RICE UNIVERSITY

Memory and Communication Optimizations for Macro-dataflow Programs

by

Dragoş Sbîrlea

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Vivek Sarkar

Vivek Sarkar, Chair
E.D. Butcher Chair in Engineering and Professor of Computer Science

[illegible]

Keith D. Cooper
Doerr Chair in Computational Engineering and Professor of Computer Science

[illegible]

Peter J. Varman
Professor of Electrical and Computer Engineering and Professor of Computer Science

Houston, Texas

May, 2015

ABSTRACT

Memory and Communication Optimizations for Macro-dataflow Programs

by

Dragoş Sbîrlea

It is now widely recognized that increased levels of parallelism are a necessary condition for improved application performance on multicore computers. However, the memory-per-core ratio is already low and, as the number of cores increases, it is expected to further decrease, making per-core memory efficiency of parallel programs an even more important concern in future systems. Further, the memory requirements of parallel applications can be significantly larger than for their sequential counterparts and their memory utilization also depends critically on the schedule used when running them.

This thesis proposes techniques that enable awareness and control of the trade-off between a program's memory usage and resulting performance. It does so by taking advantage of the computation structure that is made explicit in macro-dataflow programs which is one of the benefits of macro-dataflow as a programming model for modern multicore applications.

To address this challenge, we first introduce folding - a memory management technique that enables programmers to map multiple data values to the same memory slot. This reduces the memory requirement of the program while still preserving its macro-dataflow execution semantics.

We then propose an approach that allows dynamic macro-dataflow programs running on shared-memory multicore systems to obey a user-desired memory bound. Using the inspector/executor model, we tailor the set of allowable schedules to either guarantee that the program can be executed within the given memory bound, or throw an error during the inspector phase without running the computation if no feasible schedule can be found.

Finally, we turn our attention to distributed systems where often the memory size is not a limiting factor, but communication and load balancing are. For these systems, we show that data and task distributions can be selected automatically even for applications expressed as dynamic task graphs, freeing the programmer from the cumbersome selection process. We show that optimal selection can be achieved for certain classes of distributions and cost functions that capture the trade-off between communication and load balance.

Acknowledgments

I would like to acknowledge all members of the Habanero Exascale Research group for being insightful discussion partners during the length of my time at Rice. I especially thank my co-authors in the works presented in this thesis, Kath Knobe and Zoran Budimlić for brainstorming, detailed design discussions and feedback. I also thank Kyle B. Wheeler who was my mentor during my internship at Sandia National Labs. Kyle helped me understand that research and personal life can be balanced - a realization without which I may not have reached the point of writing this thesis. Professor John Mellor-Crummey helped me develop my analytical thinking through his courses and projects at Rice and this directly shaped the way I do research and evaluate the results of my work.

I would like to extend my sincere thanks to the members of my PhD committee. Both Professor Keith Cooper and Professor Peter Varman are, through the combination of their work and personality, models that I would like to follow.

Above all, I would like to thank my PhD advisor, Professor Vivek Sarkar, for doing everything in his power to enable me to be productive, for giving high level insights that helped crystallize and express many of the contributions described in this thesis and for giving advice that helped turn these ideas into code artifacts. His guidance towards the ideas that are now chapters in this thesis was essential in shaping my PhD contributions.

I would like to thank my parents, Gabriela and Cristian, and especially my wife, Alina. Without their emotional support I may have abandoned working on many ideas. Without their help I would not have had the time and willpower to work on others, so this PhD is dedicated to them.

Thank you all!

Contents

Illustrations

Tables

Chapter 1

Introduction

Processors and memory are the two primary architectural components of modern computing systems, that have a major impact on the behavior of applications. Year after year, architectural improvements alternate between improving one or the other so as to achieve a balance that results in reasonable performance for common applications. Algorithmic techniques such as using lookup tables versus recalculation (and dynamic programming in general), using compressed versus uncompressed data, re-rendering versus storing images or in compiler optimization smaller code versus loop unrolling have enabled software engineers to effectively control this balance for sequential machines.

With the advent of multi-core processors and the dominant use of dynamic task parallelism as a model of computation, it has become difficult to control the balance between processor and memory utilization because different program executions, even with the same input parameters, may achieve different memory utilizations and different execution times due to differences in their schedules.

Parallel execution is known to increase memory requirements compared to a serial baseline [**Blumofe:99**]. The community has been aware of this problem since (at least) the 1990s: "The amount of memory required by a parallel program may be spectacularly larger than the memory required by an equivalent sequential program .... parallel memory requirements may vary from run to run, even with the same data" [**Burton:96**]. Without mitigation techniques, this increased memory consumption can lead to an increased occurrence of out-of-memory errors [**Dooley:10**] or degraded performance due to virtual memory paging. Unfortunately, modern programming systems for parallel applications are not aware of and do not control the peak memory footprint, making it difficult for programmers to ensure their programs will not run out of memory[*].

Multicore systems, with their increasing levels of parallelism, have arrived at a time when memory capacity scaling has already slowed [**RA01**]. Currently, the memory per core is decreasing by 30% every two years [**LCM09**] and projections state that it will soon drop from gigabytes to hundreds or tens of megabytes in extreme scale systems [**LKT13**]. As expressed by IBM, this is an important challenge to overcome for exascale computing, since "our ability to sense, collect, generate and calculate on data is growing faster than our ability to to access, manage and even store that data." [**DT05**]. But this problem is not only an obstacle for future supercomputers; for the embedded multicore processors, memory is already at a premium today.

This thesis proposes approaches that alleviate the memory problem encountered by parallel programming models in use today. Since the most popular programming models are dynamic in their expression of parallelism, we focus on techniques that work for such models, whether on shared or distributed memory platforms.

The rest of this document is organized as follows. Chapter 2 describes the parallel programming model on which we build the approaches presented in this thesis. Chapters 3 and 4 describe work that applies to shared-memory memory management: chapter 3 focuses on folding - a technique for memory management that help programmers understand the memory footprint of their programs, and Chap-

[*]In contrast, embedded applications tend to be memory-aware but usually offer little flexibility in scheduling and mapping of individual components.

ter 4 describes a memory management technique that allows programmers control of the space-performance trade off. Chapter 5describes our approach to extending the ideas on controlling the space- performance trade-off for shared-memory to distributed memory, where performance is more dependent on inter-node communication and load balancing rather than memory footprint. Chapter 6 describes related work to the techniques employed in this thesis including inspector/executor scheduling, register allocation and instruction scheduling.

1.1 Thesis Statement

Our thesis is that knowledge of the intrinsic dataflow relations within a computation is key to controlling resource trade-offs for parallel execution of applications. We show that this knowledge can be obtained, acted upon and reused in an efficient manner when using dynamic macro-dataflow programming models to explore trade-offs among parallelism, memory, and communication.

Chapter 2

Background

2.1 The Concurrent Collections (CnC) programming model

2.1.1 CnC - an implicit parallel model for domain experts

The programming models proposed in my work are extensions of the Concurrent Collections (CnC) [**Budimlic:10**] model. The CnC model is influenced by stream processing, tuple spaces and dynamic dataflow languages. It takes dataflow principles and applies them at the level of tasks which can run efficiently on modern architectures.

CnC is designed to appeal to domain experts who are not expert parallel programmers. A similar goal can be achieved by using domain specific languages (DSLs) that hide the details of the parallelism from the programmer - but only when programming withing a particular application domain. Instead, CnC aims to provide an easy way to add parallelism to any host language, with the programmer writing sequential code; the premise is that a domain expert (who is not necessarily an expert in parallel programming) can express the data and control dependences of their applications and the CnC compiler and runtime, based on this information, can execute the application in parallel. The data and control dependences expressed by the domain expert are an implicit way of expressing parallelism and has the advantage of enabling the runtime to select the granularity of parallelism that is suitable for the particular machine on which the application runs.

By automatically generating the synchronization and communication operations, the CnC model relieves the programmer from the possibility of two notorious classes of bugs in parallel programming — data races and deadlocks.

2.1.2 The CnC model

CnC applications consist of tasks (called *steps*), uniquely identified by a *step collection* specifying the code that the task will run, and a tuple called the *step tag* identifying a specific instance of a step. Tasks communicate through dynamic single assignment variables called *items*. Items are grouped together into *item collections* which contain logically related items and are uniquely identified by tuples called *keys*. By enforcing the dynamic single assignment rule for items and because items cannot be modified once they are added to an item collection, CnC avoids the possibility of data races on items.

Once spawned, steps can read items by calling the `item_collection.get(key)` function which returns the item value. `Get` calls block until some other step produces the item with that key by calling `item_collection .put(key, value)`.

For example, in a stencil computation, a step with tag `(i,j)` could read data items `(i, j-1)` and `(i-1, j)` from an item collection and produce item `(i,j)` in the same item collection.

Once steps read their input items, they perform computation and then may produce new items (through `put` operations) and/or start new steps by calling the `step_collection.spawn` function.

CnC has been shown to offer especially good performance for applications with asynchronous medium-grained tasks [**Aparna:10**], an attractive target for our optimization approaches.

Since items are accessed using tuples as keys (rather than pointer-based references), it is generally not possible to automatically identify which items are dead and should be collected. Instead, the number of expected read operations (called the *get-count* of the item) is specified as an additional parameter to item `put` calls [**Sbirlea:12**].

Some variants of CnC use an additional abstraction, that of *control collections*. These are inspired by the factory [**GoF**] design pattern: a tag `put` to a control collection results in the spawning of task instances from step collections that have registered with that control collection. Because control collections are orthogonal to the techniques presented in this thesis, we use a simplified CnC model, where possible, in which step instances are spawned by calling `spawn` directly to on the desired step collection.

2.1.3 Task creation and preconditions

In macro-dataflow models, one available knob for the control of the parallelism exposed is the precise timing of task creation. Tasks that are not yet expressed to the model (or "created") consume no resources. However, manually restricting the parallelism in this way is highly machine-specific. Over-restricting parallelism leads to poor performance due to resource under-utilization, and under-restricting parallelism results in poor performance due to resource contention and other forms of overhead. The task spawning mechanisms of a given dataflow implementation determine the costs and trade-offs involved, which affects the design of a dataflow program. The following subsections describe the two dominant approaches, eager and strict, both of which are available in CnC. There has also been work down on hybrid combinations of the eager and strict policies[**Sbirlea:13**].

