Advanced Parallel Programming with OpenMP

Tim Mattson
Intel Corporation
Computational Sciences Laboratory

Rudolf Eigenmann
Purdue University
School of Electrical and Computer Engineering
SC’2000 Tutorial Agenda

- OpenMP: A Quick Recap
- OpenMP Case Studies
  - including performance tuning
- Automatic Parallelism and Tools Support
- Common Bugs in OpenMP programs
  - and how to avoid them
- Mixing OpenMP and MPI
- The Future of OpenMP
OpenMP: Recap

OpenMP: An API for Writing Multithreaded Applications

- A set of compiler directives and library routines for parallel application programmers
- Makes it easy to create multi-threaded (MT) programs in Fortran, C and C++
- Standardizes last 15 years of SMP practice
OpenMP: Supporters*

- **Hardware vendors**
  - Intel, HP, SGI, IBM, SUN, Compaq

- **Software tools vendors**
  - KAI, PGI, PSR, APR

- **Applications vendors**
  - ANSYS, Fluent, Oxford Molecular, NAG, DOE ASCI, Dash, Livermore Software, and many others

*These names of these vendors were taken from the OpenMP web site (www.openmp.org). We have made no attempts to confirm OpenMP support, verify conformity to the specifications, or measure the degree of OpenMP utilization.
OpenMP: Programming Model

Fork-Join Parallelism:

- Master thread spawns a team of threads as needed.
- Parallelism is added incrementally: i.e. the sequential program evolves into a parallel program.
OpenMP:
How is OpenMP typically used? (in C)

- OpenMP is usually used to parallelize loops:
  - Find your most time consuming loops.
  - Split them up between threads.

```c
void main()
{
    double Res[1000];
    for(int i=0; i<1000; i++) {
        do_huge_comp(Res[i]);
    }
}
```

```c
#include "omp.h"
void main()
{
    double Res[1000];
    #pragma omp parallel for
    for(int i=0; i<1000; i++) {
        do_huge_comp(Res[i]);
    }
}
```

Split-up this loop between multiple threads
OpenMP: How is OpenMP typically used? (Fortran)

- OpenMP is usually used to parallelize loops:
  - Find your most time consuming loops.
  - Split them up between threads.

Example:

```fortran
program example
    double precision Res(1000)
    do I=1,1000
        call huge_comp(Res(I))
    end do
end
```

Split-up this loop between multiple threads:

```fortran
C$OMP PARALLEL DO
    do I=1,1000
        call huge_comp(Res(I))
    end do
end
```

Sequential Program

Parallel Program
OpenMP: 
How do threads interact?

- OpenMP is a shared memory model.
  - Threads communicate by sharing variables.

- Unintended sharing of data causes race conditions:
  - race condition: when the program’s outcome changes as the threads are scheduled differently.

- To control race conditions:
  - Use synchronization to protect data conflicts.

- Synchronization is expensive so:
  - Change how data is accessed to minimize the need for synchronization.
Summary of OpenMP Constructs

- **Parallel Region**
  - `C$omp parallel`  #pragma omp parallel

- **Worksharing**
  - `C$omp do`  #pragma omp for
  - `C$omp sections`  #pragma omp sections
  - `C$ingle`  #pragma omp single
  - `C$workshare`  #pragma workshare

- **Data Environment**
  - **directive**: `threadprivate`
  - **clauses**: `shared`, `private`, `lastprivate`, `reduction`, `copyin`, `copyprivate`

- **Synchronization**
  - **directives**: `critical`, `barrier`, `atomic`, `flush`, `ordered`, `master`

- **Runtime functions/environment variables**
SC’2000 Tutorial Agenda

- OpenMP: A Quick Recap
- OpenMP Case Studies
  - including performance tuning
- Automatic Parallelism and Tools Support
- Common Bugs in OpenMP programs
  - and how to avoid them
- Mixing OpenMP and MPI
- The Future of OpenMP
Performance Tuning and Case Studies with Realistic Applications

1. Performance tuning of several benchmarks
2. Case study of a large-scale application
Performance Tuning
Example 1: MDG

- MDG: A Fortran code of the “Perfect Benchmarks”.
- Automatic parallelization does not improve this code.

