Lecture 12

Review

- Compiler Directives
  - Conditional compilation
  - Parallel construct
  - Work-sharing constructs
    - for, *section, single*
  - *Synchronization*
  - *Work-tasking*
- *Library Functions*
- *Environment Variables*

13b.cpp

```
#include <iostream>
#include <omp.h>
using namespace std;

int main() {
  int i,j;
  #pragma omp parallel for schedule(static,2) collapse(2)
  for(i=0;i<5;i++) {
    for(j=0;j<3;j++) {
      int tid = omp_get_thread_num();
      cout << i << ", " << j << " - " << tid << endl;
    }
  }

  return 0;
}
```

```
0, 0 - 0
0, 1 - 0
0, 2 - 1
1, 0 - 1
1, 1 - 2
1, 2 - 2
2, 0 - 3
2, 1 - 3
2, 2 - 0
3, 0 - 0
3, 1 - 1
3, 2 - 1
4, 0 - 2
4, 1 - 2
4, 2 - 3
```

tid in this example is similar to rank in MPI!

Compiler directives

for *construct*, collapse *clause*

- If more than one loop is associated with the loop construct, then the iterations of all associated loops are collapsed into one larger iteration space which is then divided according to the schedule clause.
- **The iteration count for each associated loop is computed before entry to the outermost loop.** If execution of any associated loop changes any of the values used to compute any of the iteration counts then the behavior is unspecified.
- Rules
  - Perfectly nested
  - Rectangular iteration space

```
for(int i=0;i<10;i++) {
   doSomething(i);
   for(int j=0;j<i;j++) {
        ….
   }
}
```

```
for(int i=0;i<10;i++) {
     for(int j=0;j<5;j++) {
        ….
     }
     i+=2;
}
```

- *The* **order** *clause causes the code region marked by "*ordered*" directive inside loops being ordered as if they were executed in sequential!*
- *Shortcut:*

#pragma omp parallel
#pragma omp for ⇨ *#pragma omp parallel for*

14a.cpp

```
#include <iostream>

int main() {
  int i;
  #pragma omp parallel for schedule(static,4)
  for(i=0;i<100;i+=5) {
    std::cout << i << " ";
  }
  return 0;
}
```

```
OMP_NUM_THREADS=4 ./14a.exe
0 5 10 15 80 85 90 95 40 45 50 55 20 25 30 35 60 65 70 75
```

```
OMP_NUM_THREADS=2 ./14a.exe
0 5 10 15 40 45 50 55 80 85 90 95 20 25 30 35 60 65 70 75
```


The **order** *clause causes the ordered region inside loops being executed in the order as if they were executed in sequential!*

15a.cpp

```
#include <iostream>

int main() {
  int i;
  #pragma omp parallel for schedule(static,4) ordered
  for(i=0;i<100;i+=5) {
    #pragma omp ordered
    std::cout << i << " ";
  }
  return 0;
}
```

```
OMP_NUM_THREADS=4 ./15a.exe
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95
```

```
OMP_NUM_THREADS=2 ./15a.exe
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95
```

Compiler directives
for construct, nowait clause

- At the end of the parallel for region, there is an implicit synchronization (barrier) point. **nowait** removes such barrier

```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp for nowait
    for (i=0; i<n-1; i++)
        b[i] = (a[i] + a[i+1])/2;
    #pragma omp for nowait
    for (i=0; i<n; i++)
        d[i] = 1.0/c[i];

  } /*-- End of parallel region --*/
                       (implied barrier)
```

Ruud van der Pas (2005), "An introduction to OpenMP", IWOMP 2005, University of Oregon, Eugene, Oregon, USA

Compiler directives
sections construct

#pragma omp sections *[clause [[,] clause] …] new-line*
```
{
  [#pragma omp section newline]
     structured-block
  [#pragma omp section newline]
     structured-block
  …
}
```

Clauses
- *private(list)*
- *firstprivate(list)*
- *lastprivate(list)*
- *reduction(operator: list)*
- *nowait*

Each section defined is executed by exactly one thread!

