OpenMP for Intranode Programming

ATPESC, 08/06/2014

Barbara Chapman, University of Houston

Deepak Eachempati, University of Houston

Kelvin Li, IBM

Agenda

❑ Morning: An Introduction to OpenMP 3.1

❑ **Afternoon: Hybrid Programming with MPI and OpenMP; Using OpenMP; OpenMP 4.0**

❑ Evening: Practical Programming

Agenda

❑ Hybrid Programming with MPI and OpenMP

❑ Using OpenMP
   ❑ Common programming errors
   ❑ Performance Topics

Programming Options for "Hybrid" Architectures

❑ **Pure MPI** – each core runs an MPI process

  ❑ new MPI-3 support for shared memory makes MPI+MPI "hybrid" programming a viable option*

❑ **Pure OpenMP**

  ❑ single process, fully multi-threaded

  ❑ virtual distributed shared address space

❑ **MPI and OpenMP**

  ❑ non-overlapped ("Masteronly") – only a master thread makes MPI calls, while no other threads are active

  ❑ overlapped - many interesting approaches here

* T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. Gropp, V. Kale, R. Thakur: MPI + MPI: a new hybrid approach to parallel programming with MPI plus shared memory. *Computing*, 95(12):1121– 1136, December 2013.

Reasons to Add OpenMP

- OpenMP can be a more efficient solution for *intra-node* parallelism
    - uses less memory than MPI
    - more efficient for *fine-grained* parallelism
    - may require use within **NUMA** nodes

- Constraint on total number of MPI processes that can be used for application
    - per-node memory limits
    - system limits on number of processes that can be spawned
    - application doesn't scale past a certain number of MPI processes

- Application exhibits hierarchical parallelization pattern
    - natural to use MPI for top-level, and OpenMP for second level

- Unbalanced MPI workloads – can assign more threads to heavily-loaded MPI processes

Reasons to be Cautious

- Interoperability issues between MPI and OpenMP implementations
  - is MPI library thread-safe?
  - how might presence of additional threads impact MPI's performance?

- Added complexity in program - beware of shared memory programming pitfalls such as data races or false sharing

- If limiting communication to a single thread, are we still able to saturate the network?

NUMA considerations

❑ NUMA, Non-Uniform Memory Access

  ❑ this is a common case for your compute nodes

  ❑ Nodes -> (NUMA nodes) Sockets -> Cores -> H/W Threads

  ❑ consideration of process/thread assignment to cores is critical for performance

🟥 MPI process/master thread
🟩 OpenMP worker threads
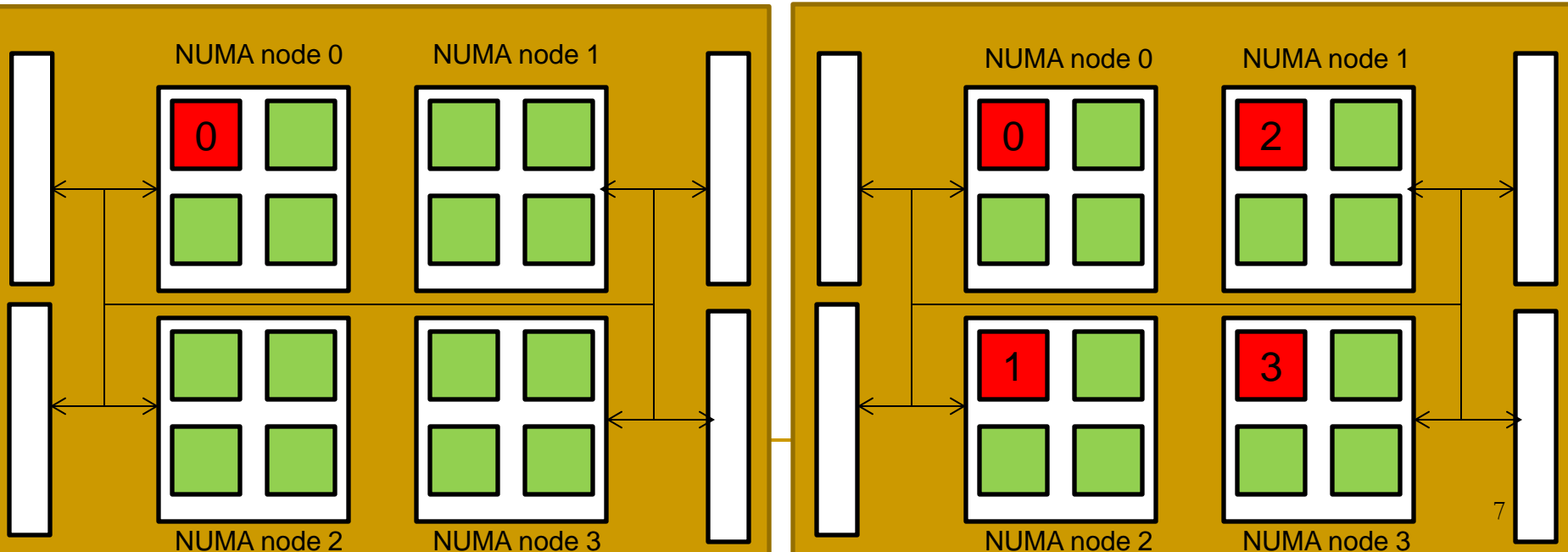
Resource Utilization Considerations

- ❑ Network Utilization
  - ❑ if only one MPI process per node, can we still saturate the network port?
  - ❑ usually yes, but maybe not if multiple network ports become commonplace in the near future
- ❑ Core Utilization
  - ❑ Threads can help overlap computation with communication
  - ❑ Can also help balance workloads through worksharing constructs
  - ❑ However: sleeping threads ("Masteronly" mode) will limit core utilization

Hybrid Programming in Practice

❑ Typically start with an MPI program, and you use OpenMP to parallelize it
- ❑ loop parallelism
- ❑ task parallelism
- ❑ SIMD and Accelerators (next talk: OpenMP 4.0)

❑ Strategies
- ❑ vary number of threads based on workload in each process
- ❑ find best mapping of threads to cores
- ❑ use threads to overlap computation with MPI calls for more asynchronous progress
- ❑ generally requires experimentation to find best combination (e.g. # processes, # threads/process, thread affinity)

MPI Thread Support Modes (Recap)

❑ Request/get thread support mode using call to MPI_Init_thread instead of MPI_Init

❑ MPI_THREAD_SINGLE (default with MPI_Init)
   ❑ assume MPI process is not multi-threaded

❑ MPI_THREAD_FUNNELED
   ❑ multi-threaded processes allowed
   ❑ only one designated thread is making MPI calls

❑ MPI_THREAD_SERIALIZED
   ❑ multi-threaded, and multiple threads may make MPI calls
   ❑ calls must be serialized

