CS420 – Lecture 8

Marc Snir

Fall 2018

Jacobi

```
...
#pragma omp parallel
 {
 do {
  #pragma omp for collapse(2) \
   reduction(max:err) private(i,j)
   for (i=1;i<N-1; i++)
    for(j=1;j<N-1;j++) {
   a[1-l][i][j]=0.25*(a[l][i-1][j]
      +a[l][i+1][j]+a[l][i][j-1]
      +a[l][i][j+1]);
    err = fmax(err,
      fabs(a[1][i][j]-a[0][i][j]));
    }
   l=1-l;
    } while(err>maxerr);
  #pragma omp single
    ll=l;
 }
```

```
printf("\n \n");
 for (i=0;i<N;i++) {
  for(j=0;j<N;j++)
   printf("%5.2f ",a[ll][i][j]);
  printf("\n");
```

- do test done at each thread
- implicit barrier at end of `for` block
- Can use `master`, rather than `single`

Slight improvement

```
#pragma omp parallel
 {
 do {
  #pragma omp for collapse(2) reduction(max:notdone) private(i,j)
   for (i=1;i<N-1; i++)
  for(j=1;j<N-1;j++) {
   a[1-l][i][j]=0.25*(a[l][i-1][j]+a[l][i+1][j]
      +a[l][i][j-1]+a[l][i][j+1]);
   if(err > maxerr || err <-maxerr) ++notdone;
  }
  l=1-l;
 } while(notdone);
 #pragma omp single
  ll=1;
 }
```

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.

A terrible example: Fibonacci

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

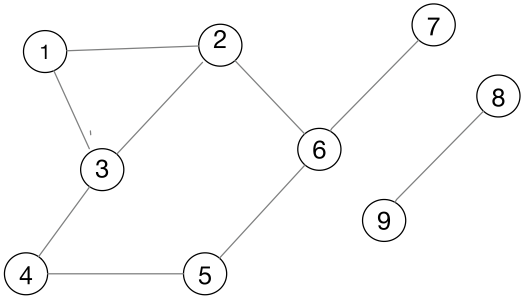
```
int fib(int n) {
 int i, j;
 if (n<2) return n;
 else {
  #pragma omp task shared(i)
   i=fib(n-1);
  #pragma omp task shared(j)
   j=fib(n-2);
  #pragma omp taskwait
   return i+j;
  }
}
```

- `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?

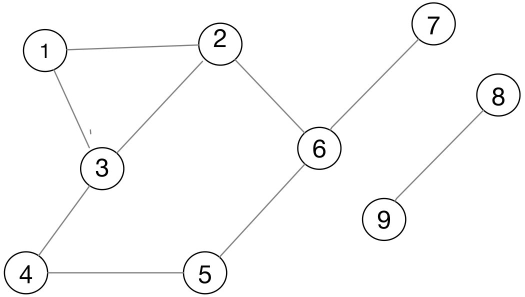
- Can be computed in constant time:
  $fib(n) = \frac{1}{\sqrt{5}}\left((\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n\right) \approx \frac{1}{\sqrt{5}}\left((\frac{1+\sqrt{5}}{2})^n\right)$
- Can be computed in linear time using the linear recursion
- Number of tasks spwaned by the parallel algorithm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

Example: Graph traversal

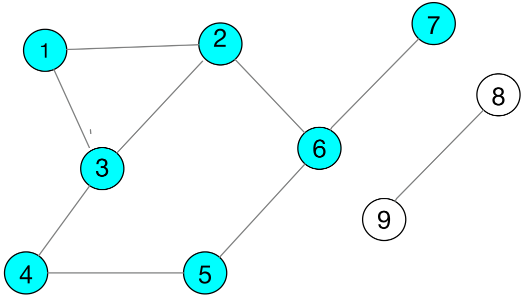


Mark all the nodes that can be reached from node 1

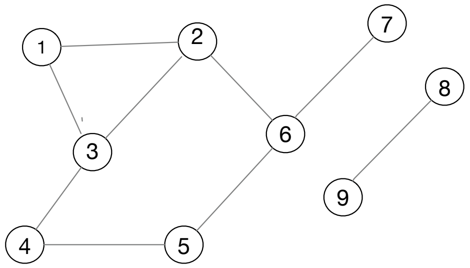
Example: Graph traversal



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.

