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 paper contains 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 programs, as well as to provide semantics for a parallel IR;
- 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-ordining of code in Figure 1.

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

\[
P_{\text{serial}}(\text{code}) = \{(A, B_1, B_2, C_1, C_2, D, 0)\}
\]

If we execute this on a relaxed runtime on one core, we see interlacing of the tasks. The possible executions are valid topological ordering of tasks. This model is called the **reordering execution model**

\[
P_{\text{reorder}}(\text{code}) = \{(A, B_1, B_2, C_1, C_2, D, 0, A, C_1, C_2, B_1, B_2, D)\}
\]

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. Serializing 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

**Optimization Theory**

![Figure 1. A parallel program in Cilk and corresponding series-parallel DAG.](image)

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**

\[
P(\text{replacement}) \subseteq P(\text{replacee})
\]

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

\[
P(\text{replacement}) \subseteq P(\text{replacee}) \wedge P_{\text{serial}}(\text{replacement}) = P_{\text{serial}}(\text{replacee})
\]

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.

\[
\text{subset and } \forall_{P(\text{replacement}) \subseteq P(\text{replacee})} |P(\text{replacement}) - P(\text{replacee})| < \Delta
\]

#### Scheduling Optimizations

- **Serialization**: Take (part of) a parallel program and run 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.

\[
pfor(int \ i=0; i<N; i++)\{
\text{spawn (}\ x = a + b; \text{)}
\text{spawn (}\ f(x); \text{)}
\}
\]

- **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 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 iterations, and then you run the remaining instructions in the loop for the remaining iterations.

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

Figure 3. Nondeterministic LICM allowing the call to \text{a.row()} to be moved out of the loop even though \text{get} could potentially modify \text{m}.

#### Parallel Models in Practice

When choosing creating a parallel IR, one must trade off between its expressivity (its ability to represent the semantics of parallel programs) and the ability of the compiler to optimize programs. This analysis of optimizations and parallel models hopes to inform choices of parallel IRs 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
