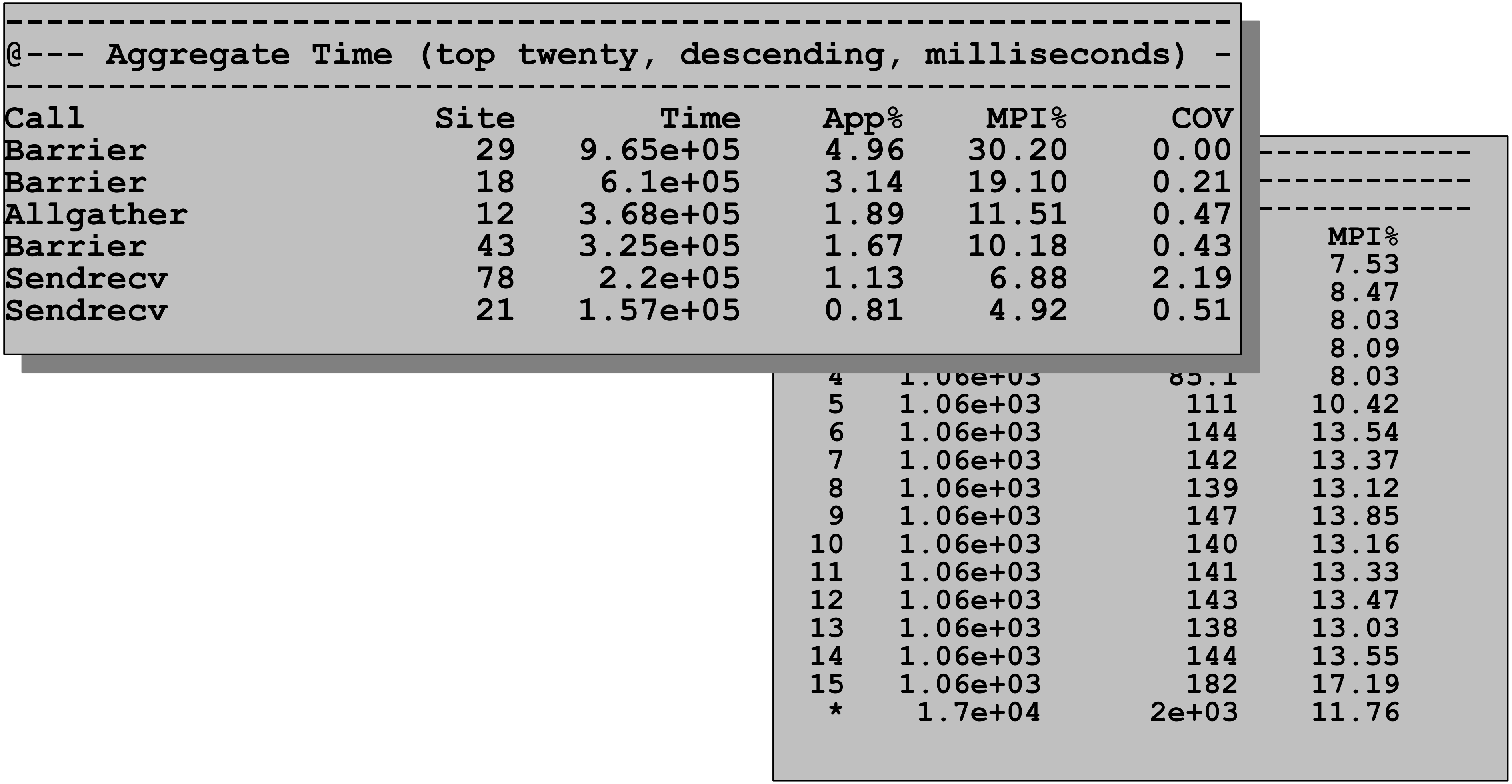
mpiP: Some output



```
@--- MPI Time (seconds) ---------------------------------------------------
Task    AppTime    MPITime     MPI%
   0      0.084     0.0523    62.21
   1     0.0481      0.015    31.19
   2      0.087     0.0567    65.20
   3     0.0495     0.0149    29.98
   *      0.269      0.139    51.69

@--- Aggregate Time (top twenty, descending, milliseconds) ----------------
Call                 Site       Time    App%    MPI%
Barrier                 1        112   41.57   80.42
Recv                    1       26.2    9.76   18.89
Allreduce               1      0.634    0.24    0.46
Bcast                   1        0.3    0.11    0.22
Send                    1      0.033    0.01    0.02

@--- Aggregate Sent Message Size (top twenty, descending, bytes) ----------
Call                 Site      Count      Total       Avrg  Sent%
Allreduce               1          8    4.8e+03        600  46.15
Bcast                   1          8    4.8e+03        600  46.15
Send                    1          2        800        400   7.69
```

```
@--- Callsite Time statistics (all,
Count      Max      Mean       Min
Allreduce           1    0      2
Allreduce           1    1      2
Allreduce           1    2      2
Allreduce           1    3      2
Barrier             1    0      3
.
.
.
@--- Callsite Message Sent statisti
Rank   Count        Max       Mean
Allreduce           1    0      2
Allreduce           1    1      2
Allreduce           1    2      2
Allreduce           1    3      2       800       600       400      1200
Bcast               1    0      2       800       600       400      1200
Bcast               1    1      2       800       600       400      1200
Bcast               1    2      2       800       600       400      1200
Bcast               1    3      2       800       600       400      1200
Send                1    0      1       400       400       400       400
Send                1    2      1       400       400       400       400
Send                1    *     18       800     577.8       400  1.04e+04
---------------------------------------------------------------------------
@--- End of Report --------------------------------------------------------
```