These performance improvements were achieved through manual tuning on a 4-processor Sun Ultra:
MDG: Tuning Steps

Step 1: Parallelize the most time-consuming loop. It consumes 95% of the serial execution time. This takes:
- array privatization
- reduction parallelization

Step 2: Balancing the iteration space of this loop.
- Loop is “triangular”. By default unbalanced assignment of iterations to processors.
MDG Code Sample

Structure of the most time-consuming loop in MDG:

Original

```
c1 = x(1)>0
c2 = x(1:10)>0

DO i=1,n
  DO j=i,n
    IF (c1) THEN rl(1:100) = …
    …
    IF (c2) THEN … = rl(1:100)
    sum(j) = sum(j) + …
  ENDDO
ENDDO
```

Parallel

```
c1 = x(1)>0
paralle 
c2 = x(1:10)>0

Allocate(xsum(1:#proc,n))
C$OMP PARALLEL DO
C$OMP+ PRIVATE (i,j,rl,id) C$OMP+ SCHEDULE (STATIC,1)
DO i=1,n
  id = omp_get_thread_num()
  DO j=i,n
    IF (c1) THEN rl(1:100) = …
    …
    IF (c2) THEN … = rl(1:100)
    xsum(id,j) = xsum(id,j) + …
  ENDDO
ENDDO
C$OMP PARALLEL DO
DO i=1,n
  sum(i)=sum(i)+xsum(1:#proc,i)
ENDDO
```
Performance Tuning
Example 2: ARC2D

ARC2D: A Fortran code of the “Perfect Benchmarks”.

ARC2D is parallelized very well by available compilers. However, the mapping of the code to the machine could be improved.
ARC2D: Tuning Steps

- **Step 1:**
  
  Loop interchanging increases cache locality through stride-1 references

- **Step 2:**
  
  Move parallel loops to outer positions

- **Step 3:**
  
  Move synchronization points outward

- **Step 4:**
  
  Coalesce loops
ARC2D: Code Samples

```
!$OMP PARALLEL DO
!$OMP+PRIVATE(R1,R2,K,J)
DO j = jlow, jup
  DO k = 2, kmax-1
    r1 = prss(jminu(j), k) + prss(jplus(j), k) + (-2.)*prss(j, k)
    r2 = prss(jminu(j), k) + prss(jplus(j), k) + 2.*prss(j, k)
    coef(j, k) = ABS(r1/r2)
  ENDDO
ENDDO
!$OMP END PARALLEL
```

Loop interchanging increases cache locality

```
!$OMP PARALLEL DO
!$OMP+PRIVATE(R1,R2,K,J)
DO k = 2, kmax-1
  DO j = jlow, jup
    r1 = prss(jminu(j), k) + prss(jplus(j), k) + (-2.)*prss(j, k)
    r2 = prss(jminu(j), k) + prss(jplus(j), k) + 2.*prss(j, k)
    coef(j, k) = ABS(r1/r2)
  ENDDO
ENDDO
!$OMP END PARALLEL
```
Increasing parallel loop granularity through NOWAIT clause

```c
!$OMP PARALLEL
!$OMP+PRIVATE(LDI,LD2,LD1,J,LD,K)
    DO k = 2+2, ku-2, 1
!$OMP DO
    DO j = jl, ju
        ld2 = a(j, k)
        ld1 = b(j, k)+(-x(j, k-2))*ld2
        ld = c(j, k)+(-x(j, k-1))*ld1+(-y(j, k-1))*ld2
        ldi = 1./ld
        f(j, k, 1) = ldi*(f(j, k, 1)+(-f(j, k-2, 1))*ld2+(-f(j, k-1, 1))*ld1)
        f(j, k, 2) = ldi*(f(j, k, 2)+(-f(j, k-2, 2))*ld2+(-f(jk-2, 2))*ld1)
        x(j, k) = ldi*(d(j, k)+(-y(j, k-1))*ld1)
        y(j, k) = e(j, k)*ldi
    ENDDO
!$OMP END DO  NOWAIT
ENDDO
!$OMP END PARALLEL
```
ARC2D: Code Samples

```fortran
$OMP PARALLEL DO
$OMP+PRIVATE(n, k,j)
  DO  n = 1, 4
  DO k = 2, kmax-1
    DO j = jlow, jup
      q(j, k, n) = q(j, k, n)+s(j, k, n)
      s(j, k, n) = s(j, k, n)*phic
    ENDDO
  ENDDO
ENDDO
$OMP END PARALLEL

$OMP PARALLEL DO
$OMP+PRIVATE(nk,n,k,j)
  DO nk = 0,4*(kmax-2)-1
    n = nk/(kmax-2) + 1
    k = MOD(nk,kmax-2)+2
    DO j = jlow, jup
      q(j, k, n) = q(j, k, n)+s(j, k, n)
      s(j, k, n) = s(j, k, n)*phic
    ENDDO
  ENDDO
ENDDO
ENDDO
$OMP END PARALLEL
```

Increasing parallel loop granularity though loop coalescing
Performance Tuning Example 3: EQUAKE

EQUAKE: A C code of the new SPEC OpenMP benchmarks.

