Intermediate OpenMP for Perl Programmers

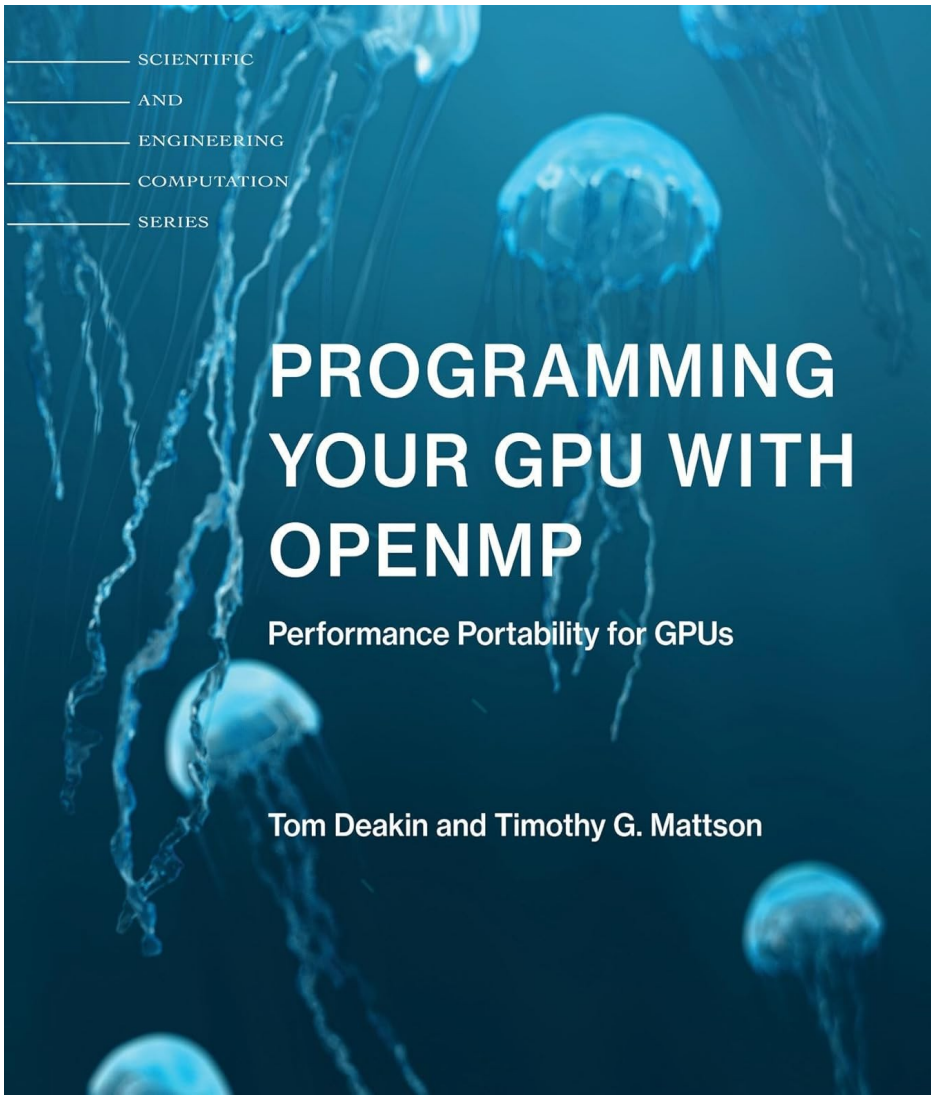
Brett Estrade
<OODLER@cpan.org>

GCC – GOMP

GNU Offloading and Multi Processing Runtime Library
https://gcc.gnu.org/wiki/openmp

- GCC 4.2 supports OpenMP spec 2.5
- GCC 4.4 supports OpenMP spec 3.0
- GCC 4.7 supports OpenMP spec 3.1
- OpenMP spec 4.1 support:
  - C/C++ support in GCC 4.9
  - Fortran support in GCC 4.9.1
- GCC 5 supports *offloading* (e.g., to GPU)
- GCC 6 supports OpenMP spec 4.5 for C/C++
- GCC 7 mostly supports OpenMP spec 4.5 for Fortran (*fully* in GCC 11)
- GCC 9 partially supports OpenMP spec 5.0 for C/C++
- GCC 10 adds OpenMP spec 5.0 support and initial Fortran support

Recommended Books



Talk Assumptions

- Discussing only C examples (so C++ or Fortran)
- Information based on OpenMP spec 4.5
- Any GCC >= 5

