CS420 – Lectures 12

Raghavendra Kanakagiri
Slides: Marc Snir

Spring 2023

Barriers

```c
double a[N], b[N], c[N];
#pragma omp parallel
{
  int tid = omp_get_thread_num();
  if (tid == 0) {
    for (int i = 0; i < N; i++) a[i] = 1.0;
  }
  #pragma omp barrier
  #pragma omp for
  for (int i = 0; i < N; i++) {
    b[i] = func(a, i);
  }
  // implicit barrier
  #pragma omp for nowait
  for (int i = 0; i < N; i++) {
    c[i] = func(a, i);
  }
  // no barrier
  a[tid] = 0.0;
}
// implicit barrier
```

Data Sharing

```
double a[N];
int main()
{
  int x = 3;
  #pragma omp parallel
  {
    double y;
    // some work
  }
  return 0;
}
// x is shared, y is private
// private -> each thread gets its own copy
```

Data Sharing

```
double a[N];
int main()
{
  int sum = 0;
  #pragma omp parallel for private(sum)
  for (int i = 0; i < N; i++)
  {
    sum += a[i];
  }
  printf("sum = %d\n", sum);
  return 0;
}
// what is wrong here?
```

Data Sharing

```
double a[N];
int main()
{
  int sum = 0;
  #pragma omp parallel for private(sum)
  for (int i = 0; i < N; i++)
  {
    sum += a[i];
  }
  printf("sum = %d\n", sum);
  return 0;
}
// sum in each thread is uninitialized
// print sum = 0
```

Data Sharing

```c
double a[N];
int main()
{
  int sum = 0;
  #pragma omp parallel private(sum)
  {
    #pragma omp for
    for (int i = 0; i < N; i++)
    {
      sum += a[i];
    }
    printf("sum = %d\n", sum);
  }
  printf("sum = %d\n", sum);
  return 0;
}
```

Tasks

- Parallel loops are convenient for nice iteration domains, but not for irregular computations where it is not clear upfront what tasks need to be generated.
- The *task* construct helps for this purpose.

Within a parallel section

```
#pragma omp task
{...}
```

will start a task that can execute on any of the available threads; the calling task may continue executing in parallel with the newly created task.

Tasks

- `task`: spawns a task that can execute separately
- `taskwait`: wait for all spawned tasks to complete before continuing
- `shared`: parent's variable shared with child

Tasks

```
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp task
    func1();
    #pragma omp task
    func2();
    #pragma omp taskwait
    #pragma omp task
    func3();
  }
}
// func3() can execute only after func1() and func2() have completed
```

Tasks

```
#pragma omp parallel shared(x) private(y)
{
  #pragma omp task
  {
    int z;
    func(x, y, z);
  }
}
// s is shared, y is firstprivate, z is private
```

A terrible example: Fibonacci

fib(0) = 0; fib(1)=1;
fib(n)=fib(n-1)+fib(n-2)

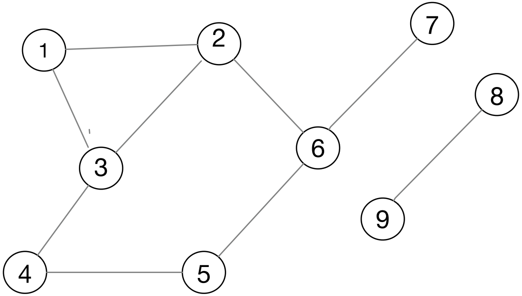
```
int fib(int n) {
 int i, j;
 if (n<2) return n;
 else {
  #pragma omp task shared(i)
   i=fib(n-1);
  #pragma omp task shared(j)
   j=fib(n-2);
  #pragma omp taskwait
   return i+j;
  }
}
```

- `task`: spawns a task that can execute separately
- `taskwait`: wait for all spawned tasks to complete before continuing
- `shared`: parent's variable shared with child

Why terrible?

