The Flexible Preconditions model for
Macro-Dataflow Execution

Dragoș Sbîrlea
Rice University
Email: dragos@rice.edu

Alina Sbîrlea
Rice University
Email: alina@rice.edu

Kyle B. Wheeler
Micron Technologies
Email: k wheeler@micron.com

Vivek Sarkar
Rice University
Email: vsarkar@rice.edu

Abstract—In this paper, we propose the flexible preconditions model for macro-dataflow execution. Our approach unifies two current approaches for managing task dependences, eager execution vs. strict preconditions. When one of the two outperforms the other, flexible preconditions can always attain, and possibly surpass, the performance of the better approach.

This work focuses on the performance of parallel programming models based on macro-dataflow, in which applications are composed of tasks and inter-task dependences. Data-flow models usually make a choice between specifying the task dependences before task creation (as strict preconditions), or during task execution, when they are actually needed (eager execution). This paper shows how the choice between eager execution and strict preconditions affects the performance, memory consumption and expressiveness of macro-dataflow applications.

The flexible preconditions model is sufficiently flexible to support both eager execution and strict preconditions, as well as hybrid combinations thereof. This capability enables programmers and future auto-tuning systems to pick the precondition combination that yields the best performance for a given application. The experimental evaluation was performed on a 32-core SMP, and is based on a new macro-dataflow implementation, QtCnC, that supports eager execution, strict preconditions and flexible preconditions in a single framework. QtCnC is an implementation of the CnC model on the QThreads library.) For applications where all dependences are known ahead of time, flexible and strict preconditions execute up to 56% faster than eager execution (for the benchmarks and platform used in our study). On the other hand, for applications where the complete set of per-task dependences is determined after the tasks are spawned, flexible preconditions and eager execution perform up to 38% better than strict preconditions.

I. INTRODUCTION

Many programming models typically limit the available parallelism to that exploitable from domain decomposition. In contrast, the dataflow model [1] is capable of exposing far greater levels of parallelism across statements, loops and procedures. Macro-dataflow models [2], [3] further extend this promise by supporting dataflow tasks expressed at granularities suitable for modern machines.

Macro-dataflow models usually consist of parallel tasks and some mechanisms for dataflow communication and synchronization among them. When implemented for multicore architectures, macro-dataflow models usually rely on a threading library (such as Pthreads, used in TFlux [4]), a task

1This work was done while the author was employed by Sandia National Laboratories.
The tasks optimistically start executing, but may have to wait if their dataflow dependences are unavailable when needed. For thread-pool based schedulers (which include work-sharing and work-stealing policies [9, 12]), eager implementations must provide a mechanism for worker threads to recover from missing dependences and continue executing the remainder of the task via a continuation or re-execution.

The main advantage of the eager model is that it is easy to understand and offers full flexibility for describing applications. Another advantage is that, if a task starts with a computation that doesn’t require dataflow dependences (or needs only some of them), that part of the task can execute without waiting for all input data to be produced. Such an example is an AND-reduction, where an eagerly spawned task may be able to finish execution using only a few of the inputs available, without the need to wait for all data to become ready.

A common problem with eager task creation approaches is that the memory and other resources needed by the tasks to execute must be allocated as soon as the task is produced, and may not be released when the task is deferred. This resource allocation puts pressure on the memory and runtime systems.

B. Strict preconditions

An alternative to eager spawning is to use the strict preconditions model in which tasks are spawned only when all their dataflow dependences are satisfied. We call this approach strict precondition because the availability of inputs becomes a precondition to running tasks, while strict refers to the fact that all input data must be available before a task starts executing.

For this approach, dependences must be known a priori; this restricts the expressiveness of the programming model and has implications on the API exposed to the user. For example, optional or data-dependent inputs are usually forbidden with strict preconditions, and have to be implemented by spawning data-dependent continuations as new tasks. Applications that rely on short-circuit reduction as a performance optimization may not be expressible in this model. Additionally, these models need a model to determine when all a task’s inputs are available these models need a mechanism to determine when a task is ready to be scheduled. For example, this support can be provided [3, 6] by maintaining one task-descriptor per task, each with an atomic counter whose value decreases when each dependence is satisfied. When the counter reaches zero, the corresponding task can safely be spawned. These counter decrements are a source of overhead and possible contention. Strict preconditions offer better performance than eager execution if the overhead of deferring and re-executing tasks in the eager runtime is larger than that of this atomic-counter based synchronization.