mpipex: Profile and Load Balance



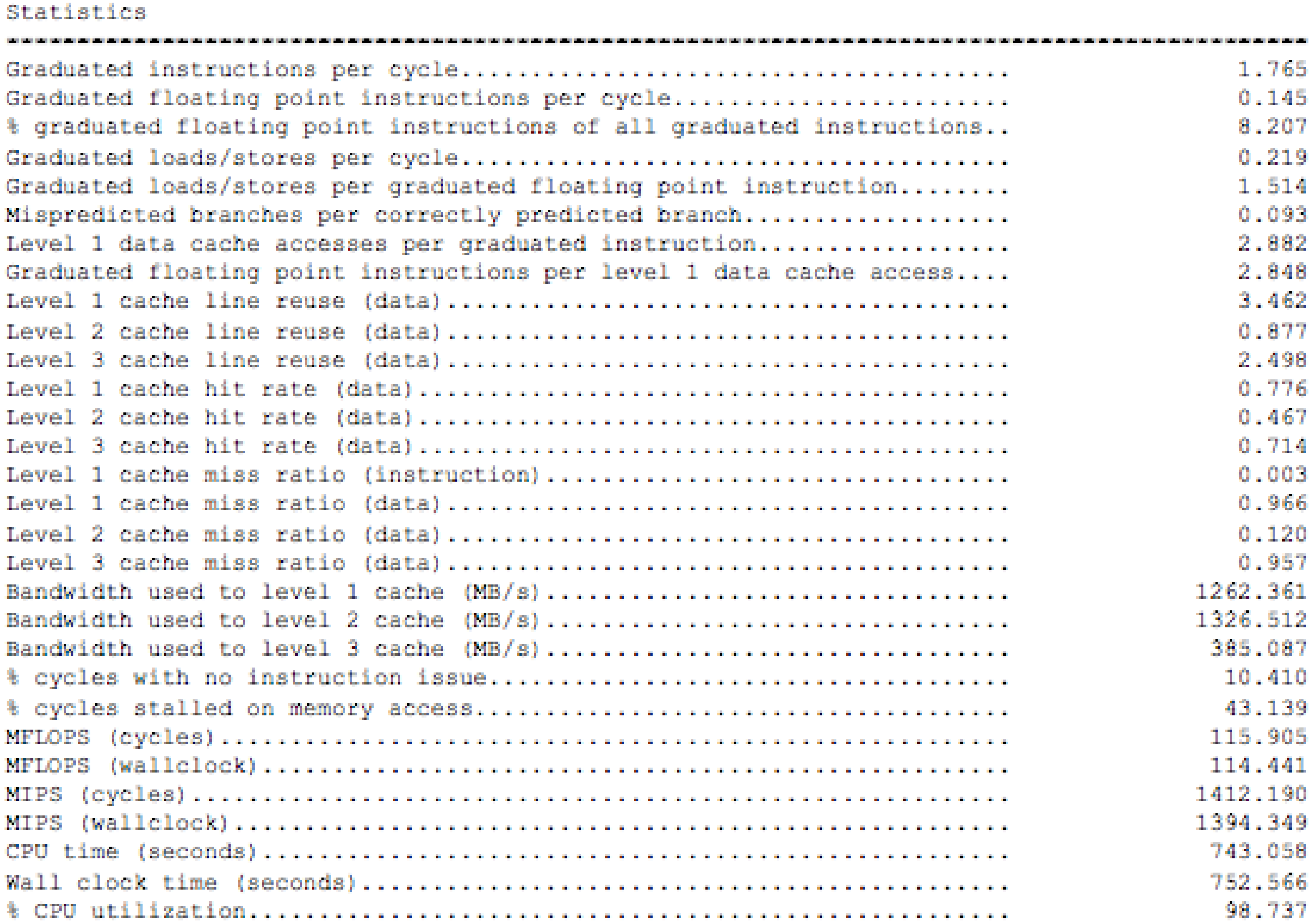
```
---------------------------------------------------------------
@--- Aggregate Time (top twenty, descending, milliseconds) -
---------------------------------------------------------------
Call                    Site       Time    App%    MPI%     COV
Barrier                   29   9.65e+05    4.96   30.20    0.00
Barrier                   18    6.1e+05    3.14   19.10    0.21
Allgather                 12   3.68e+05    1.89   11.51    0.47
Barrier                   43   3.25e+05    1.67   10.18    0.43
Sendrecv                  78    2.2e+05    1.13    6.88    2.19
Sendrecv                  21   1.57e+05    0.81    4.92    0.51
```

```
                                     ------------
                                     ------------
                                     ------------
                                         MPI%
                                         7.53
                                         8.47
                                         8.03
                                         8.09
    4   1.06e+03          85.1           8.03
    5   1.06e+03           111          10.42
    6   1.06e+03           144          13.54
    7   1.06e+03           142          13.37
    8   1.06e+03           139          13.12
    9   1.06e+03           147          13.85
   10   1.06e+03           140          13.16
   11   1.06e+03           141          13.33
   12   1.06e+03           143          13.47
   13   1.06e+03           138          13.03
   14   1.06e+03           144          13.55
   15   1.06e+03           182          17.19
    *    1.7e+04         2e+03          11.76
```

PerfSuite

- Command line tool that can
  - Provides summaries of MPI and Performance Counters
  - Provide statistical profiles as well.
  - Output is XML or ASCII
- Works on uninstrumented code.
- Well supported and documented.
- Lots of derived events for Intel processors.

psrun Output



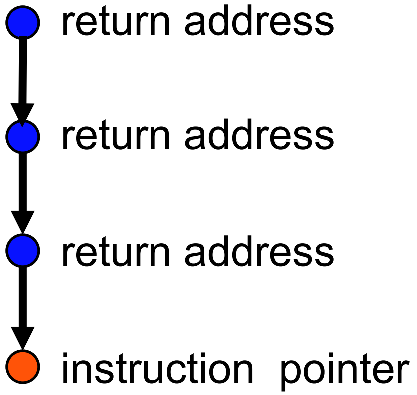
```
Statistics
--------------------------------------------------------------------------------------------
Graduated instructions per cycle.......................................                1.765
Graduated floating point instructions per cycle........................                0.145
% graduated floating point instructions of all graduated instructions..                8.207
Graduated loads/stores per cycle.......................................                0.219
Graduated loads/stores per graduated floating point instruction........                1.514
Mispredicted branches per correctly predicted branch...................                0.093
Level 1 data cache accesses per graduated instruction..................                2.882
Graduated floating point instructions per level 1 data cache access....                2.848
Level 1 cache line reuse (data)........................................                3.462
Level 2 cache line reuse (data)........................................                0.877
Level 3 cache line reuse (data)........................................                2.498
Level 1 cache hit rate (data)..........................................                0.776
Level 2 cache hit rate (data)..........................................                0.467
Level 3 cache hit rate (data)..........................................                0.714
Level 1 cache miss ratio (instruction).................................                0.003
Level 1 cache miss ratio (data)........................................                0.966
Level 2 cache miss ratio (data)........................................                0.120
Level 3 cache miss ratio (data)........................................                0.957
Bandwidth used to level 1 cache (MB/s).................................             1262.361
Bandwidth used to level 2 cache (MB/s).................................             1326.512
Bandwidth used to level 3 cache (MB/s).................................              385.087
% cycles with no instruction issue.....................................               10.410
% cycles stalled on memory access......................................               43.139
MFLOPS (cycles)........................................................              115.905
MFLOPS (wallclock).....................................................              114.441
MIPS (cycles)..........................................................             1412.190
MIPS (wallclock).......................................................             1394.349
CPU time (seconds).....................................................              743.058
Wall clock time (seconds)..............................................              752.566
% CPU utilization......................................................               98.737
```