❑ MPI_THREAD_MULTIPLE
   ❑ multi-threaded, no restrictions
   ❑ requires *fully* thread-safe MPI implementation

Example: MPI_THREAD_FUNNELED

```
#include <mpi.h>

int main(int argc, char **argv)
{
    int rank, size, ierr, i, provided;
    MPI_Init_thread(&argc,&argv,
                    MPI_THREAD_FUNNELED,
                    &provided);
  ...
   #pragma omp parallel
{
   #pragma omp master
     { ... MPI calls ... }
   #pragma barrier
   #pragma omp for
    for (i = 0; i < N; i++) {
         do_something( i );
    }
    ...
```

call MPI_Init_thread to request MPI_THREAD_FUNNELED

now we can do MPI in parallel region
(NOTE: master construct ensures its the same thread which does it)

REMEMBER: if using master, we may also need a barrier

Example: MPI_THREAD_SERIALIZED

```
...

   MPI_Init_thread(&argc,&argv,
                     MPI_THREAD_SERIALIZED,
                     &provided);
 ...
   #pragma omp parallel
{
   ...
   #pragma omp single
     { ... MPI calls ... }

   #pragma omp for
    for (i = 0; i < N; i++) {
        do_something( i );
    }
    ...
```

With SERIALIZED, we can now use a SINGLE construct for more flexibility.

NOTE: Use nowait clause if you wish to avoid implicit barrier at the end and obtain overlap

Example: MPI_THREAD_MULTIPLE

```
...

  MPI_Init_thread(&argc,&argv,
                    MPI_THREAD_MULTIPLE,
                    &provided);
 ...
 #pragma omp parallel
{
   tid = omp_get_thread_num();
   ...
   if (mpi_rank % 2) {
       MPI_Send(data, N, MPI_INT, mpi_rank-1, tid, ... );
   } else {
       MPI_Recv(data, N, MPI_INT, mpi_rank+1, tid, ... );
   }
    ...
}
```

With MULTIPLE, no restrictions on using MPI calls in a parallel region.

Hiding Communication Latency using OpenMP

- ## MPI communication is often blocking
  - ❏ even non-blocking calls may require MPI calls to achieve progress
  - ❏ hardware support and/or helper threads might help, but often not available
- ## Strategies using OpenMP
  - ❏ use an "explicit" SPMD approach
  - ❏ use nested parallel region
  - ❏ use tasks

Achieving Overlap using a SPMD approach

```
...
   MPI_Init_thread(...);
 ...
 #pragma omp parallel
{
    tid = omp_get_thread_num();
    ...
    if (tid == 0) {
        /* first thread does MPI stuff */
    } else {
        /* remaining threads carry on with independent
           computation */
    }
    #pragma omp barrier
}
```

Here we divide thread team into two "subteams" using thread ID.

Main Issue:
- work-sharing constructs in "else" block are unavailable to us
- requires explicit coding of work-sharing, cumbersome and inflexible

Achieving Overlap using Nested Parallelism

```
...
 omp_set_nested(true);
 ...
 #pragma omp parallel num_threads(2)
{
    tid = omp_get_thread_num();
    ...
    if (tid == 0) {
      /* do MPI stuff */
    } else {
      /* thread 1 spawns a new parallel region to do work */
      #pragma omp parallel
      { ... }
    }
    ...
}
```

nested parallel region here can perform all work-sharing constructs independent of the MPI communication by thread 0

Achieving Overlap using nowait clause

```
...
  MPI_Init_thread(...);
 ...
 #pragma omp parallel
{
  #pragma omp master
  { /* first thread does MPI stuff */ }

  /* remaining threads continue with other work */
  #pragma omp for schedule (...) nowait
   for(...) { ... }
  #pragma omp for schedule(...) nowait
   for(...) { ... }
   ...
}
```

This approach allows us to utilize all threads (including, eventually, the MPI-designated thread(s)) for doing computation

Achieving Overlap using explicit tasks

```
...
  MPI_Init_thread(...);
 ...
#pragma omp parallel
{
   ...
  #pragma omp master
   {
     for (...) {
     #pragma omp task
        { /* create tasks for other threads to work on */ }
     }
     /* after task creation, master does MPI stuff*/
   }

  #pragma omp barrier
   ...
}
```

Here, the master creates tasks which may be picked up by the other threads.

Recall: barriers are task scheduling points.

NPB Multi-Zone Parallel Benchmarks

❑ Compute discrete solutions of unsteady, compressible Navier-Stoke equations in 3D

❑ For each problem, a logically rectangular discretization mesh is divided into a 2D horizontal tiling of 3D zones

