Implementation of Parallelization

SIMD/SIMT & Multithreading: Numba (with CUDA), OpenMP, Python Threading & PThreads

Jascha Schewtschenko

Royal Observatory of Edinburgh, University of Edinburgh

May 15, 2025

Outline

OpenMP®

(SIMD & Threads)

OpenMP - Goals

Standardization provide standard for vatiety of platforms/shared-mem architectures

OpenMP - Goals

Standardization provide standard for vatiety of platforms/shared-mem architectures

Lean and Mean simple and limited set of directives, very few uses of directives needed

OpenMP - Goals

Standardization provide standard for vatiety of platforms/shared-mem architectures

Lean and Mean simple and limited set of directives, very few uses of directives needed

Ease of Use can incrementally parallelize program (source stays the same except for added directives), supports both coarse-grain and fine-grain parallelism

OpenMP - Goals

Standardization provide standard for vatiety of platforms/shared-mem architectures

Lean and Mean simple and limited set of directives, very few uses of directives needed

Ease of Use can incrementally parallelize program (source stays the same except for added directives), supports both coarse-grain and fine-grain parallelism

Portability public API, implementations for C, C++, Fortran

OpenMP - Structure & Implementations

OpenMP - Structure & Implementations

- Supported/shipped with various compilers for various platforms (e.g. Intel and GNU compilers for Linux), i.e. to compile simply add option:

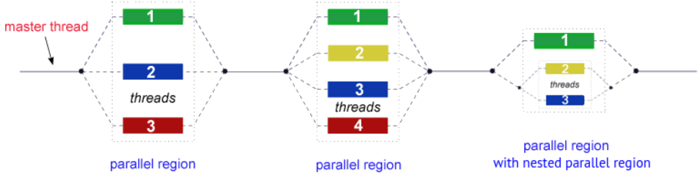
  e.g. `gcc -fopenmp`

OpenMP - Structure & Implementations

- Supported/shipped with various compilers for various platforms (e.g. Intel and GNU compilers for Linux), i.e. to compile simply add option:

  e.g. `gcc -fopenmp`

- uses a form-join model

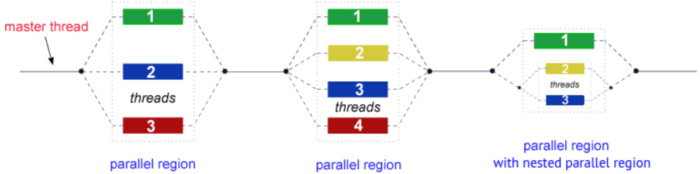


OpenMP - Structure & Implementations

- Supported/shipped with various compilers for various platforms (e.g. Intel and GNU compilers for Linux), i.e. to compile simply add option:

  e.g. `gcc -fopenmp`

- uses a form-join model



- comprised of 3 API components:
  - Compiler Directives
  - Runtime Library routines
  - Environment Variables

OpenMP - Compiler Directives

OpenMP - Compiler Directives

We will focus here on C/C++ syntax, FORTRAN syntax slightly different:

```
#pragma omp [simd] <directive name> [<clauses>]      (C/C++)
!$OMP [SIMD] [END] <directive name> [<clauses>]      (Fortran)
```

OpenMP - Compiler Directives

We will focus here on C/C++ syntax, FORTRAN syntax slightly different:

```
#pragma omp [simd] <directive name> [<clauses>]   (C/C++)
!$OMP [SIMD] [END] <directive name> [<clauses>]   (Fortran)
```

Used for:

- Defining parallel regions: requesting vectorization or spawning threads
- Specifying strip-mining / inter-thread distribution strategy of loop iterations or sections of code
- Serializing sections of code (e.g. for access to I/O or shared variables)
- Synchronizing threads

You can find a reference sheet for the C/C++ API for OpenMP 4.0 in the source code archive for this workshop.

OpenMP/SIMD: Example

```
void test(const float* A, const float* B, float* C) {
   #pragma omp simd
   for (int j = 0;j < N; j++) {
     for (int i = 0;i < N; i++) {
       C[i] = A[i] + B[i];
     }
   }
}
```

OpenMP/SIMD: Example

```
void test(const float* A, const float* B, float* C) {
   #pragma omp simd
   for (int j = 0;j < N; j++) {
     for (int i = 0;i < N; i++) {
       C[i] = A[i] + B[i];
     }
   }
}
```

```
icc vectorize.c -qopenmp-simd -no-vec -qopt-report=5
```

```
LOOP BEGIN at test-vec.c(25,4)
   remark #15540: loop was not vectorized: auto-vectorization is disabled with
-no-vec flag
LOOP END

LOOP BEGIN at test-vec.c(12,15) inlined into test-vec.c(29,4)
   remark #15305: vectorization support: vector length 4
   remark #15309: vectorization support: normalized vectorization overhead 0.500
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 8
   remark #15477: vector cost: 1.000
   remark #15478: estimated potential speedup: 7.990
   remark #15488: --- end vector cost summary ---
   remark #25015: Estimate of max trip count of loop=25000

   LOOP BEGIN at test-vec.c(13,6) inlined into test-vec.c(29,4)
      remark #25456: Number of Array Refs Scalar Replaced In Loop: 1
   LOOP END
LOOP END
```

OpenMP/Threads - Runtime Library Routines

OpenMP/Threads - Runtime Library Routines

- These routines are provided by the openmp library are used to configuring and monitoring the multithreading during execution: e.g.
  `omp_get_num_threads` returns number of threads in current team
  `omp_in_parallel` check if in parallel regions
  `omp_set_schedule` modify scheduler policy

OpenMP/Threads - Runtime Library Routines

- These routines are provided by the openmp library are used to configuring and monitoring the multithreading during execution: e.g.