2.1.4 Eager task creation

The first option is to use an eager approach, in which tasks are spawned as soon as possible. The tasks optimistically start executing, but may have to wait after starting, if their dataflow dependences are unavailable when needed. Good performance is achieved if most dependences have already been produced. For thread pool based schedulers (including work sharing and work stealing), eager implementations must provide a mechanism for worker threads to recover from missing dependences and continue executing other work. There are three mechanisms that can be used to address this situation:

continuations A task can be suspended by building its continuation, which is then registered to wait for the data needed. For this approach, the ability to build continuations is a requirement [**Imam:14**].

abort-and-restart Tasks abort upon access to data that is not yet available; the task is restarted later, ideally after the needed data becomes available. This approach involves additional restrictions, such as tasks not having side effects before aborting [**CnC:2013**].

blocking Tasks block on synchronization primitives such as locks that are aware of the thread pooling approach and cooperatively allow the thread to execute other tasks. Tasks do not need to be restarted and become available for rescheduling when the lock becomes available. This approach is different from continuations because no data external to the task function body is preserved when blocking.

A common problem with eager task creation is that the task memory and other resources needed to execute must be allocated as soon as the task is created, and are not completely released when the task is deferred. This resource allocation puts pressure on the memory and runtime systems. A possible advantage is that, if a task starts with 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.

2.1.5 Strict preconditions

With this task creation strategy, tasks are spawned only when all their dataflow dependences are satisfied. We call this approach *strict preconditions* because the availability of inputs becomes a precondition to running tasks; as such, that input can be accessed without synchronization. *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 models and has implications on the programming API exposed to the user. For example, *optional and data-dependent inputs* are usually forbidden. These restrictions can be overcome (for example) by splitting a task with data-dependent continuation tasks into separate tasks, but doing so can lead to additional performance overheads.

To support the identification of when tasks have all their inputs available — which is needed for preconditions — a synchronization mechanism separate from the one used for atomically producing dataflow dependences must exist. This support is usually provided via a set of task-descriptors which describe tasks that need to be spawned. Each descriptor has an *atomic counter* whose value decreases when each dependence is satisfied; when it reaches zero, the task can safely be spawned.

Strict preconditions offer better performance than eager execution if the overhead associated with blocking/closures/restarting in the eager case is larger than the atomic-counter based synchronization used to implement strict preconditions.

2.1.6 Example CnC program

We will describe our model while going over a simple Fibonacci example, whose C++ based CnC code[*] for computing the n-th element is shown in Listing 2.1. As described in the previous section, obtaining an item produced by another task is done via a blocking *get* operation. If the item has not been produced yet, the calls to *get* (lines 4 and 6) cause the calling task to be delayed, or blocked, until the item is produced. Producing items is done through *put* calls with the item key and value as parameters, such as in line 7.

Each task (called *step* in CnC) of type fib::execute computes a single element of the sequence. To obtain the whole sequence up to the n-th element, steps need to be created with tags corresponding to the element they compute. This is done in the main function of the program, shown in Listing 2.2. On lines 2 and 3, the initial Fibonacci elements are added to the item collection. Line 3 adds CnC tags to the fib_tags *control collection*. In CnC, control collections are paired with a task type (in our case, fib::execute) such that a *tag put* to the control collection leads to a task spawn with the same tag. On line 6 we wait for all the tasks to finish execution and on line 8 we access the n-th element computed.

The code for the CnC graph is shown in Listing 2.3. This defines the step collection `m_steps`, the item collection `m_fibs` and the control collection `m_tags` which is going to prescribe the execution of steps from the step collection `m_steps`.

The above code in Listing 2.1 executes eagerly, possibly blocking at every `get`

[*]The syntax used here is for the Qthreads version of CnC [**Sbirlea:13**].

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

Listing 2.1: Fibonacci example in CnC

```
fib_graph g;
g.m_tags.put(1, 1 );
g.m_tags.put(0, 0 );
for( int i = 2; i <= n; ++i)
        g.fib_tags.put(i);
g.wait();
fib_type res2;
g.m_fibs.get(n, res2);
```

Listing 2.2: Fibonacci in CnC: snippet of the main function

```
struct fib_graph : public CnC::context< fib_context >
{
        // step collections
        CnC::step_collection< fib_step > m_steps;
        // Item collections
        CnC::item_collection< int, fib_type > m_fibs;
        // Tag collections
        CnC::tag_collection< int > m_tags;

        // The context class constructor
        fib_context()
        : CnC::context< fib_context >(),
        // Initialize each step collection
        m_steps( *this ),
        // Initialize each item collection
        m_fibs( *this ),
        // Initialize each tag collection
        m_tags( *this )
        {
                // Prescriptive relations
                m_tags.prescribes( m_steps, *this );

        }
};
```

Listing 2.3: Fibonacci CnC graph definition

```
aligned_t** fib::get_dependences(int tag,
fib_graph g, ...) {
...
aligned_t** read = malloc(...);
g.fib_values.wait_on(tag-1, &read[0]);
g.fib_values.wait_on(tag-2, &read[1]);
return read;
}
```

Listing 2.4: Fibonacci extra CnC code for strict preconditions

call. To use a preconditions-based approach, in the CnC runtime used for this work, one needs to specify the preconditions separately, as shown in Listing 2.4.

2.1.7 Memory management for CnC programs

One approach to addressing the drawbacks of shared data structures is to enforce a *dynamic-single-assignment property* for shared data accesses, since it in turn can establish data race freedom and determinism in parallel programs. Thus, the context for our work is parallel programming models for multicore and manycore processors in which all shared data accesses are performed through put/get operations on dynamic-single-assignment data structures indexed using associative tags (keys). These models include dataflow programs with single assignment semantics such as VAL [**McGraw:82**], Id [**Arvind:78**] and Sisal [**McGraw:85**] or with I-structures [**Arvind**] as well as functional subsets of parallel programs based on tuple spaces (notably, Linda [**Gelernter**]), and programs written in CnC.

It has been shown in past work that simple reference counting schemes can suffice for memory management of single-assignment programs with object references [**Gharachorloo:88**]. However, since CnC uses tuples to identify data items and and, since (unlike object references) tuple keys can be synthesized by arbitrary program computations, it can be challenging for the runtime to determine when a piece of data will no longer be accessed in the future. This in turn means references to items may be stored indefinitely leading to continually increasing memory footprints.

To keep the memory use in check, one approach is to manually de-allocate items during the execution of steps which are known to run after all consumers of each item have finished. However, this has a high programmability cost and can lead to hard-to-debug problems when items are incorrectly freed.

Another alternative technique called get-counts [**Sbirlea:12**] allows the user to specify a reference count for selected items. This count is decremented on every read operation for that item and when it reaches zero, the item is freed. This approach is more easily used than the manual de-allocation because programmers do not need to identify steps for which de-allocation can be performed, but has the disadvantage that it cannot be used to de-allocate items when the number of read operations can vary.

In conclusion, without additional memory management techniques, CnC is unsuitable for applications for which the memory footprint can grow unboundedly without garbage collection/deallocation.. The goal of this thesis is to propose techniques to take CnC from a model with limited memory efficiency compared to mainstream models to one in which the time-space balance can be easily controlled by the programmer and runtime.

2.1.8 Distributed CnC execution

The Intel Concurrent Collections runtime supports distributed execution that is almost transparent to the programmers [**DistributedCnC:13**] [*]. By default, the distributed runtime will do a round-robin distribution of tasks to nodes and data is delivered to them after they attempt to access it the first time. The reason for this is that without producer-consumer information, the runtime cannot know where the data will be consumed.

To solve this problem, the Intel authors propose a separate tuning mechanism through which the programmers explicitly specify nodes on which data will be consumed and can assign tasks to specific nodes.

While this feature does not simplify the problem of task and data placement for the programmer, it serves as a base runtime for our distributed programming optimizations described in Chapter 5.

2.2 Runtime parallelization

Runtime parallelization consists of techniques for enabling parallelization at runtime of codes that cannot be proved statically to be either parallelizable or non-parallelizable. For irregular loops, on which runtime parallelization is traditionally used, runtime parallelization would be used on loops whose iterations cannot unambiguously be proven independent or dependent at compile time.

A classical approach, proposed by Salz [**Salz:91**] is called inspector/executor (I/E) and proposes to handle this kind of loops by compiling two versions of the loop. The first version is called the *inspector* and its role is to calculate the values of index

[*]The primary additional code needed to support distributed execution is data marshaling code to allow data transfer between nodes.

expressions that are not available statically, by simulating whole-loop execution. This is usually done by executing the loop sequentially, with many expensive calculations omitted if they do not impact the dependence structure of the loop. This code implicitly computes a graph of inter-iteration dependences and computes a parallel schedule which is a wavefront through this iteration dependence graph traversed in topological order. All iterations in a wavefront are independent and thus can be executed in parallel and the number of wavefronts gives the critical path length. The executor simply follows the schedule to execute the loop.

To amortize the inspector overhead, I/E relies on the existence of multiple iterations of the loop with different data but the same dependence structure. With sequential inspector execution, the cost of the inspector may lead to no performance benefit for the technique. If the inspected loop is not an inner loop and shows no speedup, researchers have proposed different approaches for parallelizing inspectors, such as doacross [**SM:91**] or sectioning and bootstrapping [**LZ:91**].

2.3 Integer and linear programming

2.3.1 Linear programming

Linear programming (LP) is a set of algorithmic methods that can obtain optimal solutions for a mathematical model expressed as linear relationships.

The linear function to be minimized (or maximized) is called the *objective function.* It has the form $c^T \times x = c_1 \times x_1 + \cdots + c_n \times x_n$ with $c \in \mathbb{R}^n$ being a given vector. The linear inequalities in the model are called *constraints.* Linear programs have the following form:

$$\text{Maximize the value of } c^T \times x$$
$$\text{among all vectors } x \in \mathbb{R}^n \text{ with } A \times x \leq b$$

In the above formula, A is an input matrix of $n \times m$ size, where n is the number of variables and m is the number of constraints (equations), $c \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$.

Any vector $x \in \mathbb{R}^n$ that satisfies all constraints of the linear program is a *feasible solution*, but we are interested in the solution which leads to the maximum possible value of the objective, which is called *optimal solution.* Note that, in general, a linear program may have zero, one or infinitely many optimal solutions.

2.3.2 Linear programming algorithms

The *simplex algorithm* developed in 1947 by George Dantzig is one of the most widely used algorithms for linear programming. It starts by first transforming the input to an*equational form* which is as follows:

$$\begin{aligned} &\text{Maximize the value of } c^T \times x \\ &\text{s. t. } x \in \mathbb{R}^n \text{ with } A \times x = b \\ &\qquad\qquad\qquad\qquad x \geq 0. \end{aligned}$$

Note the presence of additional $x \geq 0$ constraints called *non-negativity constraints.*

This conversion to equational form can be done by adding slack variables to eliminate "less than" or "greater than" relations. Variables that can take both positive or negative values can be decomposed into two separate non-negative variables whose difference equals the original variable.