HPCToolkit

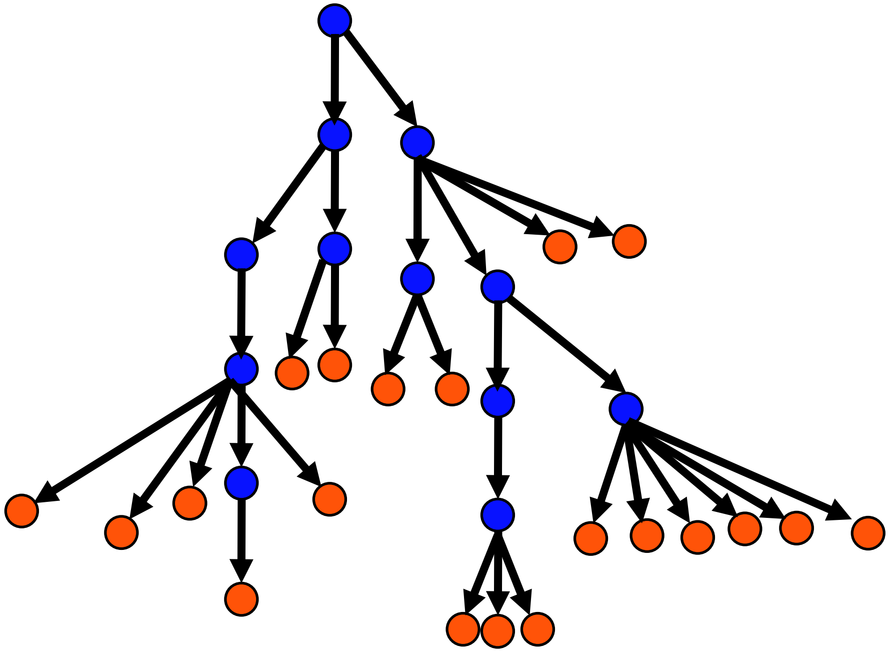
- Statistical call-stack profiling of unmodified applications.
  - Uses hardware counters and timers.
  - Produce profiles per-thread.
  - Works on **unmodified and fully optimized** code.
- Visualizer can compare multiple profiles with derived metrics.
- Concise ASCII output or with a Java GUI

Call Path Profiling

Call path sample

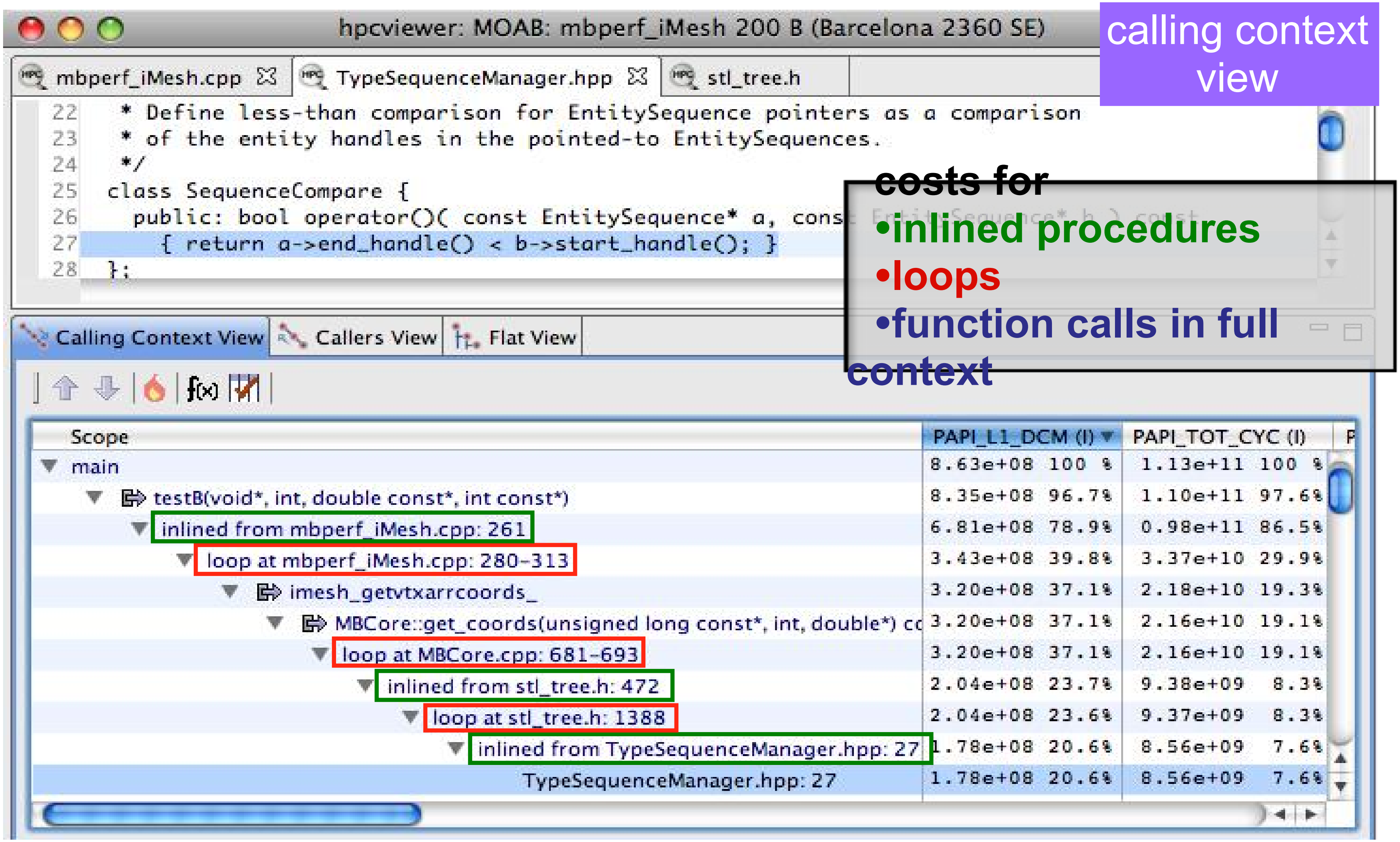


Calling Context Tree (CCT)



Overhead proportional to sampling frequency ...
... not call frequency

HPCToolkit Call Stack



hpcviewer: MOAB: mbperf_iMesh 200 B (Barcelona 2360 SE)