A second advantage is that, since all inputs are known to be available, data can be accessed without the possibility of blocking during task execution, thus avoiding the synchronization overhead for accessing those inputs. Another argument for using strict preconditions is its lower memory consumption. With all inputs known a priori, tasks can never suspend and, as a result, there is no additional memory requirement to save their intermediate state.

C. Dependency handling in previous work

This section surveys existing projects and shows that each model chooses either eager execution or strict preconditions. Some projects make the choice depending on project specific goals (such as the need to support heterogeneous CPU+accelerator execution [8, 13], but each model has its own advantages and disadvantages which makes them perform better on different classes of applications. One of the goals of this paper is to shine light on the performance and memory implications of the choice between the two, so that future projects can make a more informed decision.

Many models that evolved from task parallelism as opposed to dataflow tend to prefer eager semantics for task creation [10]. For example, both versions of Nabbit (for static and dynamic graphs, respectively) [6] use eager task creation. TFLux [4] also uses eager task creation. For this reason neither of these systems need to know the dependences of a task at task creation time. In contrast, the SMPSs [14] and Habanero datadriven-task [8] models both use strict preconditions resulting in a straightforward API for specifying preconditions. (One difference between the two is that SMPSs requires that a default sequential execution be provided, but the Habanero DDT model does not have that constraint.) Kaapi [15] is another model which opts for strict preconditions, proposing a model suitable for cluster execution as well; it builds upon Athapascan [16] which takes the same approach.

The Intel Concurrent Collections (CnC) implementation [5] offers both an eager approach and a preconditions based approach using the abort-and-restart mechanism. An alternative implementation of the CnC model, CnC-HJ [17], allows the programmer to pick between eager task creation (with different runtimes supporting continuations, blocking and abort-and-restart) and strict preconditions. CnC-HC [8] takes a similar approach by offering two alternative runtimes (strict preconditions and eager task creation), but also targets heterogeneous platforms. CnC-HC motivates the need for strict preconditions for heterogeneous computing by showing that, for GPUs, the high cost of communication can be made more efficient by grouping all dataflow dependences in a single data transfer performed before the task is started.

III. Flexible Preconditions

We propose the flexible preconditions model which is a combination of the eager task creation and the strict preconditions models: tasks do not start until the items listed as preconditions are available, but are able to wait for additional dataflow inputs that are identified during task execution. This results in behavior identical to the eager model if the preconditions list is empty and identical to the strict model when the list includes all dataflow dependences. It also opens up the possibility of supporting a wide set of intermediate behaviors not covered by either model.

In the flexible preconditions model, tasks go through the following states:

prescribed tasks are those whose creation has been requested but whose preconditions are not yet satisfied.
Fig. 1: State transition diagrams for the eager, strict and flexible preconditions models. Initial states are in bold.

- **Ready** tasks are those whose preconditions are satisfied, but have not yet been scheduled to run.
- **Running** tasks are those currently executing.
- **Blocked** tasks are those that have previously been running and have attempted to read an input that was not listed as a precondition and that input was not available.
- **Finished** tasks are those that have completed execution.

The state transition diagram in Figure 1 visually compares the flexible preconditions model (Figure 1c) with eager task creation (Figure 1a) and strict preconditions (Figure 1b). The extra state that the flexible model has compared to the eager model is the prescribed state, in which tasks are waiting for preconditions. Strict preconditions do not have a blocked state as tasks never need to wait.

We will use an AND-reduction example to illustrate the flexible preconditions model. This example is a fragment of an image processing application, in which an input image is split into multiple tiles indexed by iteration-number and tile-id. Tiles are updated independently by different tasks which produce new tiles used in the next iteration of the algorithm. Then, a reduction task checks if all tiles pass a convergence condition. If a tile is found that does not respect the condition, a new iteration is started. When such an input tile is found, any other tiles not tested need not be read. This access pattern is typical for AND-reductions. The pseudo-code for the reduction task is shown in Listing 1. The main program (Listing 2) for the AND-reduction creates the initial set of tiles and starts the first reduction task (line 4). In the pseudo-code, calls to \texttt{get} perform reads of dataflow dependences, while \texttt{put} calls produce them. The behavior of these access functions will vary depending on the macro-dataflow execution model used.