❑ Consists three algorithm benchmarks: LU, SP and BT
   ❑ LU (Lower-Upper symmetric Gauss-Seidel)
   ❑ SP (Scalar Penta-diagonal)
   ❑ BT (Block Tri-diagonal)
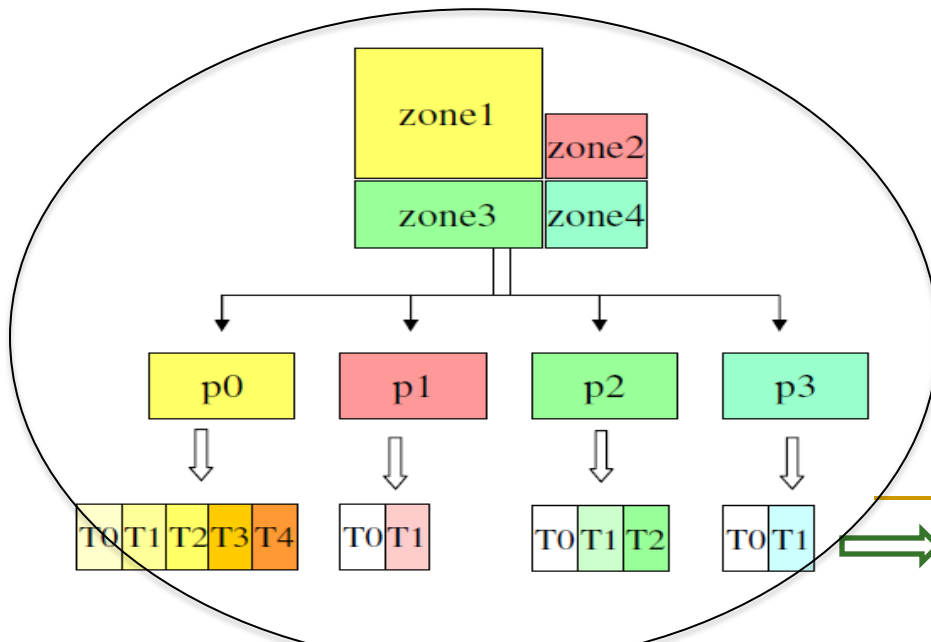
Assign more threads to larger size zones, static load balancing
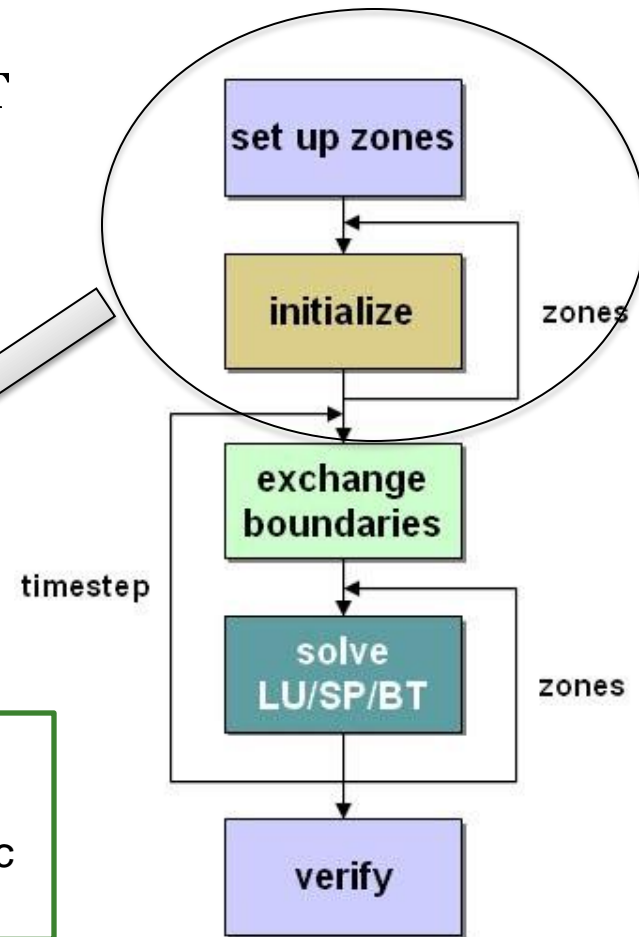
BT-MZ and SP-MZ Results
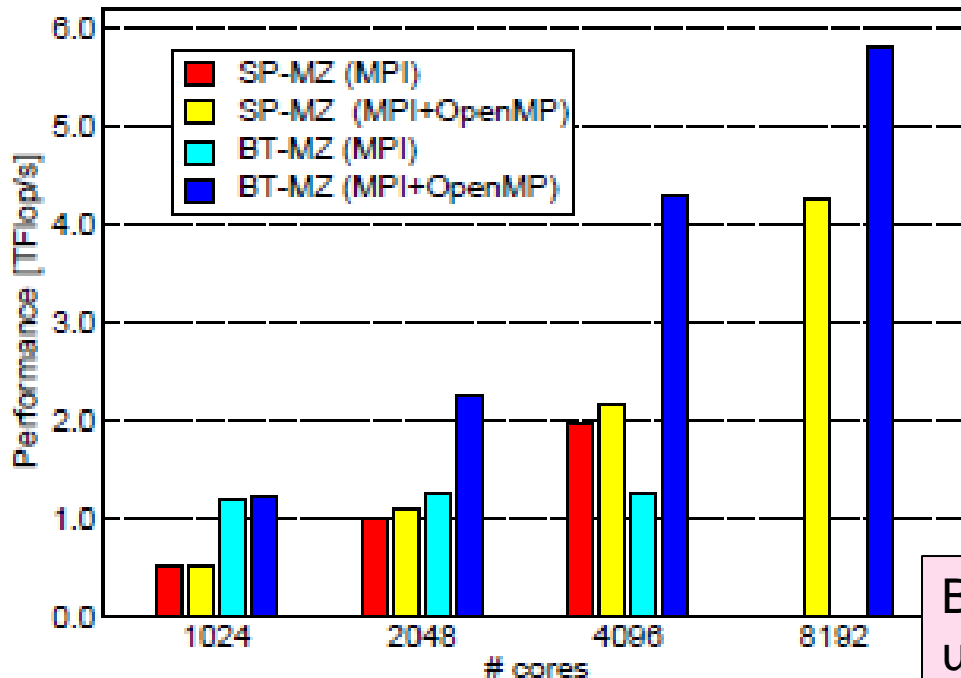
- ❑ Class E
- ❑ 4096 zones (max. number of MPI processes)
- ❑ Platform:
  - ❑ "Ranger" at TACC, Austin
  - ❑ 3936 blades, each with 4 AMD Opteron "Barcelona" quad-core chips
  - ❑ MPI: mvapich
  - ❑ numactl used for thread/core affinity

BT-MZ performance with unbalanced workload greatly improved by adding OpenMP

Rolf Rabenseifner, Georg Hager, and Gabriele Jost: **Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes.** In Didier El Baz et al. (Eds.), (PDP 2009), in Weimar, Germany, Feb. 16-18, 2009, Computer Society Press, pp. 427-236.

Summary

- Technological trends makes hybrid programming all the more important
    - "fatter" nodes with cc-NUMA characteristics
    - reduced memory available per core
    - extreme-scale computing will require dynamic, load balancing strategies
- With OpenMP, you can
    - develop more memory-efficient algorithms for within the node
    - "workshare" among threads using various scheduling policies, to curtail load imbalance
    - hide communication latency using a variety of strategies
- As always, choose the best programming system for your problem.

Agenda

❑ Hybrid Programming with MPI and OpenMP

❑ Using OpenMP
  ❑ Common programming errors
  ❑ Performance Topics

Common Sources of Errors

❑ Wrong "spelling" of sentinel

❑ Wrongly declared data attributes (shared vs. private, firstprivate, etc.)

❑ Incorrect use of synchronization constructs

  ❑ Less likely if user sticks to directives

  ❑ Erroneous use of locks can lead to deadlock

  ❑ Erroneous use of NOWAIT can lead to race conditions.

❑ Race conditions (true sharing)

  ❑ Can be very hard to find

It can be very hard to track race conditions. Tools may help check for these, but they may fail if your OpenMP code does not rely on directives to distribute work. Moreover, they can be quite slow.

