Flexible Preconditions: A Model for Efficient Macro-Dataflow Execution

Dragoș Sbîrlea*
Alina Sbîrlea+
Kyle Wheeler*
Vivek Sarkar+

+ Rice University
* Sandia National Laboratories
Agenda

- Introduction
- Motivation
- The flexible preconditions model
- Task creation optimizations
- Qthreads CnC
- Implementation
- Results
- Conclusion
Macro-dataflow task creation

• Two strategies
  – Eager task creation
  – Strict preconditions

• Eager task creation
  – Tasks are spawned as soon as they are prescribed
  – Once they start to execute, they may encounter an unavailable dataflow dependence

  – Strategies for waiting:
    – Blocking
    – Abort-and-restart
    – Continuation
Macro-dataflow task creation

- **Two strategies**
  - Eager task creation
  - Strict preconditions

- **Strict preconditions**
  - Dataflow dependences must be declared before tasks start running
  - Limited expressiveness
    - Data dependent accesses are forbidden (if values tagged $a$, $b$ are preconditions, then $b$ cannot be $f(\text{value}(a))$
    - This restriction can be overcome by creating a new task at each data dependent access point, but doing so may incur extra overheads
Task creation in previous work

**Eager spawning**
- Nabbit
- TFLux
- Intel CnC*
- Rice HJ CnC*

**Strict preconditions**
- SMPSs
- KAAPI
- Rice HJ CnC*
- Rice HC CnC*

* these models have multiple runtimes available
Motivation

• Which of the two task creation strategies is better?
• This decision seems more or less arbitrary in previous work.

• No complete study of the implications on:
  
  Performance
  +
  Memory footprint
  +
  Optimizations enabled
  +
  Programmability

• Can we get the advantages of both models?
The flexible preconditions model

- What if we allow partial specification of the preconditions of tasks?

- This way we could get:
  - performance and memory behavior of strict preconditions
  - programmability of eager spawning
Task States

(a) Eager

(b) Strict Preconditions

(c) Flexible Preconditions
Optimizations

- Optimizations drive the task spawning performance!
- Eager spawning
  - Can use lazy allocation for task resources (allocate when first ran)
- Strict preconditions
  - No need to support blocking behavior in tasks
    => no synchronization on dataflow access
    => task state can use the resources of the thread

<table>
<thead>
<tr>
<th>Task spawning method</th>
<th>Task state memory*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eager</td>
<td>$O(\text{prescribed tasks}(t) - \text{finished tasks}(t))$</td>
</tr>
<tr>
<td>Eager + lazy allocation</td>
<td>$O(\text{blocked tasks}(t) + \text{running tasks}(t))$</td>
</tr>
<tr>
<td>Strict preconditions</td>
<td>$O(\text{threads})$</td>
</tr>
<tr>
<td>Flexible preconditions</td>
<td>Between eager and strict</td>
</tr>
</tbody>
</table>

* an additional $O(\text{prescribed tasks}(t) - \text{finished tasks}(t))$ is required for tag storage
Causes for memory and time performance difference

• Task heap usage
  – If large, it makes task state insignificant

• Task stack size
  – Small amount of state is unlikely to lead to performance degradation

• Available memory

• Application task spawning pattern
  – If most tasks are spawned at the beginning, memory allocation bottleneck
From eager/flexible to strict

- Porting applications from eager to strict is possible
  - Need to split tasks
  - Number of tasks created is equal to the data dependence chain length formed by dataflow dependences
  - Additional data types needed for tags as data is explicitly inserted in tags as opposed to obtained by accessing items
Qthreads CnC

• C++ based CnC implementation
  – Includes get-counts support
  – Three runtimes:
    • eager execution
    • strict preconditions,
    • flexible preconditions

• External API similar to Intel CnC
  – Support for Intel CnC benchmarks

• Using Sandia Qthreads library
• Qthreads threading library
  – Lightweight
  – Locality-aware
  – User-level threads
• Opportunity for very low overhead
• We used
  – work-stealing scheduler for task execution
  – synchronization primitives for dataflow
    • Support for both strict and eager execution
    • Full-empty bits primitive
Eager execution in Qthreads Concurrent Collections