Note: regarding wider compiler support:

- OpenMP is supported by *many* commercial compilers
- `Inline::C` uses the compiler used to create **`perl`**
- We assume 99+% **`perl`**'s are compiled with GCC

Current State of Perl Options

OpenMP is best accessed from Perl via `Inline::C`. We assume GCC (see previous slide). Examples in this talk will use the following Perl modules:

- `Alien::OpenMP`
  - Works "`with`" Inline::C
- `OpenMP::Simple`
  - Alien::OpenMP + helper C functions & macros
- `OpenMP::Environment`
  - Perlish way to interact with `%ENV` OpenMP cares about
- No compiler support beyond GCC is planned (*is there a demand*?)

.

Example Perl Program

```
use strict;
use warnings;
use OpenMP::Simple;
use OpenMP::Environment;
use Test::More tests => 8;
use Inline (
    C    => 'DATA',
    with => qw/OpenMP::Simple/,
);
my $env = OpenMP::Environment->new;
note qq{Testing macro provided by OpenMP::Simple, 'PerlOMP_UPDATE_WITH_ENV__NUM_THREADS'};
for my $num_threads ( 1 .. 8 ) {
    my $current_value = $env->omp_num_threads($num_threads);
    is _get_num_threads(), $num_threads,
       sprintf qq{The number of threads (%0d) spawned in the OpenMP runtime via OMP_NUM_THREADS, as expected},
       $num_threads;
}
__DATA__
__C__
int _get_num_threads() {
  PerlOMP_UPDATE_WITH_ENV__NUM_THREADS
  int ret = 0;
  #pragma omp parallel
  {
    #pragma omp single
    ret = omp_get_num_threads();
  }
  return ret;
}
__END__
```

General Use Case Addressed

- An HPC developer familiar with OpenMP and Perl wants to use Perl as “host” language as the driver for OpenMP based `Inline::C` calls
- The `perl` process is not threaded, subroutines are
- Perl variables are fairly regular (e.g., 1D or 2D arrays, list of strings, etc)
- Nearly all data accesses to Perl variable is read-only

Example, OpenMP + C + GCC

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

int main() {
  int ret = 0;
  #pragma omp parallel
  {
    #pragma omp single
    {
      ret = omp_get_num_threads();
    }
  }
  printf("running with %d threads\n", ret);
  return ret;
}
```

```
 ./first-example.x
running with 48 threads
```

Example, OpenMP + Perl

```
use strict;
use warnings;

use OpenMP::Simple;
use OpenMP::Environment;
Use Test::More tests => 8;

use Inline (
    C    => 'DATA',
    with => qw/OpenMP::Simple/,
);

my $env = OpenMP::Environment->new;

for my $num_threads ( 1 .. 8 ) {
    my $current_value = $env->omp_num_threads($num_threads);
    is _get_num_threads(),$num_threads,
       sprintf qq{The number of threads (%0d) spawned in the OpenMP
            runtime via OMP_NUM_THREADS, as expected}, $num_threads;
}