The algorithm then arranges the constraints and objective in a *simplex tableau* in which each equation is of the form: *variable* = ⟨linear expression of other variables⟩. Each tableau is associated with a basic feasible solution (geometrically, basic solutions correspond to vertices of the polyhedron of the solution space). After an initial tableau is completed, the method constructs a sequence of other tableaus by rewriting the previous tableau through application of a *pivoting step*. At each pivoting step, the goal is to select a variable which enters the basis and another one to exit the basis, so as to increase the value of the current objective. Once it reaches a point where there is no such transformation that can increase the objective value, the procedure stops because it has found the optimum.

The choice of the entering and exiting variables is a critical factor in determining the number of required pivoting operations. A few possible criteria for this are: largest coefficient (choose an entering variable which has the largest coefficient in the row of the objective), largest increase, steepest edge (choose an entering variable whose adoption moves the current solution in a direction closest to c), Bland's rule (choose smallest index). Bland's rule guarantees that the pivoting operations do not end up in a cycle, but the steepest edge criteria leads to the best performance in practice.

On some examples, the original pivot rule proposed by Dantzig (largest coefficient) was shown to need exponential number of pivoting operations [**Klee:72**]. Even for the best possible pivoting rule (a "clarvoyant" rule which leads to the smallest possible number of pivot operations), the best known bound so far is $n^{1+\log n}$ [**Kalai:92**]. This is worse than polynomial, but in practice, the number of pivoting operations is between $2m$ and $3m$.

The *ellipsoid method* is of great historical significance because it was the first linear programming algorithm which was proven to run in polynomial time, but in practice it is not competitive with the simplex method. This method consists of finding successively smaller ellipsoids that, until the last step, encircle the feasible solutions.

The interior points method relies on moving through points inside the feasibility boundary carefully avoiding it until the end. The key is avoidance of the boundary while at the same time increasing the current objective value. Some interior points methods have been shown to be polynomial in the number of bit required to store the coefficients required to express the linear programming problem.

It is interesting to note that the first linear programming problem was solved in 1947. It had nine equations and seventy-seven variables and was solved with hand-operated calculators in 120 man-days [**Matousek:06**]. Today, we can solve problems with tens of thousands of variables and constraints in a few minutes.

2.3.3 Integer linear programming

There are situations when only LP solutions whose variables are integers are of practical interest, and we call these problems *integer linear programming* (ILP) problems. These problems are as follows, assuming the same variables as described for LP:

$$\begin{aligned} &\text{Maximize } c^T \times x \\ &\text{s.t. } A \times x \leq b \\ &x \in \mathbb{Z}^n \end{aligned}$$

Unfortunately, solving ILP is considerably more computationally difficult than

LP: binary ILP is one of Karp's 21 NP-hard* problems [**Karp:72**]. In practice, there are problems with only tens of variables which are not solvable even with modern computers [**Matousek:06**].

If in a LP model only some variables have integrality constraint, then that model is *a mixed-integer linear model* (MILP).

2.3.4 Integer linear programming algorithms

MILP problems are generally solved with a linear programming variation of the *branch and bound* algorithm. Before applying the algorithm itself, a *presolve* stage applies a set of transformations to reduce the size of the input MILP problem.

The algorithm starts by solving the original MILP problem with the integrality constraints dropped. This modified problem is called *the linear programming relaxation* of the MILP problem and it can be solved efficiently with LP techniques. The resulting optimal solution will be optimal for the LP relaxation, but because it likely has variables with non-integer values it is not a feasible solution of the MILP problem.

The branch and bound algorithm picks a variable with fractional value called *branching variable* (say, x) and then eliminates the fractional value (say, $x = 4.7$) from the feasible domain by splitting the problem into two MILP problems, each with an additional constraint forcing the chosen variable to exclude the current value without eliminating any integer solutions ($x \leq 4$ and respectively $x \geq 5$). The optimal solution for the original MILP problem must be the optimal solution for one of these two subproblems. By repeating this process with the two subproblems, we obtain a search tree which expands as the algorithm continues.

*The NP-hardness of ILP means that we can reduce any problem in NP to the ILP problem. If there were a polynomial solution to the ILP problem, then we could efficiently solve any NP problem also, which would mean $P = NP$.

If a subproblem has been solved and happens to have a linear programming solution which also respects the integrality constraints, then the node is called *fathomed* and we do not need to branch on it - it remains a leaf of the tree. If the objective of this subproblem is better than the previously known best objective, then we call the subproblem *incumbent* and the subproblem with this property is updated as the algorithm progresses.

A subproblem can be infeasible in which case it is a leaf of the tree. Alternatively, if its LP solution may be worse than the incumbent, we do not need to branch since that subtree cannot possible offer a better MILP solution that the current one, so again the subproblem remains a perpetual leaf of the tree.

Some constraints (or "cuts") that reduce the number of undesirable fractional solutions could be applied during the presolve stage but are are instead applied during the algorithm itself. This is because there may be a huge number of such constraints and not only it would be exceedingly expensive to find them all, but they also would make the LP relaxation harder to solve. The techniques known as *cutting planes* consist of judiciously adding these constraints during the solving process only if they are known to help. Gomory's original cuts [**Gomory:58**] were cuts that separated the vertex representing the objective of the LP relaxation on one side and all the integer feasible points on the other.

If we only use cuts and apply them repeatedly until all the variables in the optimal solution becomes integers, then the method is a *cutting planes method.* If we combine the branch and bound techniques with cuts as described above, the combined approach is called *branch and cut.*

2.4 Parametrized distribution functions

Traditionally, in distributed systems, distribution functions map elements from an input array to a $\langle node, offset \rangle$ pair, where *node* is the processor where the element would be found during execution and *offset* is the offset within the node where that particular element resides. Since the offset only impacts local computations, we will focus on the node mapping and ignore the offset in the discussion of distribution functions below.

The data distribution is an essential performance choice in the design of distributed applications and has received considerable attention from the research community with many different distributions that have been proposed [**Brent:92**, **Huss:94**, **Hendrickson:94**]. The traditional block, cyclic and block-cyclic distribution patterns shown in Figure 2.1 have proven useful in previous distributed programming models.

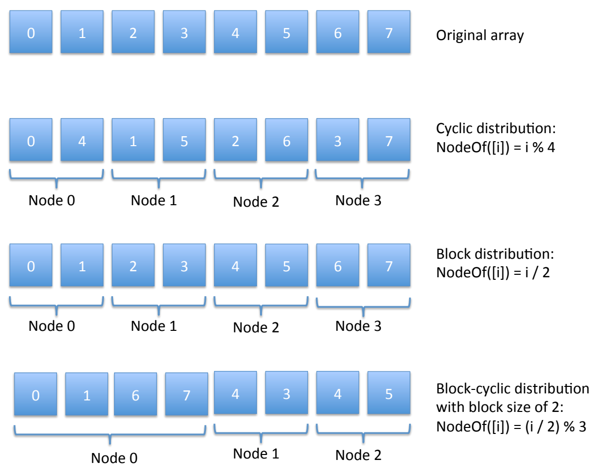


Figure 2.1 : The cyclic, block and block-cyclic distributions are widely known.

The first distribution shown is the *cyclic* (also called *scattered*) distribution, in which consecutive array elements are assigned to consecutive nodes (also traditionally called processors). After reaching the last processor, the process wraps around to the first one. For example, assuming P nodes and an input array indexed by m with $0 \leq m < M$ with $M \in \mathbb{N}$, the distribution would be $\beta(m) = m \bmod P$.

The second distribution is called the *block distribution* because the elements are grouped into a number of blocks equal to the number of nodes and the block are then assigned to nodes consecutively. For example, assuming P nodes and an input array indexed by m with $0 \leq m < M$ with M a multiple of P, the block size would be $BS = \frac{M}{P}$. The distribution is $\beta(m) = \lceil \frac{m}{BS} \rceil$.

The third distribution is the *block-cyclic distribution* in which $\beta(m) = \lceil \frac{m}{BS} \rceil \bmod P$ with the $BS \in \mathbb{N}$. Notice that for $BS = \frac{M}{P}$, the block-cyclic distribution equals the block distribution (when M is a multiple of P) and for $BS = 1$ it equals the cyclic distribution. Because it includes an unknown parameter value (BS) we consider it to be a distribution pattern rather than a distribution function.

The distribution functions described above assume an input data indexed by a single integer value (or ignore other dimensions in the index). For this reason, they are called $1D$ distribution functions. Below, we present the n-dimensional definition of distribution functions as presented by Thomas Rauben and Gudula Rungen [**Rauben:11**].

We assume the input data to be a d-dimensional array A with index set $I_A \subset \mathbb{N}^d$. The size of the array is $n_1 \times \cdots \times n_d$ and the array elements are referred to by using $A[i_1, \ldots, i_d]$ with an index $i = (i_1, \ldots, i_d) \in I_A$. Array elements distributed to an d-dimensional grid of processors $p_1 \times \cdots \times p_d$ with $p = \prod_{i=1}^{d} p_i$. A data distribution function $\gamma_A : I_A \subset \mathbb{N}^d \to 2^P$ (2^P denotes the power set of processors P). According to γ_A, the i-th element of A, with $i = (i_1, \ldots, i_d)$ should be distributed to all processors in $\gamma_A(i) \subseteq P$.

The ScaLAPACK [**ScaLAPACK**] library proposes the $2D$ block-cyclic distribution pattern as a scalable distribution for linear algebra kernels [**Dongarra:94**, **Dongarra:92**] and is widely used by other libraries, but part of the reason may be historic [**Anderson:91**]. Some researchers have questioned the use of this distribution pattern for distributed linear algebra problems [**Edwards:95**, **Sundaresh:07**], but it is unclear if better alternatives exists.

Chapter 3

Folding of Dynamic Single Assignment Values

3.1 Motivation

The multicore revolution has increased the urgency for developing programming models that deliver scalable parallelism with minimal effort by programmers. The use of shared data structures by parallel tasks has proved to be a two-edged sword in pursuing this goal. On the one hand, a shared address space can reduce the semantic gap between a sequential program and its parallel version. On the other, uncoordinated accesses to shared data structures are a notorious source of bugs that arise from data races and other sources of nondeterminism leading to the *programmability wall.*

One approach to addressing the drawbacks of shared data structures is to enforce a *dynamic-single-assignment property* for shared data accesses, since it in turn can establish data race freedom and determinism in parallel programs. Thus, the context for our work is parallel programming models for multicore and manycore processors in which all shared data accesses are performed through put/get operations on dynamic-single-assignment data structures indexed using associative tags (keys). These models include dataflow programs with I-structures [**Arvind**], functional subsets of parallel programs based on tuple spaces (notably, Linda [**Gelernter**]), and programs written in the Concurrent Collections (CnC) coordination language [**Budimlic:10**].