Care with Synchronization

❑ Recall that a thread's temporary view of memory may vary from shared memory

   ❑ Value of shared objects updated at synchronization points

   ❑ User must be aware of the point at which modified values are (guaranteed to be) accessible

❑ Compilers routinely reorder instructions that implement a program

   ❑ Helps exploit the functional units, keep machine busy

❑ Compiler cannot move instructions past a barrier

   ❑ Also not past a flush on all variables

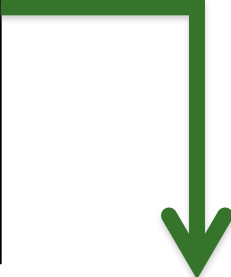   ❑ But it can move them past a flush on a set of variables so long as those variables are not accessed

Race Condition

❏ Several threads access and update shared data concurrently

  ❏ One thread writes and one or more threads read or write same memory location at about the same time

  ❏ Outcome depends on relative ordering of operations and may differ between runs

❏ User is expected to avoid race conditions

  ❏ insert synchronization constructs as appropraite, or

  ❏ privatize data

❏ Some tools exist to detect data races at runtime

  ❏ e.g. Intel Thread Checker, Oracle Solaris Studio Thread Analyzer

Global Data – An Example/1

```fortran
      program global_data
            ....
      use mod_global_data
            ....
!$omp parallel do private(j)
      do j = 1, n
         call suba(j)
      end do
!$omp end parallel do
         ......
```

Arrays "a" and "b" are shared

```fortran
module mod_global_data

   implicit none

   integer, parameter:: m= .., n= ..
   integer            :: a(m,n), b(m)

end module mod_global_data
```

Global Data – An Example/2

```
subroutine suba(j)
  .....

use mod_global_data

  .....

do i = 1, m
   b(i) = j
end do

do i = 1, m
  a(i,j) = func_call(b(i))
end do

return
end
```

Data Race !

Global Data - A Data Race!

Thread 1

Thread 2

```
          call suba(1)
```

```
          call suba(2)
```

Shared

```
subroutine suba(j=1)
```

```
subroutine suba(j=2)
```

```
do i = 1, m
   b(i) = 1
end do
```

```
do i = 1, m
   b(i) = 2
end do
```

```
do i = 1, m
  a(i,1)=func_call(b(i))
end do
```

```
do i = 1, m
  a(i,2)=func_call(b(i))
end do
```
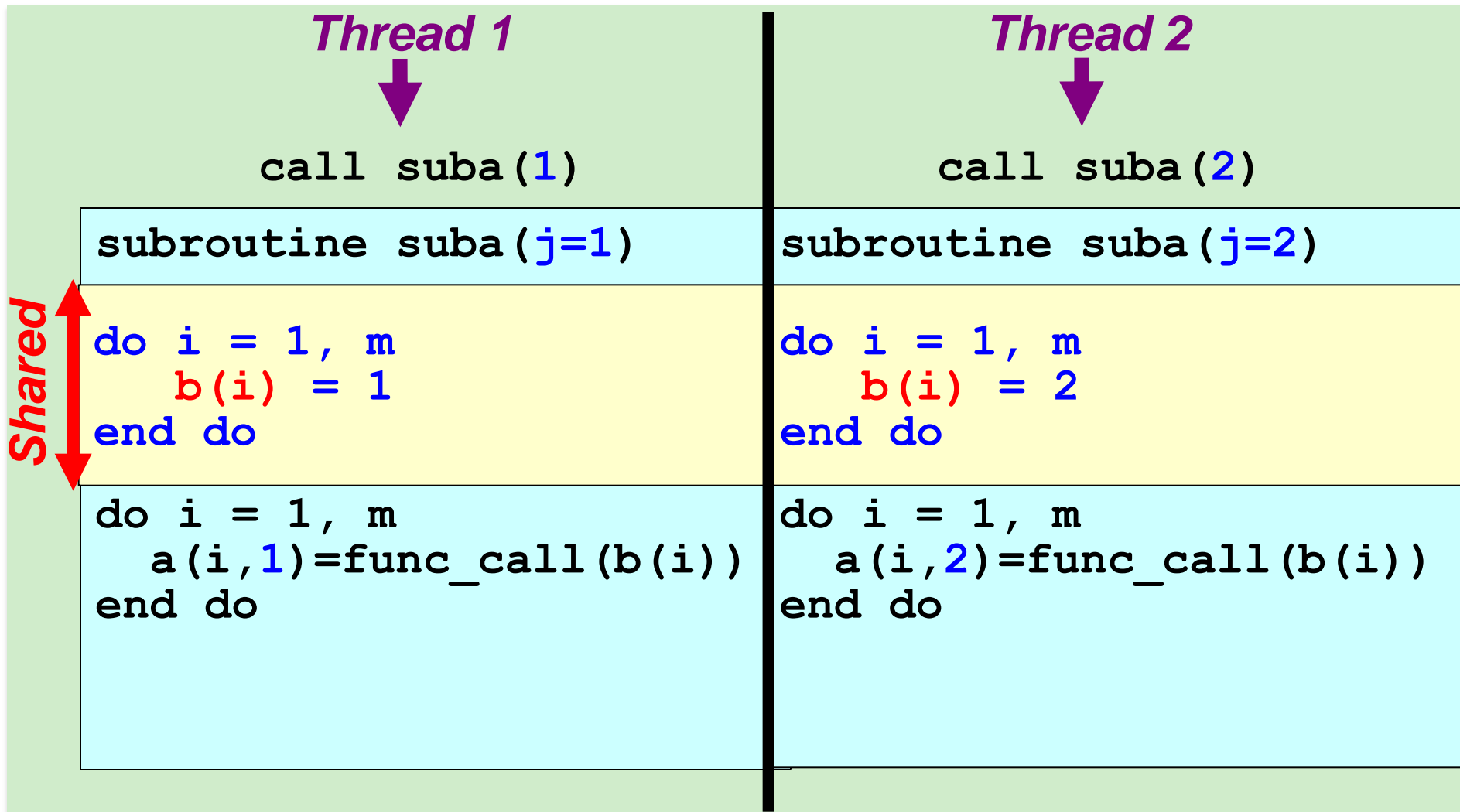
Global Data – A Solution/1

```fortran
      program global_data
              ....
      use mod_global_data
              ....
!$omp parallel do private(j)
      do j = 1, n
         call suba(j)
      end do
!$omp end parallel do
         ......
```

Make array "b"
2-dimensional

```fortran
module mod_global_data

   implicit none

   integer, parameter:: m= .., n= ..
   integer, parameter:: nthreads = ...
   integer            :: a(m,n), b(m,nthreads)

end module mod_global_data
```

Global Data – A Solution/2

```fortran
subroutine suba(j)
  .....

use omp_lib
use mod_global_data

  .....

TID = omp_get_thread_num()+1
do i = 1, m
   b(i,TID) = j
end do

do i = 1, m
   a(i,j) = func_call(b(i),TID)
end do

return
end
```
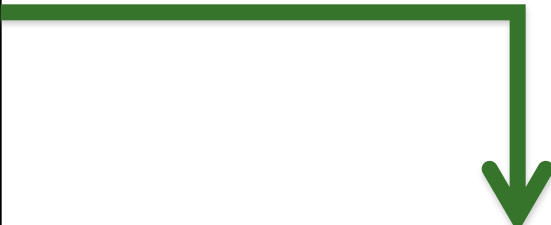
A lot of work and not very portable

Global Data – The Preferred Solution

```fortran
      program global_data
             ....
      use mod_global_data
             ....
!$omp parallel do private(j)
      do j = 1, n
          call suba(j)
      end do
!$omp end parallel do
      ......
```

Only add the "threadprivate" directive to the module file; no other changes needed !

