Parallel Lattice Programming

Session 8—Parallel Lattice Programming

Pierre Talbot

pierre.talbot@uni.lu

21th June 2024

University of Luxembourg

UNIVERSITÉ DU LUXEMBOURG

What in this presentation?

We are going to overview two parallel programming models:

1. Pessimistic Parallel Programming (state of the art).
2. Optimistic Parallel Programming (contribution).

Characteristics of our model

- Lock-free and correct.
- Based on fixpoint over lattices.
- Useful for programming parallel constraint solvers.

Pessimistic Parallel Programming

Running example: parallel maximum

Each thread computes its local max (map), then we compute the max of all local max (reduce).

- Map:

3	2	10	23	2	7	91	1	0	0	42	11	8	1	32

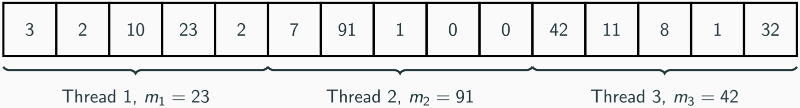
Thread 1, $m_1 = 23$ Thread 2, $m_2 = 91$ Thread 3, $m_3 = 42$

- Reduce: $max([23, 91, 42]) = 91$.

Running example: parallel maximum

Each thread computes its local max (map), then we compute the max of all local max (reduce).

- Map:



- Reduce: $max([23, 91, 42]) = 91$.

Sequential bottleneck: With 100 elements (10 threads), the reduce step takes as much time as the map step.

How to program the reduce step in parallel?

Parallel max

```
/** Suppose as many threads as elements in `data`. */
void max(int tid, const int* data, int* m) {
  if(data[tid] > *m) {
    *m = data[tid];
  }
}
```

Then you run:

```
*m = MIN_INT;
max(0, data, m) || ... || max(n-1, data, m)
```

where `p || q` is the parallel composition.

Parallel max

```
/** Suppose as many threads as elements in `data`. */
void max(int tid, const int* data, int* m) {
  if(data[tid] > *m) {
    *m = data[tid];
  }
}
```

Then you run:

```
*m = MIN_INT;
max(0, data, m) || ... || max(n-1, data, m)
```

where `p || q` is the parallel composition.

Good? No! Data-race.

Parallel max fixed!?

```
/** Suppose as many threads as elements in `data`. */
void max(int tid, const int* data, int* m) {
  if(data[tid] > *m) {
    lock(m) {
      *m = data[tid];
    }
  }
}
```

Parallel max fixed!?

```
/** Suppose as many threads as elements in `data`. */
void max(int tid, const int* data, int* m) {
  if(data[tid] > *m) {
    lock(m) {
      *m = data[tid];
    }
  }
}
```

Good? No!

Can produce wrong results.

Parallel max fixed again!?

```
/** Suppose as many threads as elements in 'data'. */
void max(int tid, const int* data, int* m) {
  lock(m) {
    if(data[tid] > *m) {
      *m = data[tid];
    }
  }
}
```

Parallel max fixed again!?

```
/** Suppose as many threads as elements in 'data'. */
void max(int tid, const int* data, int* m) {
  lock(m) {
    if(data[tid] > *m) {
      *m = data[tid];
    }
  }
}
```

Good? Yes!

But our “parallel” algorithm is now sequential.

OKayish in a map-reduce context

Not that bad if each thread performs work on `data[tid]` and “desynchronize”.

```
void map_then_max(int tid, const int* data, int* m) {
  int r = f(data[tid]);
  lock(m) {
    if(r > *m) {
      *m = r;
    }
  }
}
```

Still, locks are expensive.

Atomics to the rescue (?)

C++11 atomics can unlock lock-free programming for better efficiency :)

```
void max(int tid, const int* data, std::atomic<int>& m) {
  m.max(data[tid]);
}
```

Atomics to the rescue (?)

C++11 atomics can unlock lock-free programming for better efficiency :)

```
void max(int tid, const int* data, std::atomic<int>& m) {
  m.max(data[tid]);
}
```

`std::atomics` does not provide a `max` function.

Atomics to the rescue...

