



Introduction to OpenMP  
Ulrich Drepper  
Consulting Engineer, Red Hat, Inc.

# Before we start...

Clarify a few phrases:

- Process
  - Scheduled execution unit with its own address space
- Thread
  - Scheduled execution unit, sharing address space with other threads
- Future, (Task)
  - Description of work, not scheduled

# Forms of Parallelism

## Multi-process:

- Unix `fork()`
- Separate address space: sharing is explicit
  - Linux's `clone()` and `unshare()` provide finer granularity
- More robust:
  - No accidental memory corruption
  - No complete tear-down on crash
- Fast Linux Inter-Process Communication (IPC)
  - Pipes, message queues, shared memory
  - Robust mutexes for crash handling

# Forms of Parallelism

## Multi-process:

- Exploit multiple machines with few additional changes
- Not well suited for automatically generated parallelism
  - Exception: using of MPI

# Forms of Parallelism

## Multi-threaded:

- Widely available through `pthread_create()`
- Share everything except register content (implies stack pointer)
- Accidental corruptions felt by every thread
- Thread crash causes complete tear-down
- Communication costs minimal
  - Only synchronization cost
- Limited to single machine

# Explicit Parallelism

Processes and threads require explicit handling

- Start explicitly
  - How many?
  - What to do?
- Not scalable
  - Programmers cannot keep track of more than a handful of execution paths
- Parametrize explicitly
  - Where to run (affinity)?
- Machine architecture changes and becomes more important
  - Programmers cannot adjust each program individually

# Parallel Code

Parallel code looks like serial code to tools

- Programmer's responsibility to use synchronization
- Hard to check for all kinds of mistakes

Better model: tell tools about parallelism

- Requires integration into language
- Tools can
  - Warn about some incorrect uses
  - Use optimal mechanisms without hardcoding in sources
- After adjustment of tools for new machine architecture only recompilation needed

# OpenMP

## *Language Extension*

- C, C++, Fortran
- Compiler gets insight into parallelism
- Same program can work sequentially
  - Makes debugging easier
  - Allows using older tools on same code

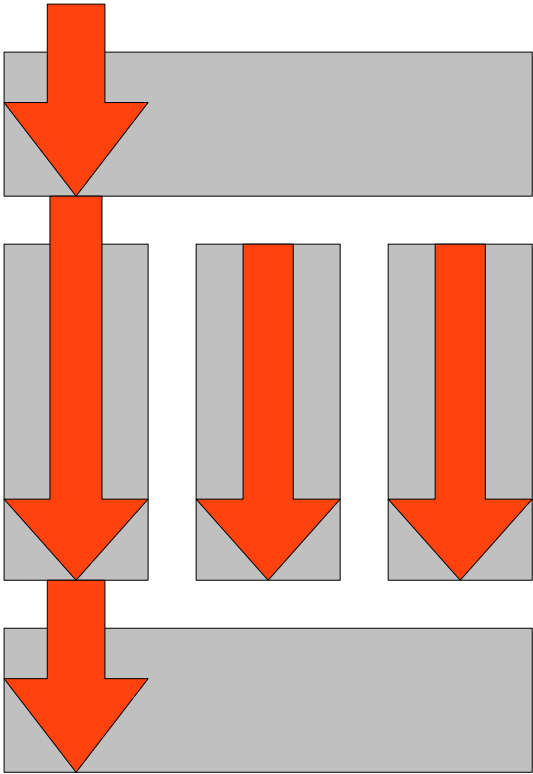
Openly developed specification

Central place for many optimizations

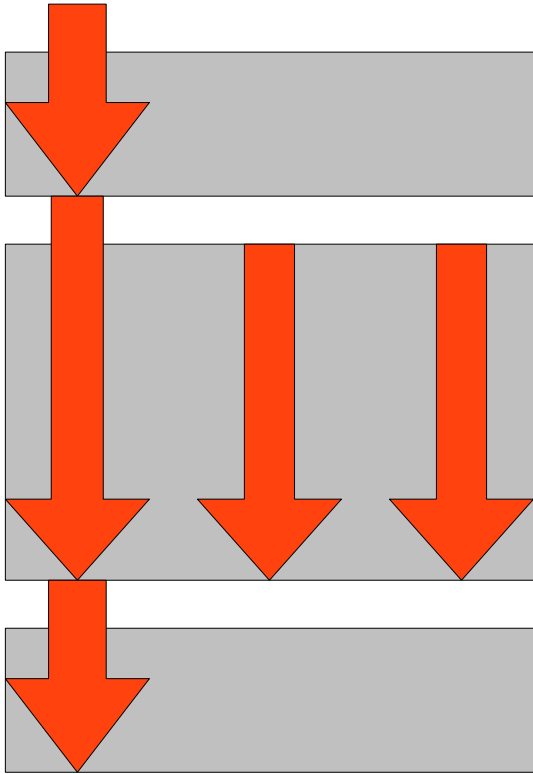
- OpenMP comes with runtime parts
- Specification allows runtime to make decisions about number and placement of threads

