OpenMP – an overview

Seminar Non-uniform Memory Access (NUMA), WS2014/15

Matthias Springer

Hasso Plattner Institute, Operating Systems and Middleware Group

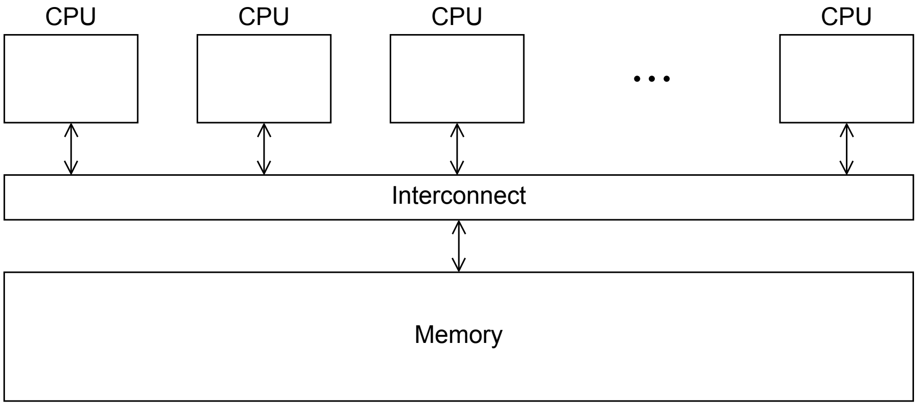
January 14, 2015

Overview

What is OpenMP? [5]

- OpenMP = Open Multi-Processing
- API for multi-platform shared memory multiprocessing
- Set of **compiler directives, library routines, and environment variables**
- Programing languages: C, C++, Fortran
- Operating Systems: e.g. Solaris, AIX, Linux, Mac OS X, Windows
- OpenMP Architecture Review Board: group of hardware and software vendors

Shared Memory Multiprocessing [4]



Overview

OpenMP vs. pthreads vs. MPI

- pthreads: low-level programming
  - Programmer specifies behavior of each thread
  - Links against `libpthread`: no compiler support required
- OpenMP: higher-level programming
  - Programmer specifies that a piece of code should be executed in parallel
  - Required compiler support (e.g. preprocessor)
- MPI: Message Passing Interface
  - Communication based on message sending and receiving
  - No shared memory, designed for distributed systems

Overview

Running Function on Multiple Threads [4]

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void hello_world(void)
{
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    printf("Thread %d of %d says Hello!\n", my_rank, thread_count);
}

int main(int argc, char* argv[])
{
    int thread_count = strtol(argv[1], NULL, 10);

    #pragma omp parallel num_threads(thread_count)
    hello_world();

    return 0;
}
```

Compiling OpenMP Programs

- Compile: `gcc -fopenmp -o hello hello.c`
- Run: `./hello 3`

```
Thread 1 of 3 says Hello!
Thread 0 of 3 says Hello!
Thread 2 of 3 says Hello!
```

OpenMP Compilation Process

- Annotated Source Code $\rightarrow$ OpenMP Compiler $\rightarrow$ Parallel Object Code
- Compiler can also generate sequential object code
- Compiler Front End: parse OpenMP directives, correctness checks
- Compiler Back End: replace constructs by calls to runtime library, change structure of program (e.g., put parallel section in a function to fork it)
- See `https://iwomp.zih.tu-dresden.de/downloads/OpenMP-compilation.pdf` for more details

Notation (Syntax)

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void hello_world(void)
{
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    printf("Thread %d of %d says Hello!\n", my_rank, thread_count);
}

int main(int argc, char* argv[])
{
    int thread_count = strtol(argv[1], NULL, 10);

    #pragma omp parallel num_threads(thread_count)
    hello_world();

    return 0;
}
```

Notation (Syntax) [1]