C++11 atomics can unlock lock-free programming for better efficiency :)

```
void max(int tid, const int* data, std::atomic<int>& m) {
  int prev_max = m;
  while(prev_max < data[tid] &&
    !m.compare_exchange_weak(prev_max, data[tid]))
      {}
}
```

Finally OK using a compare-and-swap operation.

Multithreading programming is pessimistic.

For a data race that happens once in million instructions, this model:

- Makes parallel programming painful and difficult.
- Slows down computation.
- Prevents us from thinking with a true parallel mindset.

Optimistic Parallel Programming

Let's be optimistic

Instead of being afraid of data races, let's welcome them as part of the programming model itself.

```
void max(int tid, const int* data, int* m) {
  if(data[tid] > *m) {
    *m = data[tid];
  }
}
```

What happens in case of a data race?

- Suppose two threads with `data = [1, 2]`.
- If a data race occurs, `*m == 1`.
- But if we run `max` again, then we must obtain `*m == 2`.

Let's do extra work only when data races occur (optimistic)

In case of n data races, we run the algorithm $n + 1$ times:

```
int old = *m + 1;
while(old != *m) {
  max(0, data, m) || ... || max(n-1, data, m);
  old = *m;
}
```

This is called the *fixed point loop*.

Fixing optimistic max in parallel

However, for other reason than data races, we still need atomic load and store:

```
void max(int tid, const int* data, std::atomic<int>& m) {
  if(data[tid] > m.load()) {
    m.store(data[tid]);
  }
}
```

Note that, we only need atomic load and store, every other operation can be performed non-atomically.

C++ Abstraction: Lattice Land Project

`lattice-land` is a collection of libraries abstracting our parallel model.

It provides various data types and fixpoint loop:

- `ZInc`, `ZDec`: increasing/decreasing integers.
- `BInc`, `BDec`: Boolean lattices.
- `VStore`: Array (of lattice elements).
- `IPC`: Arithmetic constraints.
- `GaussSeidelIteration`: Sequential CPU fixed point loop.
- `AsynchronousIteration`: GPU-accelerated fixed point loop.
- ...

```
void max(int tid, const int* data, ZInc& m) {
  m.tell(data[tid]);
}
AsynchronousIteration::fixpoint(max);
```

`https://github.com/lattice-land`

Conclusion

Data races occur rarely, so we should avoid working so much to avoid them.

Further properties of the model

A Variant of Concurrent Constraint Programming on GPU (AAAI 2022)[1].

- Correct: Proofs that $P; Q \equiv P||Q$, parallel and sequential versions produce the same results.
- Restartable: Stop the program at any time, and restart on partial data.
- Modular: Add more threads without fear of breaking existing code.
- Weak memory consistency: Very few requirements on the underlying memory model $\Rightarrow$ wide compatibility across hardware, unlock optimization.

[1] `http://hyc.io/papers/aaai2022.pdf`

Interlude: atomicity of load and store...

At start, suppose `x,y = 0`.

T1	T2
`x` $\leftarrow$ `512`	`x` $\leftarrow$ `1`

What are the possible outcomes?

Interlude: atomicity of load and store...

At start, suppose `x,y = 0`.

T1	T2
x ← 512	x ← 1

What are the possible outcomes? $x = 512$, $x = 1$ and... $x = 513$.

Really? 513?

Assignment is not necessarily atomic. View x as an array of two bytes `x[0]x[1]`:

```
T2: x[0] ← 0
T1: x[0] ← 1  (x = 512)
T2: x[1] ← 0
T1: x[1] ← 1  (x = 513)
```

But in practice, most architectures (x86, x64, ARM, ...) will atomically load and store 32 bits values (if correctly aligned).

Interlude II: atomicity of load and store...

A thread notifies another that it should stop (initially `b = 1`):

```
while(b) { f(); }    ||      g(); b = 0;
```

The compiler is allowed to optimize the first part as:

```
if(b) {
  while(1) { f(); }
}
```

provided *f* does not modify *b*.

Indeed, in C++ any concurrent access to shared variable, with at least one write, is **undefined behavior**.

Conclusion: We still need atomic load and store, for the correctness of our model.