Introduction to Parallel and Distributed Processing

Shared Memory Challenges

Jaroslaw ‘Jaric’ Zola
http://www.jzola.org/

Shared Memory Challenges

- Parallelism is “easy” when only memory-read operations are involved
- In realistic scenarios tasks have to communicate and have to be synchronized
- Synchronization almost always penalizes efficiency
- Lack of proper synchronization may cause non-deterministic behavior
- Or lead to deadlock/livelock

Race Conditions

- Two tasks try to perform operation on the same memory at the same time, and one involves memory-write

```
int add(int n, int x[]) {
  int S = 0;
  // race on S
  cilk_for (int i = 0; i < n; ++i) S = S + x[i];
  return S;
} // add
```

Race Conditions

- X and Y are shared between tasks and initially $X = Y = 0$, x and y are task local

T1	$X = 1$	$x = Y$
T2	$Y = 1$	$y = X$

x and y both can be 0 or 1

- Modern architectures/compilers can reorder memory access – (almost) no consistency guarantees

Mutual Exclusion

- At the thread level race conditions can be addressed via mutual exclusion
- A thread locks access to the critical section using mutex

```
std::mutex g_mtx;

int add(int n, int x[]) {
  int S = 0;
  // DO NOT TRY THIS AT HOME!!!
  cilk_for (int i = 0; i < n; ++i) {
    g_mtx.lock();
    S = S + x[i];
    g_mtx.unlock();
  }
  return S;
} // add
```

Mutual Exclusion

- Lock/unlock are sequentially consistent and appear as atomic
- Critical section is executed only by one thread at a time
- Mutex can be unlocked only by the thread that locked it
- Hence, it is easy to get disaster:

M_X and M_Y are two mutexes

T1	$M_X.lock$	$M_Y.lock$	$X = 1$	$x = Y$	$M_Y.unlock$	$M_X.unlock$
T2	$M_Y.lock$	$M_X.lock$	$Y = 1$	$y = X$	$M_X.unlock$	$M_Y.unlock$

Deadlocks/Livelocks

- Deadlock: two or more threads block waiting for each other (e.g. on mutex)
- Livelock: two or more threads switch states but remain deadlocked
- Easy to get trapped if not careful: do not lock more than one mutex, always lock in the same order, avoid mutexes if possible

Cilk+ Approach

- Mutual exclusion is too low level and does not guarantee sequential consistency
- Instead, provide hyper-objects to enable lock-free access to shared variables
- General idea: each task sees a “local” version of a global object, final view is achieved via reduction
- User can define new reducers: reducer is always a monoid, i.e. a set with associative operator and identity value

Cilk+ Reducer Example

- Simple sum:

```
int add(int n, int x[]) {
  cilk::reducer<cilk::op_add<int>> S(0);
  cilk_for (int i = 0; i < n; ++i) *S += x[i];
  return S.get_value();
} // add
```

More Interesting Example

```
cilk::reducer_list_append<int> out;

struct node {
    int x;
    node* left;
    node* right;
};

void process(const node* p) {
    out->push_back(p->x);
}

void visit(const node* p) {
    if (p->left != 0) cilk_spawn visit(p->left);
    if (p->right != 0) visit(p->right);
    process(p);
}
```

For Fun

- Implement prefix scan using Cilk+