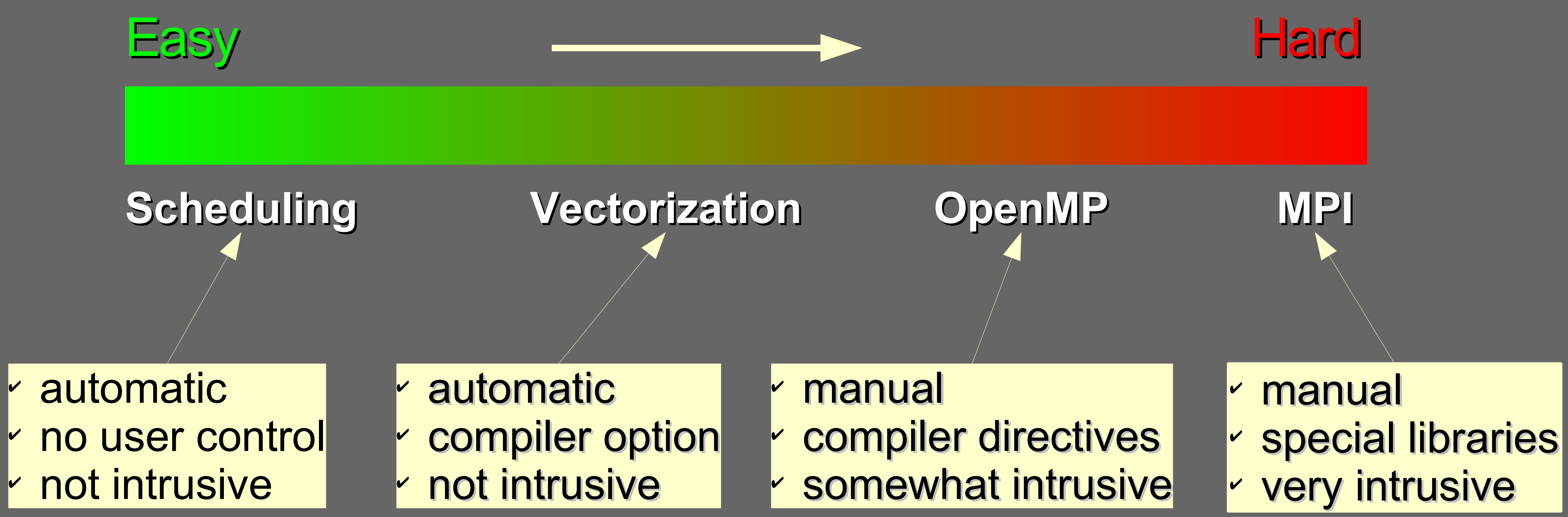
Parallel Programming with GCC

Diego Novillo
dnovillo@redhat.com
Red Hat Canada

Gelato Itanium Conference and Expo
San Jose, California, April 2006

Introduction

- GCC supports four concurrency models



Ease of use not necessarily related to speedups!

Vectorization - 1

- Perform multiple array computations at once
- GCC currently works only on loops
- Two distinct phases
  - Analysis → high-level
  - Transformation → low-level
- Successful analysis depends on
  - Data dependency analysis
  - Alias analysis
  - Pattern matching
- Successful transformation will depend on hardware capabilities
- Performance gains only expected on loop intensive code

Vectorization - 2

```
for (n = 0; n < 2e8; n++)
  for (i = 0; i < 16; i++)
    a[i] = b[i];
```

Original (21 secs)

```
        mov ar.lc = 15
.L2:
        .mmi
        adds r15 = 16, r12
        add r14 = r15, r16
        add r15 = r36, r16
        .mmi
        adds r16 = 4, r16
        ldfs f6 = [r14]
        nop 0
        .mib
        stfs [r15] = f6
        nop 0
        br.cloop.sptk.few .L2
```

Vectorized (12 secs)

```
        mov ar.lc = 7
.L2:
        .mmi
        adds r15 = 16, r12
        add r14 = r15, r16
        add r15 = r16, r36
        .mmi
        adds r16 = 8, r16
        ldf8 f6 = [r14]
        nop 0
        .mib
        stf8 [r15] = f6
        nop 0
        br.cloop.sptk.few .L2
```

