HPCSE II

OpenMP

Tasks in OpenMP 3.0

- We have seen a few ways to parallelize a block
  - `#pragma omp parallel`
  - `#pragma omp sections`
  - `#pragma omp parallel for`
- Parallel for was great for for-loops, but what about unstructured data?
  - Traversal through lists and trees?
  - while loops?
- Spawning threads dynamically is expensive
- Tasks are more lighweight:
  - new tasks get put onto a task queue
  - idle threads pull tasks from the queue

Example: Fibonacci sequence

- Parallelize recursive function $F_n = F_{n-1} + F_{n-2}$
  (yes we know that this is an artificial example)
- The serial code first

```
#include <iostream>

int fibonacci(int n)
{
  int i, j;

  if (n<2)
    return n;
  else {
    i = fibonacci(n-1);
    j = fibonacci(n-2);
    return i + j;
  }
}

int main()
{
  int n;
  std::cin >> n;

  std::cout << fibonacci(n) << std::endl;
}
```

Example: Fibonacci sequence

- Parallelize recursive function $F_n = F_{n-1} + F_{n-2}$
  (yes we know that this is an artificial example)
- First attempt using sections

```
#include <iostream>

int fibonacci(int n)
{
  int i, j;

  if (n<2)
    return n;
  else {
    #pragma omp parallel sections shared (i,j)
    {
      #pragma omp section
      i = fibonacci(n-1);

      #pragma omp section
      j = fibonacci(n-2);
    }
    return i + j;
  }
}
```

```
int main()
{
  int n;
  std::cin >> n;

  std::cout << fibonacci(n) << std::endl;
}
```

Problem: uncontrolled spawning of expensive threads

The task directive

- Spawns tasks and puts them into a queue for the threads to work on:

```
#pragma omp task [clause ...]
```

`if (scalar_expression)`	Only parallelize if the expression is true. Can be used to stop parallelization if the work is too little
`private (list)`	The specified variables are thread-private
`shared (list)`	The specified variables are shared among all threads
`default (shared | none)`	Unspecified variables are shared or not
`firstprivate (list)`	Initialize private variables from the master thread
`mergeable`	If specified allows the task to be merged with others
`untied`	If specified allows the task to be resumed by other threads after suspension. Helps prevent starvation but has **unusual memory semantics**: after moving to a new thread all private variables are that of the new thread
`final (scalar_expression)`	If the expression is true this has to be the final task. All dependent tasks are included into it.

Example: Fibonacci sequence

- Parallelize recursive function $F_n = F_{n-1} + F_{n-2}$
  (yes we know that this is an artificial example
- First attempt using tasks

```
#include <iostream>

int fibonacci(int n)
{
  int i, j;

  if (n<2)
    return n;
  else {
    #pragma omp task shared(i) firstprivate(n)
    i = fibonacci(n-1);

    #pragma omp task shared(j) firstprivate(n)
    j = fibonacci(n-2);

    return i + j;
  }
}
```

```
int main()
{
  int n;
  std::cin >> n;

  std::cout << fibonacci(n) << std::endl;
}
```

Problem 1: no parallel region

Example: Fibonacci sequence

- Parallelize recursive function $F_n = F_{n-1} + F_{n-2}$
  (yes we know that this is an artificial example
- Second attempt using tasks

```
#include <iostream>

int fibonacci(int n)
{
  int i, j;

  if (n<2)
    return n;
  else {
    #pragma omp task shared(i) firstprivate(n)
    i = fibonacci(n-1);

    #pragma omp task shared(j) firstprivate(n)
    j = fibonacci(n-2);

    return i + j;
  }
}
```

```
int main()
{
  int n;
  std::cin >> n;

  #pragma omp parallel shared(n)
  {
    std::cout << fibonacci(n) << std::endl;
  }
}
```

Problem 2: now we have too many calls to fibonacci

Example: Fibonacci sequence

- Parallelize recursive function $F_n = F_{n-1} + F_{n-2}$
  (yes we know that this is an artificial example
- Third attempt using tasks

```
#include <iostream>

int fibonacci(int n)
{
  int i, j;

  if (n<2)
    return n;
  else {
    #pragma omp task shared(i) firstprivate(n)
    i = fibonacci(n-1);

    #pragma omp task shared(j) firstprivate(n)
    j = fibonacci(n-2);

    return i + j;
  }
}
```

```
int main()
{
  int n;
  std::cin >> n;

  #pragma omp parallel shared(n)
  {
    #pragma omp single nowait
    std::cout << fibonacci(n) << std::endl;
  }
}
```

Problem 3: i and j get added before the tasks are done

Problem 4: when i and j are written the variables no longer exist

Example: Fibonacci sequence

- Parallelize recursive function $F_n = F_{n-1} + F_{n-2}$
  (yes we know that this is an artificial example
- Fourth attempt using tasks

```
#include <iostream>

int fibonacci(int n)
{
  int i, j;

  if (n<2)
    return n;
  else {
    #pragma omp task shared(i) firstprivate(n)
    i = fibonacci(n-1);

    #pragma omp task shared(j) firstprivate(n)
    j = fibonacci(n-2);

    #pragma omp taskwait
    return i + j;
  }
}
```

```
int main()
{
  int n;
  std::cin >> n;

  #pragma omp parallel shared(n)
  {
    #pragma omp single nowait
    std::cout << fibonacci(n) << std::endl;
  }
}
```

Now it works

Task-related directives and functions

- Wait for all dependent tasks:

```
#pragma omp taskwait
```

- Yield the thread to another task

```
#pragma omp taskyield
```

- Check at runtime whether this is a final task

`int omp_in_final()`	Returns true if the task is a final task

Optimizing tasking using the final clause

- Parallelize recursive function $F_n = F_{n-1} + F_{n-2}$
  (yes we know that this is an artificial example
- Fifth attempt using tasks

```
#include <iostream>

int fibonacci(int n)
{
  int i, j;

  if (n<2)
    return n;
  else {
    #pragma omp task shared(i) firstprivate(n) untied final (n<=5)
    i = fibonacci(n-1);

    #pragma omp task shared(j) firstprivate(n) untied final (n<=5)
    j = fibonacci(n-2);

    #pragma omp taskwait
    return i + j;
  }
}
```

Now it will not spawn tasks for n<=5

A more complex example: quicksort

- See source code repository for the full code

```
// this may not be optimal but is short
template <class It>
void quicksort(It first, It last)
{
  // empty sequence or length 1: we are done
  if (last-first <= 1)
    return;

  // pick a pivot, here randomly choose the last
  typedef typename std::iterator_traits<
                     It>::value_type value_type;
  value_type pivot = *(last-1);

  // partition the sequence
  It split = std::partition(first,last,
       [=](value_type x) { return x < pivot;});

  // move the pivot to the center
  std::swap(*(last-1),*split);

  // sort the two partitions individually
  #pragma omp task final (split-first<=1)
  quicksort(first,split);
  #pragma omp task final (last-split-1<=1)
  quicksort(split+1,last);
}
```

```
int main()
{
  int n;
  std::cin >> n;

  // create random numbers
  std::mt19937 mt;
  std::uniform_int_distribution<int> dist(0,100000000);
  std::vector<int> data(n);
  std::generate(data.begin(),data.end(),
                std::bind(dist,mt));

  // call quicksort in parallel
  #pragma omp parallel
  #pragma omp single nowait
  quicksort(data.begin(),data.end());

  // check if it is sorted
  if (std::is_sorted(data.begin(), data.end()))
      std:: cout << "Final data is sorted.\n";
  else
      std:: cout << "Final data is not sorted.\n";

}
```