# Forms of Structured Parallelism

Independent sections



Loops



# Parallel Sections

```
sum = 0;
```

```
for (i = 0; i < N; ++i)
```

```
    sum += count_whatever(some_data1[i]);
```

```
for (i = 0; i < N; ++i)
```

```
    sum += count_whatever(some_data2[i]);
```

# Parallel Sections

Available in OpenMP v2.5 (RHEL5):

```
sum = 0;
```

```
# pragma omp parallel sections  
{
```

```
    for (i = 0; i < N; ++i)
```

```
#        pragma omp atomic  
        sum += count_whatever(some_data1[i]);
```

```
# pragma omp section
```

```
    for (i = 0; i < N; ++i)
```

```
#        pragma omp atomic  
        sum += count_whatever(some_data2[i]);
```

```
}
```

# Parallel Sections

Available in OpenMP v2.5 (RHEL5):

```
sum = 0;
```

```
# pragma omp parallel  
{
```

Runtime may  
start threads

```
for (i = 0; i < N; ++i)
```

```
# pragma omp atomic
```

```
sum += count_whatever(some_data1[i]);
```

```
# pragma omp section
```

```
for (i = 0; i < N; ++i)
```

```
# pragma omp atomic
```

```
sum += count_whatever(some_data2[i]);
```

```
}
```

Implicit barrier

# Parallel Sections

Available in OpenMP v2.5 (RHEL5):

```
    sum = 0;
# pragma omp parallel sections reduction(+:sum)
{

    for (i = 0; i < N; ++i)

        sum += count_whatever(some_data1[i]);
# pragma omp section
    for (i = 0; i < N; ++i)

        sum += count_whatever(some_data2[i]);
}
```

# Parallel Sections

Available in OpenMP v2.5 (RHEL5): **NEW!**

```
sum = 0;  
# pragma omp parallel sections reduction(+:sum)  
{
```

```
for (i = 0; i < N; ++i)
```

```
sum += count_whatever(some_data1[i]);
```

```
# pragma omp section  
for (i = 0; i < N; ++i)
```

```
sum += count_whatever(some_data2[i]);
```

```
}
```

**No  
Atomic**

# Parallel Sections

Available in OpenMP v2.5 (RHEL5):

```
sum = 0;
```

```
# pragma omp parallel for reduction(+:sum)
```

```
for (i = 0; i < N; ++i)
```

```
    sum += count_whatever(some_data1[i]);
```

```
# pragma omp parallel for reduction(+:sum)
```

```
for (i = 0; i < N; ++i)
```

```
    sum += count_whatever(some_data2[i]);
```

# Parallel Sections

Available in OpenMP v2.5 (RHEL5):

```
sum = 0;
```

```
# pragma omp parallel for reduction(+:sum)  
for (i = 0; i < N; ++i)
```

```
    sum += count_whatever(some_data1[i]);
```

```
# pragma omp parallel for reduction(+:sum)  
for (i = 0; i < N; ++i)
```

```
    sum += count_whatever(some_data2[i]);
```

Implicit barrier

Implicit barrier

# Parallel Sections

Available in OpenMP v2.5 (RHEL5):

```
    sum = 0;
#   pragma omp parallel
{
#   pragma omp for reduction(+:sum) nowait
    for (i = 0; i < N; ++i)

        sum += count_whatever(some_data1[i]);
#   pragma omp for reduction(+:sum)
    for (i = 0; i < N; ++i)

        sum += count_whatever(some_data2[i]);
}
```

# Parallel Sections

Available in OpenMP v2.5 (RHEL5):

```
    sum = 0;
#   pragma omp parallel
{
#   pragma omp for reduction(+:sum) nowait
    for (i = 0; i < N; ++i)
        sum += count_whatever(some_data1[i]);
#   pragma omp for reduction(+:sum)
    for (i = 0; i < N; ++i)
        sum += count_whatever(some_data2[i]);
}
```