__DATA__
__C__
int _get_num_threads() {
  PerlOMP_UPDATE_WITH_ENV__NUM_THREADS
  int ret = 0;
  #pragma omp parallel
  {
    #pragma omp single // first thread to reach here runs it, no others
    ret = omp_get_num_threads();
  }
  return ret;
}
```

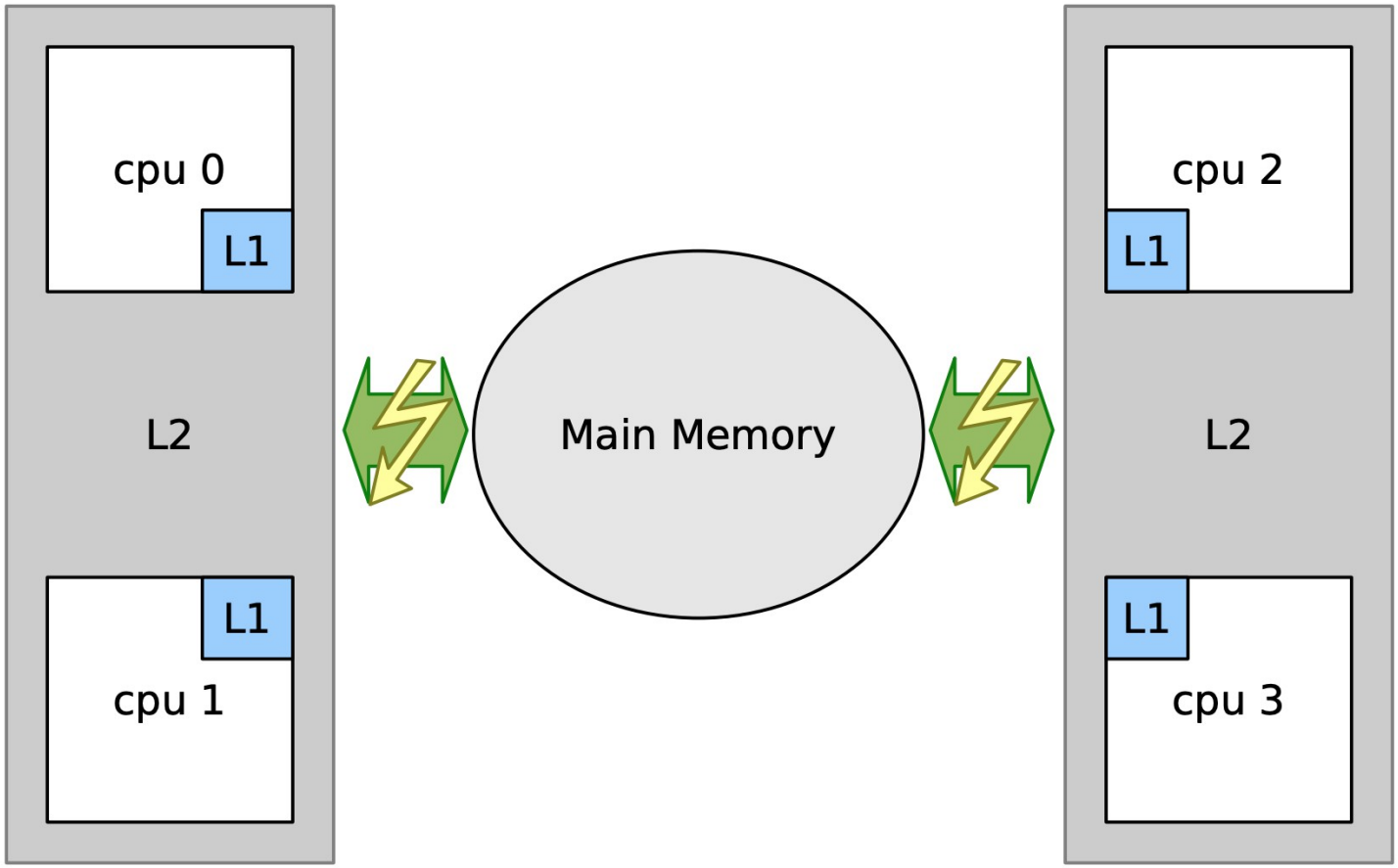
```
1..8
ok 1 - The number of threads (1) spawned in the OpenMP
#         runtime via OMP_NUM_THREADS, as expected
ok 2 - The number of threads (2) spawned in the OpenMP
#         runtime via OMP_NUM_THREADS, as expected
ok 3 - The number of threads (3) spawned in the OpenMP
#         runtime via OMP_NUM_THREADS, as expected
ok 4 - The number of threads (4) spawned in the OpenMP
#         runtime via OMP_NUM_THREADS, as expected
ok 5 - The number of threads (5) spawned in the OpenMP