In an eager implementation, any instance of the \texttt{get} call on line 4 of Listing 1 may block waiting for the tile parameter to be produced by \texttt{UpdateTileTask}. When the \texttt{UpdateTileTask} instance performs a \texttt{put} on that tile, the reduction task will become unblocked and continue execution until it blocks again on another unavailable tile.

Getting strict precondition behavior requires the user to write additional code, such as in the pseudocode in Listings 1 and 2. The function \texttt{declare-get}, once called from \texttt{ReductionTask-dependences(iteration)} registers the value of its parameter as a dataflow dependence for the \texttt{ReductionTask} with the same iteration number. Notice that all the tiles that could possibly be accessed by the \texttt{ReductionTask} are marked as dependences - this is a requirement of the strict preconditions model. Strict tasks will only be spawned when all the dependences (tiles 1 to N of each iteration) have been put. If a task only reads a few tiles, waiting on all of them to be produced would lead to unnecessarily delaying the start of the next iteration and performance degradation.

The flexible preconditions model enables a partial specification of preconditions, so that the choice of which tiles are required can be made by taking advantage of the programmer’s knowledge about the application. Ideally we want to list as preconditions only the last items to be produced that will definitely be needed by the task. This minimizes the overhead of managing the preconditions by minimizing their number and does not introduce artificial latency by waiting for unneeded items. In general though, because of parallelism, there is no single dataflow dependence guaranteed to be the last one to be produced in any possible schedule. These problems make choosing an efficient preconditions list challenging. For our AND-reduction example, all instances of \texttt{ReductionTask} will read \texttt{tile[iteration, 0]}, so having only this tile as a precondition will not lead to any artificial delays (as opposed to the strict preconditions model) and also decreases the management overhead for the preconditions list from N to 1. Let us look now at how flexible preconditions compares to the eager approach. By specifying a subset of dataflow dependences as preconditions we decrease the maximum number of blocking operations required for task execution because the task will not need to block on the inputs listed as preconditions. From a memory consumption point of view, we are able to postpone

\begin{verbatim}
function ReductionTask(iteration)
    i = 0
    boolVar = false
    while( !boolVar )
        crtTile = get( tile [iteration, i] )
        boolVar = ConvergenceCheck(crtTile)
        i = i + 1
        if(boolVar)
            spawn new ReductionTask(iteration+1)
        for i from 0 to N
            spawn new UpdateTileTask([iteration,i])

Listing 1: Pseudocode for AND reduction for eager execution

function ReductionTask-dependences(iteration)
    declaration of \texttt{get} and \texttt{put}
    for i from 0 to N
        put ( tile [iteration, i] )
    spawn new ReductionTask(iteration)

Listing 2: Pseudocode for starting the execution of an AND-Reduction

function ReductionTask-dependences(iteration)
    for i from 0 to N
        declare-get( tile [iteration, i] )

Listing 3: Pseudocode for specification of preconditions for AND-reduction in the strict preconditions model
\end{verbatim}
the allocation of task memory by keeping the task in the prescribed state as opposed to marking it as ready from the start. In the AND-reduction application, the eager execution will allocate memory before \texttt{tile[it, 0]} is available; flexible preconditions (with the \texttt{tile[it, 0]} as precondition) may still need to block, but only for tiles 1 to \texttt{N}, so the wait for \texttt{tile[it, 0]} will be done without unnecessarily consuming memory.

As most data-flow models have an implementation similar to the pseudocode presented here, we note that the task code from Listing 1 remains unchanged in the flexible preconditions implementation, making it straightforward to port eager/strict applications to the flexible preconditions model.

Flexible preconditions are useful for several types of applications. In these applications a partial specification of the dependences can be used to choose between the behavior of either eager task creation or strict preconditions, depending on which performs best for each application. In addition, the use of flexible preconditions opens up a number of intermediate behaviors to pick from: if one model is better for performance and the other is better for memory we may need a balanced choice between the two. We give examples, results, and discuss characteristics of these applications in Section VI.

IV. QtCnC Design

To compare the flexible preconditions model with the eager task creation and strict preconditions we needed a runtime framework that would subsume all three approaches. We chose to extend an existing macro-dataflow model called Concurrent Collections (CnC) [5]. The main reasons for choosing CnC is its generality (it does not specify the behavior as corresponding to either the strict or the eager models) and the ease of separating the specification of the preconditions from the task logic. Our implementation of CnC is a C++ library built on the Qthreads [18] runtime, similar in interface to the Intel CnC distribution [19]. The following two subsections give an overview of the CnC model and of the additions we propose to support the three macro-dataflow models.