Back to Gauss-Seidel

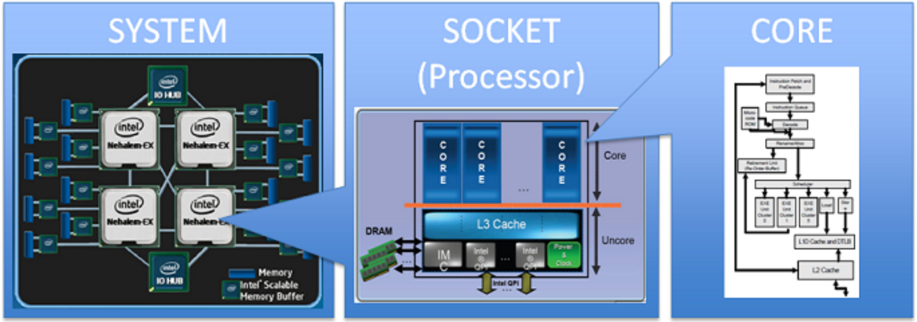
```
#pragma omp parallel
 #pragma omp single
  for(ii=1;ii<N-1;ii+=T)
   for(jj=1;jj<M-1;j+=T) {
  #pragma omp task depend(in:a[ii-1][jj],a[ii][jj-1]) \
    depend(out:a[ii][jj+T-1],a[ii+T-1][jj])
    {
      for(i=ii;i<ii+T;i++)
      for(j=jj;j<jj+T;j++)
      a[i][j] =0.2*(a[i-1][j]+a[i+1][j]+a[i][j-1]+a[i][j+1]+a[i][j]);
    }
```

NUMA

Non-Uniform Memory Access

cc-NUMA

Multisocket system



- *cc-NUMA*: Cache-Coherent Non Uniform Memory Access.
- All caches are coherent
- Access to local memory (memory on same socket) is faster and has higher bandwidth than access to remote memory.
- Need to organize data so that core mostly accesses local data

- How does one control where data goes?
- How does one control where threads run?

Environment variable `OMP_PLACES` define what is a location (place) and provides names (numbers) for places in the system

- `setenv OMP_PLACES threads` (or `export OMP_PLACES=threads`) – each place is a hardware thread; places are numbered 0,1,2...
- `setenv OMP_PLACES cores` – each place is a core
- `setenv OMP_PLACES socket` – each place is a socket
- `setenv OMP_PLACES "cores(4)"` – the computation will use four cores
- `setenv OMP_PLACES "{0,1},{2,3},{4,5},{6,7}"` – the computation has 4 places, each with 2 HW threads.

We have the usual plethora of ICV querrying functions

```
omp_get_num_places()
omp_get_place_num_procs()
omp_get_place_proc_ids()
omp_get_place_num()
omp_get_partition_num_places.()
omp_get_partition_place_nums()
```

We can control where a thread runs

```
#pragma omp parallel proc_bind(...)
```

- `proc_bind(master)` – all threads in the parallel thread team run in the same place as the master (same hardware thread/same core / same socket)
- `proc_bind(close)` – assign threads to consecutive places, starting with the master place
- `proc_bind(spread)` – assign threads to places that are as distant from each other as possible

There is no explicit way of managing data location in OpenMP for cc-NUMA systems. There is a numa library (`#include <numa.h>`) that can be used to control where threads run and where memory is allocated.
NUMA memory allocators

```
numa_alloc_onnode(size, node)
numa_alloc_local(size)
numa_alloc_interleaved(size)
```

