Leveraging OpenMP Infrastructure for Language Level Parallelism

Darryl Gove
Senior Principal Software Engineer
Outline

• Proposal and Motivation
• Overview of OpenMP
• Open Issues and Initial Proposal
• Concluding Remarks and Next Steps
Proposal and Motivation

Why we should do this
What is OpenMP

- OpenMP “describes” parallelism
  - Parallelism is “added” to the serial code
- Supported by all major compilers
- 1.0 in 1997, version 4.0 this year
- Widely used (even has benchmark suites!)
OpenMP Infrastructure

Syntax
Source

Actions
Compiler

Object file

Reuse

Runtime Environment

API

OpenMP API
Compiler API

Existing Infrastructure

OpenMP Runtime Library
Leveraging OpenMP
For Developers

- Familiarity for OpenMP developers
  - Easy adoption

- No conflict with OpenMP
  - Can mix parallelisation methods
  - Don’t force a choice on developers
  - Large existing code base using OpenMP

- Gentle transition *both ways*
  - “Standardise” parallel code
  - Utilise “leading edge” OpenMP
Reuse of OpenMP Infrastructure
For compilers

• Widely available on most/all platforms
  – Low development cost
  – Rapid availability

• 15 years of production use
  – Few remaining bugs
  – Already tuned

• Not another library
  – Compatibly leverages existing code
  – Low maintenance costs
Overview of OpenMP
Types of parallelisation

• Parallel region
• Parallel sections
• Parallel for
• Parallel tasks
#pragma omp parallel
{
  ...
}

1 2 \ldots N
Parallel Region Example

```c
#include <stdio.h>
int main()
{
    #pragma omp parallel num_threads(3)
    {
        printf("'Ello\n");
    }
}
$ cc -O -xopenmp copper.c
'Ello
'Ello
'Ello
'Ello
```
Parallel Sections Example

```c
#pragma omp parallel sections
{
    #pragma omp section
    {
        ...
    }
    #pragma omp section
    {
        ...
    }
}
```
Parallel Sections

```c
#pragma omp parallel sections num_threads(2)
{
    #pragma omp section
    { printf("Hello\n"); }  
    #pragma omp section
    { printf("There\n"); } 
}
...
$ cc -O -xopenmp sections.c 
$ .a.out 
There 
Hello
```
Parallel For

#pragma omp parallel for
for (...)
{
    ...
    1
    2
    ........
    N
}
Parallel For Example

double sum(double * values, int length)
{
    double total = 0.0;
    #pragma omp parallel for reduction(+:total)
    for (int i = 0; i < 100000; i++)
    {
        total += values[i];
    }
    return total;
}
Parallel Tasks

```c
#pragma omp parallel
#pragma omp single
while (...) {
    #pragma omp task
    {
        ...
    }
    #pragma omp task
    {
        ...
    }
}
```
Parallel Tasks Example

```c
int fib(int n) {
    int i, j;
    if (n<2) return n;
    else {
        #pragma omp task shared(i) firstprivate(n)
        i=fib(n-1);
        #pragma omp task shared(j) firstprivate(n)
        j=fib(n-2);
        #pragma omp taskwait
        return i+j;
    }
}

int main() {
    #pragma omp parallel
    {
        #pragma omp single
        printf("fib(%d) = %d\n", 10, fib(10));
    }
}
```
Core OpenMP Parallelisation

- `#pragma omp parallel for`
  - Most commonly used
- `#pragma omp task`
  - Most flexible
- Both have “clauses”
  - How parallelisation is performed
  - How variables are scoped
OpenMP Variable scoping

- **shared(var)**
  - All threads share original

- **reduction(operation:var)**
  - All threads contribute to result

- **private(var)**
  - Each thread has private copy

- **firstprivate(var)**
  - Private but initialised to value before parallel region

- **lastprivate(var)**
  - Private but last value preserved after parallel region
Useful OpenMP Clauses

- `proc_bind` – bind threads to processors
- `if` – should multiple threads be used
- `num_threads` – number of threads used
- `depend` – enable ordering of tasks
- `collapse` – merge multiple loops
Core of OpenMP API