However, past experiences with implementations of functional languages have shown that memory management can be challenging with the dynamic-single-assignment property. It becomes even more challenging when objects can be accessed through user-computable tags, since standard reference-based garbage collection cannot be applied in that case. In this chapter, we propose a new memory management approach based on user-specified *folding functions* that map logical dynamic-single-assignment (DSA) tags into dynamic-multiple-assignment (DMA) tags. We also compare folding with *get-counts*, an approach in which the user supplies a function that maps tags to integers indicating the number of gets that will occur on the item. Both approaches are *fail-safe* i.e., an exception is thrown if the program performs accesses that are inconsistent with the folding functions or get-counts.

There has been a lot of past work focused on converting a multiple-assignment program to dynamic single assignment form so as to simplify program optimization and transformation. An early paper [**feautrier91dataflow**] described several applications of dynamic single assignment, such as conversion of a program to a set of recurrence equations, scalar expansion, array expansion [**arrayExpansion**], program verification and parallel program construction. In contrast, folding addresses the dual problem of converting a dynamic single assignment program to multiple-assignment form with reduced memory requirements. Based on the well known challenges in transforming static single assignment form to multiple assignment form [**outOfSsa**], it is natural to expect that translating out of dynamic single assignment form will be a challenging problem too, especially when the original non-DSA program is unavailable. To the best of our knowledge, we are the first to propose a user-specified "folding" approach to address this problem.

In summary, this section describes the following contributions:

- *Basic folding* (Section 3.2), a novel memory management technique for accesses to associative dynamic-single-assignment data structures (item collections). This technique relies on user-specified folding functions with fail-safe checks for correctness at runtime.

- *Update-in-place memory reuse* (Section 3.3), an extension that allows the user to specify *GetForUpdate* operations that allow an input item to be rewritten as an output. This approach can be used both with folding functions and get-counts, and includes fail-safe checks as well.

- *Extended folding* (Section 3.6), an extension to basic folding for items that are written but never read.

- *A design and implementation* (Section 3.7) of the above folding and get-count techniques for the CnC model.

- *Empirical results* (Section 3.8) that show that folding and get-counts can offer significant improvements in memory efficiency over the baseline version without these techniques.

3.2 Basic folding

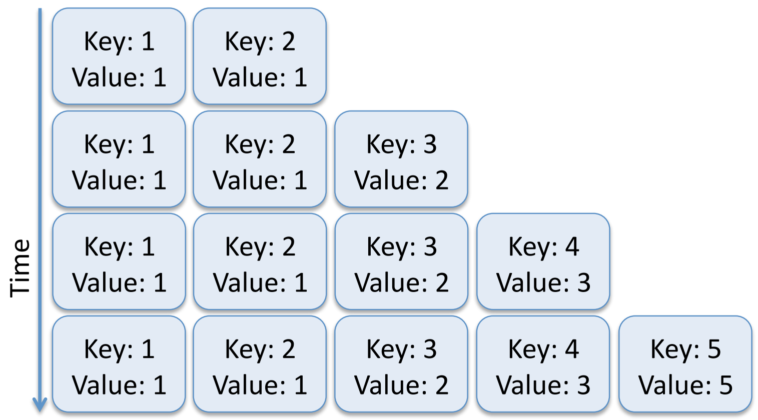
The intuition behind folding is as follows: if we know that two values have non-overlapping lifetimes, we can assign them to the same physical storage thereby reducing the maximum memory requirement for the application. Following the terminology used in the CnC model, we refer to the associative dynamic-single-assignment (DSA) data structures assumed in this work as *item collections*, to keys as *tags*, values as *items*, and computational tasks as *steps*. The two operations supported by item collections are *put(tag, item)* and *get(tag)*. The DSA property requires that dynamically at most one *put()* operation be performed for a given tag. Further, each *get()* operation is assumed to be blocking i.e., it only returns a value after a *put()* operation has been performed with that tag.

Definition 1 (Folding function)**.** *A folding function f transforms a* logical tag t_1 *to a* physical tag*, $f(t_1)$. Thus, the logical* put(t$_1$, i_1) *operation is transformed into a* physical put(t_1, f(t$_1$), i$_1$) *operation, where $f(t_1)$ is the physical location used to store the item and the original tag t_1 is stored as an auxiliary value. Likewise, the logical* get(t$_1$) *operation is transformed into a* physical get(t_1, f(t$_1$)) *operation.*

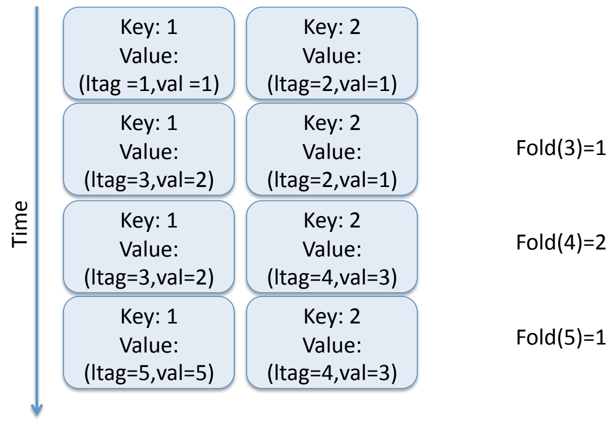
Thus, the folding function maps DSA tags to dynamic multiple assignment (DMA) tags which are associative indices into a physical store. When a new item i_2 is mapped to the same physical store location as a previous item i_1 (because $f(t_1) = f(t_2)$), the space of i_1 is freed. Example executions of a program that computes the n-th Fibonacci element are in Figures 3.1a and 3.1b (without and with folding, respectively). Item n can fold over item $n-2$. The folding function used is: $fold(n) = (n+1)\%2+1$.

This use of a folding function is called basic folding. As discussed later in Section 3.4, a runtime error may be thrown if the folding function is specified incorrectly, but a *get()* operation will never return an incorrect logical value.

We now identify the conditions under which folding is legal. As an example, consider the following sequence of logical *get()* and *put()* operations: "*put(t_1, i_1); get(t_1); put(t_2, i_2); get(t_1)*". In this case, it would be illegal to fold items i_1 and i_2 on the same location because they have interfering live ranges [**Torczon**]. To ensure safety for folding two items, they must have disjoint lifetimes in any possible schedule of the program.



(a) Item collection content for a baseline execution of Fibonacci.


Time
Key: 1 Value: (ltag =1,val =1)
Key: 2 Value: (ltag=2,val=1)
Key: 1 Value: (ltag=3,val=2)
Key: 2 Value: (ltag=2,val=1)
Fold(3)=1
Key: 1 Value: (ltag=3,val=2)
Key: 2 Value: (ltag=4,val=3)
Fold(4)=2
Key: 1 Value: (ltag=5,val=5)
Key: 2 Value: (ltag=4,val=3)
Fold(5)=1

(b) Item collection content for a folding execution of Fibonacci.

Figure 3.1 : CnC Fibonacci execution.

Definition 2 (Item lifetime)**.** *The* lifetime of an item *in a program execution is the interval between the execution point at which the item is produced by a* put() *operation and the execution point of the last* get() *operation performed on the item. If there are no* get() *operations, the lifetime begins and ends at the* put()*.*

Definition 3 (Legal program)**.** *A legal program is one that always completes execution with all* get() *operations having successfully completed, for all possible schedules.*

Definition 4 (Correct folding transformation)**.** *A folding transformation for a legal program P specified by folding function f is correct if, for every input I, an execution* *of P with input I and folding function f is also legal (no blocked gets()) and results in the same result for each* get() *operation as the original execution of program P without folding.*

Theorem 1 (Folding correctness requirement)**.** *For a folding transformation of a legal program to be correct, the folding function must not fold together any two items whose lifetimes may overlap.* [Proof omitted due to space limitations.]

3.3 Folding with memory reuse

Basic folding ensures that memory can be reclaimed after the end of a computational step that performs the last logical *get()* operation on an item. However, many steps have the following computational structure: "$i_1 = get(t_1);$ *allocate* i_2*;* i_2*.set(G(*i_1*)); put(*t_2*,* i_2*);*". With basic folding, both i_1 and i_2 will be assumed to be simultaneously live and will contribute to the maximum memory requirement for the program. However, if function G can be implemented as an *update-in-place* function, then i_1's storage can be reused for i_2 if get(t_1) is the last get operation performed with logical tag t_1. To enable this optimization, we allow the user to use a *getForUpdate()* operation instead of *get()*, as an indication that this is the last *get()* operation for the given tag in any schedule, thereby making it possible for item i_1 to be updated in place to obtain item i_2. Figure 3.2 is an example. As with the folding function, the correctness of a *getForUpdate()* operation will also be checked at runtime so as to guarantee fail-safe behavior (see Section 3.4).

3.4 Error detection

The folding error detection mechanisms are based on the assumption that the original program is legal (Definition 3) without the folding optimization. We define any

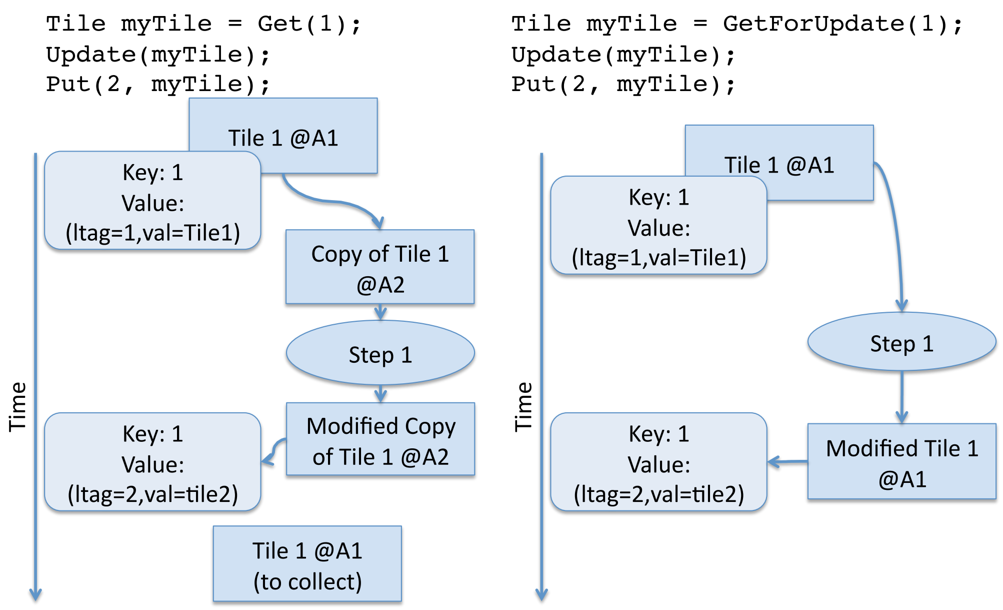


Figure 3.2 : Left: With a get() call, the item memory is copied before being returned to the step, which can modify it and put() it with some other tag. This leaves the old item memory to be collected when an item folds over its entry in the store. Right: With getForUpdate, the copy is not performed and no memory will need to be collected, as it is reused by the new item.

behavior of a legal program in the presence of folding that differs from the behavior of a non-folded execution as an error.