  `omp_get_num_threads` returns number of threads in current team
  `omp_in_parallel` check if in parallel regions
  `omp_set_schedule` modify scheduler policy
- There are further routines for locks for synchronization/access control (see later)

OpenMP/Threads - Runtime Library Routines

- These routines are provided by the openmp library are used to configuring and monitoring the multithreading during execution: e.g.
  `omp_get_num_threads` returns number of threads in current team
  `omp_in_parallel` check if in parallel regions
  `omp_set_schedule` modify scheduler policy
- There are further routines for locks for synchronization/access control (see later)
- as well as timing routines for recording elapsed time for each thread.

OpenMP/Threads - Environment variables

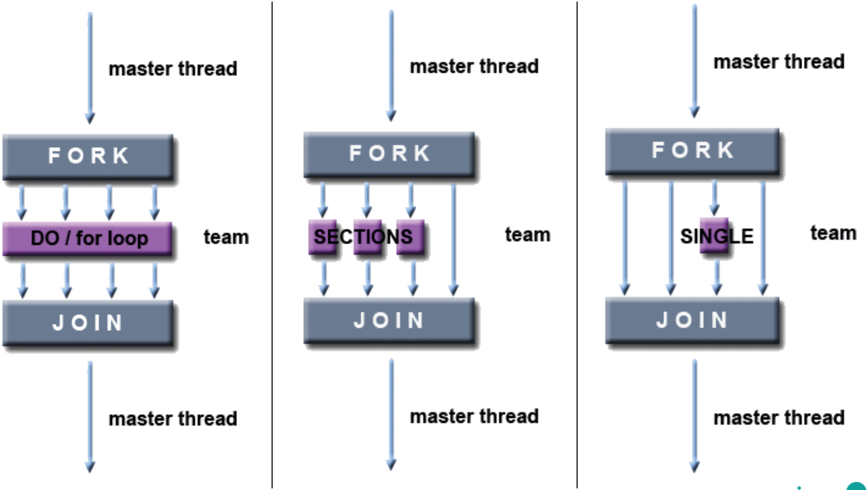
OpenMP/Threads - Environment variables

- Like for most programs in the UNIX world, environmental variables are used to store configurations needed for running the program. In OpenMP, they are used for setting e.g. the number of threads per team (`OMP_NUM_THREADS`), maximum number of threads (`OMP_THREAD_LIMIT`) or the scheduler policy (`OMP_SCHEDULE`).

OpenMP/Threads - Environment variables

- Like for most programs in the UNIX world, environmental variables are used to store configurations needed for running the program. In OpenMP, they are used for setting e.g. the number of threads per team (`OMP_NUM_THREADS`), maximum number of threads (`OMP_THREAD_LIMIT`) or the scheduler policy (`OMP_SCHEDULE`).
- While most of these settings can also be done using clauses in the compiler directives of runtime library routines, environmental variables provide a user an easy way to change these crucial settings without the need of an additional config file (parsed by your program) or even rewritting/recompiling the openmp-enhanced program.

OpenMP/Threads - Worksharing



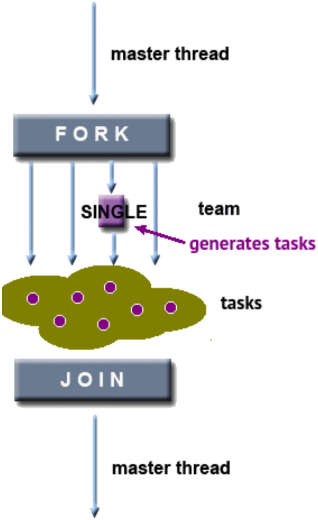
OpenMP/Threads - Worksharing: Examples

```c
#include <omp.h>
#define N 1000
#define CHUNKSIZE 100

main(int argc, char *argv[]) {

int i, chunk;
float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;

#pragma omp parallel shared(a,b,c,chunk) private(i)
  {

  #pragma omp for schedule(dynamic,chunk) nowait
  for (i=0; i < N; i++)
    c[i] = a[i] + b[i];

  }   /* end of parallel region */

}
```

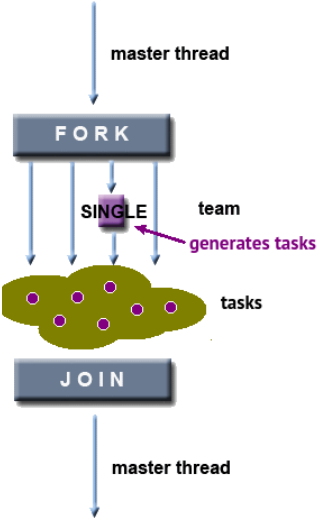
```c
#include <omp.h>
#define N 1000

main(int argc, char *argv[]) {

int i;
float a[N], b[N], c[N], d[N];

/* Some initializations */
for (i=0; i < N; i++) {
  a[i] = i * 1.5;
  b[i] = i + 22.35;
  }

#pragma omp parallel shared(a,b,c,d) private(i)
  {

  #pragma omp sections nowait
    {

    #pragma omp section
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];

    #pragma omp section
    for (i=0; i < N; i++)
      d[i] = a[i] * b[i];

    }  /* end of sections */

  }  /* end of parallel region */

}
```

```c
#include <omp.h>
#define N       1000
#define CHUNKSIZE   100

main(int argc, char *argv[]) {

int i, chunk;
float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;

#pragma omp parallel for \
  shared(a,b,c,chunk) private(i) \
  schedule(static,chunk)
  for (i=0; i < n; i++)
    c[i] = a[i] + b[i];
}
```