#         runtime via OMP_NUM_THREADS, as expected
ok 6 - The number of threads (6) spawned in the OpenMP
#         runtime via OMP_NUM_THREADS, as expected
ok 7 - The number of threads (7) spawned in the OpenMP
#         runtime via OMP_NUM_THREADS, as expected
ok 8 - The number of threads (8) spawned in the OpenMP
#         runtime via OMP_NUM_THREADS, as expected
```

Recap

- Following slides recap the last few talks on OpenMP
- Please find and watch my other TPRC talks on OpenMP for Perl programmers
- And here we go...

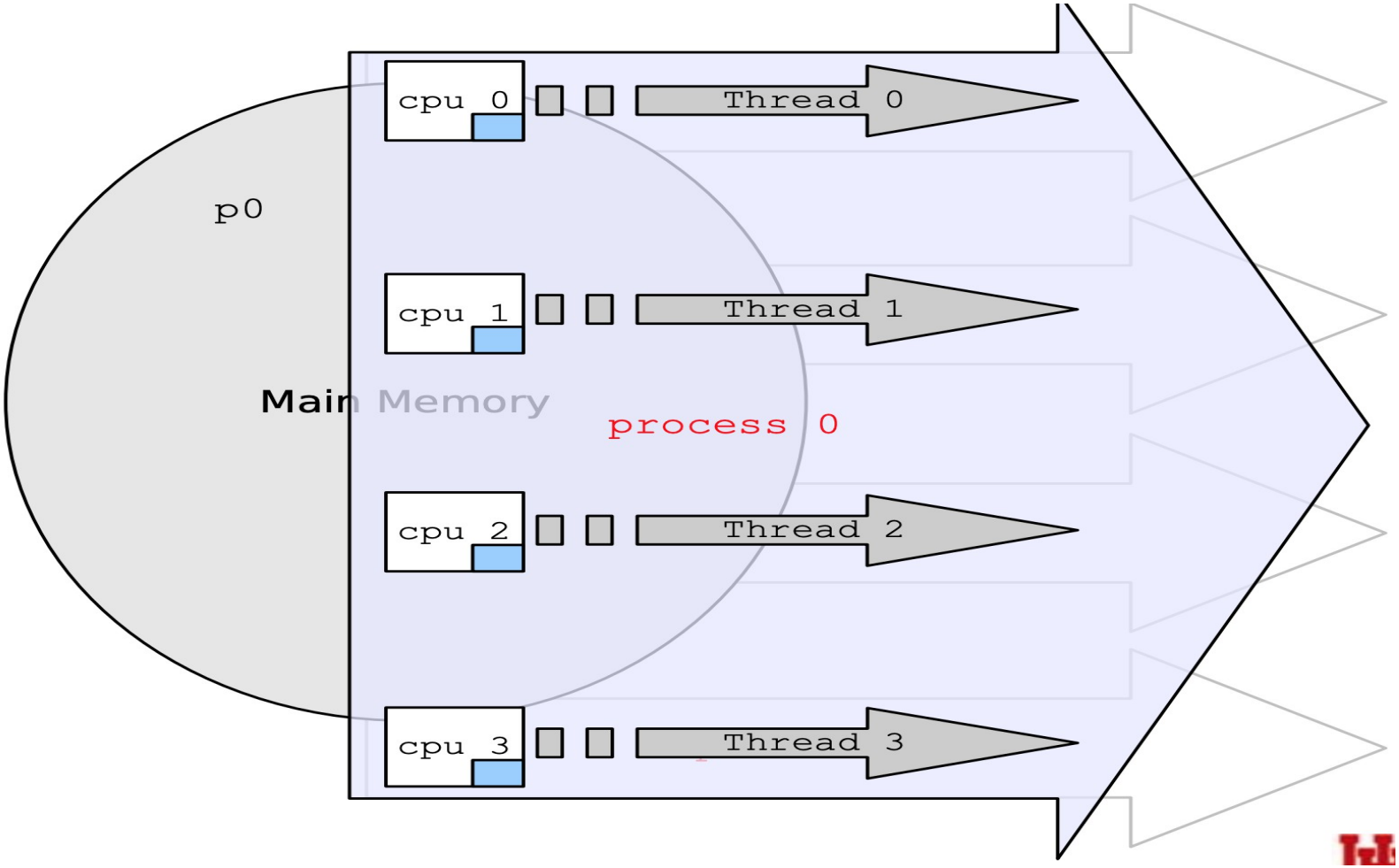
Recap!

OpenMP Memory Models



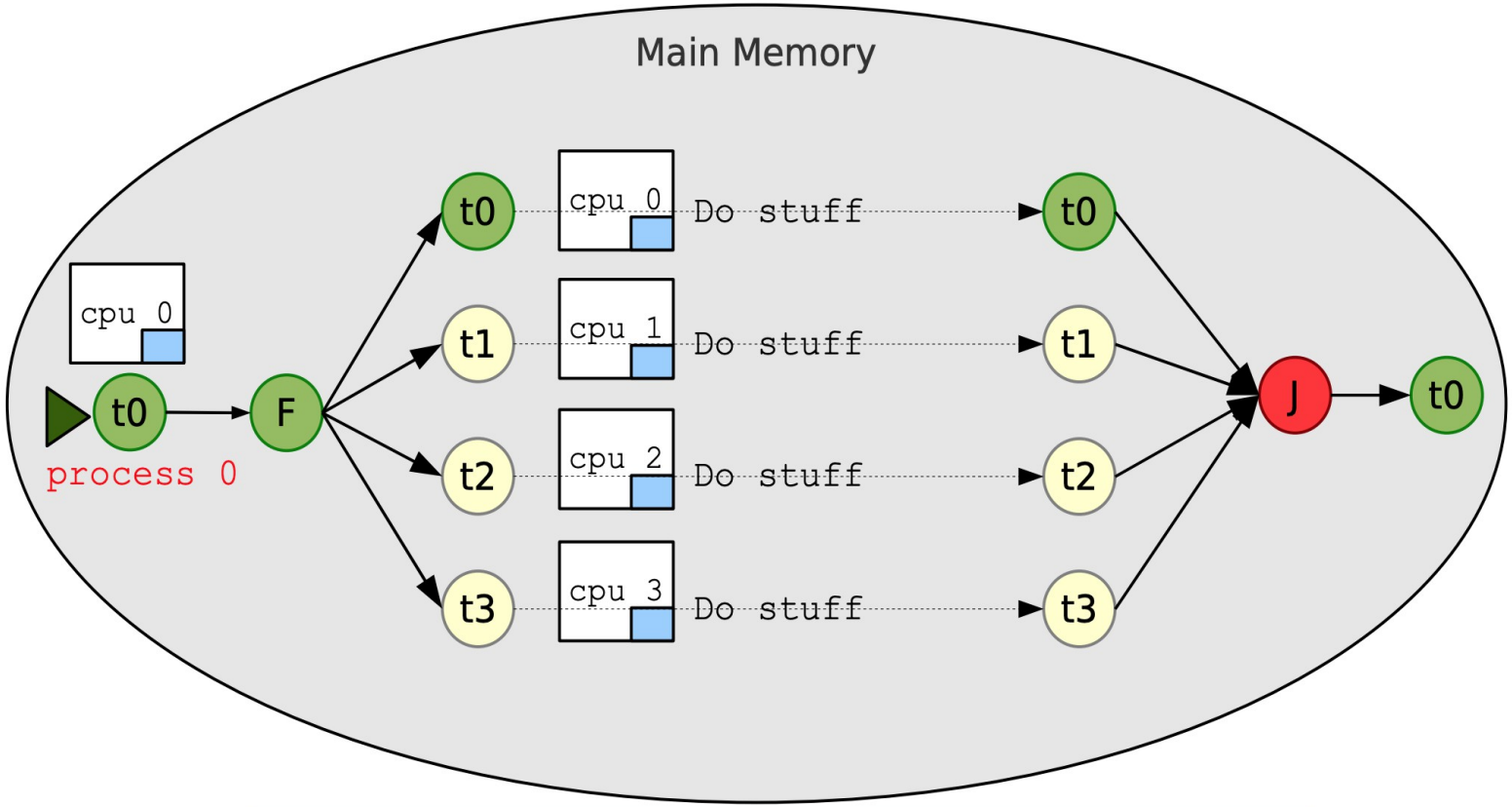
Recap!

OpenMP Memory Models



Recap!

OpenMP Execution Model



Here, thread 0 is on CPU 0; at the fork, 3 new threads are created and are distributed to the remaining 3 CPUs.

Recap!

Variables

- Variables can shared among threads
- Or can be private to a thread
- Variable initializations can be specified

Recap!

Worksharing

Parallelizing loops across threads

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

int main() {
    int matrix[10][15];
    int i, j;

    // Initialize matrix
    for (i = 0; i < 10; ++i) {
        for (j = 0; j < 15; ++j) {
            matrix[i][j] = i * M + j;
        }
    }

    // Parallel region with shared clause
    #pragma omp parallel shared(matrix) private(i, j)
    {
        #pragma omp for
        for (i = 0; i < 10; ++i) {
            printf("Thread %d handles row %d\n", omp_get_thread_num(), i);
            for (j = 0; j <15; ++j) {
                matrix[i][j] *= 2; // Modify matrix element
            }
        }
    }

    return 0;
}
```

