CS420 – Lecture 24

Marc Snir

Fall 2018

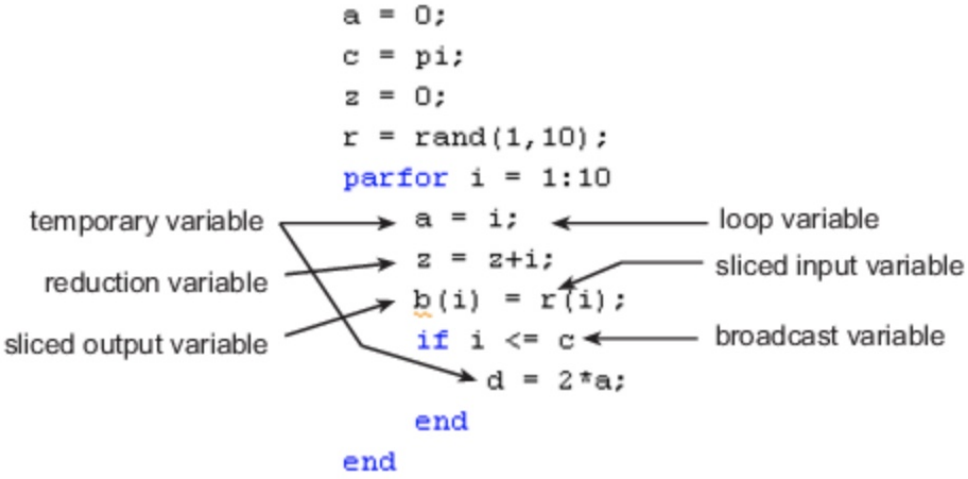
Matlab Parallel Toolkit

Matlab has *limited* support for *coarse grain* parallel constructs

```
parfor i = 1:numprocs
S(:, i) = foo();
end
```

- Executes the function calls independently – no synchronization operations are available.
- Matlab identifies different types of variables

Loop Variables	Loop indices
Sliced Variables	Arrays whose segments are operated on by different iterations of the loop
Broadcast Variables	Variables defined before the loop whose value isused inside the loop, but never assigned inside the loop
Reduction Variables	Variables that accumulates a value across iterations of the loop, regardless of iteration order
Temporary Variables	Variables created inside the loop, and not accessed outside the loop


a = 0;
c = pi;
z = 0;
r = rand(1,10);
parfor i = 1:10
a = i;
z = z+i;
b(i) = r(i);
if i <= c
d = 2*a;
end
end
temporary variable
reduction variable
sliced output variable
loop variable
sliced input variable
broadcast variable

Asynchronous computing – futures

```
% Retrieve pool of workers
p = gcp();
for idx = 1:10
% Evaluate on some worker function foo with input idx and return one value
f(idx) = parfeval(p,@foo,1,idx);
end
% Collect the results as they become available.
fooResults = cell(1,10);
for idx = 1:10
% fetchNext blocks until next results are available.
[completedIdx,value] = fetchNext(f);
fooResults{completedIdx} = value;
fprintf('Got result with index: %d.\n', completedIdx);
end
```

```
F = parfeval(p,fcn,numout,in1,in2,...)
F = parfeval(fcn,numout,in1,in2,...)
```

- Evaluate function `fcn` on some worker in pool `p` (or default pool) with inputs `in1`, `in2`, `...`, returning `numout` arguments
- `F` is a *future*, a "promise" to return eventually some results
- Previous example created a vector of futures in a the `for` loop

```
[idx,B1,B2,...,Bn] = fetchNext(F)
[idx,B1,B2,...,Bn] = fetchNext(F,TIMEOUT)
```

- `fetchNext(F)` waits until a new result is available and then returns the index of that future and the outputs of the function evaluation
- Results may be returned in arbitrary order

Matlab tall arrays

Arrays that may not fit in memory; first dimension can be very long. Tall table is stored on disk and brought to memory in chunks.

```
ds = datastore('airlines/*.csv');
ds.TreatAsMissing = 'NA';
ds.SelectedFormats{strcmp(ds.SelectedVariableNames,'TailNum')} = '%s';
ds.SelectedFormats{strcmp(ds.SelectedVariableNames,'CancellationCode')} = '%s'

tt = tall(ds)
a = tt.ArrDelay
```

Create a tall table and select a column of the table. No computation is done yet

```
% mean delay
m = mean(a, 'omitnan');
% standard deviation
s = std(a, 'omitnan');
%one sigma interval
one_sigma = [m-s,m,m+s];
% now compute
sig1 = gather(one_sigma);
```

The computation is deferred until the `gather` command is encountered.
There will be only one pass over the tall table.

Other Parallel Programming Languages

Multithreading with C++ (or Java)

Spawning and joining thread

```
#include<vector>
#include<thread>

// function to be called by threads
void say_hello(int id) {
 std::out << "Hello␣from␣thread:␣" << id << std:endl;
}

// start master thread
int main(int argc, char *argv[]) {

 const int num_threads = 4;
 std::vector<std::thread> threads; // vector of threads
```

```
// spawn
for(int id = 0; id<num+_threads; id++)
 // emplace the thread objects in vector
 threads.emplace_back(say_hello, id);

// join
for(auto& thread: threads)
 thread.join();
 }
```

- Threads can be spawned recursively by spawned threads
- A parent that spawned a thread needs to either
  - *Join* back the thread (and wait for its completion); or
  - *Detach* the spawned thread

```
...
std::thread t(foo);
t.detach();
...
```

Parent thread does not "know" when child tread will terminate – need to synchronize.