Vectorization - 3

- Enable vectorizer

```
$ gcc -ftree-vectorize -O2 prog.c
```

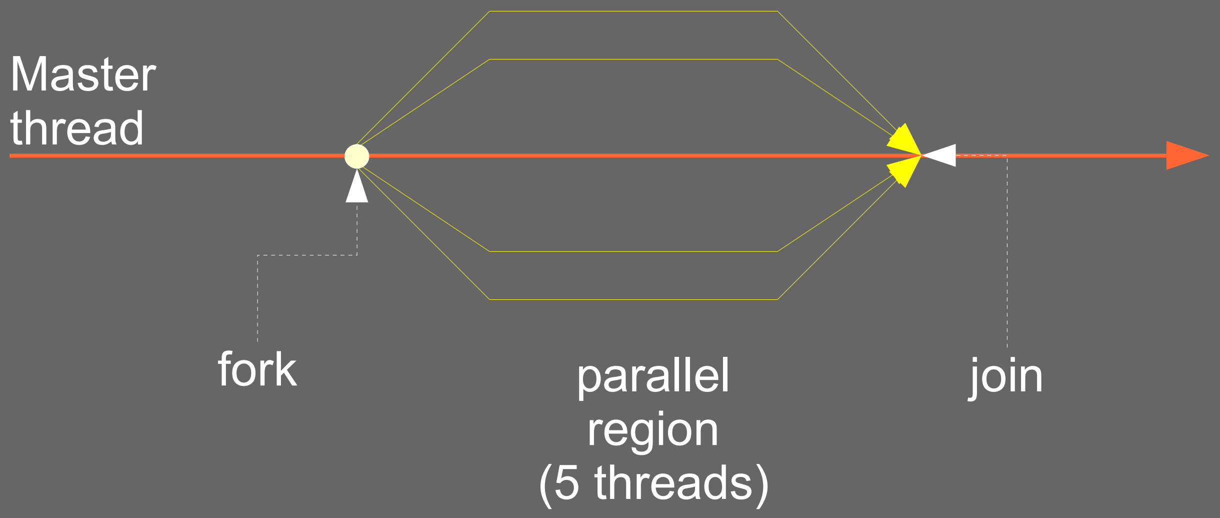
- Requires additional `-m` flags on some architectures
- Speedups depend greatly on
  - Regular, compute-intensive loops
  - Data size and alignment
  - “Simple” code patterns in inner loops
  - Aliasing
- Debugging
  - `-fdump-tree-vect` enables dump
  - `-ftree-vectorizer-verbose=[0-7]` controls verbosity

Parallel Programming

- Parallelism explicitly controlled by user
- Different mental model
  - Look for macro parallelism (tasks)
  - Tasks mapped to threads or processes
  - Profitable granularity of task dictated by target
  - Tasks have shared or private data
- Parallel programming environment provides
  - Task creation
  - Data sharing
  - Synchronization
- Two main environments
  - Shared memory
  - Message passing

OpenMP

- Language extensions for shared memory concurrency
- Supports C, C++ and Fortran
- Designed around compiler pragmas
  - **Directives** specify parallelism and work sharing
  - **Clauses** specify attributes for data sharing and scheduling
- Based on master-slave model



Programming Model

- Directives → **`#pragma omp`** (C, C++) or !**`$omp`** (Fortran)
- Compiler replaces directives with calls to runtime library (**`libgomp`**)
- Library offers API for querying/controlling threads and scheduling
- Runtime controls in program or environment variables
  - **`OMP_NUM_THREADS`**, **`OMP_SCHEDULE`**, **`OMP_DYNAMIC`**, **`OMP_NESTED`**
- Programmer responsible for synchronization and sharing
  - Sharing with variables marked with sharing clauses
  - Synchronization specified with synchronization directives
- Original intent: Same program runs sequential or in parallel
  - Compiler switch enables/disables the pragmas
  - Invalid sequential programs are possible too: parallel algorithms