A. The Concurrent Collections Model

CnC applications consist of collections which encapsulate tasks (step collections), control of tasks (control collections) and values (item collections) [5]. Item collections enable communication among tasks. They can be thought of as repositories for dynamic-single-assignment values that are indexed by keys. For example, to perform a read of a value from item collection \texttt{ic}, given its key \texttt{[it, i]}, one would use \texttt{ic.get([it, i])}; to produce that value, another task will need to perform a put operation: \texttt{ic.put([it, i], new Tile(...))}.

The CnC runtime has two main responsibilities:

- enforcing the dynamic single assignment rule by throwing a runtime error if two different put operations are performed on the same item collection with the same key. This ensures datarace-freedom for CnC programs.
- implementing the dataflow dependences between items and tasks. If a task performs a get on an item with a particular key, that task will not proceed until the

```c
int Reduce::execute(const int & it, ... ) {
for(int k = 0; k < N ; k++) {
    Tile *crt;
    tile_item_collection.get(pair(k, it), *crt);
    ... 
reduce_control_collection.put(it+1);
return CnC::CNC_Success;
}
Listing 4: QtCnC code for AND-reduction.
```

```c
aligned_t** Reduce::get_dependences ( 
    const int & it, int & no ) { 
    aligned_t** preconds;
    read = malloc( N*sizeof(aligned_t*) )
for(int k = 0; k < N; k++) {
    tile_item_collection.wait_on(
        pair(k, it),
        &{preconds[k]} );
} 
    return read;
}
Listing 5: Specification of preconditions in QtCnC.
```

item with that key has been produced. This makes CnC programs deterministic by default.

As an example, Listing 4 shows part of the QtCnC step collection code for the reduction tasks in AND-reduction. The execute method is the code that is called for each put on the reduce_control_collection. It receives as parameter the tag, in this case the iteration number for which the reduce should be performed. On line 6 the call to get is performed to access a tile with key \texttt{[k, it]}; on return, its second parameter \texttt{(crt)} contains a pointer to the tile. On line 11, the next reduction task is started by calling put on the appropriate control collection with \texttt{it+1} as tag.

B. Strict, eager and flexible models in QtCnC

As described in the previous subsection, QtCnC can support the eager task creation model. To express strict and flexible preconditions we enhanced it with additional APIs.

To specify preconditions in QtCnC, one needs to implement a get_dependences function for each step collection. Listing 5 shows such an implementation for the strict preconditions model in AND-reduction. On line 5, an array is allocated with one entry for each precondition; then, on line 7, it is filled in by calling the runtime function wait_on and specifying two parameters: the key of the item that will be a precondition and the array position in which to put this precondition. The array is then returned on line 10.

To use the flexible model as opposed to the strict preconditions one, there is no additional API needed - one just needs to write a smaller list of items as preconditions and compile with a flag specifying that flexible preconditions behavior is desired.

V. QtCnC Implementation

The QtCnC runtime is an open-source runtime\footnote{https://code.google.com/p/qthreads/wiki/QtCnC} built on top of the Qthreads tasking library [18] which we use for
task scheduling and synchronization support. The precon-
ditioned tasks and the full-empty memory mechanisms of
Qthreads are well suited for enforcing the preconditions in
preconditions-based models. Preconditioned tasks have the
advantage that stacks are assigned only after the task has
its preconditions available and is scheduled to run. They are
spawned by calling the qthread_fork_precond function
and providing a pointer parameter to an array of memory
words on which to wait. Only when these words have been
marked as full will the task be spawned. The words can
be marked full by using the full-empty mechanism API call
qthread_fill(aligned_t* word).

New APIs in the QtCnC runtime include the
get_dependencies() and wait_on() public functions,
which are used for specifying preconditions. The following
paragraphs discuss the implementation of these functions.

To track item availability, shadow state is maintained for
each item, consisting of a single memory word; this state
is abstracted away from the user through the item collection
classes and the ic.wait_on API, which writes into
its second parameter a pointer to this shadow state. When
ic.put(key, value) is called, it records the availability of
the item by marking the shadow word for that item as full
(through a qthread_fill call).

The user function get_dependencies fills the array
preconds with pointers to shadow words and then returns
it to its caller. The QtCnC runtime performs a call to this
function before spawning a task, to obtain the preconditions
it needs before running. It sends the array of preconditions
to Qthreads when it performs the spawn through a call to
qthread_fork_precond.