Recap!

Atomicity

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

int main() {
    int sum = 0;
    int i;
    // Parallel loop where each thread updates sum atomically
    #pragma omp parallel for shared(sum) private(i)
    for (i = 0; i < 1000; ++i) {
        #pragma omp atomic
        sum += i;
    }
    printf("Sum: %d\n", sum);
    return 0;
}
```

- Memory can be set aside and updated safely, using, **`#pragma omp atomic`**
- **`#pragma omp critical`** is also an option

New!

Asynchronous Tasking

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

int main() {
    #pragma omp parallel
    {
      #pragma omp single     // Only 1 thread is running
      {
         #pragma omp task    // Task #1
         {
           printf("Perl ");
         }
         #pragma omp task    // Task #2
         {
           printf("rocks! ");
         }
       } // <~ other threads begin executing
            tasks at the end of the 'single'
            block
    }
    return 0;
}
```

```
$ gcc -fopenmp ./basic-task.c
$ OMP_NUM_THREADS=4 ./a.out
Perl rocks!
$ OMP_NUM_THREADS=4 ./a.out
rocks! Perl
$
```

- Tasking was introduced in 2008 in OpenMP 3.0 (GCC 4.4)
- Generated tasks are atomically assigned to an available (real) CPU thread
- 2 different execution orderings are possible (see output)

Asynchronous Tasking

- Any thread can generate a task
- **`#pragma omp single`** when starting out so that you can think about your code more easily
- All threads in the parallel region are available to pick up a task
- **`#pragma omp taskwait`** is a synchronization barrier – when no more new tasks are available, all threads wait until all theads are idle to proceed

Asynchronous Tasking

private (list)
firstprivate (list)
default (shared | none)
if ([task:] *scalar-logical-expression*)
final (*scalar-logical-expression*)
mergeable
depend (*dependency-type: list*)
priority (*priority-value*)
untied

Affected by:

OMP_MAX_TASK_PRIORITY (*supported by OpenMP::Environment*)

Asynchronous Tasking

Synchronizing Tasks

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

int main() {
    #pragma omp parallel
    {
      #pragma omp single     // Only 1 thread is running
      {
         #pragma omp task    // Task #1
         {
           printf("Perl ");
         }
         #pragma omp taskwait
         #pragma omp task    // Task #2
         {
           printf("rocks! ");
         }
       }
    }
    return 0;
}:w
```

```
$ gcc -fopenmp ./basic-task.c
$ OMP_NUM_THREADS=4 ./a.out
Perl rocks!
$ OMP_NUM_THREADS=4 ./a.out
Perl rocks!
$ OMP_NUM_THREADS=4 ./a.out
Perl rocks!
```

- Tasking was introduced in 2008 in OpenMP 3.0 (GCC 4.4)
- Generated tasks are atomically assigned to an available (real) CPU thread
- 2 different execution orderings are possible (see output)

OpenMP Tasking + Perl

```
use strict;
use warnings;

use OpenMP::Simple;
use OpenMP::Environment;

use Inline (
    C    => 'DATA',
    with => qw/OpenMP::Simple/,
);

my $env = OpenMP::Environment->new;

for my $num_threads ( 1 .. 16 ) {
    my $current_value = $env->omp_num_threads($num_threads);
    my $got_threads = run_tasks();
}

__DATA__
__C__
int run_tasks() {
    int got_num_threads = 0;
    PerlOMP_UPDATE_WITH_ENV__NUM_THREADS
    #pragma omp parallel
    {
        got_num_threads = omp_get_num_threads();
        #pragma omp single     // Ensure only one thread creates the initial task
        {
          #pragma omp task
          {printf("Perl ");}   // Task #1
          #pragma omp task
          {printf("rocks ");} // Task #2
          #pragma omp task
          {printf("on %d threads!\n", got_num_threads);} // Task #3
          #pragma omp taskwait
        }
    }
    return got_num_threads;
}
```

```
on 1 threads!
rocks Perl on 2 threads!
rocks Perl Perl rocks on 3 threads!
Perl rocks on 4 threads!
Perl rocks on 5 threads!
Perl rocks on 6 threads!
Perl rocks on 7 threads!
Perl rocks on 8 threads!
Perl rocks on 9 threads!
Perl rocks on 10 threads!
Perl rocks on 11 threads!
Perl rocks on 12 threads!
Perl rocks on 13 threads!
Perl rocks on 14 threads!
Perl rocks on 15 threads!
Perl rocks on 16 threads!
```

New!

Thread Affinity

- Setting thread affinity (**`OMP_PROC_BIND`**) helps optimize performance by improving cache and memory access patterns on NUMA architectures.
- **`OMP_PROC_BIND`** has 4 possible values:
  - **`true`** or **`close`** (as in *near*)
  - **`master`**
  - **`spread`**
  - **`false`** (or *unset*)

Thread Affinity

- **`OMP_PROC_BIND`**`=true` or **`OMP_PROC_BIND`**`=close`:
  - **Threads are bound close to where the master thread is running.** It is useful for ensuring threads operate near each other to minimize cross-socket memory access.
- **`OMP_PROC_BIND`**`=spread`:
  - **Threads are spread out across the available sockets (NUMA nodes).** This setting aims to balance the workload across all available CPU sockets.
- **`OMP_PROC_BIND`**`=master`:
  - **Threads are bound to the same CPU core as the master thread.** This can be useful in scenarios where thread locality with respect to the master thread is critical.
- **`OMP_PROC_BIND`**`=false` or unset:
  - **OpenMP runtime system decides the thread binding strategy based on the system default or runtime decisions.** This may vary between different OpenMP implementations and system configurations.

Thread Affinity