16a.cpp
```
#include <iostream>
#include <omp.h>
using namespace std;
int main() {
  #pragma omp parallel default(none) shared(cout)
  {
    int ID = omp_get_thread_num();
    cout << "ThreadID: " << ID << endl;
    #pragma omp sections
    {
      #pragma omp section
        for(int i=0;i<5;i++)
          cout << "(" << ID << " doing AI)" << endl;
      #pragma omp section
        for(int i=0;i<7;i++)
          cout << "(" << ID << " getting user inputs)" << endl;
      #pragma omp section
        for(int i=0;i<3;i++)
          cout << "(" << ID << " doing drawings)" << endl;
    }
  }
  return 0;
}
```


OMP_NUM_THREADS=6 ./16a.exe
```
ThreadID: 0
(0 doing AI)
(0 doing AI)
(0 doing AI)
(0 doing AI)
(0 doing AI)
ThreadID: 1
(1 getting user inputs)
(1 getting user inputs)
(1 getting user inputs)
(1 getting user inputs)
(1 getting user inputs)
(1 getting user inputs)
(1 getting user inputs)
ThreadID: 5
ThreadID: 2
(2 doing drawings)
(2 doing drawings)
(2 doing drawings)
ThreadID: 3
ThreadID: 4
```

OMP_NUM_THREADS=2 ./16a.exe
```
ThreadID: 0
(0 doing AI)
(0 doing AI)
(0 doing AI)
(0 doing AI)
(0 doing AI)
(0 getting user inputs)
(0 getting user inputs)
(0 getting user inputs)
(0 getting user inputs)
(0 getting user inputs)
(0 getting user inputs)
(0 getting user inputs)
ThreadID: 1
(1 doing drawings)
(1 doing drawings)
(1 doing drawings)
```

OMP_NUM_THREADS=3 ./16a.exe
```
ThreadID: 0
(0 doing AI)
(0 doing AI)
(0 doing AI)
(0 doing AI)
(0 doing AI)
ThreadID: 2
(2 doing drawings)
(2 doing drawings)
(2 doing drawings)
ThreadID: 1
(1 getting user inputs)
(1 getting user inputs)
(1 getting user inputs)
(1 getting user inputs)
(1 getting user inputs)
(1 getting user inputs)
(1 getting user inputs)
```


Compiler directives
single construct

#pragma omp single *[clause [[,] clause] …] new-line*
   structured-block

Clauses
- *private(list)*
- *firstprivate(list)*
- *copyprivate(list)*
- *nowait*

- Exactly one unspecified thread executes the code in the single construct.
- There is an implicit barrier at the end of the single construct.
- Suitable for variable initialization, I/O, etc.

17a.cpp
```
#include <iostream>
#include <omp.h>
using namespace std;
int main() {
  int i, n=10, sum=0;

  #pragma omp parallel default(none) \
      shared(cin, cout, n, sum) private(i)
  {
    int ID = omp_get_thread_num();
    #pragma omp single
    {
      cout << "(" << ID << ") Please enter n: ";
      cin >> n;
    }
    #pragma omp for reduction(+:sum)
    for(i=0;i<=n;i++) {
      sum += i;
    }
  }
  cout << "Sum: " << sum;
  return 0;
}
```

Summary

- Compiler Directives
  - *Conditional compilation*
  - *Parallel construct*
  - *Work-sharing constructs*
    - *for, section, single*
  - **Synchronization**
  - *Work-tasking*
- *Library Functions*
- *Environment Variables*

13

Synchronization

- Synchronization refers to cooperate/coordiate multiple threads to work in a desired manner/order. OpenMP provides the following directives for synchronization:
  - #pragma omp master
  - #pragma omp barrier
  - #pragma omp critical
  - #pragma omp atomic
  - #pragma omp flush
  - #pragma omp ordered
- Note synchronization implies threads to be coordinated, and usually results in performance degradation.

14

Compiler directives for sychronization
#pragma omp master

- The **master** construct specifies a structured block that is executed by the master thread (thread number 0).

#pragma omp master *new-line*
structured-block

- Very much similar to **#pragma omp single** previously introduced, but:
  - The structured block is executed by the master thread.
  - There is no implicit barrier at the end of the structured block.