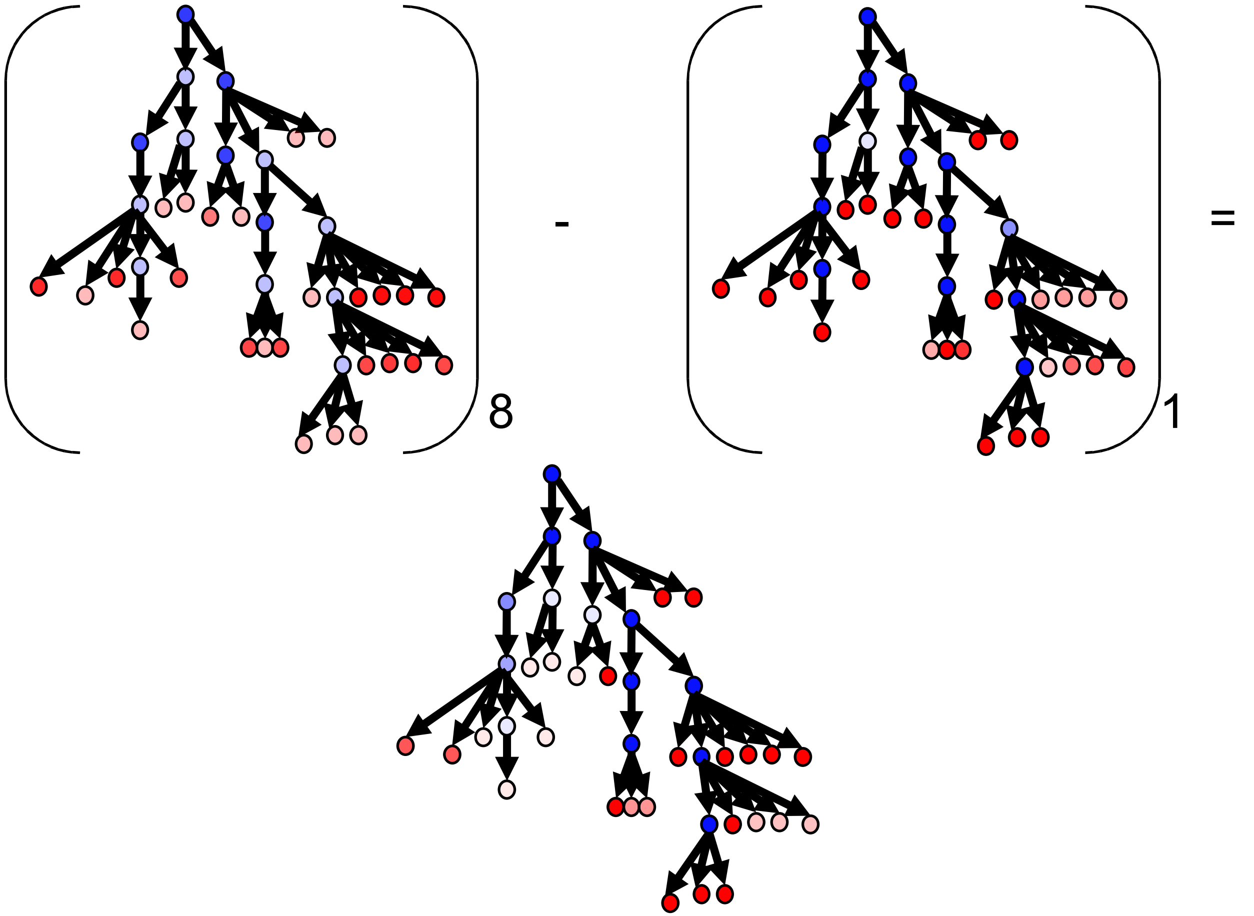
```
  * Define less-than comparison for EntitySequence pointers as a comparison
  * of the entity handles in the pointed-to EntitySequences.
  */
class SequenceCompare {
  public: bool operator()( const EntitySequence* a, const EntitySequence* b ) const
    { return a->end_handle() < b->start_handle(); }
};
```

Scope	PAPI_L1_DCM (I) ▼	PAPI_TOT_CYC (I)	P
▼ main	8.63e+08 100 %	1.13e+11 100 %	
▼ testB(void*, int, double const*, int const*)	8.35e+08 96.7%	1.10e+11 97.6%	
▼ inlined from mbperf_iMesh.cpp: 261	6.81e+08 78.9%	0.98e+11 86.5%	
▼ loop at mbperf_iMesh.cpp: 280-313	3.43e+08 39.8%	3.37e+10 29.9%	
▼ imesh_getvtxarrcoords_	3.20e+08 37.1%	2.18e+10 19.3%	
▼ MBCore::get_coords(unsigned long const*, int, double*) cc	3.20e+08 37.1%	2.16e+10 19.1%	
▼ loop at MBCore.cpp: 681-693	3.20e+08 37.1%	2.16e+10 19.1%	
▼ inlined from stl_tree.h: 472	2.04e+08 23.7%	9.38e+09 8.3%	
▼ loop at stl_tree.h: 1388	2.04e+08 23.6%	9.37e+09 8.3%	
▼ inlined from TypeSequenceManager.hpp: 27	1.78e+08 20.6%	8.56e+09 7.6%	
TypeSequenceManager.hpp: 27	1.78e+08 20.6%	8.56e+09 7.6%	