EQUAKE is hand-parallelized with relatively few code modifications. It achieves excellent speedup.
EQUAKE: Tuning Steps

● **Step1:**
  Parallelizing the four most time-consuming loops
  – inserted OpenMP pragmas for parallel loops and private data
  – array reduction transformation

● **Step2:**
  A change in memory allocation
/* malloc \texttt{w1[numthreads][ARCHnodes][3]} */

\#pragma omp parallel for
for (j = 0; j < numthreads; j++)
    for (i = 0; i < nodes; i++) { \texttt{w1[j][i][0] = 0.0; ...;}

\#pragma omp parallel private(my\_cpu\_id,exp,...)
{ 
    my\_cpu\_id = omp\_get\_thread\_num();

\#pragma omp for
for (i = 0; i < nodes; i++)
    while (...) {
        ...
        exp = loop-local computation;
        \texttt{w1[my\_cpu\_id][...][1]} += exp;
        ...
    }
}

\#pragma omp parallel for
for (j = 0; j < numthreads; j++) {
    for (i = 0; i < nodes; i++) { \texttt{w[i][0] += w1[j][i][0]; ...;}}
What Tools Did We Use for Performance Analysis and Tuning?

- Compilers
  - the starting point for our performance tuning of Fortran codes was always the compiler-parallelized program.
  - It reports: parallelized loops, data dependences.

- Subroutine and loop profilers
  - focusing attention on the most time-consuming loops is absolutely essential.

- Performance tables:
  - typically comparing performance differences at the loop level.
Guidelines for Fixing “Performance Bugs”

The methodology that worked for us:

- Use compiler-parallelized code as a starting point
- Get loop profile and compiler listing
- Inspect time-consuming loops (biggest potential for improvement)

- Case 1. Check for parallelism where the compiler could not find it
- Case 2. Improve parallel loops where the speedup is limited
Performance Tuning

Case 1: if the loop is not parallelized automatically, do this:

- **Check for parallelism:**
  - read the compiler explanation
  - a variable may be independent even if the compiler detects dependences (compilers are conservative)
  - check if conflicting array is privatizable (compilers don’t perform array privatization well)

- **If you find parallelism, add OpenMP parallel directives, or make the information explicit for the parallelizer**
Performance Tuning

Case 2: if the loop is parallel but does not perform well, consider several optimization factors:

Parallelization overhead
- High overheads are caused by:
  - parallel startup cost
  - small loops
  - additional parallel code
  - over-optimized inner loops
  - less optimization for parallel code
  - load imbalance
  - synchronized section
  - non-stride-1 references
  - many shared references
  - low cache affinity

Spreading overhead

Parallel program
- serial program

Memory
CPU
CPU
CPU
Case Study of a Large-Scale Application

Converting a Seismic Processing Application to OpenMP

- Overview of the Application
- Basic use of OpenMP
- OpenMP Issues Encountered
- Performance Results
Overview of Seismic

- Representative of modern seismic processing programs used in the search for oil and gas.
- 20,000 lines of Fortran. C subroutines interface with the operating system.
- Available in a serial and a parallel variant.
- Parallel code is available in a message-passing and an OpenMP form.
- Is part of the SPECchpc benchmark suite. Includes 4 data sets: small to x-large.
Seismic: Basic Characteristics

- Program structure:
  - 240 Fortran and 119 C subroutines.
- Main algorithms:
  - FFT, finite difference solvers
- Running time of Seismic (@ 500MFlops):
  - small data set: 0.1 hours
  - x-large data set: 48 hours
- IO requirement:
  - small data set: 110 MB
  - x-large data set: 93 GB
Basic OpenMP Use: Parallelization Scheme

- Split into $p$ parallel tasks
  ($p = \text{number of processors}$)

Program Seismic

<table>
<thead>
<tr>
<th>Initialization</th>
<th>Initialization done by master processor only</th>
</tr>
</thead>
<tbody>
<tr>
<td>C$OMP$ PARALLEL call main_subroutine()</td>
<td>Main computation enclosed in one large parallel region</td>
</tr>
<tr>
<td>C$OMP$ END PARALLEL</td>
<td></td>
</tr>
</tbody>
</table>

→ SPMD execution scheme
Basic OpenMP Use: Data Privatization

- Most data structures are *private*, i.e., Each thread has its own copy.
- Syntactic forms:

Program Seismic
...
C$OMP PARALLEL
C$OMP+PRIVATE(a)
  a = "local computation"
call x()
C$END PARALLEL