This solution also automatically adapts to the number of threads used

```fortran
module mod_global_data

   implicit none

   integer, parameter:: m= .., n= ..
   integer            :: a(m,n), b(m)

   !$omp threadprivate(b)

end module mod_global_data
```

Recap: About Global Data

❑ Global data is shared: take care when using it
❑ Potential problems if multiple threads access the same memory simultaneously:
  - ❑ Read-only data is no problem
  - ❑ Updates have to be checked for race conditions
❑ It is your responsibility to deal with this situation
❑ In general one can do the following:
  - ❑ Split the global data into a part that is accessed in serial code only and a part that is accessed in parallel
  - ❑ Manually create copies of the latter
  - ❑ Use the thread ID to access these copies
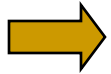❑ Alternative: Use OpenMP's threadprivate directive !

Agenda

❑ Hybrid Programming with MPI and OpenMP

❑ Using OpenMP
  ❑ Common programming errors
  ❑ Performance Topics

General Comments on Performance

❑ Be aware of overheads of OpenMP constructs, thread management
  - ❑ Microbenchmarks help here*
  - ❑ Don't create too many parallel regions
  - ❑ Dynamic loop schedules have much higher overheads than static schedules
  - ❑ Synchronization is expensive, so minimize
    - ❑ use **nowait** where possible
    - ❑ privatize data
    - ❑ minimize code in critical region
  - ❑ Choose default behavior carefully
    - ❑ Use appropriate schedules
    - ❑ Wait policy (OMP_WAIT_POLICY=passive|active)

* J. M. Bull and D. O'Neill, A microbenchmark suite for OpenMP 2.0, SIGARCH Comput. Archit. News, vol. 29, no. 5, pp. 41–48, 2001.

General Comments on Performance

❏ Thread / Data Affinity

  ❏ Check on your implementation's documentation to control for this

    ❏ e.g. KMP_AFFINITY for Intel, GOMP_CPU_AFFINITY for GNU

  ❏ Other tools (e.g. taskset, numactl, likwid) can help with this

  ❏ **OpenMP 4.0** includes features to control for this

❏ Structure and characteristics of program

  ❏ Minimize sequential part of program

  ❏ Be aware of and address load balance

  ❏ Address cache utilization and false sharing (it can kill any speedup if not addressed)

  ❏ Large parallel regions help reduce overheads, enable better cache usage and standard optimizations

❏ Quality of compiler is also a factor on achievable performance

# Briefly, on OpenMP Implementations

- ❑ **Directives implemented via code modification and insertion of runtime library calls**
  - ❑ Typical approach is outlining of code in parallel region
  - ❑ Or generation of micro tasks
- ❑ **Runtime library responsible for managing threads**
  - ❑ Scheduling loops
  - ❑ Scheduling tasks
  - ❑ Implementing synchronization
  - ❑ Collector API provides interface to give external tools state information
- ❑ **Implementation effort is reasonable**

OpenMP Code	Translation

int main(void)
{
int a,b,c;
#pragma omp parallel \ private(c)
do_sth(a,b,c);
return 0;
}
	_INT32 main()
	{
	int a,b,c;
	/* microtask */
	void __ompregion_main1()
	{
	_INT32 __mplocal_c;
	/*shared variables are kept intact, substitute accesses to private variable*/
	do_sth(a, b, __mplocal_c);
	}
	...
	/*OpenMP runtime calls */
	__ompc_fork(&__ompregion_main1);
	...
	}

Each compiler has custom run-time support. Quality of the runtime system has major impact on performance.

OpenMP and Data Locality

❑ **Implicit Data Locality**

  ❑ Thread fetches data it needs into local cache

  ❑ Emphasis on privatizing data where possible, and optimizing code for cache

  ❑ Implicit means of data layout on NUMA systems

    ❑ "First touch" as introduced by SGI for Origin

❑ **Emphasis on privatizing data where possible, and optimizing code for cache**

  ❑ This can work pretty well

  ❑ But small mistakes may be costly
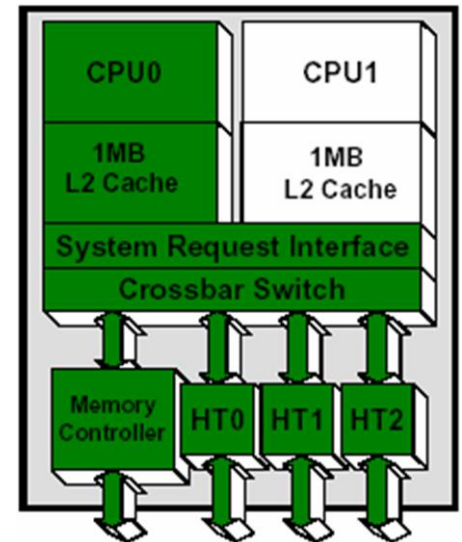
Tuning: Critical Regions

❑ It often helps to chop up large critical sections into finer, named ones

❑ Original Code

```
#pragma omp critical (foo)
{
    update( a );
    update( b );
}
```

❑ Transformed Code

```
#pragma omp critical (foo_a)
    update( a );
#pragma omp critical (foo_b)
    update( b );
```

Tuning: Locks Instead of Critical

❑ **Original Code**

```
#pragma omp critical
  for( i=0; i<n; i++ ) {
    a[i] = …
    b[i] = …
    c[i] = …
  }
```

❑ Idea: cycle through different parts of the array using locks!

❑ **Transformed Code**

```
jstart = omp_get_thread_num();
for( k = 0; k < nlocks; k++ ) {
  j = ( jstart + k ) % nlocks;
  omp_set_lock( lck[j] );
  for( i=lb[j]; i<ub[j]; i++ ) {
    a[i] = …
    b[i] = …
    c[i] = …
  }
  omp_unset_lock( lck[j] );
}
```

❑ Adapt to your situation

Tuning: Eliminate Implicit Barriers

- ❑ Worksharing constructs have implicit barrier at end
- ❑ If consecutive work-sharing constructs modify (& use) different objects, the barrier in the middle can be eliminated
- ❑ If same object modified (or used), barrier can be safely removed if iteration spaces guaranteed to align

no barriers needed here

```
#pragma omp for nowait
for (i = 0; i < N; i++) {
    d[i] = a[i] + b[i]*c[i];
}

#pragma omp for schedule(runtime)
for (i = 0; i < N; i++) {
    e[i] = c[i] + b[i]*a[i];
}
```

no dependences between these loops

```
#pragma omp for nowait
for (i = 0; i < N; i++) {
    d[i] = a[i] + b[i]*c[i];
}

#pragma omp for
for (i = 0; i < N; i++) {
    e[i] = d[i] + b[i]*c[i];
}
```