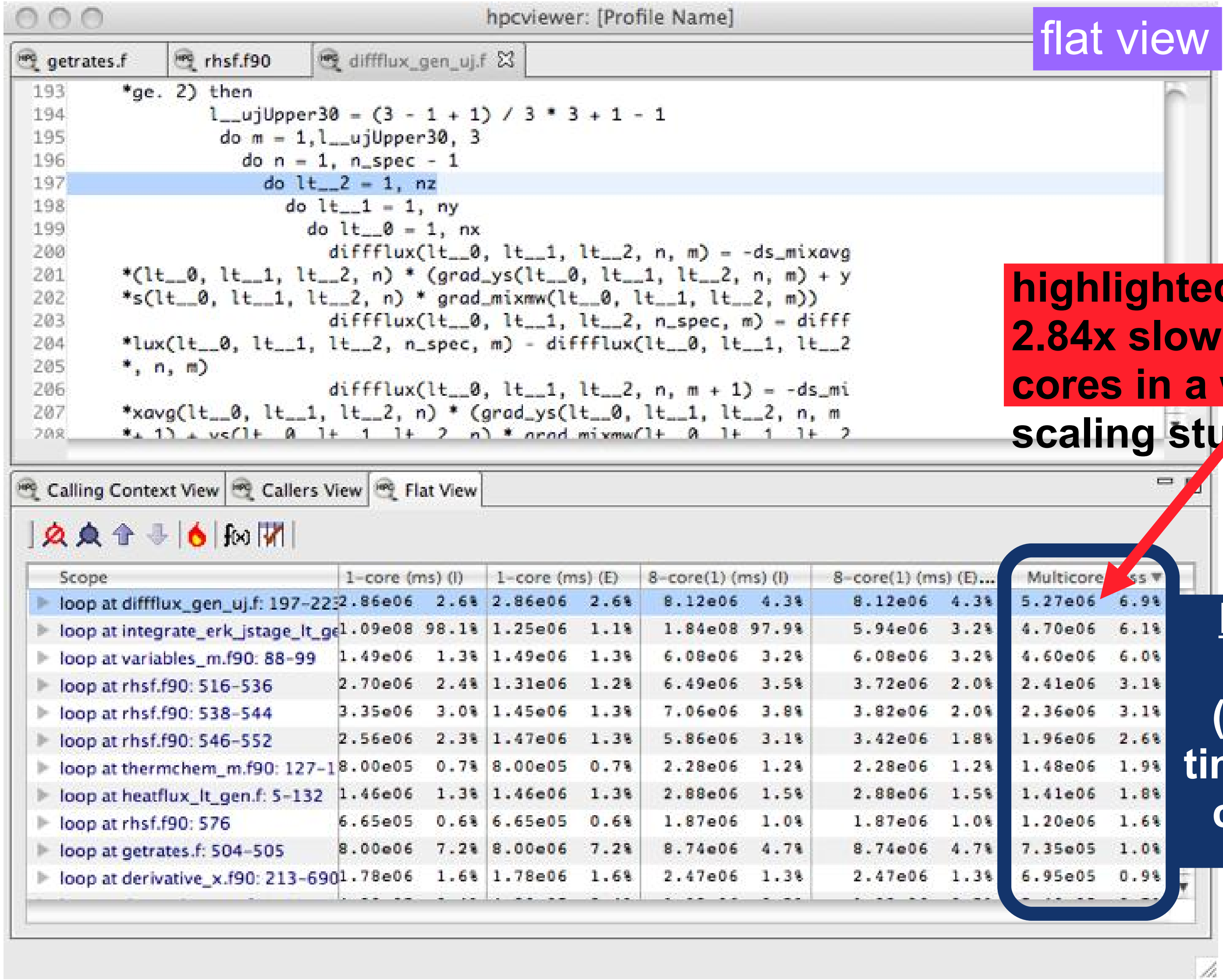
calling context view

costs for
- inlined procedures
- loops
- function calls in full context

Scaling Study with Multiple Profiles



HPCToolkit 1-core v 8-core



```
      *ge. 2) then
              l__ujUpper30 = (3 - 1 + 1) / 3 * 3 + 1 - 1
               do m = 1,l__ujUpper30, 3
                 do n = 1, n_spec - 1
                   do lt__2 = 1, nz
                     do lt__1 = 1, ny
                       do lt__0 = 1, nx
                         diffflux(lt__0, lt__1, lt__2, n, m) = -ds_mixavg
      *(lt__0, lt__1, lt__2, n) * (grad_ys(lt__0, lt__1, lt__2, n, m) + y
      *s(lt__0, lt__1, lt__2, n) * grad_mixmw(lt__0, lt__1, lt__2, m))
                         diffflux(lt__0, lt__1, lt__2, n_spec, m) = difff
      *lux(lt__0, lt__1, lt__2, n_spec, m) - diffflux(lt__0, lt__1, lt__2
      *, n, m)
                         diffflux(lt__0, lt__1, lt__2, n, m + 1) = -ds_mi
      *xavg(lt__0, lt__1, lt__2, n) * (grad_ys(lt__0, lt__1, lt__2, n, m
      *+ 1) + ys(lt__0, lt__1, lt__2, n) * grad_mixmw(lt__0, lt__1, lt__2
```

Scope	1-core (ms) (I)		1-core (ms) (E)		8-core(1) (ms) (I)		8-core(1) (ms) (E)...		Multicore ss ▼	
▶ loop at diffflux_gen_uj.f: 197-22	2.86e06	2.6%	2.86e06	2.6%	8.12e06	4.3%	8.12e06	4.3%	5.27e06	6.9%
▶ loop at integrate_erk_jstage_lt_ge	1.09e08	98.1%	1.25e06	1.1%	1.84e08	97.9%	5.94e06	3.2%	4.70e06	6.1%
▶ loop at variables_m.f90: 88-99	1.49e06	1.3%	1.49e06	1.3%	6.08e06	3.2%	6.08e06	3.2%	4.60e06	6.0%
▶ loop at rhsf.f90: 516-536	2.70e06	2.4%	1.31e06	1.2%	6.49e06	3.5%	3.72e06	2.0%	2.41e06	3.1%
▶ loop at rhsf.f90: 538-544	3.35e06	3.0%	1.45e06	1.3%	7.06e06	3.8%	3.82e06	2.0%	2.36e06	3.1%
▶ loop at rhsf.f90: 546-552	2.56e06	2.3%	1.47e06	1.3%	5.86e06	3.1%	3.42e06	1.8%	1.96e06	2.6%
▶ loop at thermchem_m.f90: 127-1	8.00e05	0.7%	8.00e05	0.7%	2.28e06	1.2%	2.28e06	1.2%	1.48e06	1.9%
▶ loop at heatflux_lt_gen.f: 5-132	1.46e06	1.3%	1.46e06	1.3%	2.88e06	1.5%	2.88e06	1.5%	1.41e06	1.8%
▶ loop at rhsf.f90: 576	6.65e05	0.6%	6.65e05	0.6%	1.87e06	1.0%	1.87e06	1.0%	1.20e06	1.6%
▶ loop at getrates.f: 504-505	8.00e06	7.2%	8.00e06	7.2%	8.74e06	4.7%	8.74e06	4.7%	7.35e05	1.0%
▶ loop at derivative_x.f90: 213-690	1.78e06	1.6%	1.78e06	1.6%	2.47e06	1.3%	2.47e06	1.3%	6.95e05	0.9%

flat view

highlighted loop is 2.84x slower on 8 cores in a weak scaling study

Multicore Loss (Multicore time - single core time)

Performance Experiment Tools

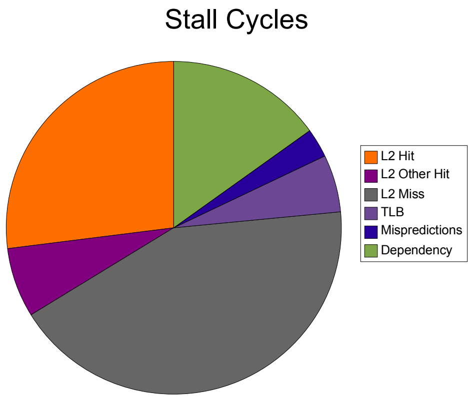
- A set of tools, easy to use as **time**.
- Provide a uniform interface to a number of underlying tools.
- Largely work on uninstrumented code.
- Mostly take the same arguments.
- papiex, mpipex, ioex, hpcex, gptlex, tauex

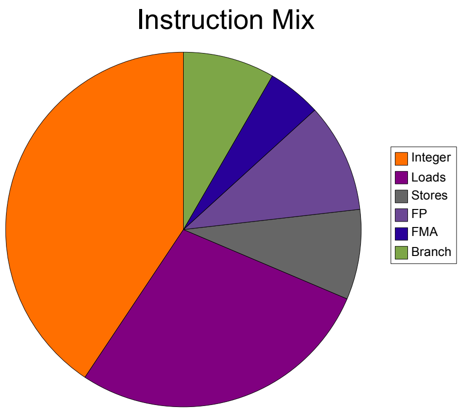
papiex

- A simple to use tool that generates performance measurements for the entire run of a code, including summaries for job, task and thread.
  - Hardware performance metrics
  - I/O
  - Thread synchronization
  - MPI
- Simple instrumentation API
- No recompilation

Papiex: Workload Characterization

```
Est. L2 Private Hit Stall % ..................          10.76
Est. L2 Other Hit Stall % ....................           2.79
Est. L2 Miss (private,other) Stall % .........          17.24
Total Est. Memory Stall % ....................          30.79
Est. L1 D-TLB Miss Stall % ...................           2.26
Est. L1 I-TLB Miss Stall % ...................           0.04
Est. TLB Trap Stall % ........................           0.15
Total Est. TLB Stall % .......................           2.45
Est. Mispred. Branch Stall % .................           1.15
Dependency (M-stage) Stall % .................           6.17
Total Measured Stall % .......................           9.77
Total Underestimated Stall % .................          34.39
Total Overestimated Stall % ..................          40.56
Actual/Ideal Cyc (max. dual) .................           2.29
Ideal IPC (max. dual) ........................           1.07
Ideal MFLOPS (max. dual) .....................         148.88
Actual/Ideal Cyc (cur. dual) .................           2.40
Ideal IPC (cur. dual) ........................           1.12
Ideal MFLOPS (cur. dual) .....................         156.29
MPI cycles % .................................           8.85
MPI Sync cycles % ............................           0.00
I/O cycles % .................................           0.00
Thr Sync cycles % ............................           0.00
```





GPTL

- Used to easily instrument applications for the generation of performance data.
- Optimized for usability.
- Provides access to timers as well as PAPI events.
- Thread-safe and per-thread statistics.
- Provides estimates of overhead.
- Call-tree generation.