OpenMP Hello World

```
#include <omp.h>
main()
{
    #pragma omp parallel
    {
        printf (“[%d] Hello\n”, omp_get_thread_num());
    }
}

$ gcc -fopenmp -o hello hello.c
$ export OMP_NUM_THREADS=4
$ ./hello
[2] Hello
[3] Hello
[0] Hello
[1] Hello
```

fork → `{`

join → `}`

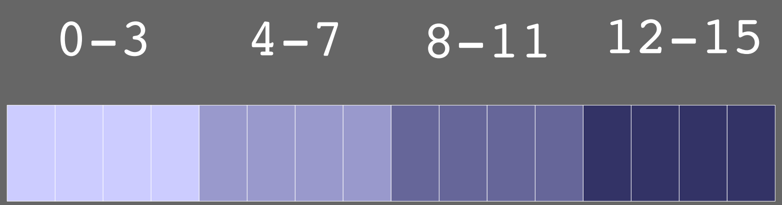
Optional → `$ export OMP_NUM_THREADS=4`

Master thread → `[0] Hello`

Worksharing - 1

- Distributes pieces of work to threads in region
- Worksharing does **not** create new threads
- Most common distribution mechanism: loop iterations

```
#pragma omp parallel
#pragma omp for
for (i = 0; i < 16; i++)
   a[i] = i;
```



- Each thread executes a subset of the iteration space
- Scheduling determines distribution of iteration subsets
- No synchronization, other than implicit barrier at the end

Worksharing - 2

- `#pragma omp for`
- data/loop parallelism
- Partitions iteration space with `schedule(type, chunk)`
- `chunk` is optional. Number of iterations for each thread.
- `type` may be
  - `static` Static round-robin distribution by thread ID
  - `dynamic` Iterations on a first-come, first-served queue
  - `guided` Same as `dynamic` but varying chunk size
  - `runtime` Taken from environment var `OMP_SCHEDULE`.
- Dynamic and guided schedules achieve better load balancing
- Runtime useful to avoid re-compiling.

Worksharing - 3

- **`#pragma omp sections`**
  - cobegin/coend style parallelism
  - Sections are delimited with **`#pragma omp section`**
  - Each section is executed by a different thread

```
#pragma omp parallel sections
{
    #pragma omp section
        t1();
    #pragma omp section
        t2();
    #pragma omp section
        t3();
}
```

Can be combined

Worksharing - 4

- `#pragma omp workshare`
  - Distributes execution of Fortran FORALL, WHERE and array assignments
  - Only valid in Fortran
  - Distribution of units of work is up to the compiler

```
integer :: a (10), b (10)
!$omp parallel workshare
  a = 10
  b = 20
  a(1:5) = max (a(1:5), b(1:5))
!$omp end parallel workshare
```

Data Sharing

- Sharing specified at variable level
- Three sharing methods
  - Shared

    ```
    #pragma omp parallel shared (x,y)
    ```
  - Semi-private

    ```
    #pragma omp parallel firstprivate (x,y)
    #pragma omp parallel lastprivate (x,y)
    #pragma omp single copyprivate (x)
    ```
  - Private

    ```
    #pragma omp parallel private (x,y)
    ```
- Various rules to determine sharing properties.
  - Globals and heap allocated variables are shared
  - Locals declared inside a directive body are private
  - Loop iteration variables for parallel loops are private