- **Directive:** `pragma` statement
  e.g. `#pragma omp parallel` *[clause [[,] clause] . . .]*
  structured-block
- **Runtime Library Routine:** function defined in `omp.h`
  e.g. `omp_get_thread_num()`
- **Structured Block:** Single statement or compound statement with a single entry at the top and a single exit at the bottom
- **Clause:** modifies a directive's behavior
  e.g. `num_threads(` *integer-expression* `)`
- **Environment Variable:** defined outside the program
  e.g. `OMP_NUM_THREADS`

Notation (OpenMP)

- **Master Thread:** original thread
- **Slave Threads:** all additional threads
- **Team:** master thread + slave threads

Scope of Variables

- **shared** scope: variable can be accessed by all threads in team
  variables declared outside a structured block following a `parallel` *directive*
- **private** scope: variable can be accessed by a single thread
  variable declared inside a structured block following a `parallel` *directive*

```
int foo = 42;
int bar = 40;

#pragma omp parallel private(foo) shared(bar) default(none)
{
    int x;

    /* foo and x are private */
    /* bar is shared */
}
```

Handout only: Scope of Variables

- Private variables are uninitialized.
- Initialize variables with values from master thread: `firstprivate`.
- `default(none)` requires programmer to specify visibility for all variables implicitly (good practice).

parallel for Directive

```
#include <stdlib.h>
#include <omp.h>

int main(int argc, char* argv[])
{
    #pragma omp parallel
    {
        #pragma omp for
        for (int i = 0; i < 3; ++i)
        {
            printf("Thread %d of %d says Hello!\n",
                omp_get_thread_num(), omp_get_num_threads());
        }
    }
}
```

Handout only: `parallel for` Directive

- Runs loop iterations in parallel.
- Shortcut: `#pragma omp parallel for`
- Loop iterations must be data-independent.
- Loop must be in canonical form.
  - E.g.: test condition is $<$, $<=$, $>$, etc.; operation is increment.
  - OpenMP must be able to determine the number of iterations before the loop is executed.

`parallel for` Directive

Mapping of iterations to threads controlled by `schedule` clause.

- `schedule(static [, chunksize])`: block of `chunksize` iterations statically assigned to thread
- `schedule(dynamic [, chunksize])`: thread reserves `chunksize` iterations from queue
- `schedule(guided [, chunksize])`: same as dynamnic, but chunk size starts big and gets smaller and smaller, until it reaches `chunksize`.
- `schedule(runtime)`: scheduling behavior determined by environment variable

Example: Sum of List of Integers

```c
#include <stdlib.h>
#include <omp.h>

int main(int argc, char* argv[])
{
    int sum = 0;
    int A[100];
    int i;

    for (i = 0; i < 100; ++i) A[i] = i;

    #pragma omp parallel for
    for (i = 0; i < 100; ++i)
    {
        sum += A[i];
    }

    printf("Sum: %d\n", sum);
}
```

reduce Clause

```
#include <stdlib.h>
#include <omp.h>

int main(int argc, char* argv[])
{
    int sum = 0;
    int A[100];
    int i;

    for (i = 0; i < 100; ++i) A[i] = i;

    #pragma omp parallel for reduction (+:sum)
    for (i = 0; i < 100; ++i)
    {
        sum += A[i];
    }

    printf("Sum: %d\n", sum);
}
```

`reduce` Clause

- Compiler creates local private copy per variable
- $+$, initial value 0
- $-$, initial value 0
- $*$, initial value 1
- Also support for &, |, ˆ, &&, || in C/C++

Critical Sections

```
#include <stdlib.h>
#include <omp.h>

int main(int argc, char* argv[])
{
    int sum = 0;
    int A[100];
    int i;

    for (i = 0; i < 100; ++i) A[i] = i;

    #pragma omp parallel for
    for (i = 0; i < 100; ++i)
    {
        #pragma omp critical
        sum += A[i];
    }

    printf("Sum: %d\n", sum);
}
```