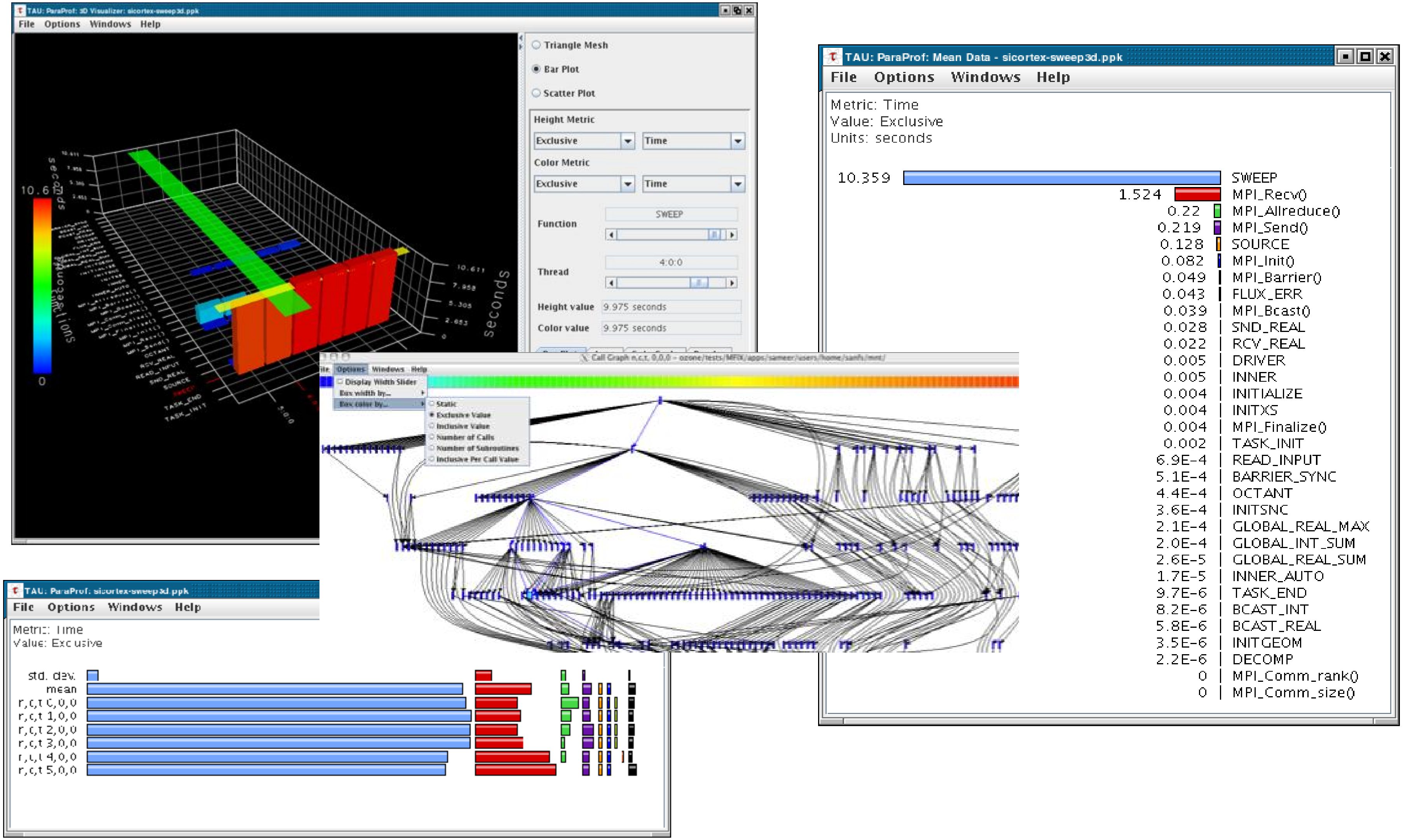
TAU

- Entire toolkit for parallel and serial performance instrumentation, measurement, analysis and visualization.
- Steep(ish) learning curve, but payoff can be worth it.
- Works via source instrumentation and limited dynamic instrumentation
- Very good at OpenMP instrumentation

TAU Parallel Performance System

- Parallel Performance Evaluation Tool for Fortran, C, C++ and Python
- Used for in-depth performance studies of an application throughout its lifecycle.
- Supports all forms of parallel profiling
  - Flat, callpath, and phase based profiling
  - PAPI counters, wallclock time, CPU time, memory
- PerfExplorer cross experiment analysis tool

TAU



TAU: ParaProf: 3D Visualizer: sicortex-sweep3d.ppk
File Options Windows Help
10.6
0
seconds
Call Graph n,c,t, 0,0,0 - ozone/tests/MFIX/apps/sameer/users/home/sanfs/mnt/
File Options Windows Help
☐ Display Width Slider
Box width by...
Box color by...
TAU: ParaProf: sicortex-sweep3d.ppk
File Options Windows Help
std. dev.
mean
n,c,t 0,0,0
n,c,t 1,0,0
n,c,t 2,0,0
n,c,t 3,0,0
n,c,t 4,0,0
n,c,t 5,0,0
TAU: ParaProf: Mean Data - sicortex-sweep3d.ppk
File Options Windows Help

○ Triangle Mesh
● Bar Plot
○ Scatter Plot

Height Metric: Exclusive, Time

Color Metric: Exclusive, Time

Function: SWEEP

Thread: 4:0:0

Height value: 9.975 seconds

Color value: 9.975 seconds

○ Static
● Exclusive Value
○ Inclusive Value
○ Number of Calls
○ Number of Subroutines
○ Inclusive Per Call Value

