SAARLAND UNIVERSITY

COMPUTER SCIENCE

Advanced C Programming

Winter Term 2008/09

Guest Lecture by Markus Thiele <thiele@st.cs.uni-sb.de>

Lecture 14:

Parallel Programming with OpenMP

Motivation: Why parallelize?

“The free lunch is over.”

– Herb Sutter (2005)

- Tact frequencies are reaching their limit
- Multi-core systems are becoming more and more common
- Automatic parallelization is still in its infancy

Options for Parallelization

- Platform specific APIs
- POSIX Threads
- …

- Generally provide:
  - Forking into threads (often functions as threads)
  - Joining threads
  - Synchronization with Mutexes, Semaphores, etc.

Our focus today: OpenMP

- Cross-platform API for C/C++ and Fortran
- In development since about 1997
- Open standard
- Widely supported (in recent compiler versions)
  - GNU (GCC 4.2 and higher)
  - Microsoft (Visual C++ 2005 and newer)
  - Intel, Sun, IBM, etc.
- High-level API

http://www.OpenMP.org/

Experience Report: OpenMP

- Parallelizing libsvm

```
…

#pragma omp parallel for private(j)
for(j=start;j<len;j++)
   data[j] = (Qfloat)(y[i]*y[j]*
              (this->*kernel_function)(i,j));

…

#pragma omp parallel for private(i)
for(i=0;i<l;i++)
   kvalue[i] =
      Kernel::k_function(x,model->SV[i],model->param);
…
```

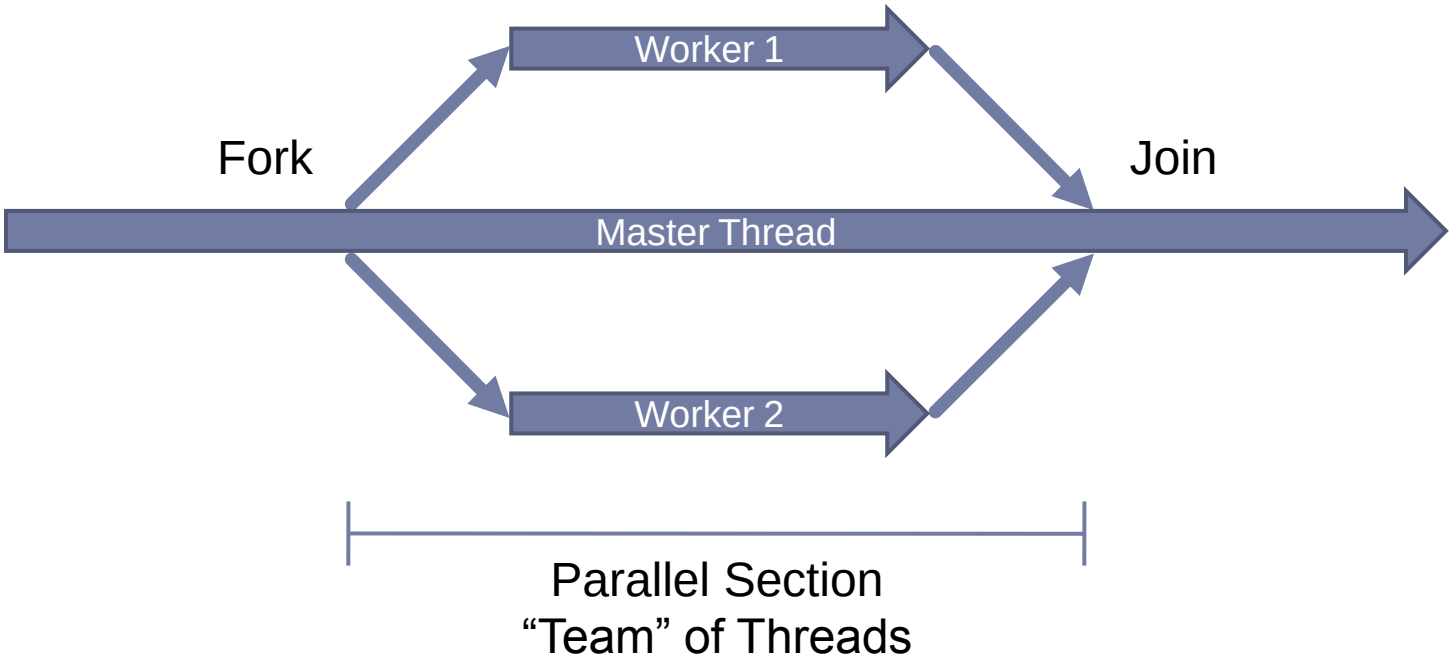
- Effect: **2 to 5 times faster on 16 cores**

Integration

- Compiler Feature
  `gcc -fopenmp …`
- Directives
  `#pragma omp …`
- Library
  `#include <omp.h>`
- Conditional Compilation
  `_OPENMP` macro
- Environment Variables
  e.g. `OMP_NUM_THREADS`

