Parallel Programming with GCC

Diego Novillo
dnovillo@redhat.com

Red Hat Canada
Outline

• Introduction to parallel computing
• Parallel programming models
  – Automatic parallelization
  – Shared memory
  – Message passing
• Vectorization in GCC
• Introduction to OpenMP
• Status and Conclusions
Parallel Computing

• Use hardware concurrency for increased
  – Performance
  – Problem size

• Two main models
  – Shared memory
  – Distributed memory

• Nature of problem dictates
  – Computation/communication ratio
  – Hardware requirements
Shared Memory

- Processors share common memory
- Implicit communication
- Explicit synchronization
- Simple to program but hidden side-effects
Distributed Memory

- Each processor has its own private memory
- Explicit communication
- Explicit synchronization
- Difficult to program but no/few hidden side-effects
Programming Models

• Shared/Distributed memory often combined
  – Networks of multi-core nodes
  – Parallelism available at various levels

• Additional requirements over sequential
  – Task creation
  – Communication
  – Synchronization

• How do we program these systems?
Automatic Parallelization

- Holy grail for a long time
- Limited success
- Hampered by need to preserve sequential semantics
- Useful in certain application domains
  - Loop intensive codes
  - No “complex” data dependencies across iterations
- Vectorization, instruction-level parallelism (ILP), loop parallelism
Explicit Parallelism

• User controls: Tasks, communication and synchronization

• Increased programming complexity
  – Often require different algorithms

• Many different approaches
  – Parallel languages or language extensions: HPF, Occam, Java
  – Compiler annotations: OpenMP
  – Libraries: Pthreads, MPI
Parallelism in GCC

GCC supports four concurrency models

- **Easy**
  - ILP
    - automatic
    - no user control
    - not intrusive
  - Vectorization
    - automatic
    - compiler option
    - not intrusive
  - OpenMP
    - manual
    - compiler directives
    - somewhat intrusive
  - MPI
    - manual
    - special libraries
    - very intrusive

Ease of use not necessarily related to speedups!
Vectorization

- Perform multiple array computations at once
- Two distinct phases
  - Analysis → high-level
  - Transformation → low-level
- Successful analysis depends on
  - Data dependency analysis
  - Alias analysis
  - Pattern matching
- Suitable only on loop intensive code
Vectorization

• Enable vectorizer
  
  $ gcc -ftree-vectorize -O2 prog.c

• Additional \texttt{-m} flags on some architectures
  
  − PowerPC → \texttt{-maltivec}
  − x86 → \texttt{-msse2}

• Speedups depend greatly on
  
  − Regular, compute-intensive loops
  − Data size and alignment
  − “Simple” code patterns in inner loops
  − Aliasing
Vectorization

• Debugging
  \- `fdump-tree-vect` enables dump
  \- `ftree-vectorizer-verbose=[0-7]` controls verbosity

• Features and limitations
  - Multi-platform vectorization: x86, ppc, ia64, etc
  - Recognized patterns grow with each release
  - Only works on loops (straight-line code in progress)
Vectorization

\[
\begin{align*}
\text{int } &\text{ a}[256], b[256], c[256]; \\
\text{foo } () &
\begin{cases}
\text{ for (i = 0; i < 256; i++)} \\
\quad a[i] = b[i] + c[i];
\end{cases}
\end{align*}
\]

Vectorized

\[
\begin{align*}
.L2: & \\
& \text{movdqa } c(\%eax), \%xmm0 \\
& \text{paddd } b(\%eax), \%xmm0 \\
& \text{movdqa } \%xmm0, a(\%eax) \\
& \text{addl } 16, \%eax \\
& \text{cmpl } 1024, \%eax \\
& \text{jne } .L2
\end{align*}
\]

Scalar

\[
\begin{align*}
.L2: & \\
& \text{movl } c(,\%edx,4), \%eax \\
& \text{addl } b(,\%edx,4), \%eax \\
& \text{movl } \%eax, a(,\%edx,4) \\
& \text{addl } 1, \%edx \\
& \text{cmpl } 256, \%edx \\
& \text{jne } .L2
\end{align*}
\]

\(~2x\text{ on P4)\)
OpenMP - Introduction

- Language extensions for shared memory concurrency
- Supports C, C++ and Fortran
- Embedded directives specify
  - Parallelism
  - Data sharing semantics
  - Work sharing semantics
- Standard and increasingly popular
OpenMP – Programming Model

- Based on fork/join semantics
  - Master thread spawns teams of children threads
  - All threads share common memory
- Allows sequential and parallel execution

![Diagram showing fork/join semantics with master thread and parallel region]
OpenMP - Programming Model

- Compiler directives via pragmas (C, C++) or comments (Fortran).
- Compiler replaces directives with calls to runtime library (libgomp)
- Runtime controls available via library API and environment variables
- Environment variables control parallelism