Subroutine x()
common /cc/ d
C$omp threadprivate (/cc/)
real b(100)
...
b() = "local computation"
d = "local computation"
...
Basic OpenMP Use: Synchronization and Communication

- compute
- communicate
- compute
- communicate

Copy to shared buffer; barrier_synchronization; copy from shared buffer;

Copy-synchronize scheme corresponds to message send-receive operations in MPI programs.
OpenMP Issues: Mixing Fortran and C

- Bulk of computation is done in Fortran
- Utility routines are in C:
  - IO operations
  - data partitioning routines
  - communication/synchronization operations
- OpenMP-related issues:
  - IF C/OpenMP compiler is not available, data privatization must be done through “expansion”.
  - Mix of Fortran and C is implementation dependent

Data privatization in OpenMP/C
```c
#pragma omp thread private (item)
float item;
void x()
{
    ... = item;
}
```

Data expansion in absence a of OpenMP/C compiler
```c
float item[num_proc];
void x()
{
    int thread;
    thread = omp_get_thread_num_();
    ... = item[thread];
}
```
OpenMP Issues:
Broadcast Common Blocks

common /cc/ cdata
common /dd/ ddata

C$OMP PARALLEL
C$OMP+COPYIN(/cc/, /dd/)
call main_subroutine()
C$END PARALLEL

Issues in Seismic:
- At the start of the parallel region it is not yet known which common blocks need to be copied in.

Solution:
- copy-in all common blocks → overhead
OpenMP Issues: Multithreading IO and malloc

IO routines and memory allocation are called within parallel threads, inside C utility routines.

- OpenMP requires all standard libraries and intrinsics to be thread-safe. However the implementations are not always compliant.
  → system-dependent solutions need to be found

- The same issue arises if standard C routines are called inside a parallel Fortran region or in non-standard syntax.

  Standard C compilers do not know anything about OpenMP and the thread-safe requirement.
OpenMP Issues: Processor Affinity

- OpenMP currently does not specify or provide constructs for controlling the binding of threads to processors.

- Processors can migrate, causing overhead. This behavior is system-dependent. System-dependent solutions may be available.
Performance Results

Speedups of Seismic on an SGI Challenge system

- small data set
- medium data set
SC’2000 Tutorial Agenda

- OpenMP: A Quick Recap
- OpenMP Case Studies
  - including performance tuning
- Automatic Parallelism and Tools Support
- Common Bugs in OpenMP programs
  - and how to avoid them
- Mixing OpenMP and MPI
- The Future of OpenMP
Generating OpenMP Programs Automatically

Source-to-source restructurers:
- F90 to F90/OpenMP
- C to C/OpenMP

Examples:
- SGI F77 compiler (-apo -mplist option)
- Polaris compiler
The Basics About Parallelizing Compilers

- Loops are the primary source of parallelism in scientific and engineering applications.
- Compilers detect loops that have independent iterations.

DO I=1,N
   A(expression1) = ...
   ... = A(expression2)
ENDDO

The loop is independent if, for different iterations, expression1 is always different from expression2.
Basic Program Transformations

Data privatization:

```fortran
DO i=1,n
    work(1:n) = ....
    ....
    .... = work(1:n)
ENDDO
```

```fortran
C$OMP PARALLEL DO
C$OMP+ PRIVATE (work)
DO i=1,n
    work(1:n) = ....
    ....
    .... = work(1:n)
ENDDO
```

Each processor is given a separate version of the private data, so there is no sharing conflict.
Basic Program Transformations

Reduction recognition:

```
DO i=1,n  
   ...  
   sum = sum + a(i)  
   ...  
ENDDO
```

```
C$OMP PARALLEL DO
C$OMP+ REDUCTION (+:sum)
DO i=1,n  
   ...  
   sum = sum + a(i)  
   ...  
ENDDO
```

Each processor will accumulate partial sums, followed by a combination of these parts at the end of the loop.
Basic Program Transformations

Induction variable substitution:

\[
\begin{align*}
  &i1 = 0 \\
  &i2 = 0 \\
  &\text{DO } i = 1, n \\
  &\quad i1 = i1 + 1 \\
  &\quad B(i1) = \ldots \\
  &\quad i2 = i2 + i \\
  &\quad A(i2) = \ldots \\
  &\text{ENDDO}
\end{align*}
\]

C$OMP PARALLEL DO
\[
\begin{align*}
  &\text{DO } i = 1, n \\
  &\quad B(i) = \ldots \\
  &\quad A((i**2 + i)/2) = \ldots \\
  &\text{ENDDO}
\end{align*}
\]

The original loop contains data dependences: each processor modifies the shared variables \(i1\), and \(i2\).
Compiler Options

Examples of options from the KAP parallelizing compiler (KAP includes some 60 options)