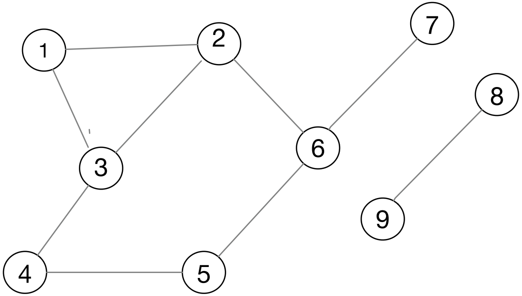
- Can be computed in constant time:
  $fib(n) = \frac{1}{\sqrt{5}}\left((\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n\right) \approx \frac{1}{\sqrt{5}}\left((\frac{1+\sqrt{5}}{2})^n\right)$
- Can be computed in linear time using the linear recursion
- Number of tasks spwaned by the parallel algorithm is $ntasks(n) = ntasks(n-1) + ntasks(n-2)$; i.e., $ntasks(n) = fib(n)$. Exponential amount of compute work!

Example: Graph traversal

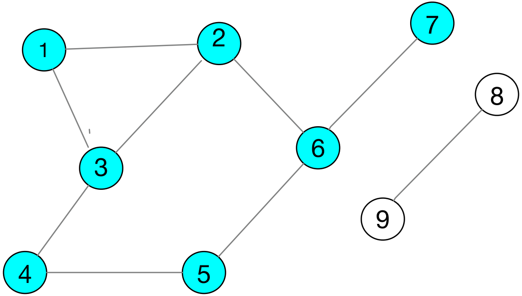


Mark all the nodes that can be reached from node 1

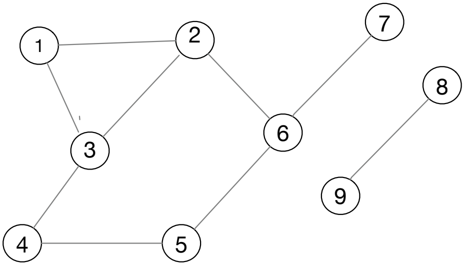
Example: Graph traversal



Mark all the nodes that can be reached from node 1



Adjacency list representation



1	2	3	
2	1	3	6
3	1	2	4
4	3	5	
5	4	6	
6	2	5	7
7	6		
8	9		
9	8		

Parallel traversal

```
\\ structure for node
typedef struct {
 int visited; \\ mark for visited node
 int numneighbors; \\ number of neighbors (degree)
 int neighbors[]; \\ array of neighbor ids
} Node;

Node * graph[N]; \\ array of pointers to nodes

void visit(int i) {
 int j,k,mark;
  for(j=0; j<graph[i]->numneighbors; j++) {
   k = graph[i]->neighbors[j];
   #pragma omp atomic
    mark = graph[k]->visited++;
  if(mark==0)
   #pragma omp task
    visit(k);
  }
}
}
```

```
int main() {
 #pragma omp parallel \\ need to start all threads
  #pragma omp single \\ need to call only once
   visit(0);
}
```

Task Dependences

True dependence (aka RAW, aka *flow dependence*)

```
int main() {
 int x = 1;
 #pragma omp parallel
  #pragma omp single {
  #pragma omp task shared(x) depend(out: x)
    x = 2;
  #pragma omp task shared(x) depend(in: x)
    printf("x = %d\n", x); } return 0;
  }
```

Will print x=2

Task Dependences

Anti-dependence (aka WAR)

```
int main() {
 int x = 1;
 #pragma omp parallel
  #pragma omp single
   {
   #pragma omp task shared(x) depend(in: x)
   printf("x = %d\n", x);
   #pragma omp task shared(x) depend(out: x)
   x = 2;
   }
return 0;
}
```

Will print `x=1`

Task Dependences

Output-dependence (aka WAW)

```
int main() {
 int x;
 #pragma omp parallel
  #pragma omp single
   {
    #pragma omp task shared(x) depend(out: x)
      x = 1;
    #pragma omp task shared(x) depend(out: x)
      x = 2;
    #pragma omp taskwait
    printf("x = %d\n", x);
  }
 return 0;
}
```

Will print `x=2`
If a dependence exists then tasks are executed in the order they were spawned.

Not a dependency

RAR

```
int main() {
 int x = 1;
 #pragma omp parallel
  #pragma omp single
   {
   #pragma omp task shared(x) depend(in: x)
   printf("x = %d\n", x);
   #pragma omp task shared(x) depend(in: x)
   x = 2;
   }
return 0;
}
```

Can print x=1 or x=2