threads can return values

```
int power(int a, int k) {
// function computes a^k, a and b integers
if(k==0)
 return 1;
else if(k%2==0)
 return power(a*a,k/2);
 else
  return a*power(a*a,k/2);
}
```

Need function that returns result in a specified location

```
void rpower(int a, int k, int* result) {
 *result = power(a,k);
}
```

Compute in parallel multiple exponents

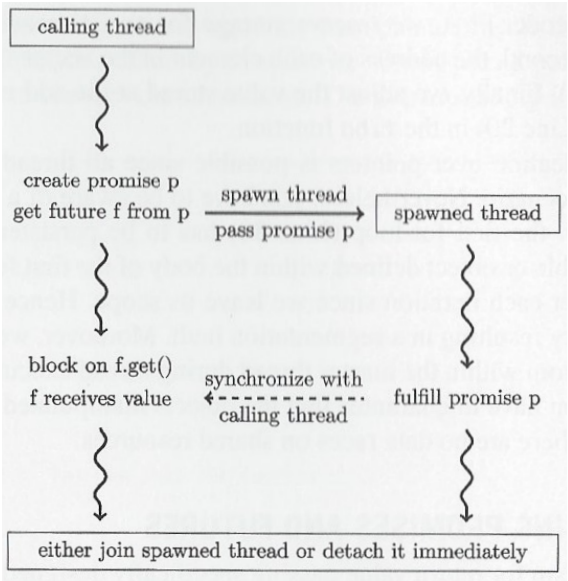
```
#include <iostream>
#include <vector>
#include <thread>

//define power and rpower
...
int main(int argc, char *argv[]) {
 const int num_threads=32;
 const int power = 5;
 std:vector<std::threads> threads;
 std::vector<int> results(num_threads.0) // returned results
```

```
 for (int id=0;id<num_threads; id++)
  threads.emplace_back( apower, id, power,  &(results[id]))

 for(auto& thread: threads)
  thread.join();

 for(const auto& result: results)
  std:cout << result << std:endl;
}
```

0
1
32
243
1024
...

Promises and Futures

- Create *promise* for future value
- Pass promise to another thread
- Create *future* (reference to promise)
- When future is requested, thread blocks until promise is fulfilled



```
// define power as before
...
// define rpower to fulfil a promise
void rpower(int a, int k, std::promise<int> && result) {
 result.set_value(power(a,k)); // fulfill promise
}
```

```
...
int main(int argc, char *argv[]) {
 const int num_threads=32;
 std::vector<std::thread> threads;

 std::vector<std::future<int>> results; // vector of future results

 for(int id=0;id<num_threads,id++) {
  // create promise and store the associated future
  std::promise<int> promise;
  results.emplace_back(promise.get_future());

  // create thread and pass future to created thread
  threads.emplace_back(rpower, id, std::move(promise));
 }
```

```
 // each invocation of get causes a synchronization with producer
 for (auto& result: results)
  std::cout << result.get() << std::endl;
 }

// need to join or detach threads
 for(auto& thread: threads)
  thread.detach();
}
```

Tasks

Similar to OpenMP tasks: can be executed by any thread.
`auto future = std::async(power,a,k)` is creating a task that will run on some thread (possibly the calling thread) and fulfill the future promise with its return value
Task may be scheduled eagerly (spawned as soon as created) or lazily (when a get is invoked on the future). User can control choice
eager: `auto future = std::async(std::launch::async, rpower,a,k)`
lazy: `auto future = std::async(std::launch::deferred, rpower,a,k)`

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

 const int num_values = 128;
 std::vector<std::future<int>> results;

 for(int id=0; id<num_values; id++)
  results.emplace_back(std::async(std::launch::async, power, id, 5));

 for (auto& result : results)
  std::cout << result.get() << std.endl;
}
```

Atomics

- As in OpenMP: can prevent races by protecting conflicting access with locks.
- More efficiently, can use atomics

Histogram

```
#include<iostream>
#include <vector>
#include <thread>
#include <atomic>

void count(std::vector<int>& grades, int first, int next,
    volatile std::atomic<int> *pass, volatile std::atomic<int> *fail) {
 for(int i=first; i<next; i++)
  if(grade[i]>50) (*pass)++;
  else (*fail)++;
}
```

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

 std::vector<int> grades;
 int numgrades;
 const int num_threads = 32;
 std::atomic<int> pass, fail;

// read numgrades and grades
...

 for(int i=0; i<num_threads; i++) {
  int first = numgrades*i/num_threads;
  int last = numgrades*(i+1)/num_threads;
  threads.emplace_back(count, grades, first, last, *pass, *fail);
 }

 for(auto& thread: threads)
  thread.join;
}
std::cout << pass << fail <<std::endl;
```

Atomics

- `std::atomic<T> A` works for any type `T`, including structures or vectors.
- Loads and stores to `A` will be atomic (no races)
  - But of `A` is large then atomic loads and stores will be implemented via locks; short structures do not require locks
  - `is_loc_free()` method can be used to query if a lock is needed.
- for integral types `T`, C++ also supports atomic arithmetic (`a++, a-,a+=x, a-=, a&=, a|=`)

`int atomic_fetch_add (volatile atomic<int>* a, int incr):`
Atomically replaces a with `a+incr` and returns the old value of `a`

Example: queue

Ignore underflow and overflow!

```
...
template<typename T>
class queue
{
private:
 volatile std::atomic<int> head, tail;
 std::vector<T> Q;
public:
 queue(int size) {
  Q(std::vector<T>(size);
 }
 void enqueue(T& entry) {
 int index = fetch_and_add(tail,1);
 Q[index] = T;
 }
  T dequeue(void) {
    int index = fetch_and_add(head, -1);
    return Q[index];
  }
}
```