15

18a.cpp

```
#include <iostream>
#include <omp.h>
using namespace std;
int main() {
  int i, n=10, sum=0;
  #pragma omp parallel default(none) \
       shared(cin, cout, n, sum) private(i)
  {
    int ID = omp_get_thread_num();
    #pragma omp master
    {
      cout << "(" << ID << ") Please enter n: ";
      cin >> n;
    }
    #pragma omp barrier
    #pragma omp for reduction(+:sum)
    for(i=0;i<=n;i++) {
      sum += i;
    }
  }
  cout << "Sum: " << sum;
  return 0;
}
```

16

Compiler directives for sychronization
#pragma omp barrier

- The **barrier** construct specifies an explicit barrier at the point at which the construct appears.

#pragma omp barrier *new-line*

- **Barrier**: *A point in the execution of a program encountered by a team, beyond which no thread in the team may execute until all threads in the team have reached that point.*
- Recall: MPI_Barrier().
- The following constructs have implicit barrier at the end of the structured block: **parallel, for, sections, *critical, single***

17

Compiler directives for sychronization
#pragma omp critical

- The **critical** construct restricts execution of the associated structured block to a single thread at a time.

#pragma omp critical *[(name)] new-line*
structured-block

- *In concurrent programming, a **critical section** is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution.*
- Different threads may enter critical sections of different names.
- Often used to update shared variables, or to call **thread-unsafe** functions (e.g. rand(), file I/O, …).
  - Thread unsafe functions are usually those who keep their own "states" (e.g. static variables, rand())

18

```
04a.cpp
#include <iostream>
using namespace std;

int main() {
  int count=17;
  #pragma omp parallel shared(count)
  {
    for(int i=0;i<1000000;i++) {
      count++;
    }
    cout << "My count: " << count << endl;
  }
  cout << "Main count: " << count << endl;
  return 0;
}
```

```
OMP_NUM_THREADS=8 ./04a.exe
My count: My count:
10919331091933
My count: 2170209

My count: 3213347
My count: 3751776
My count: 4269572
My count: 4272164
My count: 4347902
Main count: 4347902
```

```
OMP_NUM_THREADS=8 ./04a.exe
My count: 1501215
My count: 2501215
My count: 3501215
My count: 4501215
My count: 5501215
My count: 6501215
My count: 7501215
My count: 1000017
Main count: 1000017
```

```
OMP_NUM_THREADS=8 ./04a.exe
My count: 1089747
My count: 2126479
My count: 2240935
My count: 3165806
My count: 3345987
My count: 3364074
My count: 4210268
My count: 4425132
Main count: 4425132
```

```
OMP_NUM_THREADS=8 ./04a.exe
My count: 1152417
My count: 2249624
My count: 2274868
My count: 3346454
My count: 3858762
My count: 4402852
My count: 4406426
My count: 4465075
Main count: 4465075
```

```
OMP_NUM_THREADS=8 ./04a.exe
My count: 1000017
My count: 2000017
My count: 3000017
My count: 4000017
My count: 5000017
My count: 6000017
My count: 7000017
My count: 8000017
Main count: 8000017
```

19

```
19a.cpp
#include <iostream>
using namespace std;

int main() {
  int count=17;
  #pragma omp parallel shared(count)
  {
    for(int i=0;i<1000000;i++) {
      #pragma omp critical
      count++;
    }
  }
  cout << "Main count: " << count << endl;
  return 0;
}
```

```
OMP_NUM_THREADS=8 ./19a.exe
Main count: 8000017
```

```
OMP_NUM_THREADS=8 ./19a.exe
Main count: 8000017
```

```
OMP_NUM_THREADS=8 ./19a.exe
Main count: 8000017
```

```
OMP_NUM_THREADS=8 ./19a.exe
Main count: 8000017
```

NOTE: Critical section induces a great synchronization between threads, and may significantly reduce performance due to this synchronization!

Compiler directives for sychronization

#pragma omp atomic

- The atomic construct ensures that a specific storage location is updated **atomically**, rather than exposing it to the possibility of multiple, simultaneous writing threads.