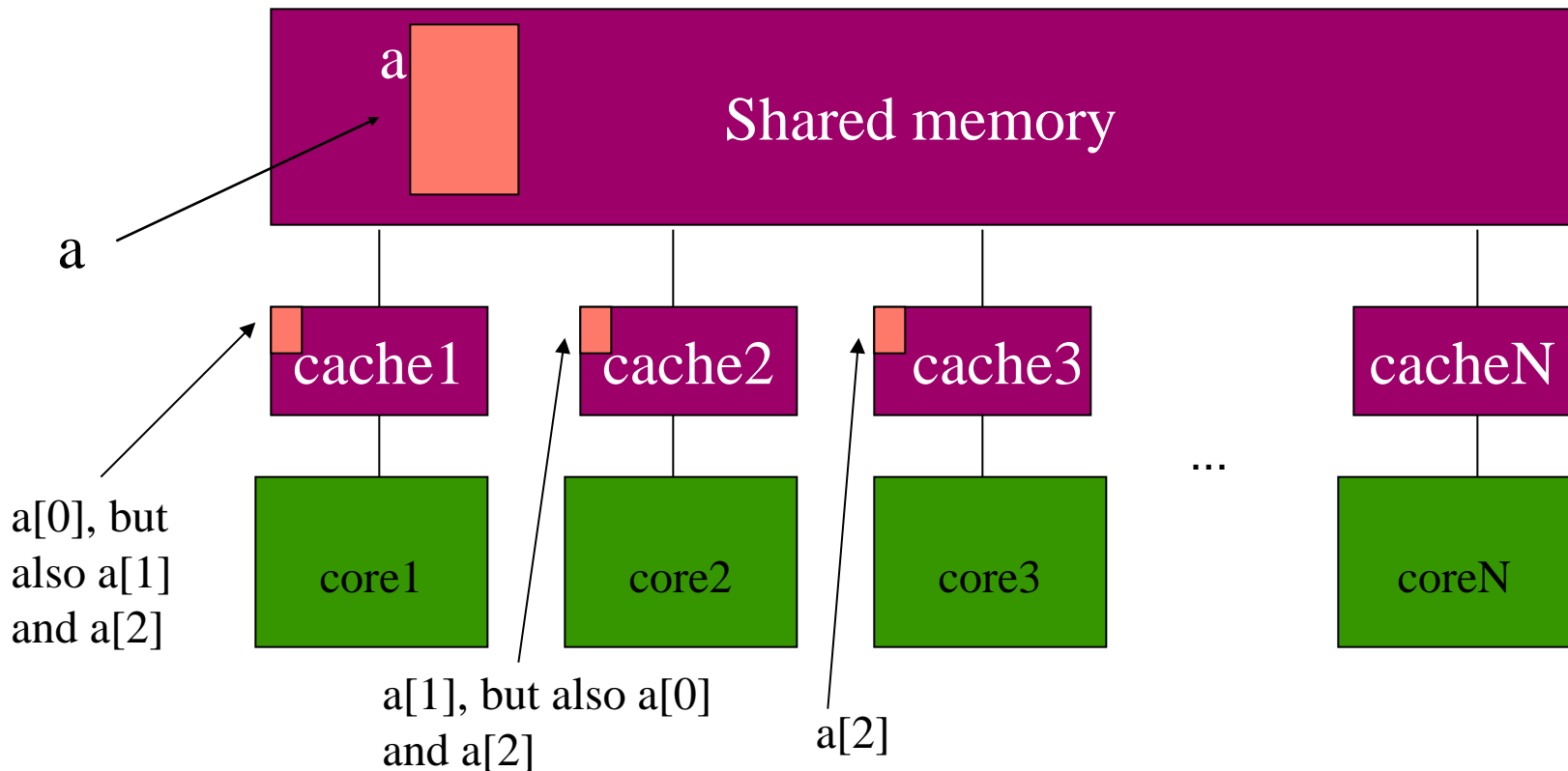spec guarantees same iteration-to-thread mapping

Cache Coherence and False Sharing

- ❑ Blocks of data are fetched into cache lines
- ❑ What happens if multiple threads access different data, but on same cache line, at same time?
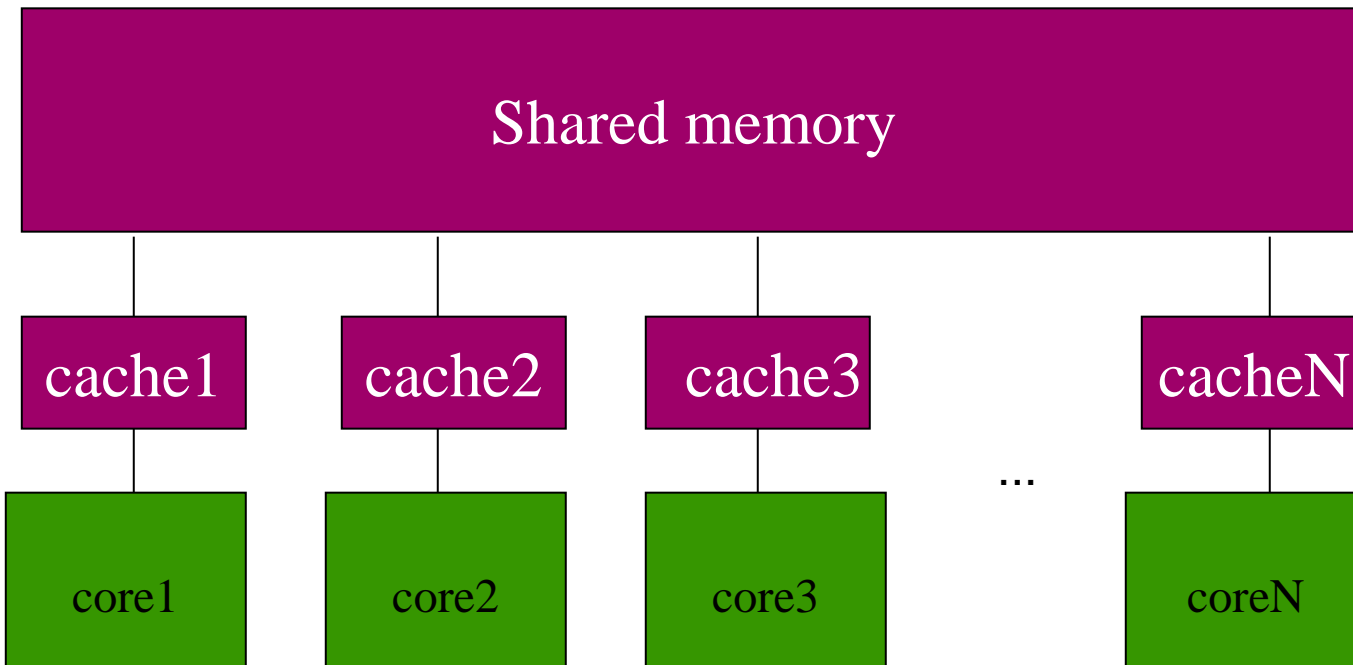
Updates to Shared Data