```
use strict;
use warnings;

use OpenMP::Simple;
use OpenMP::Environment;

use Inline (
    C    => 'DATA',
    with => qw/OpenMP::Simple/,
);

my $env = OpenMP::Environment->new;

foreach my $policy (qw/true close spread master false/) {
  for my $num_threads ( 1 .. 16 ) {
    my $current_value  = $env->omp_num_threads($num_threads);
    my $current_policy = $env->omp_proc_bind($policy);
    my $got_threads = runit();
  }
}

__DATA__
__C__
int runit() {
    PerlOMP_UPDATE_WITH_ENV__NUM_THREADS
    // Start a parallel region
    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num();
        printf("Thread %d is running on CPU %d\n", thread_id, sched_getcpu());
    }

    return 0;
}

__END__
```

OpenMP *runtime*

New!

SIMD

Single Instruction Multiple Data

- SIMD and Performance: SIMD OpenMP directives are essential for achieving high performance on modern processors by exploiting parallelism at the instruction level. Proper usage of SIMD can significantly accelerate computational tasks that operate on large arrays or matrices.
- Limitations: Not all loops can be effectively vectorized using SIMD directives. Loop-carried dependencies, complex control flow, or irregular memory access patterns may hinder vectorization.
- Evolution: OpenMP continues to evolve with newer versions (5.0 and beyond), introducing enhancements and new features for SIMD support, such as SIMD-awareness in array sections (simd clause with aligned, linear, etc.).
- The **`simd`** construct has been introduced, but no new environmental variables or runtime functions have been introduced to support it.

SIMD Example

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

int main() {
    int matrix[100][50];
    int result[100][50];
    int i, j, k; // Declare loop indices outside of the loops

    // Initialize matrix
    for (i = 0; i < 100; ++i) {
        for (j = 0; j < 50; ++j) {
            matrix[i][j] = i * 50 + j;
        }
    }

    // Parallel region with nested loop vectorization
    #pragma omp parallel for collapse(2) private(i, j, k)
    for (i = 0; i < 100; ++i) {
        for (j = 0; j < 50; ++j) {
            #pragma omp simd
            for (k = 0; k < 4; ++k) { // Assume vector length is 4
                result[i][j] += matrix[i][j] * k;
            }
        }
    }
    return 0;
}
```

SIMD Support in CPUs

- x86 (Intel and AMD):
  - **SSE** (Streaming SIMD Extensions): Introduced with Pentium III processors, SSE provides SIMD support for operations on floating-point and integer data types. SSE has evolved through multiple versions (SSE2, SSE3, SSE4) with increased capabilities for parallel processing.
  - **AVX** (Advanced Vector Extensions): AVX builds upon SSE and offers wider SIMD registers (256-bit and 512-bit), enabling higher throughput for vectorized operations. AVX2 and AVX-512 further enhance vectorization performance.
- ARM (Various Manufacturers):
  - **NEON**: NEON is ARM's SIMD architecture extension, providing similar functionality to SSE on x86 processors. It supports vector operations on integers and floating-point data types, enhancing performance for multimedia and signal processing applications.
- Compiler vendors such as GCC, Intel ICC, Clang, and others implement SIMD support for various architectures. They provide tools and flags (**-fopenmp**, -mavx, -mfpu=neon, etc.) to enable and control SIMD vectorization, ensuring compatibility with OpenMP directives for parallel programming.

New!