Computation Model

- Fork-and-Join model



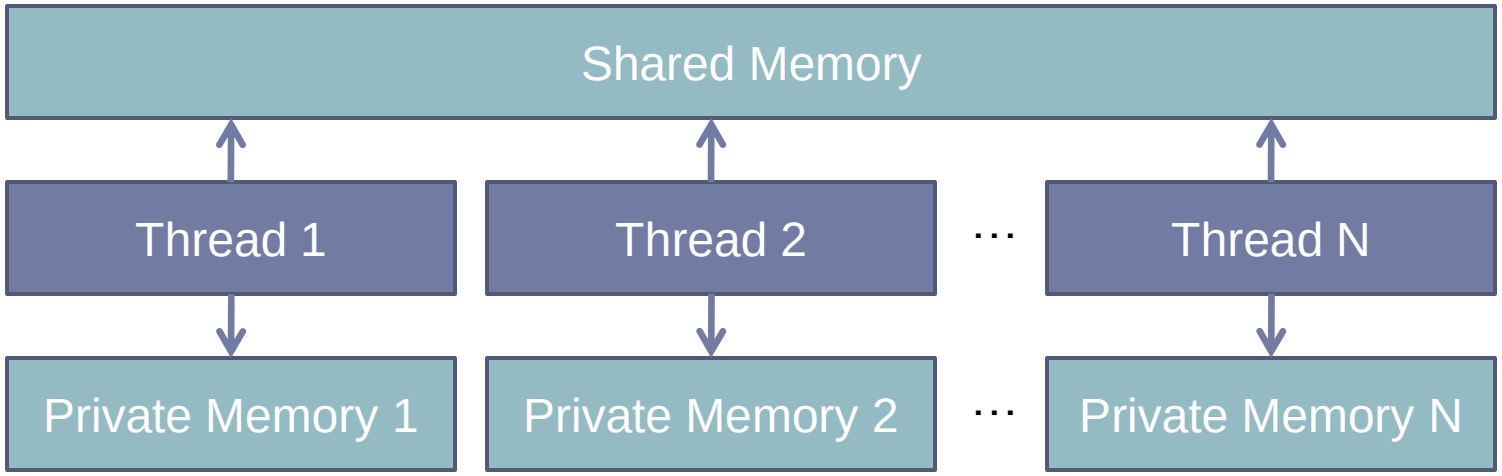
Team Size

- How many threads are created?
  - By default as many as there are cores

- Can be overridden by
  - `OMP_NUM_THREADS` environment variable
  - `omp_set_num_threads()` library function
  - `num_threads` clause in a specific directive

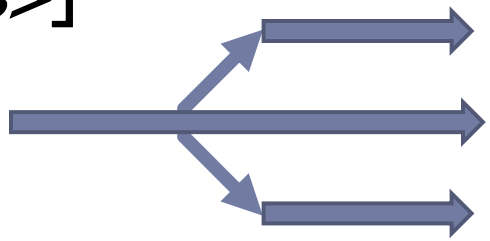
Memory Model

- Shared memory with thread local storage



Forking

```
#pragma omp parallel [<clauses>]
{
      …
}
```



- Creates a team of threads, executing the following code block (i.e. the exact same code)
- Clauses change behavior
- Block may contain synchronization directives
- Block may contain work sharing constructs to distribute work over threads in a specific way
- Implicit join at the end of the block

Clauses: Parallelism

- Normally, the default number of threads (or the number specified by the environment variable `OMP_NUM_THREADS`) is created and run in parallel

- Clauses may change this:
  - **`num_threads(`*`<integer expression>`*`)`**
    The number of threads created will correspond to the number the given expression evaluates to.
  - **`if(`*`<boolean expression>`*`)`**
    If the given expression evaluates to false, the block will be executed sequentially in the master thread only

Clauses: Storage Association

- By default, all variables are shared among threads

- Clauses may change this:
  - **`private(`*`<list of variables>`*`)`**
    Each thread will operate on a private version of the listed variables (note that the value of the variable is undefined on entry and exit)
  - **`Shared(`*`<list of variables>`*`)`**
    All threads will explicitly (this is the default) operate on the same original version of the variable
  - **`default({shared|none})`**
    Changes the default storage association; If `none` is specified, all used variables must explicitly be declared `private` or `shared`.