  #pragma omp atomic *new-line*
  expression-statement

- This is a special case of a critical section that can only be used for certain simple statements: =, ++, --, +=, -=, *=, /=, &=, ^=, …
- And it is more efficient …

21

Compiler directives for sychronization

#pragma omp flush

- In OpenMP, threads have temporary view of memory (in cache or in register). This temp. view of memory may be different from the actual content in the main memory.
- The flush construct enforces consistency between the temporary view and memory. (i.e. flush variables from cache to main memory, and "invalid" cached variable so that the next the variable has to be read from the main memory)

  #pragma omp flush (variable list) *new-line*

- If the list is ignored, all thread-local variables are flushed. This is implied at the following regions: barrier region; entry to/exit from parallel, critical, and ordered; exit from work-sharing; lock API function calls; before & after task scheduling point.

22

Synchronization

- Synchronization refers to cooperate/coordiate multiple threads to work in a desired manner/order. OpenMP provides the following directives for synchronization:
  - #pragma omp master
  - #pragma omp barrier
  - #pragma omp critical
  - #pragma omp atomic
  - #pragma omp flush
  - #pragma omp ordered
- Note synchronization implies threads to be coordinated, and usually results in performance degradation.

23

Guess what is the outcome?

38a.cpp

```
11 55 11 11 55 55 11 11 11 55 11 55 11 55 11 55
11 55 55 55 11 55 55 11 11 55 55 11 11 55 55 55
11 55 11 11 55 55 55 11 11 55 55 55 11 55 55 11
```

```
#include <iostream>
#include <omp.h>

using namespace std;
#define NUM_THREADS 16
int synch[NUM_THREADS];
int output[NUM_THREADS];

int main() {
  #pragma omp parallel num_threads(NUM_THREADS)
  {
    int id = omp_get_thread_num();
    int next = (id>0? id: NUM_THREADS) - 1;

    #pragma omp for
    for(int i=0;i<NUM_THREADS;i++) {
      synch[i] = 1;
    }
    int status = synch[next];

    synch[id] = 5;
    output[id] = status * 10 + synch[next];
  }

  for(int i=0;i<NUM_THREADS;i++) cout << output[i] << " ";
  return 0;
}
```

24

```
#include <iostream>
#include <omp.h>

using namespace std;
#define NUM_THREADS 16
int synch[NUM_THREADS];
int output[NUM_THREADS];

int main() {
  #pragma omp parallel num_threads(NUM_THREADS)
  {
    int id = omp_get_thread_num();
    int next = (id>0? id: NUM_THREADS) - 1;

    #pragma omp for
    for(int i=0;i<NUM_THREADS;i++) {
      synch[i] = 1;
    }
    #pragma omp barrier
    int status = synch[next];

    synch[id] = 5;
    output[id] = status * 10 + synch[next];
  }
  for(int i=0;i<NUM_THREADS;i++) cout << output[i] << " ";
  return 0;
}
```

Guess what is the outcome?

38b.cpp

11 15 15 11 15 15 11 11 15 15 11 15 11 15 11 15
11 15 15 11 11 15 15 11 15 15 11 11 11 15 15 15
11 15 11 11 15 11 11 11 11 15 15 11 15 15 11 15

25

```
#include <iostream>
#include <omp.h>

using namespace std;
#define NUM_THREADS 16
int synch[NUM_THREADS];
int output[NUM_THREADS];

int main() {
  #pragma omp parallel num_threads(NUM_THREADS)
  {
    int id = omp_get_thread_num();
    int next = (id>0? id: NUM_THREADS) - 1;

    #pragma omp for
    for(int i=0;i<NUM_THREADS;i++) {
      synch[i] = 1;
    }
    int status = synch[next];
    #pragma omp barrier

    synch[id] = 5;
    output[id] = status * 10 + synch[next];
  }
  for(int i=0;i<NUM_THREADS;i++) cout << output[i] << " ";
  return 0;
}
```

Guess what is the outcome?