OpenMP/Threads - advanced Worksharing

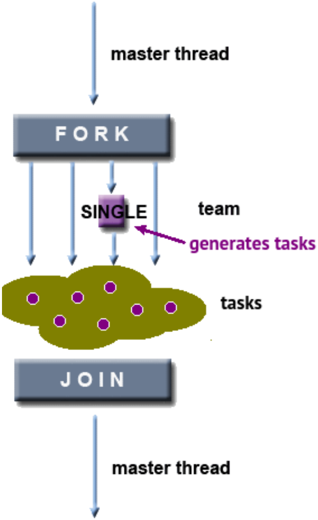


OpenMP/Threads - advanced Worksharing



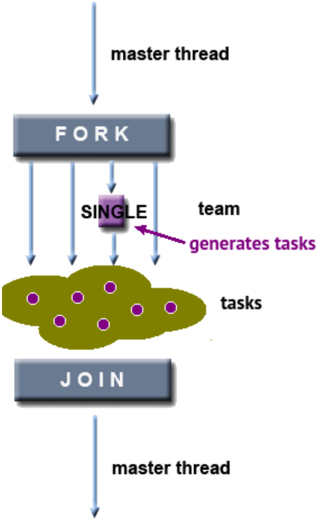
- defines explicit tasks similar to sections that are generated (usually by a single task) and then deferred to any thread in the team via a queue/scheduler

OpenMP/Threads - advanced Worksharing



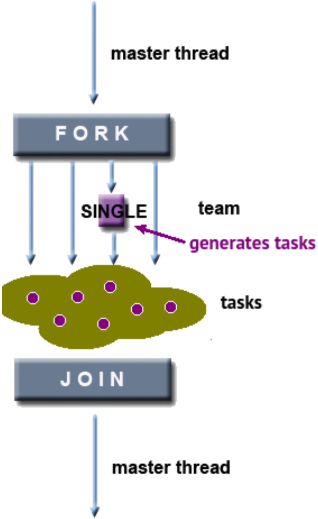
- defines explicit tasks similar to sections that are generated (usually by a single task) and then deferred to any thread in the team via a queue/scheduler
- tasks are not necessarily tied to a single thread, can be e.g. postponed or migrated to other threads

OpenMP/Threads - advanced Worksharing



- defines explicit tasks similar to sections that are generated (usually by a single task) and then deferred to any thread in the team via a queue/scheduler
- tasks are not necessarily tied to a single thread, can be e.g. postponed or migrated to other threads
- allows for defining dependencies among tasks (e.g. task X has to finish before any thread can work on task Y)

OpenMP/Threads - advanced Worksharing: Example




```c
#include <omp.h>

float sum(const float *a, size_t n)
{
    // base cases
    if (n == 0) {
        return 0;
    }
    else if (n == 1) {
        return 1;
    }

    // recursive case
    size_t half = n / 2;
    float x, y;

    #pragma omp parallel shared(x,y)

    #pragma omp single nowait
    {
        #pragma omp task shared(x)
        x = sum(a, half);
        #pragma omp task shared(y)
        y = sum(a + half, n - half);
        #pragma omp taskwait
        x += y;
    }
    return x;
}
```

OpenMP/Threads - Synchronization / Flow control

In the 'Introduction to Parallelization', we discussed the need of controlling the execution of threads at certain points to e.g. synchronize them to exchange intermediate results or to protect resources from getting accessed simultaneously with non-deterministic outcome ('race condition'). OpenMP provides two ways to do this:

OpenMP/Threads - Synchronization / Flow control

In the 'Introduction to Parallelization', we discussed the need of controlling the execution of threads at certain points to e.g. synchronize them to exchange intermediate results or to protect resources from getting accessed simultaneously with non-deterministic outcome ('race condition'). OpenMP provides two ways to do this:

- Compiler Directives:
  - (for general parallel regions) e.g. `cancel`,`single`,`master`,`critical`,`atomic`, `barrier`
  - (for loops) `ordered`
  - (for tasks) `taskwait`, `taskyield`

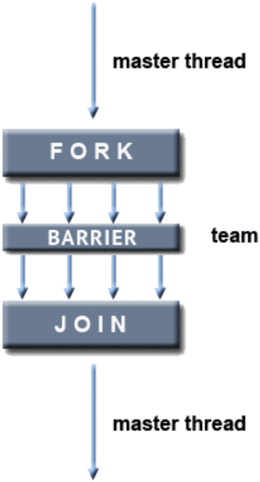
OpenMP/Threads - Synchronization / Flow control

In the 'Introduction to Parallelization', we discussed the need of controlling the execution of threads at certain points to e.g. synchronize them to exchange intermediate results or to protect resources from getting accessed simultaneously with non-deterministic outcome ('race condition'). OpenMP provides two ways to do this:

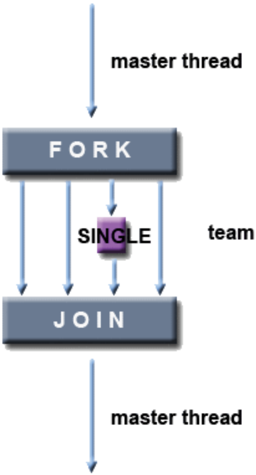
- Compiler Directives:
  - (for general parallel regions) e.g. `cancel`,`single`,`master`,`critical`,`atomic`, `barrier`
  - (for loops) `ordered`
  - (for tasks) `taskwait`, `taskyield`
- Runtime Library Routines: `omp_set_lock`,`omp_unset_lock`,`omp_test_lock`

OpenMP/Threads - Synchronization / Flow control (RESTRICTION)

BARRIER

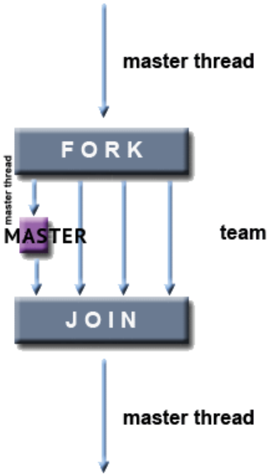


SINGLE



implicit BARRIER

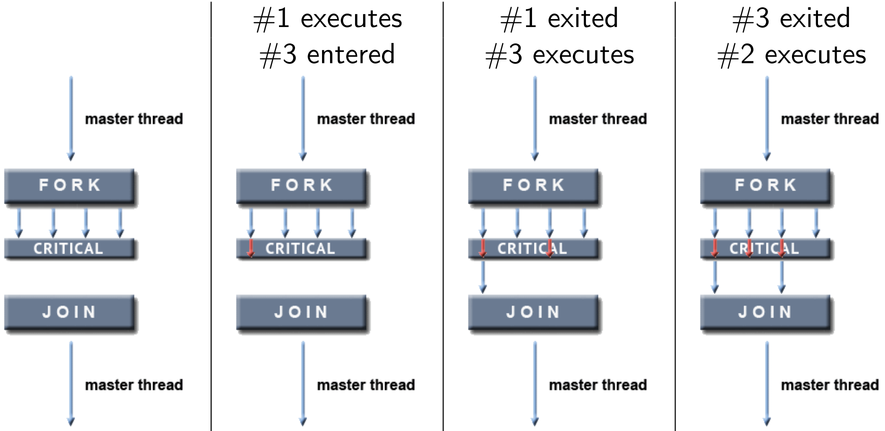
MASTER



NO implicit BARRIER

OpenMP/Threads - Synchronization / Flow control (MUTEX)