**Threads**

- `omp_[get|set]_num_threads`: Number of threads used for parallel region
- `omp_get_max_threads`: Number of threads available for use
- `omp_get_thread_limit`: Maximum number of threads available
- `omp_get_thread_num`: Get ID of current thread
- `omp_get_proc_bind`: Thread binding pattern

**Scheduling**

- `omp_[get|set]_schedule`: Scheduling algorithm used
- `omp_[get|set]_dynamic`: Demand based control of number of threads

**Hardware**

- `omp_get_num_processors`: Number of processors available (next slide)
- `omp_get_wtime`: Wall time in seconds
- `omp_get_wtick`: Precision of wall timer
A Detour into Hardware

How many threads?

• Need to know more about hardware
  – Number of hardware threads?

• Topology important
  – Threads per core?
  – Cores per socket?
  – Sockets per system?

• Performance varies depending on thread location
  – Use many cores
  – Use few sockets
Scattering or Gathering Threads

• **Spread (ie scatter)**
  − Use fewest threads per core
  − Gives each thread most core resources

• **Close (ie co-locate)**
  − Use fewest cores for threads
  − Shares core resources
  − Minimises communication costs
Open Issues and Initial Proposal
Open Issues

• Syntax
  – Many options
  – One used as illustration

• Variable scoping
  – Multiple ways to do this
  – Including *not* doing it

• What to include/exclude
  – Aiming to get most benefit
  – From fewest features
_Parallel for

_Parallel for (int i=0; i<1000; i++)
{
    // Work divided across all threads
    a[i] = b[i] * c[i];
}


```c
Parallel Task
{
    Parallel Task
    {  i=fib(n-1);  }
    Parallel Task
    {  j=fib(n-2);  }
    waitfortasks();
    return i+j;
}
```
while(1) {
    int stream = accept(s, &client, &size);
    _Parallel_Task
    {
        char buffer[1024];
        while (recv(stream, buffer, sizeof(buffer), 0))
            { send(stream, buffer, strlen(buffer)+1, 0); } 
    }
}
waitfortasks();
Variable Scoping Overview

- May not need `shared` or `private`
  - Data shared by default
  - Private variables declared in parallel region
- Need reductions
  - Important
  - Tricky to emulate
- Private variants sometimes useful
  - `firstprivate`
  - `lastprivate`
Variable Scoping

• `_Parallel <variable scoping>`
• For example:

```c
`_Parallel _Reduction(+:temp) for (...)`
{
    // Work divided across all threads
    temp += ...; // Reduction
}
```
## C API

```c
#include <parallel.h>

[get|set]numthreads()  // Number of threads used for parallel region
getmaxthreads()        // Maximum number of threads available
getthreadID()          // Get ID of current thread
[get|set]threadbinding() // Thread binding pattern
[get|set]loopschedule()  // Scheduling algorithm used
[get|set]dynamicthreads() // Demand based control of number of threads
getprocessorcount()    // Number of processors available
```
Conclusion and Next Steps
OpenMP 4.0

- Support for user defined reductions
- Support for SIMD loops
  - `simd` clause + `safelen(length)`
- Support for GPU/Accelerators
  - target a particular device
  - map data to/from device
  - array sections
- ....
Concluding Remarks

- Co-exist with OpenMP
  - Gentle integration curve
  - Many existing users
  - Focuses compiler effort

- Leverage existing OpenMP infrastructure
  - Existing code
  - Quick time to market
  - Robust and tuned implementations
C++ API

#include <parallel>

parallel::numthreads
parallel::maxthreads
parallel::threadID
parallel::threadbinding
parallel::loopschedule
parallel::dynamicthreads
parallel::processorcount

Number of threads used for parallel region
Maximum number of threads available
Get ID of current thread
Thread binding pattern
Scheduling algorithm used
Demand based control of number of threads
Number of processors available
Tasks and `std::async`

- Minimal source change (OpenMP principle)
- Thread pool
- Anonymous tasks