38c.cpp

15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15
15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15
15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15

26

Examples

Vector normalization: make a vector of unity length without changing its direction

27

Serial Implementation

```
double length = 0;
for(i=0;i<V_SIZE;i++) {
  length += a[i] * a[i];
}

length = sqrt(length);

for(i=0;i<V_SIZE;i++) {
  a[i] /= length;
}
```

21a.cpp

28

Parallel Implementation, V1

```
double
for(i=(
  lengt
}

length
#pragma omp parallel for
for(i=0;i<V_SIZE;i++) {
  a[i] /= length;
}
```

Size	Serial	V1	SpeedUp
1.E+02	0.0000	0.0080	0.0003
1.E+03	0.0000	0.0076	0.0020
1.E+04	0.0001	0.0074	0.0198
1.E+05	0.0015	0.0045	0.3371
1.E+06	0.0151	0.0144	1.0492
1.E+07	0.1401	0.0830	1.6876
1.E+08	1.4063	0.7827	1.7967

22a.cpp

29

Parallel Implementation, V2

```
double length = 0;
#pragma omp parallel for
for(i=0;i<V_SIZE;i++) {
                       a[i
                       h);
#pragma omp parallel for
for(i=0;i<V_SIZE;i++) {
  a[i] /= length;
}
```

Size	Serial	V1	SpeedUp
1.E+02	0.0000	0.0080	0.0003
1.E+03	0.0000	0.0076	0.0020
1.E+04	0.0001	0.0074	0.0198
1.E+05	0.0015	0.0045	0.3371
1.E+06	0.0151	0.0144	1.0492
1.E+07	0.1401	0.0830	1.6876
1.E+08	1.4063	0.7827	1.7967

Size	Serial	V2	SpeedUp
1.E+02	0.0000	0.0072	0.0003
1.E+03	0.0000	0.0076	0.0020
1.E+04	0.0001	0.0074	0.0197
1.E+05	0.0015	0.0086	0.1755
1.E+06	0.0151	0.0160	0.9419
1.E+07	0.1401	0.0835	1.6779
1.E+08	1.4063	0.7760	1.8122

23a.cpp

30

Parallel Implementation, V3

```
double length = 0;
#pragma omp parallel for \
   reduction(+:length)
for(i
  len
}

lengt

#prag
for(i=0;i<V_SIZE;i++) {
  a[i] /= length;
}
```

Size	Serial	V3	SpeedUp
1.E+02	0.0000	0.0084	0.0003
1.E+03	0.0000	0.0076	0.0020
1.E+04	0.0001	0.0073	0.0200
1.E+05	0.0015	0.0085	0.1768
1.E+06	0.0151	0.0157	0.9619
1.E+07	0.1401	0.0815	1.7191
1.E+08	1.4063	0.7695	1.8276

24a.cpp

31

Parallel Implementation, V4

```
double length = 0;
#pragma omp parallel
{
  #pragma omp for reduction(+:length)
  for(i=0;i<V_SIZE;i++) {
    length += a[i] * a[i];
  }

  length = sqrt(length);

  #pragma omp for
  for(i=0;i<V_SIZE;i++) {
    a[i] /= length;
  }
}
```

32

Parallel Implementation, V5

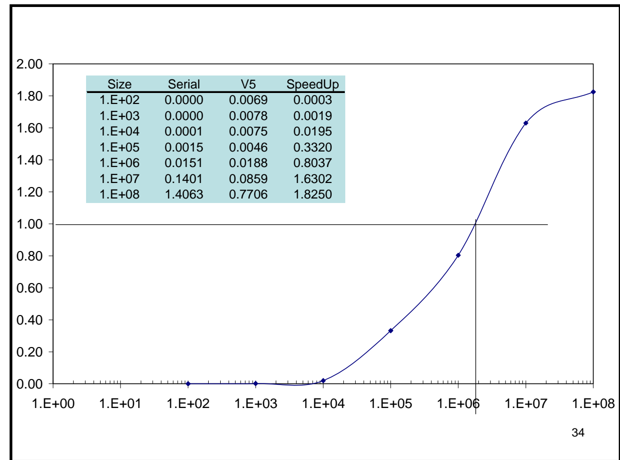
```
double length = 0;
#pragma omp parallel
{
  #pragma omp for reduction(+:length)
  for(i=0;i<V_SIZE;i++) {

  for(i=0;i<V_SIZE;i++) {
    a[i] /= length;
  }
}
```