`CRITICAL` / `ATOMIC`



- `CRITICAL`,`ATOMIC` exclusive for ALL threads, not just team
- `CRITICAL` regions can be named, regions with same name treated as same region

OpenMP/Threads - Memory management (CLAUSES)

- Certain clauses for compiler directives allow us to specify how data is shared (e.g. `shared`, `private`, `threadprivate`) and how they are initialized (e.g.`firstprivate`, `copyin`)

OpenMP/Threads - Memory management (CLAUSES)

- Certain clauses for compiler directives allow us to specify how data is shared (e.g. `shared`, `private`, `threadprivate`) and how they are initialized (e.g.`firstprivate`, `copyin`)
- Others like `copyprivate` allow for broadcasting the content of private variables from one thread to all others

OpenMP/Threads - Memory management (CLAUSES)

- Certain clauses for compiler directives allow us to specify how data is shared (e.g. `shared`, `private`, `threadprivate`) and how they are initialized (e.g.`firstprivate`, `copyin`)
- Others like `copyprivate` allow for broadcasting the content of private variables from one thread to all others
- Similarly, the `reduction` clause provides an elegant way to gather private data from the threads when joining them

OpenMP/Threads - Memory management (CLAUSES)

- Similarly, the `reduction` clause provides an elegant way to gather private data from the threads when joining them

```
#include <omp.h>

 main(int argc, char *argv[])  {

 int   i, n, chunk;
 int a[100], b[100], result;

 n = 100;
 chunk = 10;
 result = 0.0;
 for (i=0; i < n; i++) {
   a[i] = i;
   b[i] = i*2;
   }

 #pragma omp parallel for        \
   default(shared) private(i)   \
   schedule(static,chunk)       \
   reduction(+:result)

   for (i=0; i < n; i++)
     result = result + (a[i] * b[i]);

 printf("result= %d\n",result);
}
```

OpenMP/Threads - Memory management (FLUSHING DATA)

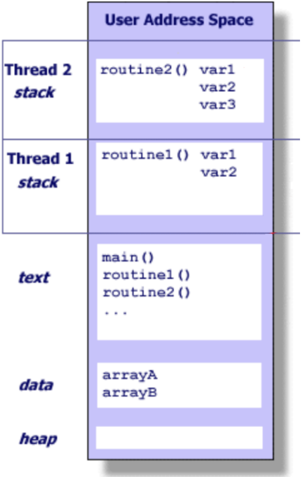
OpenMP/Threads - Memory management (FLUSHING DATA)

- even if shared, sometimes variable may not be updated in the "global" view, e.g. if kept in a register or cache of a CPU instead of the shared memory

OpenMP/Threads - Memory management (FLUSHING DATA)

- even if shared, sometimes variable may not be updated in the "global" view, e.g. if kept in a register or cache of a CPU instead of the shared memory
- while many directives (e.g. `for`, `section`, `critical`) implicitly flush variable to synchronize them with other threads, sometimes explicit flushing using the `flush` may be necessary.

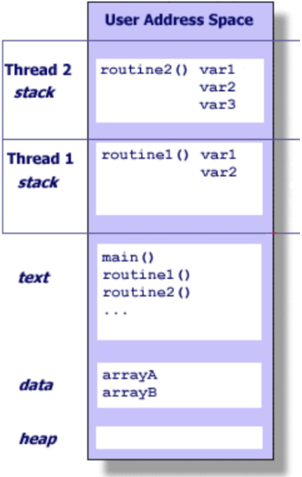
OpenMP/Threads - Memory management (STACK)



OpenMP/Threads - Memory management (STACK)

- OpenMP standard does not specify a default stack size for each thread. So depends on the compiler e.g.

Compiler	Approx Stack Limit
icc/ifort (Linux)	4 MB
gcc/gfort (Linux)	2 MB

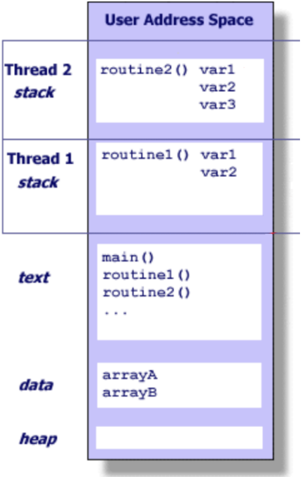


OpenMP/Threads - Memory management (STACK)

- OpenMP standard does not specify a default stack size for each thread. So depends on the compiler e.g.

Compiler	Approx Stack Limit
icc/ifort (Linux)	4 MB
gcc/gfort (Linux)	2 MB

- if stack allocation exceeded, may result in seg fault or (worse) data corruption.

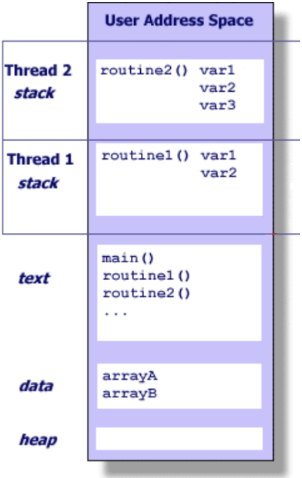


OpenMP/Threads - Memory management (STACK)

- OpenMP standard does not specify a default stack size for each thread. So depends on the compiler e.g.

