A Source-to-Source OpenMP Compiler

Mario Soukup and Tarek Abdelrahman

The Edward S. Rogers Sr.
Department of Electrical and Computer Engineering
University of Toronto

{soukup|tsa}@eecg.toronto.edu
A portable and efficient implementation of the Fortran 77 OpenMP 1.0 standard for compiler-assisted parallel programming on shared memory multiprocessors.
Outline

- Motivation.
- System overview.
- Implementation details.
- Experimental Evaluation.
- Conclusions and future work.
The OpenMP Standard

- Compiler-assisted shared-memory parallel programming.
  - User inserts directives in sequential code; compiler generates explicitly parallel code.

```plaintext
program test
  integer*8 i,j
j=0
do i=1,1000
  A(i)=F(j)
end do
print *, 'Done'
end
```

- Fortran & C/C++ versions.

- Wide acceptance from both vendors and users.
The OpenMP Standard

- Directives for functionality not performance.

- Past work on improving performance of compiler generated programs on large-scale parallel systems:
  - Enhance cache locality [Manjikian and Abdelrahman, 1997].
  - Enhance memory locality [Tandri and Abdelrahman, 1997, 2000].

- We believe that the OpenMP standard can be extended with performance-enhancing directives.

- We need an implementation of OpenMP that we can extend.
OpenMP Implementations

- Commercial implementations: MIPSpro (SGI), F95 (Sun), Guide F90 from KAI.
  - Source code of the implementation not available.
  - Specific to a particular platform.

- Open-source implementations: OdinMP/CCp (Lund University), Omni (Tsukuba Research Center), NanosCompiler (Barcelona).
  - Support the C/C++ version of the standard.
  - Specific to a platform.

- There is a need for a portable and efficient implementation for the Fortran OpenMP standard.
System Overview

- Source-to-source translation.
- Standard components on Unix/Linux systems.
Implementation – PARALLEL

subroutine test (n)
  integer i,j,k,n
  i = 1
  j = 2
  k = 3
  print *, 'Started'
  c$OMP PARALLEL PRIVATE(i)
  c$OMP+SHARED(j,n)
  c$OMP+FIRSTPRIVATE(k)
  c$OMP END PARALLEL
  print *, 'Finished'
end

subroutine test (n)
  EXTERNAL create_threads
  EXTERNAL omp_sub1
  INTEGER*4 i, j, k,n
  COMMON /omp_cb2/ k
  COMMON /omp_cb1/ j
  i = 1
  j = 2
  k = 3
  PRINT *, 'Started'
  PRINT *, i,j,k,n
  PRINT *, 'Finished'
END

subroutine omp_sub1(omp_thr_num,n)
  INTEGER*4 i, j, k, omp_init1,n
  INTEGER*4 omp_thr_num
  COMMON /omp_cb2/ omp_init1
  COMMON /omp_cb1/ j
  END
Implementation – Loop Scheduling

```fortran
num_of_threads = OMP_GET_NUM_THREADS()
thread_num = OMP_GET_THREAD_NUM()
size = (end-start+1)/num_of_threads
new_start = start + thread_num*size
new_end = new_end+(size-1)

DO i = new_start, new_end
    code
ENDDO

F_OMP_BARRIER()
```

```fortran
DO i = start, end
    code
ENDDO
```
The Run-time System

- **Thread creation/termination routines.**
  - Mapped to POSIX thread creation/termination routines.
  - Packaging of subroutine parameters.

- **Synchronization routines.**
  - Locks and barriers.
  - Mapped to POSIX thread synchronization routines.

- **Scheduling routines.**
  - Determine the next set of iterations to be executed by a thread according to the scheduling scheme.
  - Run-time state.

- **OpenMP environment routines.**
  - Access to environment variables through wrapper to standard Unix/Linux calls.
Experimental Evaluation

- **Platform:** 4-processor Sun Ultra Enterprise 450 (400 MHz) with GNU g77 version 2.7.2.

- **Overhead measurements.**
  - Edinburgh Parallel Computing Centre (EPCC) micro-benchmarks designed to measure overhead of various OpenMP constructs.
    - Barrier overhead.
    - Mutual exclusion overhead.
    - Loop scheduling overhead.
  - Comparison to published results.

- **Speedup measurements.**
  - With respect to the sequential (no OpenMP directives) code.
  - Jacobi benchmark (www.openmp.org).
Overhead increases with the number of processors.

- Overhead increases with the number of processors.
- Dependent on the underlying POSIX implementation.
Overhead – Mutual Exclusion

- Overhead relatively independent of the number of processors.
Overhead – Loop Scheduling

- STATIC scheduling overhead: 6800 cycles (17 µseconds).
- Overhead increases with run-time decisions.
Overhead - Comparison

- 8-processor Sun E3500 (400 MHz).
- KAI Guide F90 version 3.7.
- Our implementation has:
  - Higher (~2X) barrier synchronization overhead.
  - Lower (~1/2 X) mutual exclusion overhead.
  - Similar loop scheduling overhead.
- Overall comparable.
Speedup – Jacobi Application

- Speedup at 4 processors: 3.89.
Conclusions and Future Work

- Described the implementation of the Fortran 77 OpenMP standard version 1.0.
  - Portable: source-to-source transformations and POSIX.
  - Efficient: overhead comparable to one commercial implementation, overall speedup is linear.

- Future work.
  - Larger number of processors.
  - Larger applications.
  - Research on performance directives.
  - Release into public domain.
Overhead increases with run-time decisions.
Barrier Synchronization Overhead

Synchronisation overheads on Sun E3500

Overhead (clock cycles) vs. No of. processors
Mutual Exclusion Overhead

Synchronisation overheads on Sun E3500

Overhead (clock cycles)

Critcal
Lock/Unlock
Atomic

No of. processors

1 2 4 8
Loop Scheduling Overhead

Scheduling overhead on Sun E3500 - 8 processors

Overhead (clock cycles) vs Chunksize

- STATIC
- DYNAMIC
- GUIDED
- STATIC