Size	Serial	V3	SpeedUp
1.E+02	0.0000	0.0084	0.0003
1.E+03	0.0000	0.0076	0.0020
1.E+04	0.0001	0.0073	0.0200
1.E+05	0.0015	0.0085	0.1768
1.E+06	0.0151	0.0157	0.9619
1.E+07	0.1401	0.0815	1.7191
1.E+08	1.4063	0.7695	1.8276

Size	Serial	V5	SpeedUp
1.E+02	0.0000	0.0069	0.0003
1.E+03	0.0000	0.0078	0.0019
1.E+04	0.0001	0.0075	0.0195
1.E+05	0.0015	0.0046	0.3320
1.E+06	0.0151	0.0188	0.8037
1.E+07	0.1401	0.0859	1.6302
1.E+08	1.4063	0.7706	1.8250

25a.cpp

33



Size	Serial	V5	SpeedUp
1.E+02	0.0000	0.0069	0.0003
1.E+03	0.0000	0.0078	0.0019
1.E+04	0.0001	0.0075	0.0195
1.E+05	0.0015	0.0046	0.3320
1.E+06	0.0151	0.0188	0.8037
1.E+07	0.1401	0.0859	1.6302
1.E+08	1.4063	0.7706	1.8250

34

Parallel Implementation, V6

```
double length = 0;
#pragma omp parallel if(V_SIZE>=2000000)
{
  #pragma omp for reduction(+:length)
  for(i=0
    lengt
  }
  #pragma
  length

  #pragma
  for(i=0
    a[i]
  }
}
```

Size	Serial	V6	SpeedUp
1.E+02	0.0000	0.0000	1.0000
1.E+03	0.0000	0.0000	1.0678
1.E+04	0.0001	0.0001	1.0754
1.E+05	0.0015	0.0014	1.0523
1.E+06	0.0151	0.0149	1.0106
1.E+07	0.1401	0.0823	1.7030
1.E+08	1.4063	0.7726	1.8203

26a.cpp

35

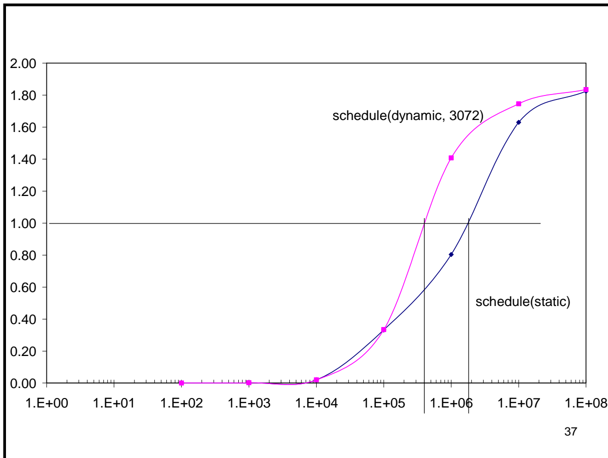
Parallel Implementation, V7

```
double length = 0;
#pragma omp parallel if(V_SIZE>=400000)
{
  #pragma omp for schedule(dynamic,3072) \
     reduction(+:length)
  for(i=0;
    length
  }
  #pragma
  length =
  #pragma
  for(i=0;
    a[i] /
  }
}
```

Size	Serial	V7	SpeedUp
1.E+02	0.0000	0.0000	1.1250
1.E+03	0.0000	0.0000	1.0678
1.E+04	0.0001	0.0001	1.0679
1.E+05	0.0015	0.0014	1.0478
1.E+06	0.0151	0.0119	1.2643
1.E+07	0.1401	0.0807	1.7356
1.E+08	1.4063	0.7677	1.8318

27a.cpp

36



37

Work Tasking

38

Tasking

- New feature in OpenMP 3.0 (200805)
- Used to parallelize very unstructured parallelism Unbounded loops, recursive functions, ...
- Tasks are work units whose execution may be deferred or may be executed immediately (
- Synchronize through #pragma omp taskwait - suspends the current task until all children tasks are completed.
  - Just direct children, not descendants