For example, a *get()* that returns an incorrect value would constitute an error. This could happen, if the content of the physical store location corresponding to a particular tag is returned without checking that the logical tag of the item in that location corresponds to the logical tag of the item we are trying to get. If the item in the store does not have the same logical tag, we need to wait for it to be produced. However, if the item was previously produced and some other item was erroneously folded over it, we will never find the item. Without an error checking mechanism, the program may finish with blocked steps instead of the correct non-folded behavior.

To enable detection of such errors, we define a debug mode for folding, in which a boolean flag is stored for each tag that is *put()* during execution. Using this flag, we can differentiate between items that are not present in the physical store because some other item was folded over them and items that have not been produced as yet. A *get()* performed on previously overwritten items should throw an exception reporting an incorrect folding function, but a *get()* should block until the item is produced if that is not the case. In debug mode the system also detects dynamic single assignment violations (on every put, if the boolean flag for that logical tag was previously *put()*, we report an exception) with or without the presence of folding.

3.5 Programmability benefits of folding

To illustrate the benefits of folding with error detection, consider a common technique used by performance-oriented C programmers where storage is reused instead of calling free() followed by malloc(). This approach can be especially error-prone for parallel programs, because the overlap in lifetime between the initial and subsequent values may be schedule-dependent. With folding, a similar reuse of memory could be achieved in a fail-safe manner by folding the two logical items and using the getForUpdate mechanism for memory reuse.

As a concrete example, consider the classic two-buffer approach used by iterative algorithms in which one buffer is used as an input and the other as the output, and their roles are swapped in each sequential iteration. With our folding approach, the programmer can think in terms of allocating a new DSA output buffer in each iteration, and a folding function can effectively perform the swap. This approach was used in our implementation of a Routing simulation application (see Section 3.8) where the routing tables for one iteration are built using the routing tables of the previous one, and a folding function was specified as follows:

```
public final Object fold(point tag) {
   int i, j, k; //i: node id, j: iteration id; k: repetition #
   i = p.get(0); j = p.get(1); k = p.get(2);
   return new point(i, j%2, k);
}
```

3.6 Extended folding: Folding with ordering

Items with empty lifetimes pose an interesting research challenge for folding. Consider a program that expects to produce and consume items in order as follows: "*Step1:[put(t_1, i_1)] Step2:[get(t_1);put(t_2, i_2)] Step3:[get(t_2); put(t_3, i_3)] Step4:[get(t_3)]* ". In such a case, it might seem reasonable to fold t_1, t_2, and t_3 to the same physical location. However, if (say) *get(t_2)* is not performed for some reason, there is no way (if using blocking-get synchronization only) to ensure that *put(t_2, i_2)* completes before *put(t_3, i_3)*, thereby making the folding incorrect (because *get(t_3)* may never find t_3 as it has been folded over).

This is an instance of the more general problem caused by optional *get()* calls but in this particular case there is a way to solve the problem. We propose an extension to folding that allows folding of items that may never be consumed. Such items can appear when control dependent gets are used, for example with short-circuit boolean operations such as "get(t_1) && get(t_2)". We observe that items that are never read have an empty lifetime and can be optimized away from the physical store. However, this may not be known at the time of the *put()* operation, but may be known when a subsequent *put()* is performed on the same physical location.

We can express this by allowing the presence of an additional user function that acts like a "compare age" operation. If an item that is being put maps to a physical location where another item resides and should be declared dead, the function returns *true* ("newer"), and the new item is stored. Otherwise, if the new item is known to never be read, it returns returns *false* ("older"), the incoming item is not stored and the old item is retained.

To perform the age comparison, the function needs two parameters: the tag of the item being put currently and the tag of the old item that exists in the location in the physical store where the new item would be inserted. The programmer has to identify if the tag of the current item in the item collection means that all of the steps that could access the incoming item have executed and did not access the incoming item. If this is the case, then the incoming item can safely be discarded. The Rician Denoising benchmark (see Section 3.8) uses this extension.

3.7 Implementation

We have implemented folding as an extension to the Habanero Java CnC runtime [**Budimlic:10**]. The Java key-value data structure used to implement item collections is now indexed by DMA tags instead of DSA tags. When an item is put() with DSA tag t_1 its corresponding DMA location in the store is determined by identifying $pt_1 = f(t_1)$, where f is the folding function. Then, the physical store is accessed to see if there is any entry at that physical location. If there is none, we create it, and label it with the logical tag t_1. If there is, we need to hold a lock on the physical store location while the following operations are performed. First, we update the logical tag of the physical store entry to the logical tag of the item that has just been put. Then, we go through the list of steps waiting on that particular physical store location and, for each step that is waiting for the current item mark it as ready for execution. The marked marked continue their execution by performing a *get()* that will succeed because the desired item is already in the physical store.

When a get() on item with DSA tag t_1 is performed, its DMA tag is determined by identifying $pt_1 = f(t_1)$. If the entry does not exist, it is created, inserted in the physical store and the step is added to its list of waiting steps. If the entry does not correspond to the logical tag of the item, it registers itself to wait also. Compared to a non-folding execution, the only extras step needed for insertion is the application of the folding function (which does not need synchronization and has minimal overhead). The bigger overhead is in the *put()* , where the list of waiting steps has to be checked linearly to unblock only the steps that are waiting for the new item and this happens while holding the lock. We chose to have the overhead in the *put()* and not *get()* as the *get()* is usually performed multiple times on a single item and our approach leads to less contention.

Both the get-counts and folding policies only remove items from item collections, so that there is no object reference pointing to them; the Java garbage collection subsequently reclaims the memory.

3.8 Results

The following results were obtained on a 16 core Xeon system with 16GB RAM, running Habanero Java implementation of Concurrent Collections [**Budimlic:10**] on a 64 bit Java 1.6, using 16 workers for the work-stealing CnC runtime and Java default garbage collection mechanism. In this section we compare the performance and memory footprint of the following CnC memory management policies:

1. *Baseline*: non-collecting CnC (items are never removed from item collections) leading to memory leaks, but also no folding overhead.

2. *Get-counts*: memory management in which the user specifies a reference count for selected items, the count is decremented on every get() operation on a specified item, and the item is freed when the count becomes zero.

3. *Folding*: the folding runtime described in Section 3.7. We used the ordering extension described in Section 3.6 as needed and the tables contain the “Ordered" specifier where this happened.

For each policy used, we obtained the following measurements:

1. Execution Time - We performed thirty repetitions of the program in the same JVM instance, and reported the average, as advocated in [**javaPerformance**].

2. Memory at end - the program footprint after the CnC graph finishes execution. With this metric, get-counts has an advantage because it removes items immediately, where as folding waits for the birth of another item, so at the end folding usually has more live items. In contrast, folding saves some work by taking a lazy approach to freeing items.

3. Items at end - similar to the previous metric, but expressed in items.

We evaluated the impact of folding and get-counts on the following applications:

1. *Microbenchmark* showing the difference in scalability between get-counts and folding with the number of reads per item.

2. *N-body simulation* for performance analysis.

3. *Routing simulation* as an application in which get-counts might lead to leaks because items have a number of accesses unknown at creation time, but folding works without needing the Ordered extension.

4. *Rician denoising* as example of an application in which folding with ordering can safely be used, but get-counts leads to leaks because some items have data-dependent accesses whose number is unknown.

5. *Cholesky factorization* as an example of memory reuse via the getForUpdate optimization.

Microbenchmark: Scalability with read/write ratio This benchmark varies the reads to write ratio to analyze the performance of the two collection mechanisms. Because folding performs most of the synchronization on *put()* as opposed to get-counts, which performs most of the synchronization on *get()*, we checked if the best performing policy might be get-counts for low read/write ratio. However, as shown in Figure 3.3, the folding version runs faster than both get-counts and baseline CnC even for a ratio of 1. Some applications may have a read/write ratio lower than one; performance for this case is analysed later using the Rician Denoising application.

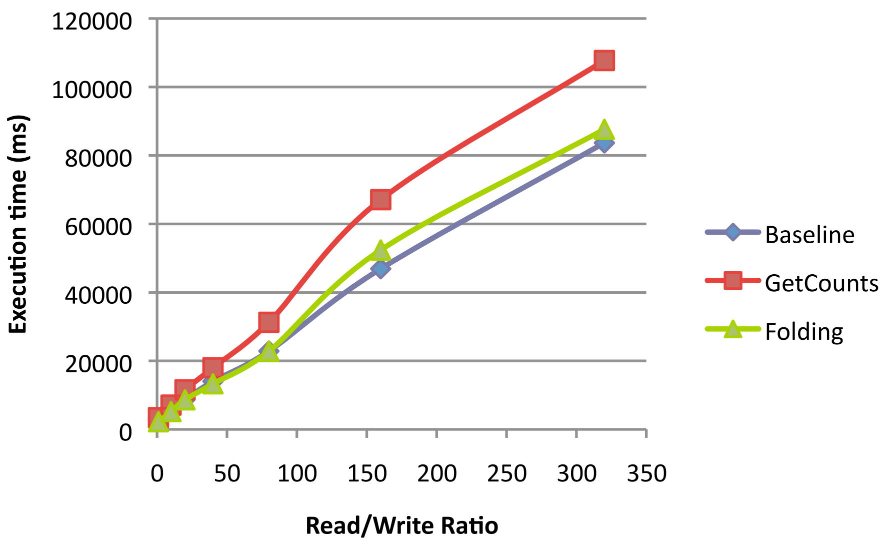


Figure 3.3 : Performance with read/write ratio (16 core Xeon)

N-Body Simulation We implemented the O(N^2) algorithm for N-body simulation with both get-counts and folding and the results are shown in Table 3.1. The folding policy performs well because this benchmark has a small step granularity, large number of items and thus more contention on the item collections. The fact that folding has less synchronization of gets (in this application there are 10 gets per item) leads to a consistent (1.3×) performance improvement compared to get-counts. The get-counts footprint is smaller because folding can only reduce the footprint to the maximum footprint of the program during its execution, and in this case, that footprint is 20 items, which is also the maximum theoretical footprint for get-counts.

CnC policy	Time (s)	Memory at end (bytes)	(items)
Baseline	16.9	277.0 MB	1,000,005
Get-Counts	18.1	3.6 KB	10
Folding	**13.0**	7.0 KB	20

Table 3.1 : Experimental results for NBody (5 bodies, 100000 timesteps)

CnC policy	Time (s)	Memory at end (bytes)	(items)
Baseline	**21**	61.0MB	102000
Get-Counts	25	10.7KB	1000
Folding	**21**	1.3MB	2000

Table 3.2 : Experimental results for Routing, with reliable links.