An additional optimization we performed in the strict
preconditions model involved the spawning of stackless tasks
(called “simple tasks” in Qthreads). These tasks use the stacks
of the thread they are scheduled on, instead of allocating
their own and are prevented from performing any blocking
synchronization operations.

Shadow word management is made more complex be-
cause the words have to be created by the first of following
three APIs that is called: ic.get, ic.put or ic.wait_on.
This management is performed by the item collections. Item
collections are implemented in QtCnC via a C++ template
class that wraps a concurrent hashtable indexed by item keys.
The concurrent hashtable ensures that concurrent calls to the
previously mentioned three functions do not cause dataarces.
We built the split-ordered-list concurrent hashtable described
in [20] with dynamic resizing. This concurrent hash table is
now included in the latest Qthreads distribution.

VI. EVALUATION

A. Experimental Setup

Our results were obtained on a 4 socket 32 core Intel
Nehalem X7550 system with 512GB memory. All tests have
been performed using the default Qthreads scheduler, with
-03 optimization and a stack size of 2MB. The stack size
was chosen in accordance with the stack size used by default
by Pthreads (2MB), with the reasoning that it is suitable for
most applications, and is considered in previous literature to
be a value that is easy to exceed [21]. The results listed are
averages of 10 runs; where appropriate, we include error bars
on graphs that correspond to the largest and smallest values
obtained during the runs.

To measure the actual memory footprint of the programs
we used the /usr/bin/time tool, for which the -v param-
eter outputs the maximum resident set size of the program. The
memory and performance numbers were collected as averages
of the same 10 runs.

B. Benchmarks

To test the performance of the precondition-based models,
we use the following benchmarks. Unless specified otherwise,
these benchmarks have been implemented in all three models.
A summary of the results is in Table I and the results are
discussed in detail in the next subsections.

Blackscholes is a financial application that computes stock
values. We use the implementation from Intel CnC [19]
to analyze the overhead of the three models. The input
size is 1,500,000 and granularity 100.

Cholesky Factorization is the non-MKL version of the linear
algebra benchmark from the Intel CnC distribution [19].
It decomposes a matrix into a lower triangular matrix and
its transpose. The input matrix size used is 4000 × 4000
tiles are 125 × 125.

Matrix Inverse is a benchmark from the Intel CnC distribution [19]. The input matrix size is 2048 × 2048 and the
tile size is 64 × 64.

File Concatenation is a benchmark that concatenates a set of
files by performing the least number of concatenations.
It builds a balanced binary tree in which the inputs
are leaves and each node represents a concatenation
operation. This application illustrates a situation when
strict preconditions cannot be directly applied, so only
the flexible preconditions and eager versions are used.
The number of input files is 32768.

Reduction is the kernel of the Rician Denoising [22] applica-
tion that we use to assess the performance of the runtimes
for cases where some gets are optional, such as in short-
circuit reductions. The input size consists of 16 tiles of
10×10 size and the algorithm performs 16 iterations.

C. Blackscholes

In Blackscholes, tasks perform a single get of inputs
produced by the environment, so tasks never block. Because of
this, there is no memory footprint difference between flexible
preconditions and eager, as seen in Figure 26. As expected,
the strict preconditions memory footprint is almost constant
because strict preconditions tasks are stackless. For flexible
preconditions and eager execution, tasks may still block on
inputs that are not listed as preconditions, which means tasks
need to be paused and their state (such as their stack or a
closure) must be saved. Because the ability of tasks to block
is not used in this benchmark (all inputs are available from the
start), the footprint difference is equal to the size of the task
stack multiplied by the number of tasks running concurrently, which is equal to the number of worker threads.

If we look at the performance comparisons from Figure 2, we see that the three runtimes have similar scalability. This is to be expected, as all items are available from the start (so no task ever blocks).

D. Cholesky Factorization

The performance results for Cholesky, Figure 3, show a performance difference of up to 56% between the eager approach and strict and flexible preconditions. For the eager runtime, the speedup reaches a maximum at around 16 threads versus more than 32 in the other models. The reason for this is visible in Figure 3. In the eager runtime, because the application spawns all tasks in the beginning and many of them block, their allocated stacks must be stored; the maximum memory pressure is a lot worse than for preconditions based models, where the use of preconditions lowers the memory requirements by 62% because tasks never block.