- **optimization levels**
  - `optimize`: simple analysis, advanced analysis, loop interchanging, array expansion
  - `aggressive`: pad common blocks, adjust data layout

- **subroutine inline expansion**
  - inline all, specific routines, how to deal with libraries

- **try specific optimizations**
  - e.g., recurrence and reduction recognition, loop fusion
  (These transformations may degrade performance)
More About Compiler Options

- **Limits on amount of optimization:**
  - e.g., size of optimization data structures, number of optimization variants tried

- **Make certain assumptions:**
  - e.g., array bounds are not violated, arrays are not aliased

- **Machine parameters:**
  - e.g., cache size, line size, mapping

- **Listing control**

Note, compiler options can be a substitute for advanced compiler strategies. If the compiler has limited information, the user can help out.
Inspecting the Translated Program

- **Source-to-source restructurers:**
  - transformed source code is the actual output
  - Example: KAP

- **Code-generating compilers:**
  - typically have an option for viewing the translated (parallel) code
  - Example: SGI f77 -apo -mplist

This can be the starting point for code tuning
Compiler Listing

The listing gives many useful clues for improving the performance:

- Loop optimization tables
- Reports about data dependences
- Explanations about applied transformations
- The annotated, transformed code
- Calling tree
- Performance statistics

The type of reports to be included in the listing can be set through compiler options.
Tuning Automatically-Parallelized Code

• This task is similar to explicit parallel programming.

• Two important differences:
  ◆ The compiler gives hints in its listing, which may tell you where to focus attention. E.g., which variables have data dependences.
  ◆ You don’t need to perform all transformations by hand. If you expose the right information to the compiler, it will do the translation for you. (E.g., C$assert independent)
Why Tuning Automatically-Parallelized Code?

Hand improvements can pay off because

- compiler techniques are limited
  E.g., array reductions are parallelized by only few compilers

- compilers may have insufficient information
  E.g.,
  - loop iteration range may be input data
  - variables are defined in other subroutines (no interprocedural analysis)
Performance Tuning Tools

- User inserts directives
- Parallelizing compiler inserts directives
- OpenMP program
- User tunes program

We need tool support
Profiling Tools

- Timing profiles (subroutine or loop level)
  - shows most time-consuming program sections
- Cache profiles
  - point out memory/cache performance problems
- Data-reference and transfer volumes
  - show performance-critical program properties
- Input/output activities
  - point out possible I/O bottlenecks
- Hardware counter profiles
  - large number of processor statistics
KAI GuideView: Performance Analysis

- Speedup curves
  - Amdahl’s Law vs. Actual times
- Whole program time breakdown
  - Productive work vs
  - Parallel overheads
- Compare several runs
  - Scaling processors

- Breakdown by section
  - Parallel regions
  - Barrier sections
  - Serial sections
- Breakdown by thread
- Breakdown overhead
  - Types of runtime calls
  - Frequency and time
GuideView

Analyze each Parallel region

Find serial regions that are hurt by parallelism

Sort or filter regions to navigate to hotspots

www.kai.com
SGI SpeedShop and WorkShop

- Suite of performance tools from SGI
- Measurements based on
  - pc-sampling and call-stack sampling
    - based on time [prof, gprof]
    - based on R10K/R12K hw counters
  - basic block counting [pixie]
- Analysis on various domains
  - program graph, source and disassembled code
  - per-thread as well as cumulative data
SpeedShop and WorkShop

Addresses the performance Issues:

- **Load imbalance**
  - Call stack sampling based on time (*gprof*)

- **Synchronization Overhead**
  - Call stack sampling based on time (*gprof*)
  - Call stack sampling based on hardware counters

- **Memory Hierarchy Performance**
  - Call stack sampling based on hardware counters
WorkShop: Call Graph View
WorkShop: Source View

```
program sample
  integer N, i, j
  parameter (N = 2048)
  real a(N, N)

  do i = 1, N
    do j = 1, N
      a(i, j) = i + N * j
    end do
  end do
  call sums(a, N)
  !$OMP PARALLEL
  call sums(a, N)
  !$OMP END PARALLEL
end

subroutine sums(a, N)
  integer N, i, j
  real a(N, N)
  !$OMP DO
  End
```

File: /x/poulos/openmp/omp-sample.f (Read Only)
Purdue Ursa Minor/Major

- Integrated environment for compilation and performance analysis/tuning
- Provides browsers for many sources of information:
  - call graphs, source and transformed program, compilation reports, timing data, parallelism estimation, data reference patterns, performance advice, etc.
- www.ecn.purdue.edu/ParaMount/UM/
TAU
Tuning Analysis Utilities