Routing Simulation The routing simulation benchmark has unknown number of gets on each item, making it a challenge for the get-counts approach. It simulates the convergence of min-distance routing protocols such as IS-IS [**isis**] and OPSF [**opsf**]. As links might go down, when a routing table is being built, we cannot know how many gets will be performed on that node. In such cases, the get-count will never reach zero and the item will become a memory leak. To see how the number of leaked items varies with the chance of links failing we varied the chance of a message not getting through from 0 to 10%, as shown in Figure 3.4: at only 1% failure rate half the items are leaked. Even in the absence of link failure, folding shows a 16% performance improvement over get-counts (Table 3.2).

Rician Denoising Rician (Poisson) denoising is an image processing application. Its global convergence check is a reduction on the convergence status of all tiles and it is sped up using a short-circuit evaluation: if a single tile changes significantly we do not need to wait for the convergence condition of all the other tiles to be evaluated,

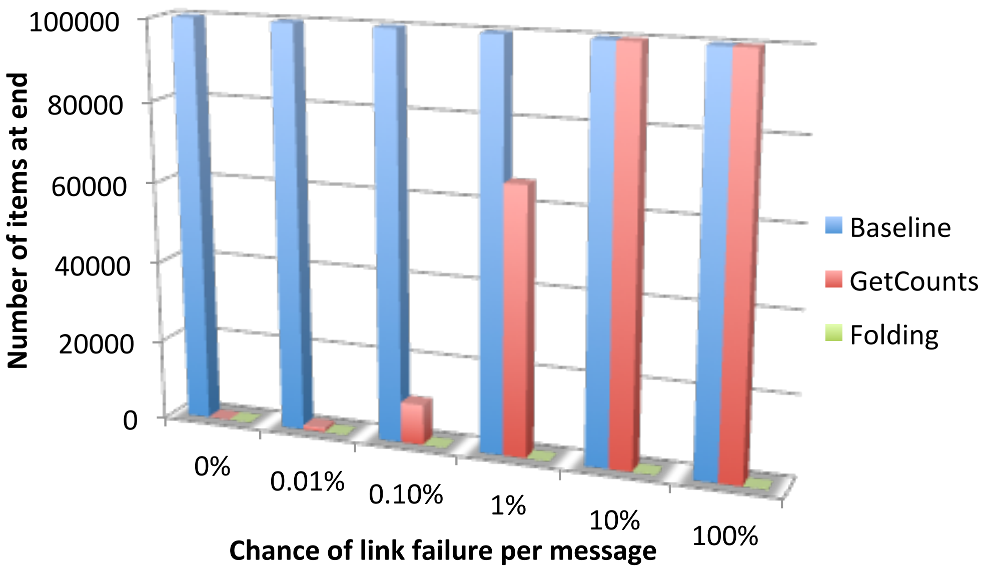


Figure 3.4 : The relation between the link fail rate and memory leaks. At only 10% link failure rate, a large majority of items are leaked when using GetCounts. Folding is able to maintain a minimal footprint relative to both baseline and GetCounts.

we immediately know we will need an additional iteration and can start spawning the corresponding steps.

The results (Table 3.3) show the performance of get-counts and folding: folding offers the best performance. Furthermore, get-counts leads to leaks of items from the ConvergenceStatus item collection in which the operands of the short-circuit operators are stored (the cause of the leaks is the unknown number of gets): 95% of items stored in that item collection are leaked, totaling 20MB. However, without short-circuit operators, get-counts collects more items because at the end of the program, all the items stored in the item collections that store intermediate results (Gradient, ImageTimesGradient, etc) can be collected. This does not affect the actual high water-mark of the program which is the same in both folding and get-counts executions.

Short circuit	CnC policy	Time (s)	(MB)	Memory at end (number of items in each collection)				
				Image	Gradient	Image× Gradient	Factor	Convergence Status
Enabled	Baseline	6.5	2888	10800	10400	10400	10400	10400
	Get-Counts	**2.6**	740	800	0	0	0	9897
	Folding (Ordered)	**2.6**	800	1200	800	800	800	800
Disabled	Baseline	8.1	2888	108006	10400	10400	10400	10400
	Get-Counts	3.7	720	800	0	0	0	0
	Folding	**3.5**	830	1200	800	800	800	800

Table 3.3 : Performance comparison for Rician Denoising with shortcircuit reductions disabled and enabled (image size 2560 × 1280, tile size 128 × 64).

Cholesky Factorization Cholesky factorization is a numerical application whose input is a symmetrical positive-definite matrix and output a lower-triangular matrix. One possible CnC implementation was previously described and benchmarked in [**Aparna:10**] and the results were encouraging.

Table 3.4 shows that the proposed update-in-place optimization, if applied on either get-counts or folding, can lead to a large performance increase. Using getForUpdate leads to a performance improvement between 10% and 20% for both collecting policies. Baseline CnC cannot safely apply this optimization without additional programmer input to ensure that whenever GetForUpdate is called, the item accessed is indeed dead. To work around this, we manually added this call only when such accesses are safe.

Memory High-watermark comparison Table 3.5 shows the maximum number of live items during the execution of the benchmarks. This metric shows, in the sched-

Input size	CnC Policy	Without update-in-place			With update-in-place		
		Time	Items at end		Time	Items at end	
		(s)	(MB)	(items)	(s)	(MB)	(items)
2000	Baseline	**0.9**	142.2	952	0.8	33.7	952
	Get-Counts	**0.9**	33.7	272	0.8	33.7	272
	Folding	**0.9**	33.7	272	**0.7**	33.7	272
4000	Baseline	7.3	1008.5	6512	5.6	133.2	6512
	Get-Counts	**6.2**	133.2	1056	5.6	133.2	1056
	Folding	**6.2**	133.2	1056	**5.0**	133.2	1056
6000	Baseline	26.6	2680.2	20776	19.5	298.4	20776
	Get-Counts	22.3	298.4	2352	19.2	298.4	2352
	Folding	**21.6**	298.4	2352	**19.1**	298.4	2352

Table 3.4 : Performance comparison for Cholesky factorization (125 $\times$ 125 tiles).

ules and with the parallelism actually used during execution, what is the maximum number of items that were live - the memory “high-water mark" of the program. To obtain these values we used atomic counters that tracked the number of stored items. The results show that maximum live items number is lower than the bound identified by folding. However, in the future, as the number of processors grows, more tasks will run concurrently and the number of live items will increase.

Benchmark		**Baseline**	**Get-Counts**	**Folding**
Nbody		1,000,005	19	20
Routing		102000	1100	2000
RicianDenoising	Image	10800	800	800
	Gradient	10400	27	800
	Image $\times$ Gradient	10400	26	800
	ConvergenceStatus	10400	9897	800
Cholesky (6000)		20776	2352	2352

Table 3.5 : The maximum number of items live during execution.

3.9 Conclusions

In this chapter, we introduced a new memory management approach based on user-specified *folding functions* that map logical dynamic-single-assignment (DSA) tags into dynamic-multiple-assignment (DMA) tags, while preserving semantic guarantees of data race freedom and determinism. Our approach is applicable to parallel programming models in which shared data accesses are coordinated by put/get operations on tagged DSA data structures. These models include dataflow programs with I-structures, functional subsets of parallel programs based on tuple spaces (notably, Linda), and programs written in the Intel Concurrent Collections (CnC) coordination language. Our conclusion, based on experimental evaluation of five CnC programs, is that folding can offer significant memory efficiency improvements, and that folding can handle cases that get-counts (an alternative approach to user-specified memory management) cannot. An interesting direction for future work is automatic generation of folding functions. In many of the benchmarks that we studied, it is possible to use static analysis of get and put function parameters to identify candidates for folding.

Chapter 4

Bounded Memory Scheduling of Dynamic Task Graphs

4.1 The bounded memory scheduling problem

We propose a programming system that targets the *bounded memory scheduling* (BMS) problem: *Given a parallel program P with input I and a memory bound M, can P complete execution in the memory bound M?*

We propose an inspector/executor [**Salz:91**] based model that enables dynamic program analysis, transformation and optimization based on the computation task graph at runtime, but before running the application. To the best of our knowledge, this work is the first to consider the BMS problem in the context of dynamic task scheduling. This problem is a more general case of the register sufficiency problem [**Garey:1979**], which has been well studied due to its importance in compiler code generation. In the context of task scheduling, additional difficulty arises from the fact that, in most programming systems, there is insufficient information at the point when a task is created to decide if it should be deferred or handed to the scheduler directly in order to maintain the memory usage within the desired bound. Without an oracle to answer this question, the BMS problem becomes intractable. We propose a scheduling approach in which the role of the oracle is performed by the inspector phase of an inspector/executor [**Salz:91**] system. Our parallel programming model (see Section 4.2) enables the inspector to build the computation graph of compliant applications without running the internals of computation steps in the application, thereby revealing both the parent-child relationships for tasks and the reader-writer relationships for data. With this knowledge, the inspector can identify scheduling restrictions that lead to bounded-memory execution. These restrictions are then enforced by the executor stage, when the application runs on a load-balancing work-stealing scheduler. The result is a hybrid scheduling approach which obeys a memory bound but retains the advantages of dynamic scheduling.

The main contributions of this work are:

- *a heuristic algorithm for BMS based on the inspector/executor model* for identifying a set of schedules that fit a desired memory bound. The BMS algorithm is run in the inspector phase and works by imposing restrictions on the executor phase.

- *an optimal algorithm for bounded memory scheduling* based on integer linear programming; as opposed to the heuristic algorithm, it is optimal in that it ensures finding a schedule that fits the memory bound if one such schedule exists. By proposing an efficient ILP formulation and by using the result of the heuristic BMS to hot-start the optimal algorithm, our formulation works on graphs that are an order of magnitude larger than those reported in previous work on ILP-based register scheduling.

- *a schedule reuse technique* to amortize the cost of the BMS inspector across multiple executions by matching new runs to previously computed schedules. This technique works whenever the runs have the same dynamic computation graph, even if their inputs differ and, to the best of our knowledge, is the first to reuse inspector-executor results across application runs.

- *experimental evaluation on several benchmarks* showing that the range of memory bounds and parallel performance delivered by BMS gracefully spans the spectrum from serial to fully parallel execution.

4.2 BMS-CnC: an inspector/executor parallel programming model

Many analyses of task-parallel programs (such as data race detection) require understanding the task-parallel structure of the computation, which is usually unknown at compile time. As a result, many of these analyses build the task graph dynamically, while the application is running. Unfortunately, this is too late for certain optimizations, such as bounding the memory consumption of the program.