We theorized that the lack of performance of the eager runtime is caused by additional time spent during concurrent allocations and we built a Pthreads-based micro-benchmark to verify our assumptions. The microbenchmark performs similar allocations from parallel threads and suffered from the same lack of scalability. The Cholesky application was the original motivating application for our work, as the high memory usage prevented us from running the application on larger inputs using a machine with only 2GB of memory; this happens because in the eager model, task memory is much larger than the application data.

For Cholesky, all dataflow item accesses are known in advance, so the preconditions list was complete even for flexible preconditions and the runtimes show the same performance.

E. Matrix Inverse

For MatrixInverse, Figures 4a and 5b, the eager approach is consistently worse (up to 44%) than both flexible and strict preconditions, while the difference between the precondition based approaches is small. The precondition-based approaches are so similar because all dataflow task preconditions for the application are known in advance; the difference is the overhead of the get calls, just as in Cholesky, and remains under 2% even at 32 threads. The memory consumption shows the footprint of the eager approach is 3.6 × larger than that of strict preconditions.

F. File Concatenation Benchmark

The File Concatenation benchmark cannot be run on the strict preconditions model, because the preconditions of each task depend on one another’s value (data-dependent gets). Note that this is not a characteristic of the runtime or model, but of the way the application is written: some applications written for the eager model cannot be converted to run in a strict preconditions model without a considerable increase in the number of tasks.

In this benchmark, tasks perform get operations to obtain the two operating system inode structures representing the input files. These structures contain the tags of the blocks that need to be concatenated. Because this information is not known until the inodes are read, it cannot be added to the list of preconditions.

As we see in Figure 5a, even though the memory consumption difference between flexible and eager is small (less than 0.6%), in absolute value it reaches 383MB. Compared to eager execution, flexible preconditions are up to 28% faster and allow scaling up to 16 cores instead of only just 4 in the case of eager execution (see Figure 5a).
Fig. 6: Porting microbenchmark performance

Using the file concatenation benchmark we analyzed the performance difference between an eager implementation and a flexible preconditions one, but the performance of the strict runtime was not included because porting the file concatenation tasks to a strict model is very time-consuming. The Porting benchmark takes a systematic look at the performance implications of porting applications from the eager/flexible runtimes to strict preconditions, which would enable applications with data-dependent gets to run on the strict runtime. In this microbenchmark, tasks perform data-dependent gets — gets whose keys are values of previous gets. This is not allowed in strict preconditions models because item values are not known at the time the preconditions are specified; for a strict implementation to be used, such tasks need to be split into sub-tasks, one for each of such data-dependent gets. Figure 6 shows the performance of the two runtimes in such a case. On the X axis, we have the number of data dependent gets per task, which is also the number of subtasks in which tasks are split. Flexible preconditions have better performance when such gets are few and the difference increases when there are more. This happens because with more subtasks, their spawning overhead dwarfs the useful work they perform. In fact, even for a single data-dependent get, flexible preconditions offer better performance for this microbenchmark. Of course, in a real application the difference between the performance of the two runtimes will depend on the granularity of the work performed in each task, relative to the overhead of task creation.

G. AND-Reduction Benchmark

Reductions are a common pattern in many applications. An interesting variation is short-circuit reductions, in which not all inputs may be read. This is the case in image processing applications such as the Rician Denoising application on which this benchmark is based. Here, the reduction is a convergence criteria tested before starting the next denoising iteration: if a tile is found not to respect the convergence criteria, the following tiles need not be read. Eager execution will wait for each tile as and when it is needed and flexible preconditions allow the programmer the freedom to choose which tiles should be waited on before spawning the reduction task (in our implementation, we list only the first tile as a precondition). Strict preconditions need to declare all input tiles as preconditions and wait for the availability of them all; this behavior is a problem because, even if it does not compromise correctness, it may compromise performance, by waiting for a larger set of preconditions than are actually needed.

Fig. 7a shows that the eager model and flexible preconditions offer significantly better performance. As far as memory consumption is concerned, as shown in Figure 7b, all runtimes have almost the same footprint for this application, with flexible preconditions being in the middle.

H. Discussion of the findings

The cause for the poor performance shown by eager task spawning may be contention in the memory allocator [23], but the problem is worse in our case, due to the quantity of memory allocated. Allocating many tasks with 2MB stacks quickly exhausts all allocation arenas and allocations are forced to call into the kernel (via sbkrk() or similar) to enlarge the process’s available memory. These kernel calls are likely serialized, given that they’re all modifying the single kernel-level memory map for the process.