Compiler	Approx Stack Limit
icc/ifort (Linux)	4 MB
gcc/gfort (Linux)	2 MB

- if stack allocation exceeded, may result in seg fault or (worse) data corruption.
- Env. variable `OMP_STACKSIZE` allows to set stacksize prior to execution. So if your program needs an significant amount of data on the stack, make sure to adapt the stacksize this way!



OpenMP/Threads - Cython wrappers

- Cython is currently using OpenMP as its (only) backend for its parallelisation.
- It provides some wrappers for it that can be used directly in cython's parallelisation module:

```
from cython.parallel import parallel
from cython.cimports.openmp import omp_set_dynamic, omp_get_num_threads

num_threads = cython.declare(cython.int)

omp_set_dynamic(1)
with cython.nogil, parallel():
    num_threads = omp_get_num_threads()
    # ...
```

Python/threading

Python/threading

- First introduced in Python 1.5.2 as an enhancement of the low-level `thread` module (based on PThreads)

Python/threading

- First introduced in Python 1.5.2 as an enhancement of the low-level `thread` module (based on PThreads)
- Goals:

  Simplicity The `threading` module provides high-level multithreading capability to one of the most popular languanges

  Portability *Threading* module available for both Python2 & 3, a various different implementations (`CPython`, `PyPy`, `Jython`, etc.) and for a wide range of platforms/OS

Python/threading

- First introduced in Python 1.5.2 as an enhancement of the low-level `thread` module (based on PThreads)
- Goals:

  Simplicity The `threading` module provides high-level multithreading capability to one of the most popular languanges

  Portability *Threading* module available for both Python2 & 3, a various different implementations (`CPython`, `PyPy`, `Jython`, etc.) and for a wide range of platforms/OS

- There are inherit design choices made for the most popular Python implementations that limit the utility of multithreading (see later)

Python/threading - Thread management: Creation & Term.

- Python threads are created explicitly by creating an instance of the `Thread` class (`threading.Thread(target=...[,args=...][,name=...],...)`)
  - `target` is the name of a function to be invoked by the Thread's `run()` method
  - `arg` is a tuple containing the arguments for the (function) call [default: ()]
  - `name` is a unique name/identifier for the thread [default: Thread-N]

Python/threading - Thread management: Creation & Term.

- Python threads are created explicitly by creating an instance of the `Thread` class (`threading.Thread(target=...[,args=...][,name=...],...)`)
    - `target` is the name of a function to be invoked by the Thread's `run()` method
    - `arg` is a tuple containing the arguments for the (function) call [default: ()]
    - `name` is a unique name/identifier for the thread [default: Thread-*N*]
- They terminate when finishing their starting routine or, if 'daemonic' once the main program finishes with only daemon threads left

Python/threading - Thread management: Creation & Term.

- Python threads are created explicitly by creating an instance of the `Thread` class (`threading.Thread(target=...[,args=...][,name=...],...)`)
  - `target` is the name of a function to be invoked by the Thread's `run()` method
  - `arg` is a tuple containing the arguments for the (function) call [default: ()]
  - `name` is a unique name/identifier for the thread [default: Thread-*N*]
- They terminate when finishing their starting routine or, if 'daemonic' once the main program finishes with only daemon threads left
- Basic threads **cannot** be terminated by other threads (but you can implement a subclass with this property if needed)
- you can explicitly wait for a Thread to finish by invoking its *join()* method.
- there is also a `Timer` class, that allows to set up a delayed thread.

Python/threading - Thread management: Example 1

```
import threading
import time
import random

def do_something():

   time.sleep(random.randrange(1,5))
   print('%s RUNNING\n' % (threading.current_thread().name))

if __name__ == '__main__':

    threads = []

    for i in range(10):
        my_thread = threading.Thread(target=do_something)
        threads.append(my_thread)

    for t in threads:
        t.start()

    for t in threads:
        t.join()

    print('Done')
```

Python/threading: Synchronization / Flow control

- certain structures allow you to control and synchronize the execution of the threads:

  `Barrier` (Python3 only) Similar to OpenMP barrier, but works based on counter rather than waiting for whole team

Python/threading: Synchronization / Flow control

- certain structures allow you to control and synchronize the execution of the threads:

  `Barrier` (Python3 only) Similar to OpenMP barrier, but works based on counter rather than waiting for whole team

  `Lock/RLock` To secure a *critical* region, locks can be used. They can be held by either a single thread or no thread at all. Should a thread try to acquire a lock on a resource that is already locked, that thread will basically pause until the lock is released. RLocks allow same thread to lock multiple times.

Python/threading: Synchronization / Flow control

- certain structures allow you to control and synchronize the execution of the threads:

  `Barrier` (Python3 only) Similar to OpenMP barrier, but works based on counter rather than waiting for whole team

  `Lock/RLock` To secure a *critical* region, locks can be used. They can be held by either a single thread or no thread at all. Should a thread try to acquire a lock on a resource that is already locked, that thread will basically pause until the lock is released. RLocks allow same thread to lock multiple times.

  `Condition` Allows to keep threads waiting until explicitely notified and associated lock released

Python/threading: Synchronization / Flow control

- certain structures allow you to control and synchronize the execution of the threads:

  `Barrier` (Python3 only) Similar to OpenMP barrier, but works based on counter rather than waiting for whole team

  `Lock/RLock` To secure a *critical* region, locks can be used. They can be held by either a single thread or no thread at all. Should a thread try to acquire a lock on a resource that is already locked, that thread will basically pause until the lock is released. RLocks allow same thread to lock multiple times.