Atomic Statements

```c
#include <stdlib.h>
#include <omp.h>

int main(int argc, char* argv[])
{
    int sum = 0;
    int A[100];
    int i;

    for (i = 0; i < 100; ++i) A[i] = i;

    #pragma omp parallel for
    for (i = 0; i < 100; ++i)
    {
        #pragma omp atomic
        sum += A[i];
    }

    printf("Sum: %d\n", sum);
}
```

Handout only: Atomic Statements

- Behavior is implementation-specific, but might use special CPU instructions (e.g. atomic fetch add).
- Supports `x binop= y`, `x++`, `++x`, `x--`, `--x`.

More Synchronization Constructs

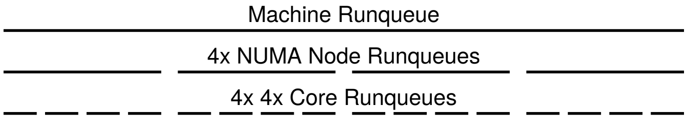
- `#pragma omp barrier`: wait until all threads arrive
- `#pragma omp for nowait`: remove implicit barrier after for loop (also exists for other directives)
- `#pragma omp master`: only executed by master thread
- `#pragma omp single`: only executed by one thread
- Sections: define a number of blocks, every thread executes one block
- Locks: `omp_init_lock()`, `omp_set_lock()`, `omp_unset_lock()`, ...

Overview

Implementation: ForestGOMP [2]

- Objectives and Motivation
  - Keep buffers and threads operating on them on the same NUMA node (reducing contention)
  - Processor level: group threads sharing data intensively (improve cache usage)
- Triggers for scheudling
  - Allocation/deallocation of resources
  - Processor becomes idle
  - Change of hardware counters (e.g., cache miss, remote access rate)

BubbleSched: Hierarchical Bubble-Based Thread Scheduler



- Runqueue for different hierarchical levels
- Bubble: group of threads sharing data or heavy synchronization
- Responsible for scheduling threads

Mami: NUMA-aware Memory Manager

- API for memory allocation
- Can migrate memory to a different NUMA node
- Supports Next Touch policy: migrate data to NUMA node of accessing thread

Handout only: Memory Relocation with Next Touch [3]

- Buffers are marked as *migrate-on-next-touch* when a thread migration is expected
- Buffer is relocated if thread touches buffer that is not located on local node
- Implemented in kernel mode

ForestGOMP: Mami-aware OpenMP Runtime

- Mami attaches memory hints: e.g., which regions are access frequently by a certain thread
- Initial distribution: put thread and corresponding memory on same NUMA node (local accesses)
- Handle idleness: steal threads from local core, then from different NUMA node (also migrates memory; prefers threads with less memory)
- Two levels of *distribution*: memory-aware, then cache-aware

References

OpenMP Architecture Review Board.
Openmp 3.1 api c/c++ syntax quick reference card, 2011.

François Broquedis, Nathalie Furmento, Brice Goglin, Pierre-André Wacrenier, and Raymond Namyst.
Forestgomp: An efficient openmp environment for numa architectures.
International Journal of Parallel Programming, 38(5-6):418–439, 2010.

Brice Goglin, Nathalie Furmento, et al.
Memory migration on next-touch.
In *Linux Symposium*, 2009.

P.S. Pacheco.
An Introduction to Parallel Programming.
An Introduction to Parallel Programming. Morgan Kaufmann, 2011.

Wikipedia.
Openmp — wikipedia, the free encyclopedia, 2014.
[Online; accessed 14-December-2014].