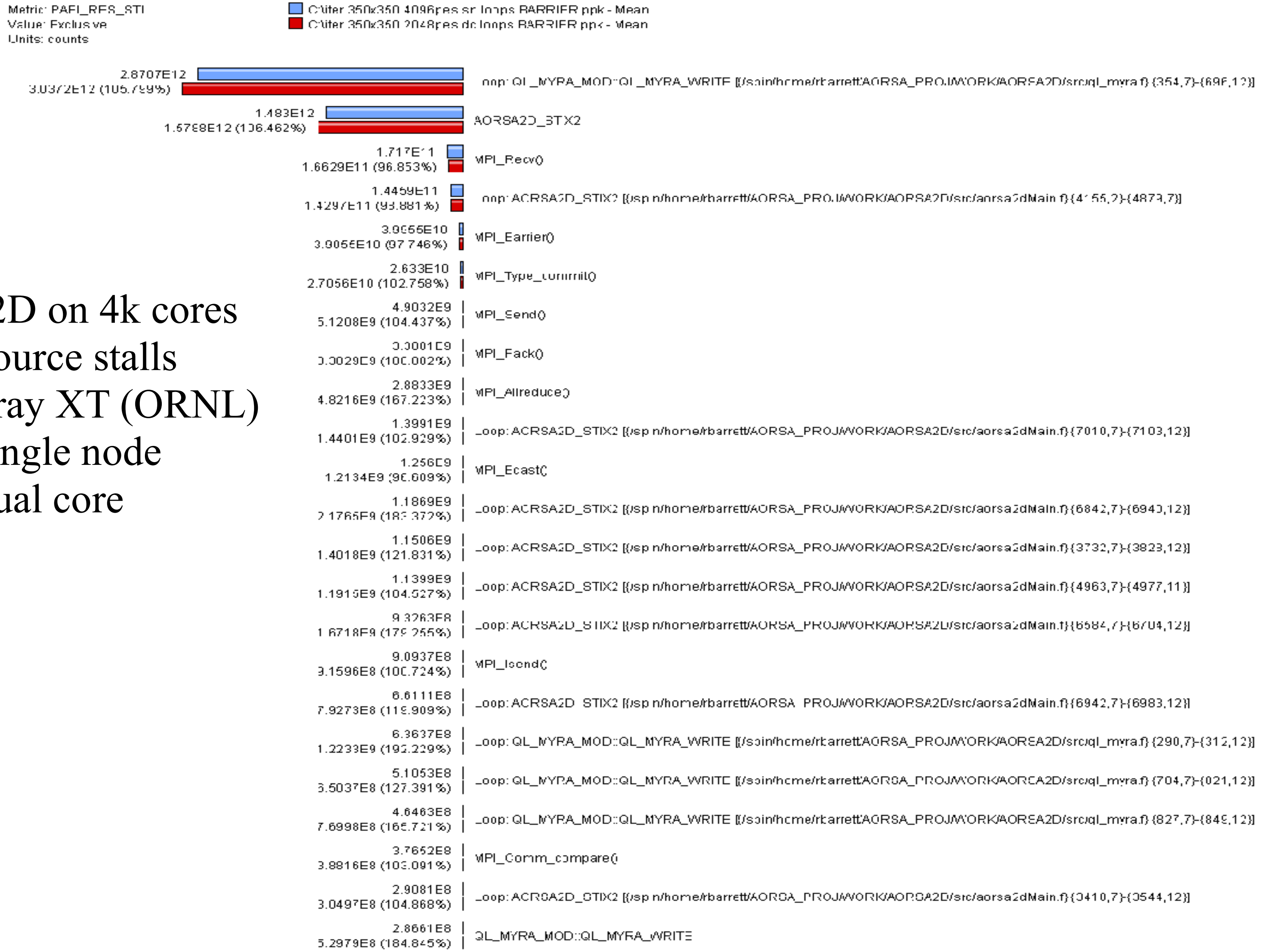
Metric: Time
Value: Exclusive

Metric: Time
Value: Exclusive
Units: seconds

10.359	SWEEP
1.524	MPI_Recv()
0.22	MPI_Allreduce()
0.219	MPI_Send()
0.128	SOURCE
0.082	MPI_Init()
0.049	MPI_Barrier()
0.043	FLUX_ERR
0.039	MPI_Bcast()
0.028	SND_REAL
0.022	RCV_REAL
0.005	DRIVER
0.005	INNER
0.004	INITIALIZE
0.004	INITXS
0.004	MPI_Finalize()
0.002	TASK_INIT
6.9E-4	READ_INPUT
5.1E-4	BARRIER_SYNC
4.4E-4	OCTANT
3.6E-4	INITSNC
2.1E-4	GLOBAL_REAL_MAX
2.0E-4	GLOBAL_INT_SUM
2.6E-5	GLOBAL_REAL_SUM
1.7E-5	INNER_AUTO
9.7E-6	TASK_END
8.2E-6	BCAST_INT
5.8E-6	BCAST_REAL
3.5E-6	INITGEOM
2.2E-6	DECOMP
0	MPI_Comm_rank()
0	MPI_Comm_size()

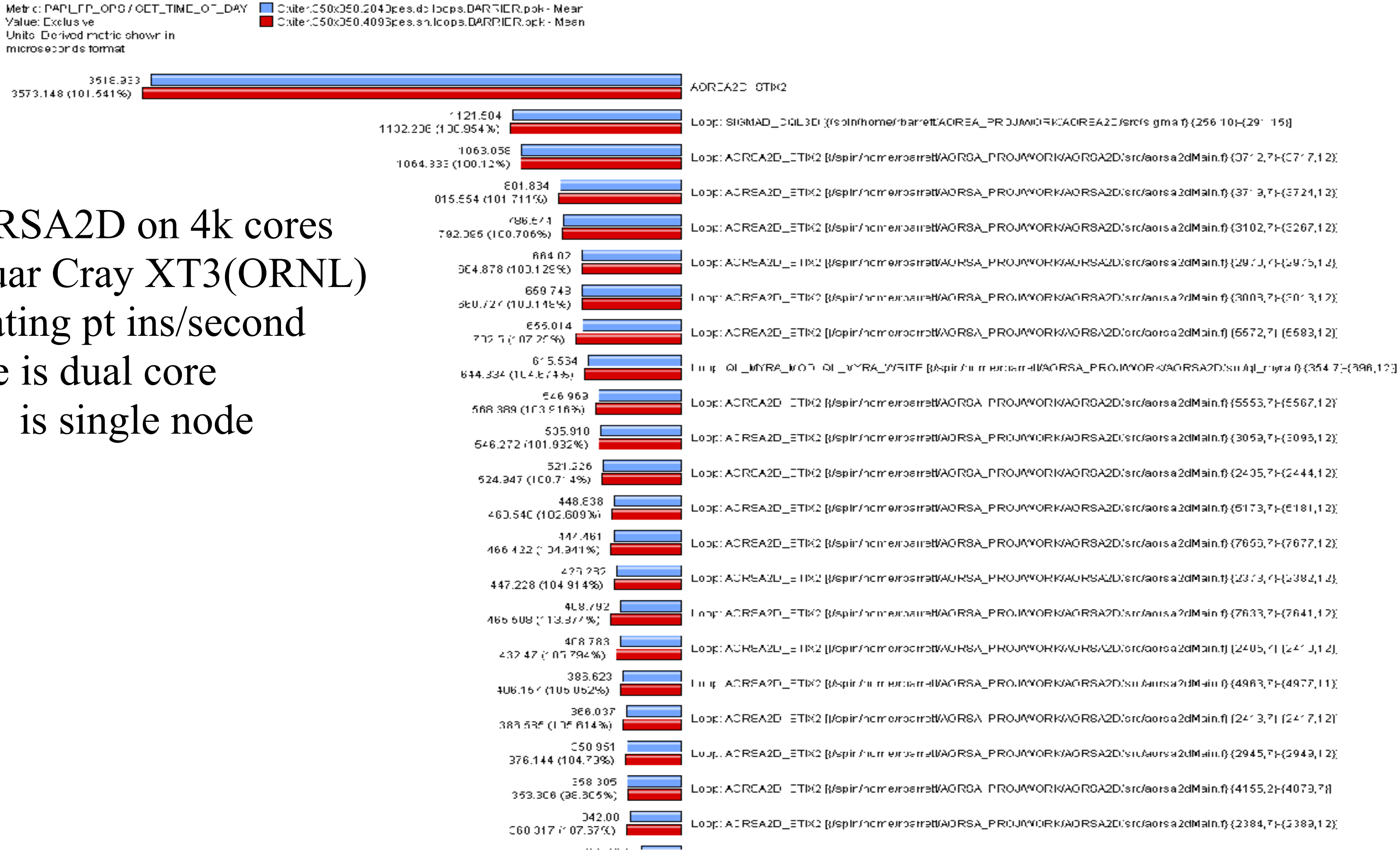
Comparing Effects of MultiCore Processors