- Blocks of data are transferred to cache lines
- When an element of cache line is updated, the entire line is invalidated: local copies are reloaded from main memory

a

Shared memory

a

cache1   cache2   cache3   ...   cacheN

a[0], but also a[1] and a[2]

core1   core2   core3   coreN

a[1], but also a[0] and a[2]

a[2]

Small "Mistakes", Big Consequences

- ❑ GenIDLEST
  - ❑ Scientific simulation code
  - ❑ Solves incompressible Navier Stokes and energy equations
  - ❑ MPI and OpenMP versions
- ❑ Platform
  - ❑ SGI Altix 3700 (NUMA)
  - ❑ 512 Itanium 2 Processors
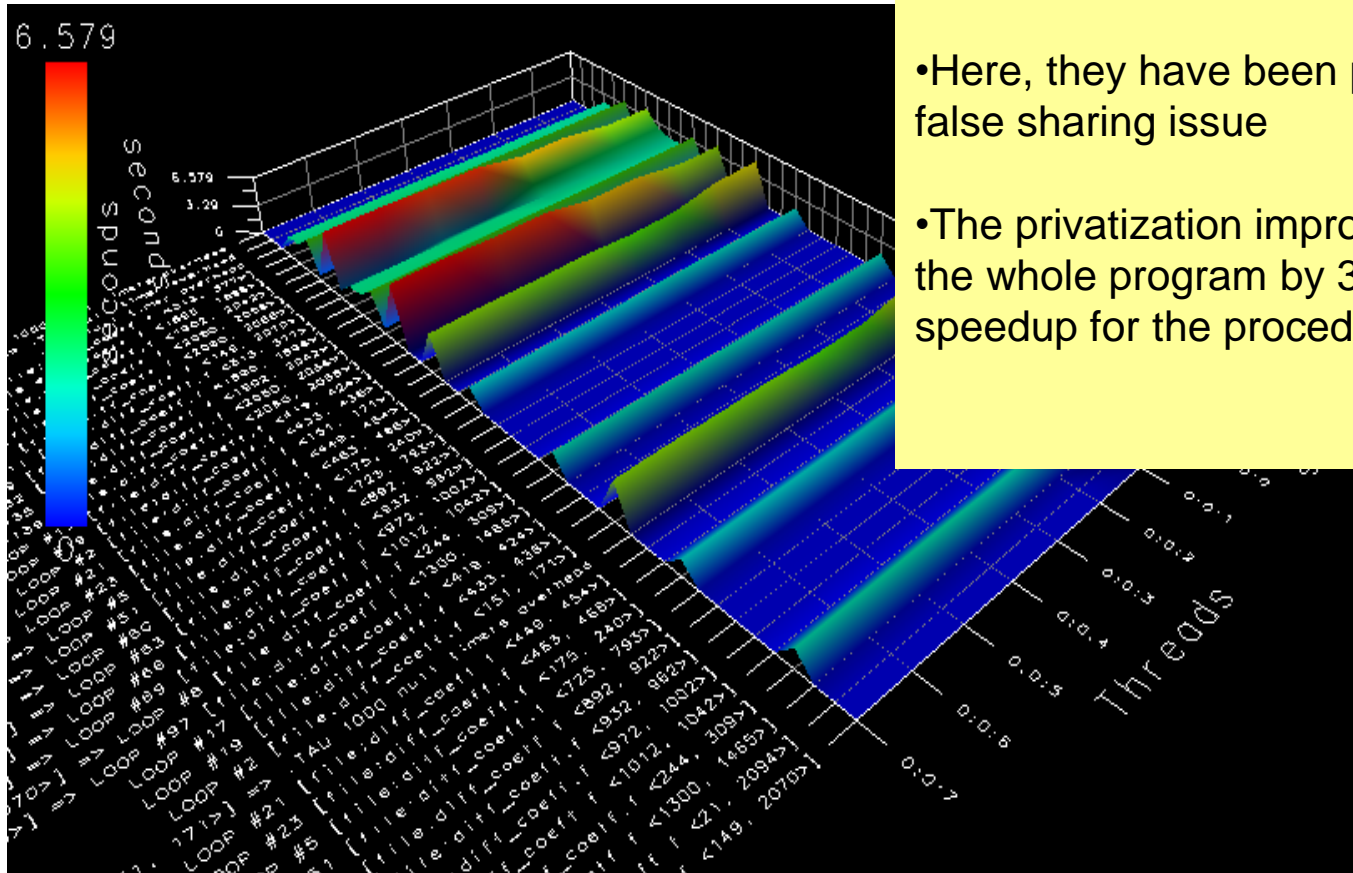- ❑ OpenMP code slower than MPI
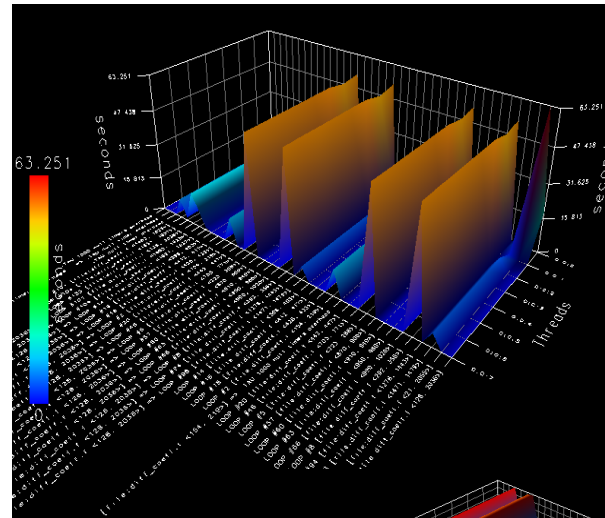
OpenMP version
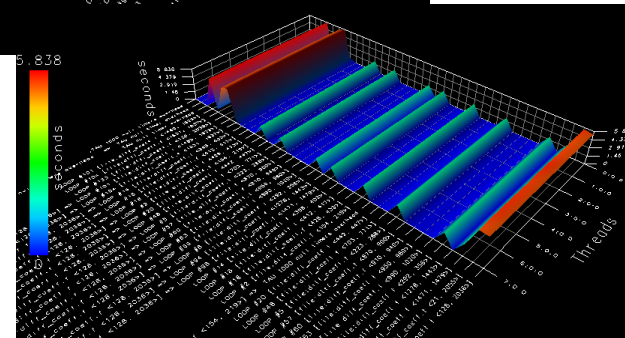
MPI version

In the OpenMP version, a single procedure is responsible for 20% of the total time and is 9 times slower than the MPI version . Its loops are up to 27 times slower in OpenMP than MPI.