Matrix Multiply with OpenMP and MPI

Seminar Non-uniform Memory Access (NUMA), WS2014/15

Carolin Fiedler, Matthias Springer

Hasso Plattner Institute, Operating Systems and Middleware Group

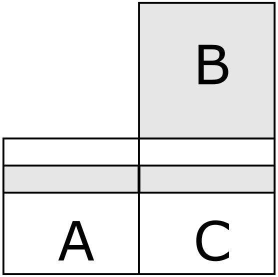
January 14, 2015

Overview

Idea

- Distribute work on nodes with MPI (no memory sharing)
  1 worker per node
- Parallelize work on a single node with OpenMP (shared memory)
  1 thread per core

Message Passing



- Replicate *B* on all MPI workers
- For *n* MPI workers, divide *A* in *n* stripes, every worker gets one stripe
- Result matrix *C* contains one stripe per worker
- Message passing (remote memory access) during send (distribute) and collect phases
- Local memory access only during multiplication and add

Live Demo and Source Code

Source Code (OpenMP)

```
#pragma omp parallel default (none) shared(A,B,C,offset,rows)
    private(i,j,k)
{
#pragma omp for
    for (j = 0; j < M; j++)
        for (i = offset; i < offset + rows; i++)
            for (k = 0; k < P; k++)
                C[i][j] += A[i][k] * B[k][j];
}
```

Performance Measurements

- Matrix size: 2048 x 2048
- (MPI) 1 x (OpenMP) 1: 110.3 s
- 1 x 2: 55.6 s
- 1 x 12: 12.7 s
- 1 x 24: 10 s
- 2 x 12: 9.8 s
- 12 x 2: 10.7 s
- 24 x 1: 11.5 s

Best Configuration

- 2 x 12: 9.8 s
- System has 2 NUMA nodes (sockets)
- Every socket has 12 cores (6 real ones + 6 HyperThreading)
- Only local memory accesses inside an OpenMP thread

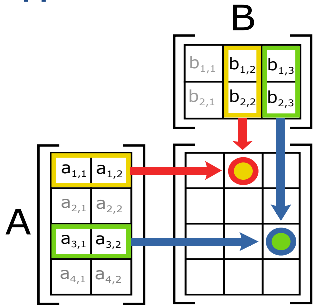
```
export OMP_NUM_THREADS=12
mpirun -np 2 --bysocket --bind-to-socket --report-bindings ./a.out
```

Hardware-specific Performance Optimizations

- `ubuntu-numa0101` machine details
  - 2x Intel Xeon E5-2620 (Sandy Bridge) CPU
    - 6 cores, 2.0 GHz each
    - 6x 32 KB L1 cache (32 KB instruction, 32 KB data)
    - 6x 256 KB L2 cache
    - 1x 15 MB shared L3 cache
  - 64 GB RAM
- Optimizations
  - Transposition: read matrices row-wise
  - Blocking: access matrices in chunks that fit into the cache
  - SSE (Streaming SIMD Extensions): add, multiply two 128-bit vectors; some CPUs have fused multiply-add units
  - Alignment: aligned (16 byte) loads are faster than unaligned loads
  - Loop Unrolling: less branches in the assembly code, instruction-level parallelism
  - Parameter Tuning: brute-force different blocking sizes per matrix size

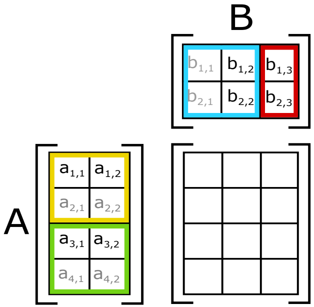
Matrix Transposition

Image taken from Wikipedia [6]



- Matrices stored row-wise in main memory
- Matrix A: reading row-wise, Matrix B: reading column-wise
- Prefetching: 64 byte cache line, will read 8 doubles from B but only use one of them
- Transpose matrix B for more cache hits

L1/L2 Blocking



- Divide matrices in block and iterate over all combinations of blocks
- L1 Blocking: cache big enough for $\frac{32768}{8} = 4096$ doubles, block size $\sqrt{\frac{4096}{2}} = 45.2$, use 40×40 blocks to ensure that entire cache line is used
- L2 Blocking: 128×128 blocks