Enforcing Sequential Consistency in SPMD Programs with Arrays

Wei Chen
Arvind Krishnamurthy
Katherine Yelick
Motivation

- Compiler and hardware optimizations are legal only if the resulting execution is indistinguishable from one that follows program order.

- In terms of memory accesses:
  - Uniprocessor: Never reorder accesses with data-dependencies.
  - Multiprocessor: Not enough to just satisfy local dependencies.

- Programmers intuitively rely on the notion of Sequential Consistency for parallel programs.
Examples of SC violation

- If we consider only one thread,
  - Access to A,B can be reordered
  - But means both threads can enter critical section
- SC prevents this by restricting reordering on shared memory accesses
  - Reordering not allowed on either T1 or T2, because the other thread may observe the effect
Problem with Sequential Consistency

- SC is easy to understand, but expensive to enforce
- Naïve approach – insert fences between every consecutive pair of shared accesses
- Bad for performance:
  - Most compiler optimizations (pipelining, prefetching, code motion) need to reorder memory accesses
  - Fences are expensive (drain processor pipeline)
- Especially for GAS (global address space) languages
  - Alternative to MPI for distributed memory machines
  - Threads can read and write remote memory directly
  - Overlapping communication overhead is critical
- Goal: Find the minimal amount of ordering needed to guarantee sequential consistency
Problem Statement

- **Input**: SPMD program with program order $P$
  - Represented as a graph, with shared accesses as its nodes
  
  **Delay**: for $u$, $v$ on the same thread, guarantees that $u$ happens before $v$
  - i.e. there is a fence between $u$ and $v$
  - A “delay set” is a subset of $P$

- **Output**: Find the minimal delay set $D$ s.t. any execution consistent with it satisfies SC

- Use the idea of **Cycle Detection**
Conflict Accesses: for $u$, $v$ on different threads
- $u$, $v$ are conflicting if they access the same shared variable, with at least one write

A parallel execution instance $E$ defines a happens-before relation for accesses to the same shared memory location.
- $E$ – memory centric, $P$ – thread centric

$E$ is correct iff it’s consistent with $P$
- In other words, can’t have cycles in $P \cup E$

But we don’t know $E$
- Use $C$, the set of conflict edges, to approximate $E$
Example

- SC restriction: \((x, y)\) on P2 can’t be \((0, 1)\)
- Analysis finds a critical cycle \(\rightarrow\) enforces all delays on the cycle.
- Figure-eight shape – only way to get cycle for straight-line code

![Diagram showing a figure-eight shape with P1 and P2 processes, where P1 writes X = 1, then writes Y = 1, and P2 reads X and Y. P edges and C edges are also shown with delays.]
Example II

- No restrictions by SC: \((x,y)\) on P2 can be either \((0,0), (0,1), (1,0), (1,1)\)

- Analysis finds no cycles in the graph \(\rightarrow\) no delays are necessary

\[(\text{initially } x = y = 0)\]

- P1
  - Write \(X = 1\)
  - Write \(Y = 1\)

- P2
  - Read \(X\)
  - Read \(Y\)

\(\text{P edges}\)
\(\text{C edges}\)
Cycle Detection for SPMD programs

- Krishnamurthy and Yelick created polynomial time algorithms for SPMD programs
- Keep two copies $P_L$ and $P_R$ of $P$
- Add internal $C$ (conflict) edges to $P_R$
- Remove all edges from $P_L$
- Consider the **conflict graph** $P_L \cup C \cup P_R$
  - For each pair $(u_L, v_L)$ in $P$, check if we can find a back-path $(v_L, u_L)$
  - Algorithm takes $O(n^3)$ time – one depth-first search for each node ($n$ is number of shared accesses)
  - Computes minimal delay set for programs with scalar variables
Algorithm at Work

While (turn != MYPROC);
numTrans++;
fund = c;
turn++;

The Code

P edges

C edges

$P_L$

Read turn

Read numTrans

Write numTrans

Write fund

Write turn

$P_R$

Read turn

Read numTrans

Write numTrans

Write fund

Write turn
Faster SPMD Cycle Detection

- For each $P$ edge $(u_L, v_L)$, we want to know if $u_L$ is reachable from $v_L$ in the conflict graph.
- Since graph is static, we can use *strongly-connected-components* to cache the reachability.
  - For $(u_L, v_L)$, back-path $(v_L, u_L)$ exists $\leftrightarrow C(u_L)$ reachable from $C(v_L)$ (or they are the same).
- A $O(n^2)$ running time.
- Compute same delay set as Krishnamurthy and Yelick’s Algorithm.
Extending Cycle Detection to Array Accesses