We propose the use of an inspector/executor programming model in which an analysis (inspector) phase is performed before any tasks start executing. The inspector uncovers the task creation and data communication patterns of the application without running the internals of computation steps; the information it uncovers can be used for program transformations. As soon as the transformation completes, the executor starts running the transformed program.

Specifically, we introduce BMS-CnC, a CnC variant that adds programmer-written *graph expansion functions*, associated with each step collection. These functions enable the inspector to query the input and output items and spawned steps of each step, without performing a complete step execution. The set of keys corresponding to items read by the step with tag `t` is returned by the programmer-written `get_inputs(t)` function. Similarly, `get_outputs(t)` and `get_spawns(t)` return the keys of items produced by the step and the tags of steps spawned by it. *

An additional expansion function deals with those items that are the output of the whole CnC computation. Before spawning the first step, programmer needs to identify items `k` that are read by the environment after all CnC steps have finished, through calls to `declare_get(k)`.

BMS-CnC uses a CnC runtime in which tasks do not start executing until all input items are available (known as strict preconditions [**Sbirlea:13**]), which means that tasks have only two states before termination: *prescribed* (expressed to the runtime by a spawn call) and *running*. In the prescribed state, tasks consume memory consisting of a function pointer and the tag tuple; during execution, they also use the stack. Because they never block, there are only as many task stacks as there are workers. Since task stacks are fixed-size†, the stack memory consumption is constant during execution.

4.2.1 Programming model characteristics useful for BMS

Several features make CnC an ideal candidate for BMS:

- CnC makes it easy to separate data and computation, simplifying the implementation of the inspector-executor approach and reducing the inspector overhead.

- Assuming there are no data-races, CnC programs are deterministic [**Budimlic:10**[redacted]], enabling BMS schedule reuse across multiple runs (Section 4.6).

*Note that tasks can make conditional `puts` and `gets` in BMS-CnC, the only requirement is that these must also be expressed in the corresponding graph expansion function, so any such condition has to be a pure function of the step tag. See subsection 4.2.2 for a discussion.

†We allocate fixed-size stacks for each task. If more stack space is needed for activation frames, the task can create additional child tasks; if it needs more space for stack data, it can create CnC items instead.

- The fact that CnC uses the item abstraction for all inter-task communication makes it easy to evaluate how much memory is used for data in the parallel program.

- CnC tasks only wait on items, before running [**Sbirlea:13**]. This minimizes the number of task states, making the memory accounting easier than in other models.

- CnC steps finish without waiting for their spawned children to finish and do not use stack variables to communicate with other tasks. This behavior is different from spawn-sync models where parent stack cannot be reclaimed until all children have finished. In BMS-CnC, there will only be as many task stacks as there are worker threads (a constant amount of memory).

- The dynamic single assignment property implies that there are no anti and output dependences between steps, which increases parallelism and gives BMS the maximum flexibility in reordering tasks.

- CnC items are usually tiles, and steps are medium-grained ("macro-dataflow") keeping the graph of the computation at a manageable size and decreasing the overhead of the inspector phase.

4.2.2 Independent control and data as a requirement for BMS

Since BMS-CnC relies on the programmer to separate the computation graph from the internals of the computation through expansion functions, an important question arises: *Is it always possible to separate the computation structure from the computation itself?* In general, the answer is *no*.

The problem can be illustrated with `step_collection. get_inputs(t)` in the case when the step reads two items. If the key of the second item depends on the value of the first item (not only on the tag of the step) then it is impossible to obtain the key of this second item without actually executing the step that produces the first item. This example is an instance of an application pattern called "data-dependent `gets`". A related pattern is that of "conditional `gets`'," in which the read operation on an item is conditional on the value of a previously read item and leads to the same issue. Similar issues happen for `puts` and can be worked-around by putting empty items instead of doing conditional `puts`.

If the keys of items read and written and tags of steps spawned are only a function of the current step tag, then the application has *independent control and data*, which is needed to accurately model an application using BMS. If the keys and tags depend on the values of items, we say that the application has *coupled control and data.*

When faced with an application with coupled control and data, one possible solution is to include more of the computation itself in the graph expansion functions. In the extreme case, by including all the computation in the expansion functions, we would be able to obtain an accurate dynamic task graph. Unfortunately, in the worst case, the computation would be performed twice, once for the expansion and once for the actual execution. However, our experience is that many application contain independent control and data, thereby supporting the BMS approach. For case studies and a discussion on the problems and benefits of independent control and data, see Sbîrlea et al. [**Sbirlea:13**].

4.3 Building the computation graph

The inspector builds a dynamic computation graph: items and tasks are nodes and the producer-consumer relationships are edges. Because of the dynamic single assignment nature of items, item nodes can only have a single producer, but may have multiple consumer tasks. Tasks can also spawn (prescribe) other tasks and each task has a unique parent.

The graph construction process starts from the node that models interactions with the parts of the program that are outside of CnC. The *environment-in* node produces initial items and spawns initial steps. After the computation completes, the *environment-out* node reads the outputs.

The tasks spawned by the *environment-in* node are added to a worklist of tasks that are expanded serially, by calling the graph expansion functions. For a single task, the process consists of the following steps:

- Call `get_inputs(t)` and add edges from the node of each consumed item to the task node.
- Call `get_outputs(t)` and add edges from the task node to each output item node.
- Call `get_spawns(t)` and add edges from the current task to the child tasks. Add children to the worklist.

The process finishes when all tasks have been expanded[*]. The *environment-out* node is added and connected to the output items of the computation (declared by using the function `item_collection.declare_get(k)`) As an example, the computation graph obtained for Cholesky factorization, is shown in Figure 4.1.

[*]During the expansion process, nodes are created when they are referenced for the first time.

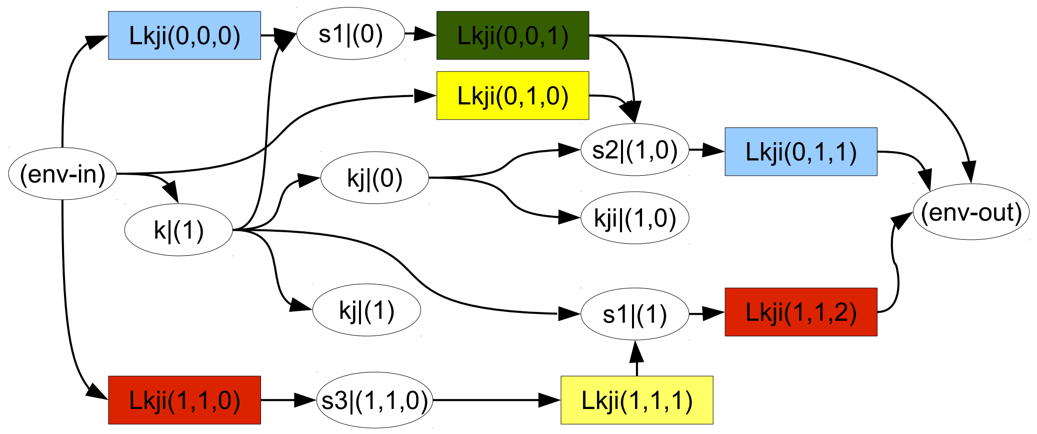


Figure 4.1 : The BMS-CnC computation graph for Cholesky factorization tiled 2×2. Data items are represented as rectangles and Circles represent steps. Nodes are labeled with the collection name followed by the key or tag. Item colors are assigned by the BMS algorithm (Section 4.4).

4.4 The heuristic BMS algorithm

After generating the computation graph, the inspector attempts to find bounded memory schedules using the heuristic BMS algorithm, which takes as input the computation graph and a memory bound M. BMS outputs an augmented partial order of tasks such that if a schedule respects the partial order, it will use at most M memory.

Even with substantial simplification, the BMS problem is NP-hard, since the register sufficiency problem [**Garey:1979**] which is well-known to be NP-Complete can be reduced to BMS*. Furthermore, the size of the computation graph is an order of magnitude larger than the basic block length (which determines the graph size in local register sufficiency). Thus, trying to find a heuristic solution before attempting

*The BMS problem has additional constraints not found in the register sufficiency problem that increase its complexity, such as items of different sizes, tasks that produce multiple items, the fact that inputs and outputs of a task (instruction in the register sufficiency case) are live at the same time.

a more expensive solution is essential. We propose a best effort approach in which, if a set of schedules that execute in less than M memory is found, the program is executed following a dynamically chosen schedule from the set. This leads to the following approximation of the BMS problem: *Given a parallel program P with input I and a computing system with memory size M, find an additional partial task ordering TO such that any schedule of P that also respects TO uses at most M peak memory.* If no schedule is found, BMS returns *false* (even though such a schedule may still exist).

In this initial description *items are assumed to be of a fixed size* and task memory is ignored. Section 4.8 extends the algorithm to address these simplifications.

Intuitively, given a serial schedule S (i.e., a total order) of the task graph, the BMS algorithm can test if it respects the memory bound by dividing the memory into item-sized slots (called *colors*) and checking that the number of available colors is larger than the maximum number of items live in the sequential schedule. The task graph can then be run in parallel if items assigned to the same color have non-overlapping lifetimes (to ensure that the memory bound is respected). This is enforced by adding ordering edges between the consumers of the item previously assigned to a color and the producer of the next item assigned to that color. To ensure adding ordering edges does not introduce deadlocks, we only add ordering edges that follow the same sequence of creation and collection as in the serial schedule S (since S is a valid topological sort of the graph, this cannot cause cycles).

The pseudocode, shown in Algorithm 1, follows the general list scheduling pattern. It picks a serial ordering of tasks in the main loop, lines 9 - 30 (by default we use a breadth-first schedule). In each iteration, it extracts one task from the priority queue of "ready to run" tasks (line 10), which is initialized with the only task ready to run

Algorithm 1 The BMS Algorithm.

```
function BMS(G, M, $\alpha$)
                                                      ▷ G is the computation graph
                                                      ▷ M is the desired memory bound
                         ▷ $\alpha$ affects the task priority queue (see Section 4.4.1)
  $noColors \leftarrow M/G.itemsize$
  $freeColors \leftarrow$ InitializeSet($noColors$)
  $freeTasks \leftarrow$ PriorityQueue($\alpha$)
  push($freeTasks, G.env$)
  while $freeTasks \neq \emptyset$ do
    $crtTask \leftarrow$ Pop($freeTasks$)
    for all $crtItem \in$ ProducedItems($crtTask$) do
      MarkAsProduced($crtItem$)
      $color \leftarrow$ Pop($freeColors, crtItem$)
      if $color = null$ then
        return false                                  ▷ Failed to find BMS schedule
      else
        $prevItem \leftarrow$ GetStoredItem($color$)
        for $prev \in$ ConsumersOf($prevItem$) do
          AddEdge($prev, crtTask$)
        for all $cTask \in$ ConsumersOf($prevItem$) do
          MarkInputAvailable($cTask, crtItem$)
          if ReadyToRun($cTask$) then
            Push($freeTasks, cTask$)
        SetStoredItem($color, crtItem$)
    for all $crtItem \in$ ConsumedItems($crtTask$) do
      if UnexecutedConsumers(crtItem) $==$ 0 then
        $availableColor \leftarrow$ ColorOf($crtItem$)
        $freeColors \leftarrow freeColors \cup availableColor$
    for all $spawn \in$ SpawnedTasks($crtTask$) do
      MarkPrescribed(spawn)
  return true                                         ▷ Found BMS schedule
end function
```

at the start: the input environment node (line 8).