No implicit  
barrier

## Convenience

```
#pragma omp parallel for  
for (...)  
    ...
```

short for

```
#pragma omp parallel  
{  
# pragma omp for  
for (...)  
    ...  
}
```

Similarly for parallel sections

# Explicit Tasks

Available in OpenMP v3 (RHEL6):

```
    sum = 0;
# pragma omp parallel
{
# pragma omp for nowait
    for (i = 0; i < N; ++i)
#         pragma omp task untied
        {
#             pragma omp atomic
                sum += count_whatever(some_data1[i]);
        }
# pragma omp for nowait
    for (i = 0; i < N; ++i)
#         pragma omp task untied
        {
#             pragma omp atomic
                sum += count_whatever(some_data2[i]);
        }
}
```

# Explicit Tasks

Available in OpenMP v3 (RHEL6):

```
sum = 0;
# pragma omp parallel
{
# pragma omp for nowait
  for (i = 0; i < N; ++i)
#      pragma omp task untied
      {
#          pragma omp atomic
            sum += count_whatever(some_data1[i]);
      }
# pragma omp for nowait
  for (i = 0; i < N; ++i)
#      pragma omp task untied
      {
#          pragma omp atomic
            sum += count_whatever(some_data2[i]);
      }
}
```

Any thread can pick up  
task

Implicit barrier, incl all tasks

# Exclusion

Producer

```
struct elem *newp
    = ...;
#pragma critical pclock
{
    newp->next = first;
    first = newp;
}
```

Consumer

```
#pragma critical pclock
{
    curp = first;
    if (curp != NULL)
        first = first-
            >next;
}
... use curp ...
```

# Only One

Executed by one thread

```
#pragma omp parallel  
{  
    fct1();  
# pragma omp single nowait  
    fct2();  
    fct3();  
}
```

Executed by master thread

```
#pragma omp parallel  
{  
    fct1();  
# pragma omp master  
    fct2();  
    fct3();  
}
```

# Only One

Executed by one thread

```
#pragma omp parallel  
{  
    fct1();  
# pragma omp single nowait  
    fct2();  
    fct3();  
}
```

No implied barrier

Executed by master thread

```
#pragma omp parallel  
{  
    fct1();  
# pragma omp master  
    fct2();  
    fct3();  
}
```

Also available  
for other  
constructs

# Extending the Range

Nested loops are “natural”

```
for (i = 1; i < N - 1; ++i)
    for (j = 1; j < M - 1; ++j)
        b[i][j] = (a[i][j-1]+a[i-1][j]+a[i][j]
                    +a[i+1][j]+a[i][j+1]) / 5;
```

# Extending the Range

Nested loops are “natural”

Available in OpenMP v3 (RHEL6):

```
#pragma omp parallel for collapse(2)
```

```
for (i = 1; i < N - 1; ++i)
```

```
    for (j = 1; j < M - 1; ++j)
```

```
        b[i][j] = (a[i][j-1]+a[i-1][j]+a[i][j]  
                  +a[i+1][j]+a[i][j+1]) / 5;
```

## Extending the Range

Nested loops are “natural”

Available in OpenMP v3 (RHEL6):

**Both loops in one  
iteration range**

```
#pragma omp parallel for collapse(2)
```

```
for (i = 1; i < N - 1; ++i)
```

```
    for (j = 1; j < M - 1; ++j)
```

```
        b[i][j] = (a[i][j-1]+a[i-1][j]+a[i][j]  
                  +a[i+1][j]+a[i][j+1]) / 5;
```

# Scheduling

Parallel section has rules for how many threads to create

- Programmer can request number
- User can control through environment variable
- Or: OpenMP runtime can be in complete control

Loop scheduling:

- Reliable assignment of iterations to threads
- Fair distribution
- Or: also under control of the runtime

# Outlook

OpenMP compiler & runtime become more intelligent:

- Runtime knows about machine architecture
- Compiler tells runtime about cost and memory behavior of the code

→ Runtime in good position to make decision

- No adjust of program for new machine needed

Example:

- Tightly coupled tasks, writing to same memory
  - Run on one socket or cache domain

# Outlook

## Example: Large Working Set

- Choose loop iterations to touch different pages, allocate pages on different NUMA nodes, set thread affinity

## Coordination:

- Many uncoordinated OpenMP programs unnecessarily stress machine
- With coordination between all OpenMP processes runtime could ensure machine resources are not oversubscribed

## Result:

If possible, let runtime decide!



**Questions?**