- ❑ AORSA2D on 4k cores
- ❑ PAPI resource stalls
- ❑ Jaguar Cray XT (ORNL)
- ❑ Blue is single node
- ❑ Red is dual core



Comparing FLOPS: MultiCore Processors

- ❑ AORSA2D on 4k cores
- ❑ Jaguar Cray XT3(ORNL)
- ❑ Floating pt ins/second
- ❑ Blue is dual core
- ❑ Red is single node



Other Performance Tools

- Oprofile
  - Hardware counter profiling for Linux
    - But you need to have dedicated access to the node.
- Scalasca
  - Tracing for OpenMP and MPI

Ways to Avoid Multicore Performance Problems

- Don't write your own solvers.
  - Know what libraries are available and **plan** your data structures.
  - Spend your time on innovation not implementation.
  - Libraries are well documented and well publicized.
  - Remember the 80/20 rule.

Ways to Avoid Multicore Performance Problems

- Don't use more data and memory bandwidth than necessary.
  - Do you need **double** or can you live with **float**?
  - Do you need > 2GB of address space?
  - After comm., memory bandwidth is always the biggest bottleneck.
- Running in 32-bit mode does not mean you can't use double precision floating point.

Ways to Avoid Multicore Performance Problems

- Help the compiler help you (optimize for cache).
  - Flags
  - Directives
  - Good code structure
  - Compilers are better at optimizing **simple code** than you are.
  - Reading the manual **is** worth it.

Ways to Avoid Multicore Performance Problems

- If you have to write your own code, tune it for cache on a single processor.
  - Make sure the algorithm scales first.
  - If you get good cache utilization, it will make multi-core performance that much easier.

Ways to Avoid Multicore Performance Problems

- Maximize granularity and minimize synchronization (and communication).
  - Larger, longer and more independent the computations, the greater the speedup.

Ways to Avoid Multicore Performance Problems

- Don't violate the usage model of your programming environment.
  - If something seems 'hard' to get right, you may be doing something wrong.
  - Have reasonable expectations.
  - Recall the CUDA comment.

References

- http://www.cs.utk.edu/~mucci/MPPopt.html
- http://www.cs.utk.edu/~mucci/latest/mucci_talks.html
- Multithreaded Algorithms
  - http://www.cilk.com/resources/multithreaded-algorithms-textbook-chapter/
- Multicore Optimization
  - http://software.intel.com/en-us/articles/software-techniques-for-shared-cache-multi-core-systems/
  - http://drops.dagstuhl.de/opus/volltexte/2008/1374/pdf/07361.GroeszlingerArmin.Paper.1374.pdf
  - http://www.cis.udel.edu/~cavazos/cisc879/BryanY.pdf
  - http://crd.lbl.gov/~oliker/papers/ipdps08_final.pdf
  - http://isdlibrary.intel-dispatch.com/isd/1588/MC_Excerpt.pdf

References

- Pthreads
  - https://computing.llnl.gov/tutorials/pthreads/
  - http://randu.org/tutorials/threads/
  - http://www.cs.umu.se/kurser/TDBC64/VT03/pthreads/pthread-primer.pdf
  - http://www.cs.utsa.edu/~whaley/teach/cs6643/LEC/pthreads_ho.pdf
- OpenMP
  - http://www.compunity.org/events/pastevents/ewomp2004/suess_leopold_pap_ew04.pdf
  - https://computing.llnl.gov/tutorials/openMP/
  - http://infostream.rus.uni-stuttgart.de/lec/288/291/real/
  - https://fs.hlrs.de/projects/par/par_prog_ws/2004C/22_openmp_performance

References

- FFTW
  - http://www.fftw.org
- PetSC
  - http://www-unix.mcs.anl.gov/petsc
- SUPERLU
  - http://crd.lbl.gov/~xiaoye/SuperLU
- ScaLAPACK etc...
  - http://www.netlib.org
- VSIPL
  - http://www.vsipl.org

References

- GNU compilers
  - http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
- Intel Compiler
  - http://www.ncsa.edu/UserInfo/Training/Workshops/Multicore/presentations/Optimization%20for%20Performance.ppt
  - http://www.intel.com/cd/software/products/asmo-na/eng/222300.htm
  - http://www.ichec.ie/support/tutorials/intel_compiler.pdf
- CILK tutorials
  - http://supertech.csail.mit.edu/cilk
  - http://www.cilk.com/resource-library/resources/
- UPC
  - http://upc.gwu.edu/tutorials.html
- Co-Array Fortran
  - http://www.co-array.org

References

- OpenMPI and SLURM process binding
  - http://icl.cs.utk.edu/open-mpi/faq/?category=tuning
  - https://computing.llnl.gov/linux/slurm/mc_support.html
- Hoard memory allocator
  - http://www.cs.umass.edu/~emery/hoard/index.html
- PAPI
  - http://icl.cs.utk.edu/projects/papi
- ompP
  - http://www.ompp-tool.com
- Valgrind
  - http://www.valgrind.org
  - http://valgrind.org/docs/manual/hg-manual.html
  - http://valgrind.org/docs/manual/drd-manual.html

References

- mpiP
  - http://mpip.sourceforge.net
- PerfSuite
  - http://perfsuite.ncsa.uiuc.edu
- HPCToolkit
  - http://hipersoft.cs.rice.edu/hpctoolkit
- papiex
  - http://www.cs.utk.edu/~mucci/papiex
- TAU
  - http://www.paratools.com
- GPTL
  - http://www.burningserver.net/rosinski/gptl

References

- Prefetching
  - http://gcc.gnu.org/projects/prefetch.html
- Aliasing
  - http://developers.sun.com/solaris/articles/cc_restrict.html
  - http://www.redhat.com/docs/manuals/enterprise/RHEL-4-Manual/gcc/restricted-pointers.html
  - http://www.cellperformance.com/mike_acton/2006/05/demystifying_the_restrict_keyw.html
  - http://www.cellperformance.com/mike_acton/2006/06/understanding_strict_aliasing.html
- Alignment
  - http://software.intel.com/en-us/articles/align-and-organize-data-for-better-performance
  - http://www.jauu.net/data/pdf/beware-of-your-cacheline.pdf