In light of these results, we believe that a good solution is instead to opt for models based on preconditions. For applications whose performance is extremely critical and whose design allows all dependences to be known at task creation time, strict preconditions may be suitable.

A disadvantage of the strict preconditions model is that it has performance issues when some of the items that are read in some corner case are often not read, such as in the AND-reduction example. Using strict preconditions for such applications leads to considerable performance loss compared to flexible preconditions.

A second problem with the strict preconditions model is that for some applications it is a programmability impediment. One characteristic of applications not easily expressible in the strict model is the existence of data-dependent gets, which

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Runtime</th>
<th>Time (s)</th>
<th>Δ Time (%)</th>
<th>Memory (MB)</th>
<th>Δ Memory (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blackscholes</td>
<td>strict</td>
<td>1.384</td>
<td>-1.7</td>
<td>71.78</td>
<td>-39.7</td>
</tr>
<tr>
<td></td>
<td>eager</td>
<td>1.408</td>
<td>0.1</td>
<td>105.96</td>
<td>-1.4</td>
</tr>
<tr>
<td></td>
<td>flexible</td>
<td>1.407</td>
<td>0.0</td>
<td>107.46</td>
<td>0.0</td>
</tr>
<tr>
<td>Cholesky</td>
<td>strict</td>
<td>0.810</td>
<td>3.0</td>
<td>274.78</td>
<td>0.7</td>
</tr>
<tr>
<td></td>
<td>eager</td>
<td>1.814</td>
<td>56.7</td>
<td>717.84</td>
<td>62.0</td>
</tr>
<tr>
<td></td>
<td>flexible</td>
<td>0.786</td>
<td>0.0</td>
<td>272.89</td>
<td>0.0</td>
</tr>
<tr>
<td>Matrix Invert</td>
<td>strict</td>
<td>0.880</td>
<td>-3.4</td>
<td>127.63</td>
<td>-3.0</td>
</tr>
<tr>
<td></td>
<td>eager</td>
<td>1.648</td>
<td>44.8</td>
<td>461.39</td>
<td>71.5</td>
</tr>
<tr>
<td></td>
<td>flexible</td>
<td>0.910</td>
<td>0.0</td>
<td>131.44</td>
<td>0.0</td>
</tr>
<tr>
<td>File Concatenation</td>
<td>strict</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>eager</td>
<td>37.786</td>
<td>28.5</td>
<td>64.41</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>flexible</td>
<td>27.030</td>
<td>0.0</td>
<td>64.03</td>
<td>0.0</td>
</tr>
<tr>
<td>Reduction</td>
<td>strict</td>
<td>2.193</td>
<td>38.0</td>
<td>2143.12</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td>eager</td>
<td>1.329</td>
<td>-2.4</td>
<td>2124.89</td>
<td>-0.5</td>
</tr>
<tr>
<td></td>
<td>flexible</td>
<td>1.361</td>
<td>0.0</td>
<td>2135.92</td>
<td>0.0</td>
</tr>
</tbody>
</table>

TABLE I: Summary of results for execution on 32 threads. Δ values show the relative improvement of flexible preconditions over the other two runtimes.
are get operations whose keys are derived from the value of an item obtained through a previous get. Such accesses are used mostly for convenience in applications built for the eager model, in cases where tasks follow the natural structure of the program data (for example, in File Concatenation, reading a file’s inode is followed by reading the blocks listed in the inode structure to access the file data). Similarly, in another application, Routing simulation [22], tasks that represent routers in a network with link failures need to read the network topology to find their neighbors and then they read the routing tables of those neighbors. The network topology and routing tables of the neighbors should be preconditions, but the neighbor routing tables cannot be strict preconditions because we do not know which will be read until the topology is obtained. One solution is to add the tables of all nodes as preconditions, but that slows down the computation just as in the AND-reduction case.

Another characteristic of applications that are not easily expressed in a strict preconditions model is that these applications often perform a variable number of gets, depending on conditions identified in the task computation. The best example is the AND-reduction discussed in Section III and File Concatenation, where the inode of a long file contains the id of an extra block that must be read.

Our experimental results showed that the proposed flexible preconditions model performs on par with the best of the strict preconditions or eager models, while maintaining the expressiveness of the eager model making it a good alternative to both models.