OMP_NUM_THREADS
OMP_SCHEDULE
OMP_DYNAMIC
OMP_NESTED
OpenMP – Programming Model

• Explicit sharing and synchronization
• Threads interact via shared variables
  − Several ways for specifying shared data
  − Sharing always at the variable level
• Programmer responsible for synchronization
  − Unintended sharing leads to “data races”
  − Use synchronization directives and library API
  − Synchronization is expensive
OpenMP - Hello World

```c
#include <omp.h>

main()
{
    #pragma omp parallel
    printf ("[%d] Hello\n", omp_get_thread_num());
}
```

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

```
$ gcc -o hello hello.c
$ ./hello
[0] Hello
```

← Master thread
OpenMP – Directives and Clauses

- **Directives** are the main OpenMP construct
- **Clauses** provide modifiers and attributes to the directives
- General syntax is
  - C/C++
    
    ```
    #pragma omp directive [ clause [ clause ] ... ]
    ```
  - Fortran
    
    ```
    !$omp directive [ clause [ clause ] ... ]
    *$omp directive [ clause [ clause ] ... ]
    ```
OpenMP – Directives and Clauses

- Directives are enabled with `-fopenmp`
- Most directives only apply to structured blocks
  - No early exits except program termination
- Directives control
  - Thread creation
  - Work sharing
  - Synchronization
- Clauses control data sharing
OpenMP – Thread creation

• Exactly **one** way to specify parallelism

```
#pragma omp parallel [ clauses ]
structured-block
```

• Every thread executes the block

• Number of threads created depends on
  - Environment variable **OMP_NUM_THREADS**
  - Clauses **num_threads** and **if**
  - Library function **omp_set_num_threads**
OpenMP – Thread creation

- Number of threads involved may be dynamic
  - Environment variable `OMP_DYNAMIC`
  - Library function `omp_set_dynamic`
- No implicit synchronization between threads
- At end of parallel region all children threads disappear
- Every thread has a unique ID starting at 0
  - Useful for distributing work (*work sharing*)
OpenMP – Work Sharing

- Different threads should work on different parts of a problem
- Distribution can be specified manually using thread IDs
- Directives for common work sharing patterns
  - Data parallel loops
    ```
    #pragma omp for [ clauses ]
    ```
  - cobegin/coend
    ```
    #pragma omp sections [ clauses ]
    ```
OpenMP – Parallel loops

• Most common work sharing mechanism
• Threads execute subset of iteration space

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

• Scheduling determines distribution of chunks
• No synchronization other than implicit barrier at the end of the loop
OpenMP – Parallel loops

• `#pragma omp for schedule(type[, chunk])`

• Schedule type is
  - `static` Static round-robin distribution
  - `dynamic` First-come, first-serve queue
  - `guided` Same as dynamic but varying chunk size proportional to outstanding iterations
  - `runtime` Taken from environment `OMP_SCHEDULE`.

• Dynamic and guided schedules may achieve better load balancing

• Runtime useful to avoid re-compiling.
OpenMP – Parallel sections

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

```c
#pragma omp parallel sections
{
    #pragma omp section
    t1();
    #pragma omp section
    t2();
    #pragma omp section
    t3();
}
```
OpenMP – Fortran arrays

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

```fortran
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
```
OpenMP – Data sharing

• Sharing specified at variable level
• `#pragma omp [ ... ] shared (x,y)`
  - All threads access the same variable
• `#pragma omp [ ... ] private (x,y)`
  - All threads have their own copy
• `#pragma omp [ ... ] firstprivate (x,y)`
  - Private with initial value taken from master thread
OpenMP – Data sharing

- `#pragma omp [ ... ] lastprivate (x,y)`
  - Private with last value taken from last iteration or lexically last section
  - Only valid for parallel loops and sections
- `#pragma omp [ ... ] reduction (op:x)`
  - Apply reduction operator `op` to private copy of `x` and update original at the end
  - C/C++ → `+ * - & | ^ && ||`
  - Fortran → `+ * - .and. .or. .eqv. .neqv. max min iand ior ieor`
OpenMP – Data sharing

- `#pragma omp single copyprivate (x)`
  - Broadcast private `x` to all the threads that did not enter the region
- `#pragma omp threadprivate (x, y)`
  - Global variables `x` and `y` are private to each thread
- `#pragma omp [...] copyin(x, y)`
  - Initialize threadprivate variables with the value from the master thread.
**OpenMP – Data sharing**

- Various rules to determine default/implicit sharing properties
  - Globals and heap allocated variables are shared
  - Locals declared outside a directive body are shared
  - Locals declared inside a directive body are private
  - Loop iteration variables for parallel loops are private
OpenMP – 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
OpenMP – Synchronziation

- `#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.
Status and Future Work

• Vectorization support started in 4.0 series
  – New patterns added with every release
  – Use on loop-intensive code
• OpenMP will be released with 4.2 later this year
• Implementation available in Fedora Core 5
• Automatic parallelism planned using OpenMP infrastructure
Status and Future Work

SPEC OMP2001 (-O2)

SPEC OMP2001 scores on dual-core EM64T

May 31, 2006
**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
Message Passing

- 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
Conclusions

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