Heterogeneous Architectures

(e.g., GPUs)

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

int main() {
    int a = 10, b = 20, c;

    #pragma omp target map(from:c)
    {
        c = a + b;
    }

    printf("Result: %d\n", c);

    return 0;
}
```

Compiling for GPUs

- Requires use of **`target`** OpenMP directive
- Requires compiler support that link to external libraries (e.g., CUDA, etc)
- Enabled in **`gcc`** via **`-fopenmp`**

GCC Offloading & `nvptx-none`

- **Offloading Support:**

GCC introduced support for offloading computations to accelerators such as GPUs starting from version 4.9. This allows parallel sections of code annotated with OpenMP directives (target, parallel, etc.) to be executed on devices like NVIDIA GPUs.

- **Target Architecture:**

`-foffload=nvptx-none` specifies that the offloaded code should target NVIDIA GPUs using the NVIDIA Parallel Thread Execution (PTX) intermediate representation (nvptx). PTX is a portable assembly-like language that NVIDIA GPUs can execute.

- **Compilation Process:**

When you compile a program with **`-foffload=nvptx-none`**, GCC generates PTX code (nvptx) instead of native GPU machine code. PTX is architecture-independent and serves as an intermediate representation.

The generated PTX code can then be further compiled and optimized by NVIDIA's CUDA toolchain (nvptx-as, nvptx-nvcc) into executable machine code (cubin) that runs on *NVIDIA GPUs*.

GCC Offloading & `nvptx-none`

- **Command Usage:**

  Example command to compile a program (program.c) with OpenMP offloading to NVIDIA GPUs:

  `gcc -o program -fopenmp -foffload=nvptx-none program.c`

  `-fopenmp`: Enables OpenMP support in GCC.

  `-foffload=nvptx-none`: Specifies that the offloaded code should target NVIDIA GPUs using PTX.

- **CUDA Toolkit Requirement:**

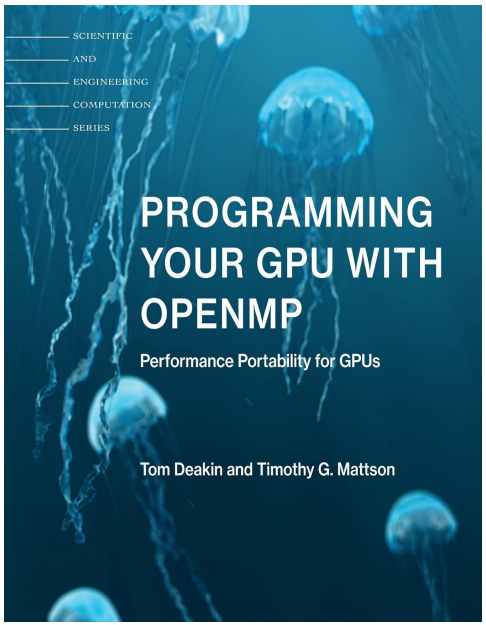
  To execute programs compiled with **`-foffload=nvptx-none`**, you need to have the NVIDIA CUDA toolkit installed on your system. The CUDA toolkit provides necessary libraries and tools (nvptx-as, nvptx-nvcc) to compile PTX code into executable GPU machine code.

- **Hardware and Driver Support:**

  Ensure that your system has compatible NVIDIA GPU hardware ad drivers installed. The CUDA toolkit supports a wide range of NVIDIA GPUs, and performance can vary based on GPU architecture and capabilities.

More OpenMP, GPUs, Offloading

- https://gcc.gnu.org/projects/gomp/
- https://gcc.gnu.org/wiki/Offloading
- Many online tutorials and examples
- Books exist, such as:



Present & Future Direction

- All Perl modules presented here are geared towards running and controlling `Inline::C` with OpenMP
- Current work is easy getting Perl variables into and out of `Inline::C`'d functions
- **Near future**: more options from `OpenMP::Simple` for OpenMP *first* programmers (variable IN,OUT; internal transforms)
- **Ultimate goal**: provide thread-safe (OpenMP) RW of variables directly via Perl API (or similar type of library via `OpenMP::Simple`)

Perl+OpenMP for Fortran?

- Very possible with gfortran and highly desirable, but ..
- I'd need a suitable Inline module for Fortran 90
- I am open to doing this pro bono if we had a working `Inline::Fortran` module

OpenACC

- *a programming standard for parallel computing developed by Cray, CAPS, Nvidia and PGI*
- Same directives/runtime based approach as OpenMP
- Also supported by GCC (9.1 offers “nearly complete OpenACC spec 2.5 support)
- Perl+OpenMP project has no intention to support it
- But the opportunity is there
- I am open to being sponsored for the work

Perl+OpenMP

Where it happens!

- https://github.com/Perl-OpenMP
- May become part of *Perl Community*'s HPC Perl Committee (details still being worked out)

Thank You

BINARY.NET

ADCIRC Live

Professional training, paid support, online tools, and ADCIRC data.

CERA Coastal Emergency Risks Assessment

Storm Surge - Wave - Compound Flood Guidance

Coastal Computing Services, LLC

Seahorse COASTAL CONSULTING