Clauses: Private Variables

- Normally, private versions of variables are uninitialized and the value of the original variable at the end of the block is undefined

- Variations of the `private` clause may change this:
  - **`firstprivate(`*`<list of variables>`*`)`**
    Listed variables are private and are initialized with the value of the original object before entry
  - **`lastprivate(`*`<list of variables>`*`)`**
    Listed variables are private and at the end of the block, the original object will receive the value of the private version from the sequentially last operation

Clauses: Shared Variables

- Normally, shared variables are subject to race conditions

- The `reduction` clause avoids race conditions for certain computations:
  - **`reduction(`*`<operator>`*`: `*`<list of variables>`*`)`**
    If the listed variables are only updated with allowed operations, code is generated to avoid race conditions
  - Allowed operations (for variable x):
    `x = x `*`<operator>`*` `*`<expression>`*
    `x = `*`<expression>`*` `*`<operator>`*` x`
    `x++, ++x, x--, --x`
    `x `*`<operator>`*`= `*`<expression>`*
  - The value of x is undefined until the end of the block

Synchronization Directives

- **`#pragma omp barrier`**
  Threads will sleep when reaching the barrier until all other threads have reached the barrier

- **`#pragma omp critical`**
  `{ … }`
  The critical section will be executed by all threads, but only by one thread at a time

- **`#pragma omp atomic`**
  `<expression statement>`
  Light weight alternative to make a single memory update atomic (executed without interruption by another thread)

Synchronization Directives

- **`#pragma omp flush [`*`<list of variables>`*`]`**
  Make sure all threads have a consistent view of the listed variables (or all shared variables, if none are listed)

- **`#pragma omp ordered`**
  `{ … }`
  Preserve apparent sequential execution order inside the given block (this is very inefficient, as it does not allow much actual parallelism)

Work Sharing: Single Execution

- **`#pragma omp single [`*`<clauses>`*`]`**
  `{ … }`
  The given block is only executed once by a single thread (there is no implicit barrier, so other threads will move on past the block as soon as they're ready)

- **`#pragma omp master [`*`<clauses>`*`]`**
  `{ … }`
  The given block is only executed once by the master thread (again with no implicit barrier)

- These directives support the Storage Association clauses as described before

Work Sharing: Sections

```
#pragma omp sections [<clauses>]
{
    …
    #pragma omp section
    …
    #pragma omp section
    …

}
```

- Each section (separated by the section directive) is executed exactly once by one thread (left-over threads wait)
- There is an implicit barrier at the end of the sections block, which may be lifted with the **`nowait`** clause

Work Sharing: Loops

- **#pragma omp for [<*clauses*>]**
  <*for loop*>
- **#pragma omp do [<*clauses*>]**
  <*do loop*>

- The loop iterations will be executed in parallel by all available threads
- There is an implicit barrier at the end of the sections block, which may be lifted with the **`nowait`** clause

Work Sharing: Shorter Notation

- `#pragma omp parallel for [<clauses>]`
  `    <for loop>`

  … is equivalent to …

- ```
  #pragma omp parallel [<clauses>]
  {
      #pragma omp for [<clauses>]
          <for loop>
  }
  ```

- The same shorter notation may be applied to do and sections blocks


  ```

Load Balancing

- The schedule clause affects the way loop iterations are assigned to threads in a team
  - **schedule({static|dynamic|guided}[,<*chunk size*>])**
    schedule(runtime)

static	Each thread is statically assigned a chunk of iterations (if no chunk size is specified, the iteration space is divided approximately equally) round-robin
dynamic	Chunks of iterations are assigned to threads that are waiting for work and every thread waits for a new chunk when its work is done
guided	Behaves like dynamic, but the chunk size starts out at an approximately even distribution and then exponentially decreases down to the specified chunk size (or 1).
runtime	The schedule to use is read from the OMP_SCHEDULE environment variable

OpenMP Runtime Library

- Provides information about the current thread and the current team of threads
  - `omp_get_thread_num()` – returns the current thread ID (the master thread is always 0)
  - `omp_get_num_threads()` – returns the current number of threads to be used by a team
  - `omp_get_num_procs()` – returns the maximum number of available processors
  - `omp_in_parallel()` – returns true if currently in parallel region, false otherwise
  - `omp_get_dynamic()` – returns true if dynamic thread adjustment is enabled
  - `omp_get_wtime()` – returns the “wall clock” time
  - `omp_get_wtick()` – returns the number of seconds between clock ticks

OpenMP Runtime Library

- Allows to change some settings
  - `omp_set_num_threads()` – sets the number of threads to be used by a team
  - `omp_set_dynamic()` – enable or disable dynamic thread adjustment (this can also be done with the environment variable OMP_DYNAMIC)

- Provides traditional locking mechanisms
  - `omp_init_lock()`, `omp_destroy_lock()`, `omp_set_lock()`, `omp_unset_lock()`, `omp_test_lock()`

More Information

- For much more information about OpenMP, visit the OpenMP website at…

http://www.OpenMP.org/

Conclusions

- Advantages
  - Simple and intuitive
  - High level (hides many ugly details)
  - Can be incrementally applied to existing code
  - Can easily be enabled and disabled

- Disadvantages
  - Requires compiler support (recent compiler versions)
  - Limited to a certain memory architecture
  - Limited fine-grained control
  - Limited error handling

Lecture 14:

Parallel Programming with OpenMP

Thank you for your attention!

Questions? Comments? Suggestions?