- Previous algorithm has many false delays due to array accesses in loops
  - Cycle detection finds backpath from S2 to S1
  - But each S1, S2 accesses different memory location, and
  - Threads iterate the loop in the same order → no SC violation
- We can improve the accuracy by incorporating array indices into our analysis

```c
for (i = 0; i < N; i++) {
    A[i] = 1;   (S1)
    B[i] = 2;   (S2)
}
```
Imagine if a loop is fully unrolled

- All cycles have figure-eight shape
- A conflict edge means two array accesses have the same subscript
- For a cycle of \((u_L, v_L)\) with backpath \((v_L, v_R, \ldots, u_R, u_L)\):
  \[\text{index}(v_L) = \text{index}(v_R), \text{index}(u_L) = \text{index}(u_R), \\text{iter}(v_L) \geq \text{iter}(u_L), \text{iter}(u_R) \geq \text{iter}(v_R)\]

- Iteration information is encoded in edge direction
- How do we incorporate information about the index into the conflict graph?
Augmenting Conflict Graph with Weights

- For edge \((A[f(i)], B[g(i)])\), assign its weight to be \(g(i) - f(i)\)
- For loop back edge, use loop increment
- Conflict edges always have zero weight

**New Goal:** for \((u_L, v_L)\), find backpath \((v_R, u_R)\) s.t.
\[W(u_L, v_L) + W(v_R, u_R) = 0\]
Three Polynomial-time Algorithms

- **Zero cycle detection**
  - When all edge weights are constants
  - Graph theory to detect zero cycles (simple and non-simple)
  - $O(n^3)$ if no negative cycles, $O(n^5)$ otherwise

- **Data-flow analysis**
  - Use the signs of the edges to approximate answer
  - $O(n^3)$ time

- **Integer Programming with 4 variables**
  - Useful for generic affine terms
  - For each $(u, v)$, find all possible pairs of $(C(u), C(v))$
  - Create linear systems with 4 equations
  - $O(n^4)$ time
Data-flow Analysis Approximation

- Check if a cycle must have non-zero weight
  - $(u_L, v_L)$ is **NOT** a delay if
    - $\text{sgn}(u_L, v_L) \cap \text{sgn}(v_L, u_L)$
      - is in $\{+,−\}$
- $O(n^3)$ time
  - Only $3 \times n$ initial conditions for data-flow analysis

• $\text{OUT}(B) = \text{IN}(B)$
• $\text{IN}(B) = \prod (\text{Sgn}(P,B) \cap \text{OUT}(P))$, where $P$ is in $\text{pred}(B)$.
if (MYTHREAD == 1)
  for (i = 0; i < N; i+= 3) {
    A[i] = c1;    \hspace{1em} \text{(S1)}
    B[i+1] = c2; \hspace{1em} \text{(S2)}
  }
if (MYTHREAD == 2)
  for (j = 2; j < N; j+=2) {
    B[j] = c3;    \hspace{1em} \text{(S3)}
    A[j-2] = c4; \hspace{1em} \text{(S4)}
  }

\begin{itemize}
  \item S1 -> S4: \hspace{1em} i = j - 2
  \item S1 -> S2: \hspace{1em} i’ = i + 3k_1, \hspace{1em} k_1 \geq 0
  \item S2 -> S1: \hspace{1em} i = i’ + 3k_1, \hspace{1em} k_1 \geq 1
  \item S2 -> S3: \hspace{1em} i’ + 1 = j’
  \item S3 -> S4: \hspace{1em} j = j’ + 2k_2, \hspace{1em} k_2 \geq 0
  \item S4 -> S3: \hspace{1em} j’ = j + 2k_2, \hspace{1em} k_2 \geq 1
\end{itemize}

- For (S1,S2), any cycle must include (S2,S3), (S4, S1)
- System has no solution \(\rightarrow\) (S1, S2) is not a delay
- Zero cycle \(\rightarrow\) no delay, data-flow \(\rightarrow\) delay
Algorithm Evaluation

- **Speed:** data-flow > IP4 > zero
- **Accuracy:** zero > IP4 > data-flow
- **Applicability:** data-flow > IP4 > zero
- **Ease of implementation:** data-flow > zero > IP4
- **Possible implementation strategy:**
  - Use data-flow for most cases
  - Use zero cycle detection when it’s applicable, and for “hot spot” of the program
  - Use Integer Programming to deal with complex affine terms
Conclusion

- Cycle detection is important for efficiently enforcing sequential consistency
- We present a $O(n^2)$ algorithm for handling scalar accesses in SPMD programs
- We present three polynomial time algorithms to support array accesses
- Plan to experiment our techniques on UPC, a global address space SPMD language
  - Communication scheduling (prefetching, pipelining)