Team Task pool

39

Compiler directives
task *construct*

#pragma omp task *[clause [[,] clause] ...] new-line*
structured-block

Clauses
default(shared|none)
private(list)
firstprivate(list)
shared(list)

if (scalar-expression)
untied – untied task, which can be executed by any thread in the team

•The task construct defines an explicit task.
•The encountering thread may immediately execute the task, or defer its execution. In the latter case, any thread in the team may be assigned the task. Completion of the task can be guaranteed using task synchronization constructs

40

34a.cpp

```
unsigned long fib(unsigned long n) {
  if(n<2) return n;
  unsigned long x, y;
  #pragma omp task shared(x)
  x = fib(n-1);
  #pragma omp task shared(y)
  y = fib(n-2);
  #pragma omp taskwait
  return x+y;
}
int main() {
  #pragma omp parallel
  {
    #pragma omp single nowait
    cout << fib(35) << " ";
  }
  return 0;
}
```

```
time ./34a.exe
9227465

real    0m11.608s
user    0m17.665s
sys     0m5.352s
```

41

35a.cpp

```
unsigned long fib(unsigned long n) {
  if(n<2) return n;
  unsigned long x, y;
  #pragma omp task shared(x)
  x = fib(n-1);
  y = fib(n-2);
  #pragma omp taskwait
  return x+y;
}
int main() {
  #pragma omp parallel
  {
    #pragma omp single nowait
    cout << fib(35) << " ";
  }
  return 0;
}
```

```
time ./35a.exe
9227465

real    0m5.806s
user    0m9.345s
sys     0m2.004s
```

42

Compiler directives
task *construct,* if *clause*

- When the if clause argument is false
  - The encountering task is suspended.
  - The new task is executed immediately by the encountering thread.
  - The data environment is still local to the new task...
  - ...and it's still a different task with respect to synchronization.
  - The parent task resumes when the task finishes.
- It's a user directed optimization
  - when the cost of deferring the task is too great compared to the cost of executing the task code.
  - to control cache and memory affinity.
  - Avoid creating small tasks

A "Hands-on" Introduction to OpenMP*, T. Mattson and L. Meadows

43

36a.cpp

```
unsigned long fib(unsigned long n) {
  if(n<2) return n;
  unsigned long x, y;
  #pragma omp task shared(x) if(0)
  x = fib(n-1);
  y = fib(n-2);

  return x+y;
}
int main() {
  #pragma omp parallel
  {
    #pragma omp single nowait
    cout << fib(35) << " ";
  }
  return 0;
}
```

if (false) makes the encountering thread immediately executes the new task (and suspend the current running task …)

```
time ./36a.exe
9227465

real    0m4.606s
user    0m4.792s
sys     0m0.000s
```

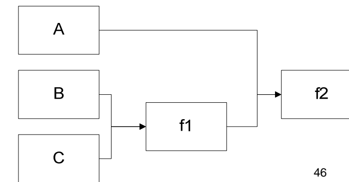
44

Compiler directives
task *construct,* untied *clause*

- By default, tasks are tied to the thread that first executes them
  - not the creator
- Tied tasks can be scheduled as the implementation wishes
  - Constraints:
    - Only the thread that the task is tied to can execute it
    - A task can only be suspended at a suspend point
      - task creation, task finish, taskwait, barrier
    - If the tasks is not suspended in a barrier it can only switch to a direct descendant of all tasks tied to the thread
- Untied tasks
  - Can be suspended at any point
  - Can switch to any task
  - Be careful: variable scoping

45

```
/* Compute f2 (A, f1 (B, C)) */
void foo () {
    int a, b, c, x, y;
    #pragma omp task shared(a)
        a = A();
    #pragma omp task if (0) shared (b, c, x) untied
    {
        #pragma omp task shared(b)
                b = B();
        #pragma omp task shared(c)
                c = C();
        #pragma omp taskwait
    }
    x = f1 (b, c);
    #pragma omp taskwait
    y = f2 (a, x);
}
```



46

http://wikis.sun.com/display/openmp/Using+the+Tasking+Feature

Assignment #6

Due: 6/5/2012