Synchronization

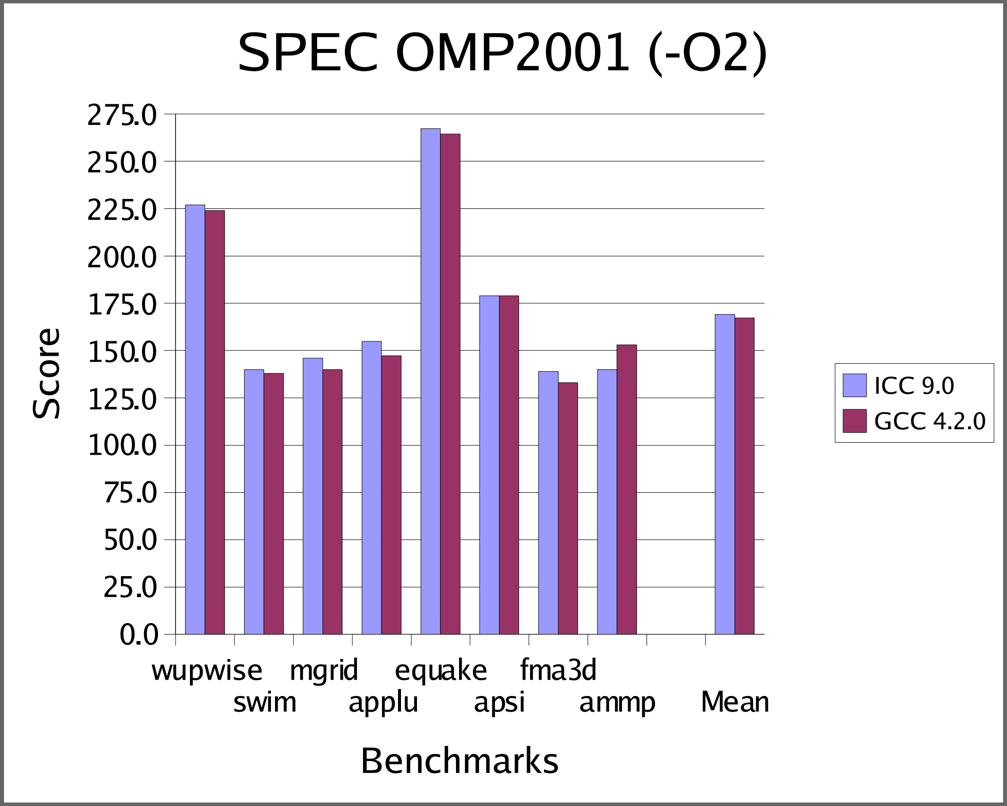
- With few exceptions user is ultimately responsible for preventing data races using OpenMP directives
- `#pragma omp single`
  - Only one thread in thread team enters block.
- `#pragma omp master`
  - Only master thread enters block.
- `#pragma omp critical`
  - Mutual exclusion.
- `#pragma omp barrier`
- `#pragma omp atomic`
  - Atomic storage update: `x` *`op`*`= expr`, `x++`, `x--`
- `#pragma omp ordered`
  - Used in loops, threads enter in loop iteration order.

Message Passing

- Completely library based
- No special compiler support required
- The “assembly language” of parallel programming
  - Ultimate control
  - Ultimate pain when things go wrong
  - Computation/communication ratio must be high
- Message Passing Interface (MPI) most popular model
- Separate address spaces
  - It may also be used on a shared memory machine
- Heavy weight processes
- Communication explicit via network messages
  - User responsible for marshalling, sending and receiving

Status and Future Work

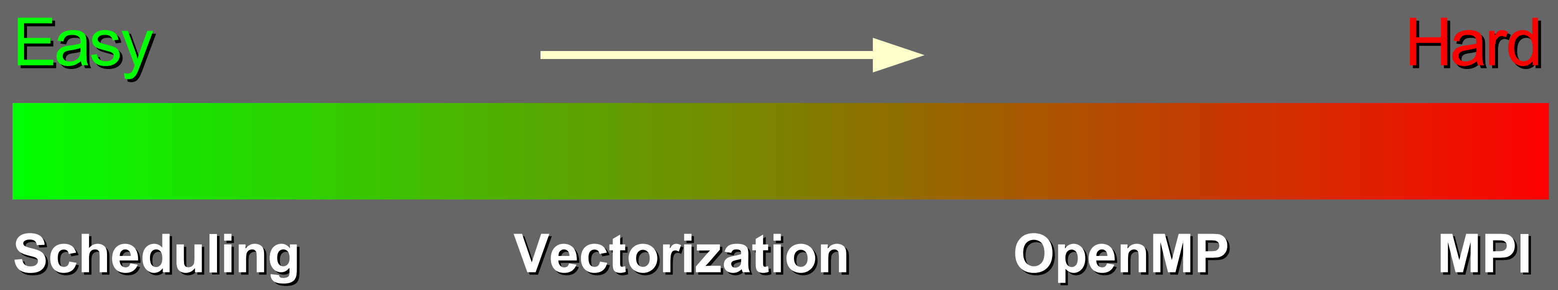
- Vectorization support started in 4.0 series
- OpenMP will be released with 4.2 later this year
- Implementation available in Fedora Core 5
- Automatic parallelism planned using OpenMP infrastructure



redhat

Conclusions

- GCC supports full spectrum of common parallel models



- There is no “right” choice
  - Granularity of work main indicator
  - Evaluate complexity ↔ speedup trade-offs
- Complex parallel applications may benefit from combined approach
- Algorithms matter!
  - Good sequential algorithms may make bad parallel ones