  `Condition` Allows to keep threads waiting until explicitely notified and associated lock released

  `Semaphore` Allows to limit access to locked are to a number of threads at a time

Python/threading: Example 2 (Barrier)

```
import threading
import time
import random

num = 4
bar = threading.Barrier(num)

def do_something():

    time.sleep(random.randrange(2, 20))
    print('%s REACHED the barrier\n' % (current_thread().name))
    try:
        bar.wait()
    except:
        pass
    finally:
        print('%s PASSED the barrier\n' % (current_thread().name))

if __name__ == '__main__':

    threads = []

    for i in range(10):
        my_thread = threading.Thread(target=do_something,args=())
        my_thread.start()

    time.sleep(30)
    print("Release stuck threads by breaking barrier…")
    bar.abort()
```

Python/threading: Example 3 (Lock/RLock)

```
import threading

total = 0
lock = threading.Lock()

def update_total(amount):

    #Updates the total

    global total
    lock.acquire()
    try:
        total += amount
    finally:
        lock.release()
    print (total)

if __name__ == '__main__':
    for i in range(10):
        my_thread = threading.Thread(
            target=update_total, args=(5,))
        my_thread.start()
```

Python/threading: Example 3 (Lock/RLock)

```
import threading

total = 0
lock = threading.Lock()

def update_total(amount):

    #Updates the total

    global total
    with lock:
        total += amount
    print (total)

if __name__ == '__main__':
    for i in range(10):
        my_thread = threading.Thread(
            target=update_total, args=(5,))
        my_thread.start()
```

Python/threading: Example 3 (Lock/RLock)

```python
import threading

total = 0
lock = threading.Lock()

def update_total(amount):

    #Updates the total

    global total
    with lock:
        total += amount
    print (total)

if __name__ == '__main__':
    for i in range(10):
        my_thread = threading.Thread(
            target=update_total, args=(5,))
        my_thread.start()
```

```python
import threading

total = 0
lock = threading.RLock()

def do_something(amount):

    with lock:
        update_total(amount)


def update_total(amount):

    #Updates the total

    global total
    with lock:
        total += amount
    print (total)

if __name__ == '__main__':
    for i in range(10):
        my_thread = threading.Thread(
            target=do_something, args=(5,))
        my_thread.start()
```

Python/threading: Example 4 (Condition/Semaphore)

```
import threading
import time

def create(cond):

    while True:
        time.sleep(3)
        print('%s: New item PRODUCED\n' %
                    (threading.current_thread().name))

        with cond:
            cond.notify()
            time.sleep(3)
            print('%s: Item AVAILABLE\n' %
                    (threading.current_thread().name))

def consume(cond):

    while True:
        with cond:
            print('%s: WAITING for new item ...\n' %
                    (threading.current_thread().name))
            cond.wait()
            print('%s: CONSUMING item\n' %
                    (threading.current_thread().name))
        time.sleep(10)

if __name__ == '__main__':

    cond = threading.Condition()

    thread_producer = threading.Thread(target=create, name='producer',
                                       args=(cond,))
    thread_consumer1 = threading.Thread(target=consume, args=(cond,))
    thread_consumer2 = threading.Thread(target=consume, args=(cond,))

    thread_creator.start()
    thread_consumer1.start()
    thread_consumer2.start()
```

Python/threading: Example 4 (Condition/Semaphore)

```
import threading
import time

def create(cond):

    while True:
        time.sleep(3)
        print('%s: New item PRODUCED\n' %
                    (threading.current_thread().name))

        with cond:
            cond.notify()
            time.sleep(3)
            print('%s: Item AVAILABLE\n' %
                    (threading.current_thread().name))

def consume(cond):

    while True:
        with cond:
            print('%s: WAITING for new item ...\n' %
                    (threading.current_thread().name))
            cond.wait()
            print('%s: CONSUMING item\n' %
                    (threading.current_thread().name))
        time.sleep(10)

if __name__ == '__main__':

    cond = threading.Condition()

    thread_producer = threading.Thread(target=create, name='producer',
                                       args=(cond,))
    thread_consumer1 = threading.Thread(target=consume, args=(cond,))
    thread_consumer2 = threading.Thread(target=consume, args=(cond,))

    thread_creator.start()
    thread_consumer1.start()
    thread_consumer2.start()
```

```
import threading
import time

total = 2
sem = threading.Semaphore(total)

def do_something():

    with sem:
        print(threading.currentThread().getName() + '\n')
        time.sleep(5)

if __name__ == '__main__':
    for i in range(10):
        my_thread = threading.Thread(
            target=do_something, args=(5,))
        my_thread.start()
```

Python/threading: Memory management

- since threads share the same process and therefore memory, global variables are automatically shared

Python/threading: Memory management

- since threads share the same process and therefore memory, global variables are automatically shared
- alternatively, there are additional structures to communicate (results) *safely* between threads:

Python/threading: Memory management

- since threads share the same process and therefore memory, global variables are automatically shared
- alternatively, there are additional structures to communicate (results) *safely* between threads:

  `Queue` The `Queue` class provides a FIFO structure that allow e.g. *producer* threads to pass on data to *consumer* threads

Python/threading: Memory management

- since threads share the same process and therefore memory, global variables are automatically shared
- alternatively, there are additional structures to communicate (results) *safely* between threads:

  `Queue` The `Queue` class provides a FIFO structure that allow e.g. *producer* threads to pass on data to *consumer* threads

  `Event` `Event` objects allow are simply binary structures to send signals across threads

Python/threading: Example 4 (Queue/Event)

```
import threading
import queue
import time

def create(data, q):

    for item in data:

        […]

        q.put(item)

def consume(q):

    while True:
        data = q.get()

        […]

        q.task_done()

if __name__ == '__main__':
    q = queue.Queue()
    data = range(10)

    thread_creator = threading.Thread(target=create, args=(data, q))
    thread_consumer = threading.Thread(target=consume, args=(q,))
    thread_consumer.daemon = True

    thread_creator.start()
    thread_consumer.start()

    time.sleep(2)
    q.join()
    print("Done")
```