Performance Analysis Environment for C++, Java, C, Fortran 90, HPF, and HPC++

- compilation facilitator
- call graph browser
- source code browser
- profile browsers
- speedup extrapolation
- [www.cs.uoregon.edu/research/paracomp/tau/](http://www.cs.uoregon.edu/research/paracomp/tau/)
TAU
Tuning Analysis Utilities
SC’2000 Tutorial Agenda

- OpenMP: A Quick Recap
- OpenMP Case Studies
  - including performance tuning
- Automatic Parallelism and Tools Support
- Common Bugs in OpenMP programs
  - and how to avoid them
- Mixing OpenMP and MPI
- The Future of OpenMP
SMP Programming Errors

- Shared memory parallel programming is a mixed bag:
  - It saves the programmer from having to map data onto multiple processors. In this sense, it's much easier.
  - It opens up a range of new errors coming from unanticipated shared resource conflicts.
2 major SMP errors

- **Race Conditions**
  - The outcome of a program depends on the detailed timing of the threads in the team.

- **Deadlock**
  - Threads lock up waiting on a locked resource that will never become free.
Race Conditions

- The result varies unpredictably based on detailed order of execution for each section.
- Wrong answers produced without warning!

```c
C$OMP PARALLEL SECTIONS
  A = B + C
C$OMP SECTION
  B = A + C
C$OMP SECTION
  C = B + A
C$OMP END PARALLEL SECTIONS
```
Race Conditions: A complicated solution

In this example, we choose the assignments to occur in the order A, B, C.

- ICOUNT forces this order.
- FLUSH so each thread sees updates to ICOUNT - NOTE: you need the flush on each read and each write.
Race Conditions

- The result varies unpredictably because the value of X isn’t dependable until the barrier at the end of the do loop.
- Wrong answers produced without warning!
- Solution: Be careful when you use NOWAIT.

```c
C$OMP PARALLEL SHARED (X)
C$OMP& PRIVATE(TMP)
   ID = OMP_GET_THREAD_NUM()
C$OMP DO REDUCTION(+;X)
   DO 100 I=1,100
      TMP = WORK(I)
      X = X + TMP
100  CONTINUE
C$OMP END DO NOWAIT
   Y(ID) = WORK(X, ID)
C$OMP END PARALLEL
```
Race Conditions

- The result varies unpredictably because access to shared variable TMP is not protected.
- Wrong answers produced without warning!
- The user probably wanted to make TMP private.

I lost an afternoon to this bug last year. After spinning my wheels and insisting there was a bug in KAI’s compilers, the KAI tool Assure found the problem immediately!
Deadlock

- This shows a race condition and a deadlock.
- If A is locked by one thread and B by another, you have deadlock.
- If the same thread gets both locks, you get a race condition - i.e. different behavior depending on detailed interleaving of the thread.
- Avoid nesting different locks.

```c
CALL OMP_INIT_LOCK (LCKA)
CALL OMP_INIT_LOCK (LCKB)
C$OMP PARALLEL SECTIONS
C$OMP SECTION
  CALL OMP_SET_LOCK(LCKA)
  CALL OMP_SET_LOCK(LCKB)
    CALL USE_A_and_B (RES)
  CALL OMP_UNSET_LOCK(LCKB)
  CALL OMP_UNSET_LOCK(LCKA)
C$OMP SECTION
  CALL OMP_SET_LOCK(LCKB)
  CALL OMP_SET_LOCK(LCKA)
    CALL USE_B_and_A (RES)
  CALL OMP_UNSET_LOCK(LCKA)
  CALL OMP_UNSET_LOCK(LCKB)
C$OMP END SECTIONS
```
### Deadlock

- This shows a race condition and a deadlock.
- If A is locked in the first section and the if statement branches around the unset lock, threads running the other sections deadlock waiting for the lock to be released.
- **Make sure you release your locks.**

```c
CALL OMP_INIT_LOCK (LCKA)
C$OMP PARALLEL SECTIONS
C$OMP SECTION
   CALL OMP_SET_LOCK(LCKA)
   IVAL = DOWORK()
   IF (IVAL .EQ. TOL) THEN
      CALL OMP_UNSET_LOCK (LCKA)
   ELSE
      CALL ERROR (IVAL)
   ENDIF
C$OMP SECTION
   CALL OMP_SET_LOCK(LCKA)
   CALL USE_B_and_A (RES)
   CALL OMP_UNSET_LOCK(LCKA)
C$OMP END SECTIONS
```
OpenMP death-traps