- **Fibonacci task code:**

  ```
  1. int fib::execute(int tag, fib_graph g) {
  2.     // get previous 2 results
  3.     fib_type f_1;
  4.     g.fib_values.get(tag - 1, f_1);
  5.     fib_type f_2;
  6.     g.fib_values.get(tag - 2, f_2);
  7.     g.fib_values.put(tag, f_1 + f_2);
  8.     return CnC::CNC_Success;
  9. }
  ```

- **Fibonacci main program code:**

  ```
  1. fib_graph g;
  2. g.m_tags.put( 1, 1 );
  3. g.m_tags.put( 0, 0 );
  4. for( int i = 2; i <= n; ++i )
  5.     g.fib_tags.put( i );
  6. g.wait();
  7. fib_type res2;
  8. g.m_fibs.get( n, res2 );
  ```
Strict Preconditions in Qthreads Concurrent Collections

- Needs extra code declaring the dataflow dependencies

```cpp
1. aligned_t** fib::get_dependences(int tag,
2.     fib_graph g, int& no) {
3.     no = 2;
4.     aligned_t** read = malloc(...);
5.     g.fib_values.wait_on(tag-1, &read[0]);
6.     g.fib_values.wait_on(tag-2, &read[1]);
7.     return read;
8. }
```
Qthreads CnC Implementation

• Needed to build concurrent dictionary
  – Support objects as keys
  – Alternatives tested
    • Static number of buckets, one lock per bucket
    • Dynamic number of buckets, one lock per bucket
    • Lockfree split-ordered-lists (Shalev and Shavit [1])
    • Lockfree concurrent tries (Prokopec et al. [2])

Results

• System:
  – 4 socket 32 core
  – Intel Nehalem X7550.
  – 128GB RAM per socket (total 512GB)

• Methodology
  – Performance: reporting average time over 10 runs
  – Memory: reporting “time –v” Unix tool output

• Applications
  – Cholesky, MatrixInvert, Primes
  – File Concatenation
Cholesky speedup and memory consumption

• Flexible preconditions
  – Items declared = items read

• Bottleneck for eager execution: memory allocation for tasks
• Flexible preconditions perform on par with strict preconditions
MatrixInvert speedup and memory consumption

- **Flexible preconditions**
  - Items declared = items read

- **Bottleneck for eager execution:** memory allocation for tasks
- **Flexible preconditions** perform on par with strict preconditions
Reduction

- Application kernel: short-circuit reduction
  - Flexible: 1 item declared out of max. N items read
  - Strict: waits for items that may not be read

- Flexible close to Eager
File Concatenation

- Concatenates a set of files by building a balanced tree of concatenation operations
- Data-dependent gets for a chain of length two
  - Porting to the strict runtime would need three times as many tasks

```
1. struct inode inode1 = graph.inodes.get(tag.inodeId1);
2. struct block firstDataBlock =
   graph.blocks.get(inode1.blocks[0]);
3. copy(firstDataBlock);
4. ...
5. struct block indirectBlock =
   graph.blocks.get(inode1.extra_blocks_ids);
6. struct block firstIndirectDataBlock =
   graph.blocks.get(indirectBlock[0])
7. copy(firstIndirectDataBlock);
8. ...
```
File Concatenation Results

- Strict runtime cannot be used because of data-dependent gets

- For flexible preconditions:
  - Items declared: 1
  - Items read: 3+ (depending on file length, up to 256)

- Flexible outperforms eager (memory bottleneck)
Conclusion

• We analyzed the memory and performance implications of the choice between the two task spawning strategies

• The proposed flexible preconditions model offers the optimum balance of the two in
  – Performance
  – Memory footprint
  – Programmability
Future work

• Automatic choice of which inputs should be preconditions

• More programmer-friendly API for preconditions
Accessing the Code

• Qthreads GIT repository:
  – http://code.google.com/p/qthreads/

• QtCnC is a branch in the same repository

• Please use them as pair (same revision for both)
Backup slides
Porting Microbenchmark

Execution Time

- **X axis**: length of data-dependence chain
- **All tasks** are empty
Performance comparison with Intel CnC

![Graph showing speedup comparison between QtCnC (strict), QtCnC (eager), and Intel CnC across different thread counts. The graph demonstrates a linear increase in speedup as the number of threads increases.]