Python/threading: Example 4 (Queue/Event)

```python
import threading
import queue
import time

def create(data, q):

    for item in data:

        […]

        q.put(item)

def consume(q):

    while True:
        data = q.get()

        […]

        q.task_done()

if __name__ == '__main__':
    q = queue.Queue()
    data = range(10)

    thread_creator = threading.Thread(target=create, args=(data, q))
    thread_consumer = threading.Thread(target=consume, args=(q,))
    thread_consumer.daemon = True

    thread_creator.start()
    thread_consumer.start()

    time.sleep(2)
    q.join()
    print("Done")
```

```python
import threading
import queue

def create(data, q):

    for item in data:

        […]

        q.put(item)

def consume(q, evt):

    while True:
        try:
            data = q.get(timeout=5)
        except:
            evt.set()

        […]

if __name__ == '__main__':
    q = queue.Queue()
    evt = threading.Event()
    data = range(10)

    thread_creator = threading.Thread(target=create, args=(data,q,))
    thread_consumer = threading.Thread(target=consume, args=(q,evt,))
    thread_consumer.daemon = True

    thread_creator.start()
    thread_consumer.start()

    evt.wait()
    print("Done")
```

Python - Multithreading: Caveats

- The reference python implementation aka `CPython` as well as popular alternatives (e.g. `PyPy`) use a *Global Interpreter Lock* (GIL) that blocks running threads simultaneously within python

Python - Multithreading: Caveats

- The reference python implementation aka `CPython` as well as popular alternatives (e.g. `PyPy`) use a *Global Interpreter Lock* (GIL) that blocks running threads simultaneously within python
- Lock-free implementations exist (e.g. `Jython`,`IronPython`), but are not suitable for every problem (some modules are not supported)

Python - Multithreading: Caveats

- The reference python implementation aka `CPython` as well as popular alternatives (e.g. `PyPy`) use a *Global Interpreter Lock* (GIL) that blocks running threads simultaneously within python
- Lock-free implementations exist (e.g. `Jython`,`IronPython`), but are not suitable for every problem (some modules are not supported)
- alternatively, external libraries written in other languanges can circumvent this problem as they release the GIL (see e.g. `NumPy` calls), so do blocking I/O calls

Python - Multithreading: Caveats

- The reference python implementation aka `CPython` as well as popular alternatives (e.g. `PyPy`) use a *Global Interpreter Lock* (GIL) that blocks running threads simultaneously within python
- Lock-free implementations exist (e.g. `Jython`,`IronPython`), but are not suitable for every problem (some modules are not supported)
- alternatively, external libraries written in other languanges can circumvent this problem as they release the GIL (see e.g. `NumPy` calls), so do blocking I/O calls
- *Take-away:* The utility of multithreading in python is very situation-dependent

Python - Multithreading: Caveats (2023 update)

- The reference python implementation aka `CPython` as well as popular alternatives (e.g. `PyPy`) use a *Global Interpreter Lock* (GIL) that blocks running threads simultaneously within python
- Lock-free implementations exist (e.g. `Jython`, `IronPython`), but are not suitable for every problem (some modules are not supported)
- alternatively, external libraries written in other languanges can circumvent this problem as they release the GIL (see e.g. `NumPy` calls), so do blocking I/O calls
- *Take-away:* The utility of multithreading in python is very situation-dependent

CPython 3.13 may finally have full support of (sub)interpreters with own GILs !

POSIX Threads

PThreads - History/Goals

PThreads - History/Goals

- standardized API for multithreading to allow for portable threaded applications

PThreads - History/Goals

- standardized API for multithreading to allow for portable threaded applications
- first defined in IEEE POSIX standard 1003.1c in 1995, but undergoes continuous evolution/revision

PThreads - History/Goals

- standardized API for multithreading to allow for portable threaded applications
- first defined in IEEE POSIX standard 1003.1c in 1995, but undergoes continuous evolution/revision
- historically implementations focused on Unix as OS, but implementations also exist now for others e.g. for Windows

PThreads - Compiling & Running

PThreads - Compiling & Running

- Like for OpenMP, POSIX Threads are included in most recent compiler suites by default

PThreads - Compiling & Running

- Like for OpenMP, POSIX Threads are included in most recent compiler suites by default
- To enable these included libraries, use e.g.

`icc -pthread` for INTEL (Linux)
`gcc -pthread` for GNU (Linux)

PThreads - API

PThreads - API

- The subroutines defined in the API can be classified into four major groups:

  Thread management For creating new threads, checking their properties and joining/destroying them and the end of their lifecycle (`pthread_`,`pthread_attr_`)

  Mutexes For creating mutex locks to control excess to exclusive resources (`pthread_mutex_`,`pthread_mutexattr_`)

  Condition variables routines for managing condition variable to allow for easy communication between threads that share a mutex (`pthread_cond_`,`pthread_condattr_`)

  Semaphores Like in Python, there are semaphores. A semaphore is a signalling mechanism and a thread that is waiting on a semaphore can be signaled by another thread and use a counter to limit access.

  Synchronization barriers, read/write locks (`pthread_barrier_`,`pthread_rwlock_`)

PThreads - Thread management: Creation & Termination

PThreads - Thread management: Creation & Termination

- POSIX threads (`pthread_t`) are created explicitly using the `pthread_create(thread,attr,start_routine,arg)` where
  - `attr` is a thread attribute structure containing settings for creating/running thread
  - `start_routine` is a procedure that works as a starting point for the thread
  - `arg` is a pointer to the argument for the starting routine (can be pointing to a single data element, an array or a custom data structure)

PThreads - Thread management: Creation & Termination

- POSIX threads (`pthread_t`) are created explicitly using the `pthread_create(thread,attr,start_routine,arg)` where
  - `attr` is a thread attribute structure containing settings for creating/running thread
  - `start_routine` is a procedure that works as a starting point for the thread
  - `arg` is a pointer to the argument for the starting routine (can be pointing to a single data element, an array or a custom data structure)
- They terminate when finishing their starting routine, calling `pthread_exit(status)` to return a status flag, by another thread by calling `pthread_cancel(thread)` with `thread` pointing to them or the host process finishing first (without `pthread_exit()` call)