- Are you using threadsafe libraries?
- I/O inside a parallel region can interleave unpredictably.
- Make sure you understand what your constructors are doing with private objects.
- Private variables can mask globals.
- Understand when shared memory is coherent. When in doubt, use FLUSH.
- NOWAIT removes implied barriers.
Navigating through the Danger Zones

- Option 1: Analyze your code to make sure every semantically permitted interleaving of the threads yields the correct results.
  - This can be prohibitively difficult due to the explosion of possible interleavings.
  - Tools like KAI’s Assure can help.
Navigating through the Danger Zones

- Option 2: Write SMP code that is portable and equivalent to the sequential form.
  - Use a safe subset of OpenMP.
  - Follow a set of “rules” for Sequential Equivalence.
Portable Sequential Equivalence

- What is Portable Sequential Equivalence (PSE)?
  - A program is sequentially equivalent if its results are the same with one thread and many threads.
  - For a program to be portable (i.e. runs the same on different platforms/compilers) it must execute identically when the OpenMP constructs are used or ignored.
Portable Sequential Equivalence

- Advantages of PSE
  - A PSE program can run on a wide range of hardware and with different compilers - minimizes software development costs.
  - A PSE program can be tested and debugged in serial mode with off the shelf tools - even if they don’t support OpenMP.
2 Forms of Sequential Equivalence

- Two forms of Sequential equivalence based on what you mean by the phrase “equivalent to the single threaded execution”:
  - Strong SE: bitwise identical results.
  - Weak SE: equivalent mathematically but due to quirks of floating point arithmetic, not bitwise identical.
Strong Sequential Equivalence: rules

- Control data scope with the base language
  - Avoid the data scope clauses.
  - Only use private for scratch variables local to a block (e.g., temporaries or loop control variables) whose global initialization don’t matter.

- Locate all cases where a shared variable can be written by multiple threads.
  - The access to the variable must be protected.
  - If multiple threads combine results into a single value, enforce sequential order.
  - Do not use the reduction clause.
Strong Sequential Equivalence: example

Everything is shared except I and TMP. These can be private since they are not initialized and they are unused outside the loop.

The summation into RES occurs in the sequential order so the result from the program is bitwise compatible with the sequential program.

Problem: Can be inefficient if threads finish in an order that’s greatly different from the sequential order.

C$OMP PARALLEL PRIVATE(I, TMP)

C$OMP DO ORDERED
DO 100 I=1,NDIM
  TMP = ALG_KERNEL(I)
C$OMP ORDERED
  CALL COMBINE (TMP, RES)
C$OMP END ORDERED
100 CONTINUE

C$OMP END PARALLEL
Weak Sequential equivalence

- For weak sequential equivalence only mathematically valid constraints are enforced.
  - Floating point arithmetic is not associative and not commutative.
  - In most cases, no particular grouping of floating point operations is mathematically preferred so why take a performance hit by forcing the sequential order?
    - In most cases, if you need a particular grouping of floating point operations, you have a bad algorithm.
- How do you write a program that is portable and satisfies weak sequential equivalence?
  - Follow the same rules as the strong case, but relax sequential ordering constraints.
Weak equivalence: example

- The summation into RES occurs one thread at a time, but in any order so the result is not bitwise compatible with the sequential program.

- Much more efficient, but some users get upset when low order bits vary between program runs.

```c
C$OMP PARALLEL PRIVATE(I, TMP)
C$OMP DO
    DO 100 I=1,NDIM
        TMP = ALG_KERNEL(I)
    C$OMP CRITICAL
        CALL COMBINE (TMP, RES)
    C$OMP END CRITICAL
100   CONTINUE

C$OMP END PARALLEL
```
Sequential Equivalence isn’t a Silver Bullet

- This program follows the weak PSE rules, but it’s still wrong.
- In this example, RAND() may not be thread safe. Even if it is, the pseudo-random sequences might overlap thereby throwing off the basic statistics.

```c
C$OMP PARALLEL
C$OMP& PRIVATE(I, ID, TMP, RVAL)
  ID = OMP_GET_THREAD_NUM()
  N  = OMP_GET_NUM_THREADS()
  RVAL = RAND (ID)
C$OMP DO
  DO 100 I=1,NDIM
    RVAL = RAND (RVAL)
    TMP = RAND_ALG_KERNEL(RVAL)
C$OMP CRITICAL
  CALL COMBINE (TMP, RES)
C$OMP END CRITICAL
100   CONTINUE
C$OMP END PARALLEL
```
SC’2000 Tutorial Agenda

- OpenMP: A Quick Recap
- OpenMP Case Studies
  - including performance tuning
- Automatic Parallelism and Tools Support
- Common Bugs in OpenMP programs
  - and how to avoid them
- Mixing OpenMP and MPI
- The Future of OpenMP
What is MPI?
The message Passing Interface