A Solution: Privatization

OpenMP Optimized Version

• Lower and upper bounds of arrays used privately by threads are shared, **stored in same memory page and cache line**

•Here, they have been privatized to eliminate false sharing issue

•The privatization improved the performance of the whole program by 30% and led to a 10x speedup for the procedure.

False Sharing: Monitoring Results

❏ Phoenix codes ported from Pthreads to OpenMP

❏ 5 out of 8 apps show symptoms of false sharing

Cache Invalidation Count

Program name	1-thread	2-threads	4-threads	8-threads
histogram	13	**7,820,000**	**16,532,800**	**5,959,190**
kmeans	383	28,590	47,541	54,345
linear_regression	9	**417,225,000**	**254,442,000**	**154,970,000**
matrix_multiply	31,139	31,152	84,227	101,094
pca	44,517	46,757	80,373	122,288
reverse_index	4,284	89,466	217,884	**590,013**
string_match	82	**82,503,000**	**73,178,800**	**221,882,000**
word_count	4,877	**6,531,793**	**18,071,086**	**68,801,742**

False Sharing: Data Analysis Results

❑ Determining the variables that cause misses

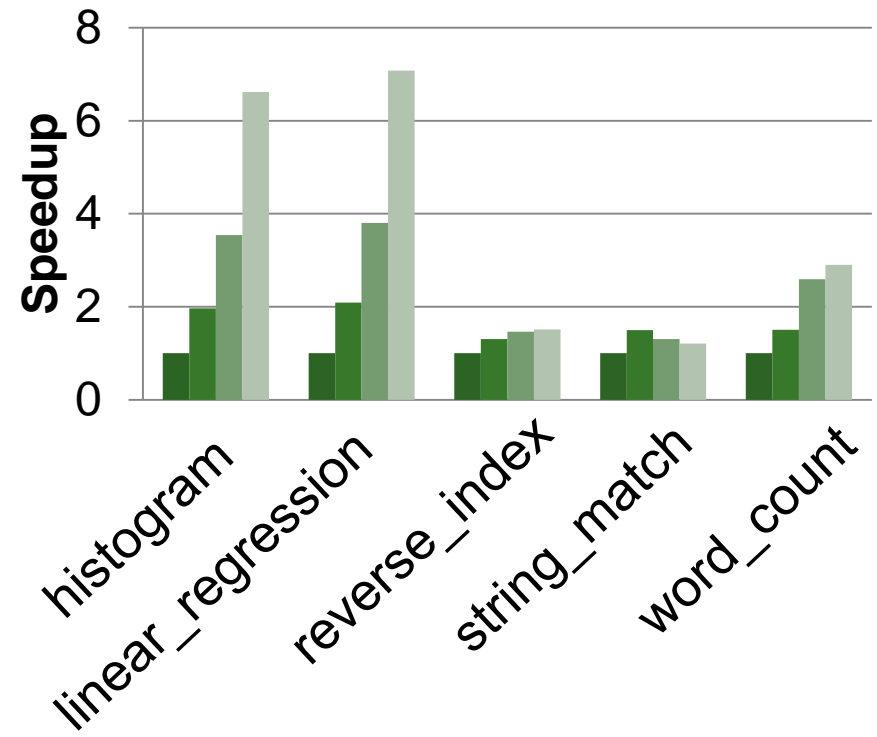| Program Name	Global/static data	Dynamic data
histogram	-	main_221
linear_regression	-	main_155
reverse_index	use_len	main_519
string_match	key2_final	string_match_map_266
word_count	length, use_len, words	-

Runtime False Sharing Detection

Original Version

- 1-thread
- 2-threads
- 4-threads
- 8-threads
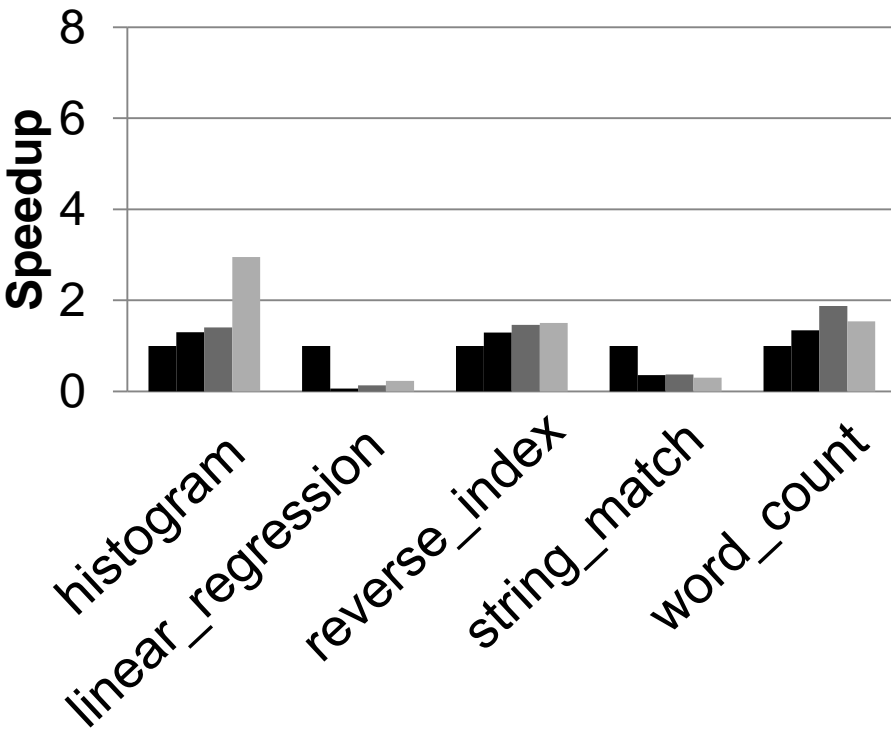
Optimized Version

- 1-thread
- 2-threads
- 4-threads
- 8-threads

B. Wicaksono, M. Tolubaeva and B. Chapman. "Detecting false sharing in OpenMP applications using the DARWIN framework", LCPC 2011

Summary

- ❑ OpenMP is designed to be easy to use, but there are several pitfalls to avoid
  - ❑ Data races are a common programming error in shared memory programming which can be hard to spot – know when to privatize your data!
  - ❑ Beware of subtle synchronization error
    - ❑ unless you're very careful, stick to OpenMP directives
  - ❑ Know the overheards associated with the constructs you're using
  - ❑ Know how to control thread and data placement
  - ❑ False sharing can also kill performance