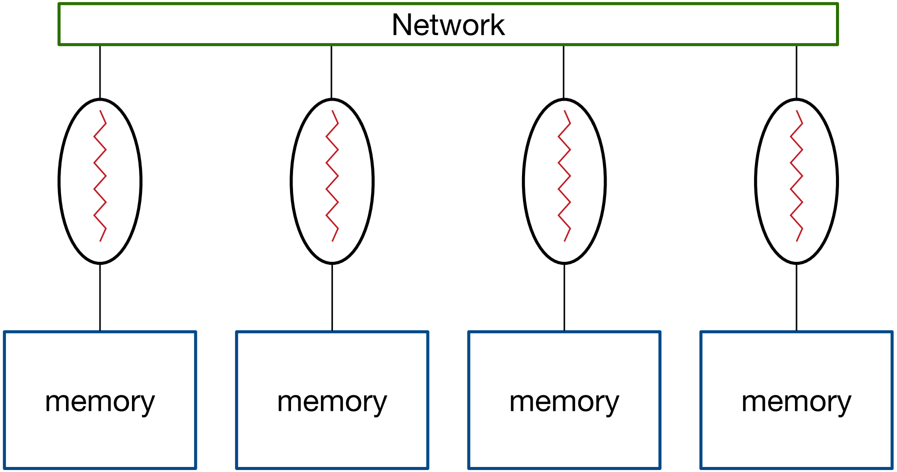
One can also control where memory is allocated by the OS when page faults occcur.
Control of affinity of computing to data in cc-NUMA systems is "work in progress"

Message-Passing

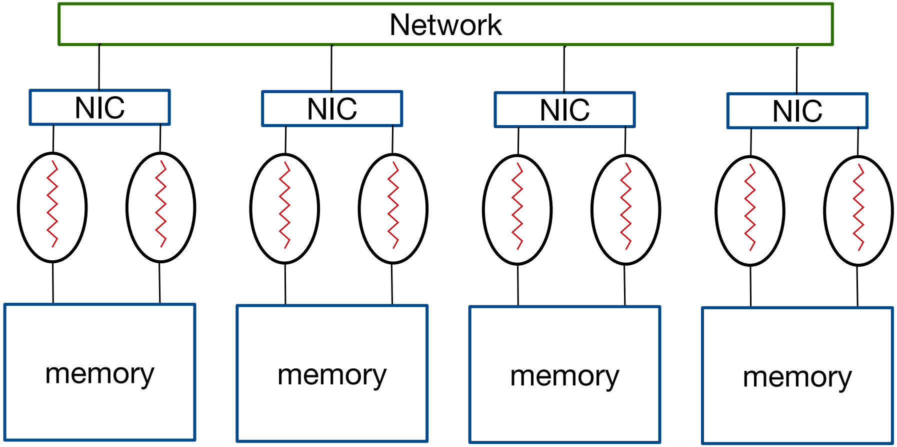
Beyond shared memory

Shared memory becomes expensive and hard to scale beyond a few sockets. Largest NUMA system ever built had 1024 HW threads.
Instead we use *distributed memory*

Distributed memory



Hybrid shared memory/distributed memory



Basic communication mechanism

Hardware:	• Moves data from the memory of one node to the memory of another node. • Provides indication that transfer is complete
Software:	• Calls to move data from one memory to another • Calls to synchronize

Message Passing

Communication is achieved by a matching pair of a *send* and a *receive*:

`send(to, data)` $\longrightarrow$ `recv(from,data)`

The communication

- Moves data (from sender to receiver)
- Synchronizes (receive will complete after send started)

MPI

- MPI (Message Passing Interface) is a Standard Message passing library designed by an open forum that is broadly used in HPC.
- Standardization effort started in 1992, MPI-1 was published late 93. Current version is 3.1 and work is ongoing on version 4.0
- Has C and Fortran binding