VII. Conclusions

In this paper, we analyzed the performance of two widely used models for macro-dataflow execution (eager tasks and strict preconditions) and found that their performance and scalability are highly sensitive to application behavior and algorithm design. We proposed a new model — flexible preconditions — that can always match the performance of the better of the two models. On applications where all dependences are known ahead of time, strict preconditions are 38% faster than eager execution and flexible preconditions match the strict model. On the other hand, for applications where the complete set of per-task dependences is determined after the tasks are spawned, eager execution performs 57% better than strict preconditions and flexible preconditions match this model. This improvement is achieved by enabling programmers to to pick the preconditions combination that yields the best performance for each application. As future work, we plan to build an auto-tuning system that can automatically pick the best preconditions by analyzing task behavior at runtime, and also extend the scope of our work to encompass task cancellation and demand-driven evaluation as additional scheduling options.

Acknowledgment

Sandia National Laboratories is a multiprogram laboratory operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Company, for the U.S. Department of Energy’s National Nuclear Security Administration under Contract DE-AC04-94AL85000.

References


**APPENDIX: COMPARISON WITH INTEL CnC**

The above experiments show that QtCnC flexible preconditions offer consistently good performance, competitive with the better of the two previous models. However, we need to compare QtCnC with the state of the art. We benchmarked the strict preconditions and eager runtimes of QtCnC with the eager Intel CnC, which is based on Intel Threading Building Blocks, recognized for its good performance [24]. The Intel CnC runtime is very efficient because it is custom built for the CnC model. It imposes restrictions on what dataflow tasks can do in order to improve performance. The following restrictions enable efficient use of the abort-and-restart mechanism:

**puts follow gets** This means gets and puts cannot be intertwined in any task, which ensures that items are not produced multiple times if tasks get restarted because of unavailable inputs.

**computation follows gets** Any operation that takes a long time relative to the task lifetime must be performed only after all gets have completed. This ensures that any work performed before gets will not be very expensive, so that work wasted on a restart is minimized.

**no side effects** Atomic counters, locks or synchronization between tasks and global data cannot be used to interact with the environment. As a result, re-running tasks will be possible with no side effects.

The important advantage of abort-and-restart is that tasks never need to be stored while they are deferred — they are simply re-computed. This also means that no task-specific memory (e.g. a stack) needs to be allocated and the worker threads’ stacks can be reused even in an eager runtime.

We wanted to keep the implementation as general as possible, so that results obtained on it reflect performance that can be obtained by any model, not just CnC. Qtthreads proved very useful thanks to its support for tasks that can block without loss of generality; the restrictions mentioned for Intel CnC do not apply for our runtimes.

The results shown in Figure 8 have been obtained by running Cholesky factorization on a 4000 × 4000 input with 125 × 125 tile size. Both runtimes use the same step code. Because Intel CnC does not offer a way to set the stack size for its tasks, we used the default stack size for runtimes. The Intel CnC implementation is known [25] to outperform OpenMP and Cilk++ on this application. Our results show that all three runtimes offer similar speedup, with the Intel runtime leading by under 10%. Considering that QtCnC offers a much more general model and that Cholesky is an application suitable to the restricted Intel CnC model, we think this is a reasonable generality-performance trade-off. If however, the dataflow model and user applications follow the three restrictions imposed by Intel CnC, then the abort-retry mechanism should be considered as a mechanism for deferring tasks, as it does offer a performance benefit.

<table>
<thead>
<tr>
<th>Threads</th>
<th>QtCnC (strict)</th>
<th>QtCnC (eager)</th>
<th>Intel CnC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>4</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
</tr>
<tr>
<td>8</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
</tr>
<tr>
<td>12</td>
<td>0.006</td>
<td>0.006</td>
<td>0.006</td>
</tr>
<tr>
<td>16</td>
<td>0.008</td>
<td>0.008</td>
<td>0.008</td>
</tr>
<tr>
<td>20</td>
<td>0.010</td>
<td>0.010</td>
<td>0.010</td>
</tr>
<tr>
<td>24</td>
<td>0.012</td>
<td>0.012</td>
<td>0.012</td>
</tr>
<tr>
<td>28</td>
<td>0.014</td>
<td>0.014</td>
<td>0.014</td>
</tr>
<tr>
<td>32</td>
<td>0.016</td>
<td>0.016</td>
<td>0.016</td>
</tr>
</tbody>
</table>

Fig. 8: Eager Intel CnC versus eager and strict preconditions QtCnC on Cholesky Factorization.