Optimizing Nondeterminacy: Exploiting Race Conditions in Parallel Programs

William S. Moses (wmoses@mit.edu)

MIT CSAIL

Parallelism in the Compiler

A parallel IR (i.e. Tapir [2], HPVM [1], etc) allows for better optimization and analysis of parallel programs and shared parallel infrastructure (i.e. compile OpenMP to Cilk, single place to implement parallel optimizations).

We can place optimizations into three categories:

- **Serial Optimizations**: Don't use parallelism information at all. Most of these optimizations are enabled by a parallel IR.
- **Scheduling Optimizations**: Rearrange the parallelism in a computation
- **Nondeterminacy Optimizations**: Exploit the undefined behavior that exist in parallel programs to improve performance.

This poster has three contributions:

- A theoretical framework to ensure correctness of optimizations on parallel programs, as well as to provide semantics for a parallel IR;
- The development of optimizations that exploit nondeterminacy along with proofs of correctness;
- An in-progress implementation of said optimizations

Parallel Execution Environment

Let us consider the types of semantics one can expect from a parallel framework by looking at the time-orderings of code in Figure 1.

If we execute this on a strict runtime on one core, we only see the serialization execution of the program. This model is called the **serial execution model**

$$\mathcal{P}_{\text{serial}}(code) = \{[A, B1, B2, C1, C2, D]\} \tag{1}$$

If we execute this on a relaxed runtime on one core, we see reorderings of the parallel tasks. This model is called the **reordering execution model**

$$\mathcal{P}_{\text{reorder}}(code) = \{[A, B1, B2, C1, C2, D], [A, C1, C2, B1, B2, D]\} \tag{2}$$

If we execute this on a relaxed runtime on many cores, we expect an interleaving of the tasks. The possible executions are valid topological ordering of tasks. This model is called the **interleaving execution model**

$$\mathcal{P}_{\text{inter}}(code) = \{[A, B1, B2, C1, C2], [A, C1, C2, B1, B2], [A, B1, C1, B2, C2], [A, B1, C1, C2, B2], [A, C1, B1, B2, C2], [A, C1, B1, C2, B2]\} \tag{3}$$

Theorem 1: *The reordering model, permitting inlining and serialization, is the same as the interleaving model.* We can construct any interleaved execution in the following manner: first, inline all function calls. Next "reorder" the parallel tasks such that the parallel task executes after any of the continuation tasks that precede the first subtask. Serialize the subtask out of the top of parallel function. Repeat until the parallel task is empty. The resultant program will be a serial program whose execution is the same as the desired interleaved execution.

Optimization Theory

```
void B() {
    B1();
    B2();
}

void C() {
    C1();
    C2();
}
```

```
void program() {
    A();
    cilk_spawn B();
    C()
    cilk_sync;
    D();
}
```

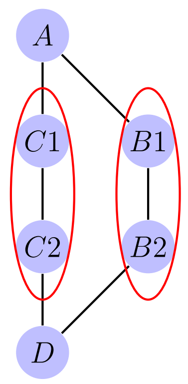

Figure 1. A parallel program in Cilk and corresponding series-parallel DAG.

To create sound optimizations, we must develop an acceptability metric that deems whether one parallel program is an acceptable replacement for another program.

One such metric considers if all of a replacement program's executions could have happened in the original program is called the **subset metric**.

$$\mathcal{P}(replacement) \subseteq \mathcal{P}(replacee) \tag{4}$$

The **serial subset metric** adds the constraint that the serial execution is the same.

$$\mathcal{P}(replacement) \subseteq \mathcal{P}(replacee) \text{ and } \mathcal{P}_{\text{serial}}(replacement) = \mathcal{P}_{\text{serial}}(replacee) \tag{5}$$

The **fair subset metric** adds the constraint that the probability of any given execution of the replacement is close to the probability of that execution of the replacee.

$$\text{subset and } \forall_{e \in \mathcal{P}(replacement)} |P(e|replacement) - P(e|replacee)| < \Delta \tag{6}$$

Scheduling Optimizations

- **Serialization**: Take (part of) a parallel program and runs it program serially to reduce the overhead from the runtime or employ more efficient operators (strength reduction). Many parallel optimizations are simply serialization: coarsening, spawn-switching, etc.

```
pfor(int i=0; i<N; i++) {
  work(i);
}

pfor(int i=0; i<N; i+=M) {
  for(int j=i; j<N && j<i+M; j++)
    work(j);
}
```

```
spawn {
  if (x) code();
}

if (x) {
  spawn code();
}
```

```
x = a + b;
spawn {
    f(x);
}

spawn {
    x0 = a + b;
    f(x0);
}
```

Figure 2. Loop coarsening on the left, spawn unswitching in the middle, and parallel region expansion on the right.

- **Parallel Region Expansion**: Move code into a parallel region.
- **Synchronization Motion**: Remove tastwait barriers where possible, moving code around them or making them less expensive where possible.

Nondeterministic Optimizations

Nondeterministic LICM/Unswitching: LICM/Unswitching can be run on a loop that could modify memory used later in the loop. This is permissible as you could choose an ordering where you execute up to that instruction for all of the loops interations, and then you run the remaining instructions in the loop for the remaining iterations.

```
using Eigen::MatrixXd;
double get(MatrixXd& m, int j);

double test(MatrixXd m) {
  double sum = 0;
  size_t len = m.rows();
  pfor(int i=0; i < len; i++)
    sum += m.size() / get(m, i);
  return sum;
}
```

```
using Eigen::MatrixXd;
double get(MatrixXd& m, int j);

double test(MatrixXd m) {
  double sum = 0;
  size_t len = m.rows();
  pfor(int i=0; i<len; i++)
    sum += len / get(m, i);
  return sum;
}
```

Figure 3. Nondeterministic LICM allowing the call to `m.rows()` to be moved out of the loop even though `get` could potentially modify m.

Loop interchange/reordering: The iterations of a parallel loop can be reordered in any way and iterations of parallel loops can be interchanged without additional checks.

Nondeterministic Vectorization: A parallel loop can always be vectorized by choosing an interleaving where all of the loop's first instructions are executed first, then all of the loop's second instructions, and so on.

Nondeterministic Mem2Reg/SROA: Loads to memory locations set by a previously spawned task can be set to either a value of a store in or before the continuation, or a store in the spawned task if this assertion isn't used by a later store. This is especially helpful in tandem with other optimizations (i.e. CSE/DCE) as we can now choose which value is preferable to replace with.

Parallel Models in Practice

When choosing creating a parallel IR, one must trade off between its expressibility (is ability to represent the semantics parallel languages) and the ability of the compiler to optimize programs. This analysis of optimizations and parallel models hopes to inform choices of parallel IR's by bringing awareness to the limitations of certain models.

- Any model with a serialization metric is forbidden from doing the nondeterministic optimizations listed as they may change the serial execution.
- A fairness metric may (specifically including the one provided) inhibits both parallel and existing serial optimizations on purely serial code as any optimization changing the timing of a program may substantially change the rate that race conditions resolve a specific way.

References

[1] Maria Kotsifakou, Prakalp Srivastava, Matthew D Sinclair, Rakesh Komuravelli, Vikram Adve, and Sarita Adve. Hpvm: heterogeneous parallel virtual machine. In *ACM SIGPLAN Notices*, volume 53, pages 68–80. ACM, 2018.

[2] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. Tapir: Embedding fork-join parallelism into LLVM's intermediate representation. In *Proc. 22nd Symposium on Principles and Practice of Parallel Programming*, 2017.