- MPI created by an international forum in the early 90’s.
- It is huge -- the union of many good ideas about message passing API’s.
  - over 500 pages in the spec
  - over 125 routines in MPI 1.1 alone.
  - Possible to write programs using only a couple of dozen of the routines
- MPI 1.1 - MPIch reference implementation.
- MPI 2.0 - Exists as a spec, full implementations?
How do people use MPI?
The SPMD Model

A sequential program working on a data set

- Replicate the program.
- Add glue code
- Break up the data

- A parallel program working on a decomposed data set.
- Coordination by passing messages.
#include <mpi.h>
void main (int argc, char *argv[])
{
    int i, my_id, numprocs; double x, pi, step, sum = 0.0;
    step = 1.0/(double) num_steps;
    MPI_Init(&argc, &argv);
    MPI_Comm_Rank(MPI_COMM_WORLD, &my_id);
    MPI_Comm_Size(MPI_COMM_WORLD, &numprocs);
    my_steps = num_steps/numprocs;
    for (i=myrank*my_steps; i<(myrank+1)*my_steps; i++)
    {
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
    sum *= step;
    MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
               MPI_COMM_WORLD);
}
How do people mix MPI and OpenMP?

A sequential program working on a data set

- Replicate the program.
- Add glue code
- Break up the data

- Create the MPI program with its data decomposition.
- Use OpenMP inside each MPI process.
Pi program in MPI

```c
#include <mpi.h>
#include "omp.h"

void main (int argc, char *argv[]) {
    int i, my_id, numprocs;  double x, pi, step, sum = 0.0 ;
    step = 1.0/(double) num_steps ;
    MPI_Init(&argc, &argv) ;
    MPI_Comm_Rank(MPI_COMM_WORLD, &my_id) ;
    MPI_Comm_Size(MPI_COMM_WORLD, &numprocs) ;
    my_steps = num_steps/numprocs ;

    #pragma omp parallel do
    for (i=myrank*my_steps; i<(myrank+1)*my_steps ; i++)
    {
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
    sum *= step ;
    MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                MPI_COMM_WORLD) ;
}
```

Get the MPI part done first, then add OpenMP pragma where it makes sense to do so.
Mixing OpenMP and MPI
Let the programmer beware!

- Messages are sent to a process on a system not to a particular thread
  - Safest approach -- only do MPI inside serial regions.
  - ... or, do them inside MASTER constructs.
  - ... or, do them inside SINGLE or CRITICAL
    - But this only works if your MPI is really thread safe!

- Environment variables are not propagated by mpirun. You’ll need to broadcast OpenMP parameters and set them with the library routines.
SC’2000 Tutorial Agenda

- OpenMP: A Quick Recap
- OpenMP Case Studies
  - including performance tuning
- Automatic Parallelism and Tools Support
- Common Bugs in OpenMP programs
  - and how to avoid them
- Mixing OpenMP and MPI
- The Future of OpenMP
The future of OpenMP is in the hands of the OpenMP Architecture Review Board (the ARB)

- Intel, KAI, IBM, HP, Compaq, Sun, SGI, DOE ASCI

The ARB resolves interpretation issues and manages the evolution of new OpenMP API’s.

Membership in the ARB is Open to any organization with a stake in OpenMP.

- Research organization (e.g. DOE ASCI)
- Hardware vendors (e.g. Intel or HP)
- Software vendors (e.g. KAI)
The Future of OpenMP

- OpenMP is an evolving standard. We will see to it that it is well matched to the changing needs of the shared memory programming community.

- Here’s what’s coming in the future:
  - **OpenMP 2.0 for Fortran:**
    - This is a major update of OpenMP for Fortran95.
    - Status. Specification released at SC’00
  - **OpenMP 2.0 for C/C++**
    - Work to begin in January 2001
    - Specification complete by SC’01.

To learn more about OpenMP 2.0, come to the OpenMP BOF on Tuesday evening
Reference Material on OpenMP

OpenMP Homepage www.openmp.org:
The primary source of information about OpenMP and its development.

Books:

Research papers:


Scherer A, Honghui Lu, Gross T, Zwaenepoel W. Transparent adaptive parallelism on NOWS using OpenMP. ACM. Sigplan Notices (Acm Special Interest Group on Programming Languages), vol.34, no.8, Aug. 1999, pp.96-106. USA.


Still CH, Langer SH, Alley WE, Zimmerman GB. Shared memory programming with OpenMP. Computers in Physics, vol.12, no.6, Nov.-Dec. 1998, pp.577-84. Publisher: AIP, USA.