PThreads - Thread management: Example 1

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0; t<NUM_THREADS; t++){
      printf("In main: creating thread %ld\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }

   /* Last thing that main() should do */
   pthread_exit(NULL);
}
```

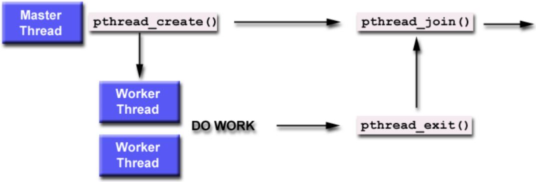
PThreads - Thread management: Joining & Detaching

PThreads - Thread management: Joining & Detaching

- "Joining" threads allows the master thread to synchronize with its worker threads on completion of their task

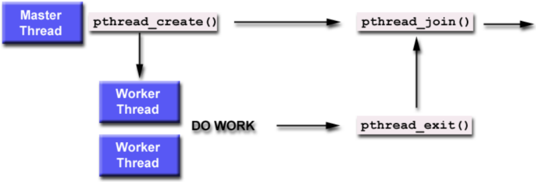
PThreads - Thread management: Joining & Detaching

- "Joining" threads allows the master thread to synchronize with its worker threads on completion of their task



PThreads - Thread management: Joining & Detaching

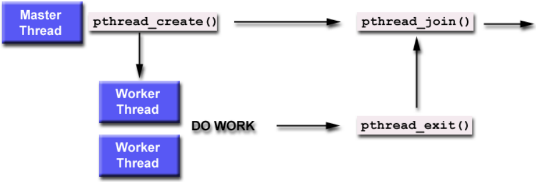
- "Joining" threads allows the master thread to synchronize with its worker threads on completion of their task



- threads can be declared "joinable" on creation

PThreads - Thread management: Joining & Detaching

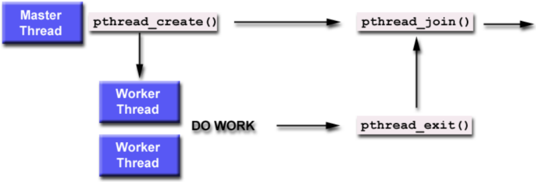
- "Joining" threads allows the master thread to synchronize with its worker threads on completion of their task



- threads can be declared "joinable" on creation
- data (Thread Control Block) remains in memory after completion of a thread until `pthread_join` is called on this dead thread and the clean-up is triggered

PThreads - Thread management: Joining & Detaching

- "Joining" threads allows the master thread to synchronize with its worker threads on completion of their task



- threads can be declared "joinable" on creation
- data (Thread Control Block) remains in memory after completion of a thread until `pthread_join` is called on this dead thread and the clean-up is triggered
- "detached" threads do not keep such (potentially unnecessary) data, i.e. get cleaned up directly on completion

PThreads - Mutexes

- Mutexes work in similar way as the OpenMP and Python locks: once claimed by one thread, other threads encountering it will be hold until the mutex released again.

PThreads - Joining & Mutexes: Example

```
#define NUMTHRDS 4
#define VECLEN 100000
   DOTDATA dotstr;
   pthread_t callThd[NUMTHRDS];
   pthread_mutex_t mutexsum;

void *dotprod(void *arg)
{
   […]

   mysum = 0;
   for (i=start; i<end ; i++)
    {
      mysum += (x[i] * y[i]);
    }

   pthread_mutex_lock (&mutexsum);
   dotstr.sum += mysum;
   printf("Thread %ld did %d to %d:  mysum=%f global sum=
%f\n",offset,start,end,mysum,dotstr.sum);
   pthread_mutex_unlock (&mutexsum);

   pthread_exit((void*) 0);
}

int main (int argc, char *argv[])
{
long i;
double *a, *b;
void *status;
pthread_attr_t attr;

a = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));
b = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));
```

```
[…]

pthread_mutex_init(&mutexsum, NULL);

/* Create threads to perform the dotproduct  */
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

for(i=0;i<NUMTHRDS;i++)
  {
  /* Each thread works on a different set of data.
   * The offset is specified by 'i'. The size of
   * the data for each thread is indicated by VECLEN.
   */
   pthread_create(&callThd[i], &attr, dotprod, (void *)i);
   }

pthread_attr_destroy(&attr);
/* Wait on the other threads */

for(i=0;i<NUMTHRDS;i++) {
  pthread_join(callThd[i], &status);
  }
/* After joining, print out the results and cleanup */

printf ("Sum =  %f \n", dotstr.sum);
free (a);
free (b);
pthread_mutex_destroy(&mutexsum);
pthread_exit(NULL);
}
```

PThreads - Condition variables

PThreads - Condition variables

- Conditions variables control the flow of threads like Mutexes

PThreads - Condition variables

- Conditions variables control the flow of threads like Mutexes
- instead of claiming a lock, it allows threads to wait (`pthread_cond_wait()`) until another thread send a signal (`pthread_cond_signal()`) through the condition variable to continue.

PThreads - Synchronization: Barriers

PThreads - Synchronization: Barriers

- POSIX Threads also feature a synchronization barrier similar to OpenMP and Python.

PThreads - Synchronization: Barriers

- POSIX Threads also feature a synchronization barrier similar to OpenMP and Python.
- Since there are no "team" structure like in OpenMP, on creation a number of threads is defined, that has to reach the barrier before any of them is allowed to pass.

PThreads - Memory management

PThreads - Memory management

- As for OpenMP, POSIX does not dictate the (default) stack size for a thread and thus can vary greatly.

PThreads - Memory management

- As for OpenMP, POSIX does not dictate the (default) stack size for a thread and thus can vary greatly.
- So better explicitly allocate enough stack to provide portability and avoid segmentation faults or data corruption

PThreads - Memory management

- As for OpenMP, POSIX does not dictate the (default) stack size for a thread and thus can vary greatly.
- So better explicitly allocate enough stack to provide portability and avoid segmentation faults or data corruption
- use `pthread_attr_setstacksize` to set the desired stacksize in the attribute object used for creating the thread.