We propose two techniques that help the algorithm cope with the different requirements of the bounded memory scheduling problem. These techniques are: successive relaxation of schedules and color assignment for minimum serialization. They are discussed in the next sections.

Tasks become ready to run when all their input items are available and the task has been prescribed. The output items of the current task are marked as produced (line 12) and assigned a color. Then, the consumer tasks of each output item are tested to see if they just became ready to run and any ready tasks are added to the priority queue (line 23). This process finishes when all tasks have been scheduled.

To maintain an upper bound on the memory consumption of the schedule, we use a list scheduling algorithm and apply a register allocation pass on the schedule as we are building it. We try to color items with C registers (colors), where $C \times ITEM\_SIZE = M$ (line 6). Instead of the widely used graph coloring allocator [**Chaitin:82**, **Briggs:94**] with a worst-case space complexity of $\mathcal{O}(n^2)$, we opted for the more memory-efficient linear scan register allocator [**Poletto:99**].

When the algorithm visits a task, it assigns each of its output items a color from a free color pool (line 13), which is only returned to the pool when all consumer tasks for that item have been scheduled (line 28). Since input and output items are simultaneously live during task execution, it is important to assign colors to output items before collecting input items. If an item is produced and the color pool is empty, then we consider that the schedule cannot be executed in the memory bound available (line 15).

After finding a serial task order that fits the memory bound, we add *ordering edges* between tasks, such that the lifetime of items with the same color cannot overlap. To do this, we need to record, for each color, the item currently stored in it, using the `SetStoredItem` and `GetStoredItem` functions. For each color, these edges restrict parallel schedules to follow the sequence of item allocations and de-allocations as in the serial order chosen above; to do this ordering edges are added (line 19) starting from each of the consumers of the item previously allocated to color C to the producer of the current item assigned to the same color C.

One challenging problem when restricting schedules with serialization edges is ensuring the absence of cycles in the resulting task graph, because such cycles would mean deadlock. The edges we insert are directed only towards tasks scheduled later in the serial schedule, so even with these additional edges, the serial schedule we build remains a topological sort of the application graph. The existence of this topological sort mean there are no cycles.

4.4.1 Successive relaxation of schedules

If the desired memory bound is small, it is possible that the serial schedule chosen by BMS will not fit the memory bound. It is essential to find a heuristic that enables us to identify a schedule which fits the memory bound and the approach must also be fast, since the executor cannot start before the inspector finishes. Our approach, called *successive relaxation of schedules*, is to sample schedules in a way that trades parallelism for lower memory requirements. We do this by varying the ranking function used to prioritize ready to run tasks in the BMS algorithm. The ranking function varies from the breadth-first (default) to depth first, since we found that breadth-first schedules usually lead to more parallelism/more memory, while depth-first leads to less parallelism/less memory. [*]

Because the default parallel schedule is based on a breadth-first task ordering, one possible concern is the loss of cache locality this implies. To address this concern, recall that this ordering only affects the partial order output by the BMS algorithm and and does not enforce a total order in which the dynamic scheduler of the executor handles tasks.

This choice of different schedules is done by varying α (line 7) from 1 (breath-first) to 0 (depth-first) which is then used by the priority queue comparison function (lines 12-16) in which the available tasks are stored. If the depth first schedule ($\alpha = 0$) does not fit the bound, we abort the scheduling operation (line 8).

Algorithm 2 Successive relaxation of schedules.

```
1: function SCHEDULE(G, M)
2:   $\alpha \leftarrow 1$
3:   while $\alpha \neq 0$ do
4:     $success \leftarrow \text{BMS}(G, M, \alpha)$
5:     if success then
6:       return true
7:     $\alpha \leftarrow \alpha - \Delta\alpha$
8:   return false
9: end function
10:
11: // Used for the task priority queue:
12: function PRIORITYQUEUE.COMPARE( $task_1$, $task_2$)
13:   $rank_1 \leftarrow \alpha \times \text{RANKBF}(task_1) + (1 - \alpha) \times \text{RANKDF}(task_1)$
14:   $rank_2 \leftarrow \alpha \times \text{RANKBF}(task_2) + (1 - \alpha) \times \text{RANKDF}(task_2)$
15:   return $rank_1 - rank_2$
16: end function
```

[*]Depth-first and breath-first ordering of tasks are done on a graph where items are treated as direct edges from producer to consumer. The breadth-first ranking of a node is one larger than its lowest ranking predecessor node. For depth-first, a queue of ready tasks is maintained and nodes are numbered in the order in which they are removed from this queue. Nodes are added to the queue when their predecessors have been numbered.

4.4.2 Color assignment

The color assignment is important because it drives the insertion of serialization edges, which in turn can affect performance: inserting too many edges increases overhead and bad placement can decrease parallelism. Moreover, a slow coloring heuristic delays the start of the executor stage slowing down the completion of the execution.

Since many steps are already ordered by producer-consumer and step spawning relationships, not all edges inserted by BMS in line 19 of Algorithm 1 actually restrict parallelism. We call these edges *transitive edges*, whereas those that restrict the parallelism are *serialization edges* and need to be enforced during execution. As described below, this distinction is also important for color assignment.

How can one quantify the parallelism decrease caused by coloring? Remember that the resulting schedule runs on a dynamic work stealing scheduler with provable performance guarantees [**Spoonhower:2009**] as long as the parallel slack assumption holds. This assumption holds as long as the critical path is not increased too much, so we attempt to insert serialization edges in such a way as to not increase the critical path.

Theorem 2. *Assuming unit-length tasks and a breadth-first schedule, BMS will increase the critical path length with at most the number of serialization edges it inserts.*

Proof. Since tasks are processed in breadth-first order and the tail of serialization edges is a task that has already been allocated, a serialization edge whose head task is at level k, must start at level $k - i$, with $i \geq 0$ thereby the edge can increase the critical path with at most one. □

Algorithm 3 shows our greedy minimum-serialization heuristic: for each item, we pick the color that leads to the insertion of the fewest serialization edges from steps with breadth first level smaller than the current task. Only if no such edges exist we consider serialization edges that start from the current breadth-first level, since that increases the critical path. If the source of an edge that would be added by BMS is already a predecessor of the destination, then the edge is transitive and is not counted as a serialization.

Storing the predecessors set of a task can take up to $\mathcal{O}(n)$ memory; we need to record this information for all consumers of items whose color has not been reassigned to another item ($\mathcal{O}(M)$), leading to a total of $\mathcal{O}(n \times M \times consumers\_per\_task)$.

Our experiments show that his approach greatly reduces the number of serialization edges to insert compared to a round-robin approach, but has an associated memory cost. In our experiments, the inspector overhead is not large enough to justify replacing the coloring heuristic, but, if needed, it can be replaced by a simple round robin approach of allocating colors.‘

Algorithm 3 Assigns item colors.

```
function POP(freeColors, crtItem)
  $producer \leftarrow \textsc{GetProducer}(crtItem)$
  $minColor \leftarrow null$
  $minEdges \leftarrow MAX\_INT$
  for $color \in freeColors$ do
    $prevItem \leftarrow \textsc{GetStoredItem}(color)$
    $edges \leftarrow 0$
    for $consum \in ConsumersOf(prevItem)$ do
      if $!\textsc{IsPredecessor}(consum, producer)$ then
        if $\textsc{BFRank}(prod) \geq \textsc{BFRank}(consum)$ then
          $edges \leftarrow edges + \textsc{ConsumersCount}(prevItem)$
        $edges \leftarrow edges + 1$
    if $edges < minEdges$ then
      $minEdges \leftarrow edges$
      $minColor \leftarrow color$
  return $minColor$
end function
```

Variable Name	**Meaning**
$issue[task\_id]$	In which cycle is task *task* issued?
$death[item]$	In which cycle can item *item* be collected?
$color[item]$	To which color is item *item* assigned?
indicators	Auxiliary binary variables for disjunction support. At most $5 \times NO\_ITEMS^2$ variables.

Table 4.1 : Variables used in the ILP formulation.

Constraint name	**Maximum number of constraints**
1. Define time of item death	NO_GETS
2. Data dependence	NO_GETS
3. Color assignment	$5 \times NO\_ITEMS^2$
4. Max_bandwidth	NO_ITEMS
5. Earliest start time	NO_TASKS
6. Latest start time	NO_TASKS

Table 4.2 : Constraints used in the ILP formulation.

4.5 Optimal BMS through integer linear programming

Heuristic BMS is fast, but offers no guarantees regarding how much memory reduction it can achieve. If it fails to find a schedule that fits the desired memory bound, we apply an integer linear programming formulation that guarantees finding a schedule for any input memory bound if such a schedule exists. The challenge in using integer linear programming is to formulate the problem in a time and memory-efficient way, so that it can be used for large computation graphs. The formulation and optimizations are described in Section 4.5.1. Section 4.5.2 proposes specific lower bounds used to speed up optimal BMS. An additional performance benefit is obtained by using the results of heuristic BMS to speed-up the optimal BMS, as shown in Section 4.5.3.

We propose a disjunctive formulation with variables and constraints shown in Tables 4.1 and 4.2.

Constraints 1 to 4 are necessary for correctness, and constraints 5 and 6 are memory/performance optimizations and may be omitted depending on the size of the linear system. For example, constraint 5 ensures that only serial schedules are considered, but this restriction is not needed. Adding these constraint will only consider serial schedules, decreasing the search space, but at the cost of an increased footprint of the linear system, so they are disabled for large input graphs.

ILP formulation attempts to find a schedule with the minimum memory bound by minimizing the number of colors used, but interrupts the search as soon as it finds a solution that fits the user-specified memory bound. As shown in Section 4.9, we are able to solve graphs that are an order of magnitude larger than those in previously published results for the problem of minimum register scheduling.

4.5.1 Optimization of color assignment constraints

We focused our optimization effort on color assignment constraints because they represent a large majority of the total number of constraints. Color assignment constraints enforce that two items assigned to the same color cannot be live at the same time and could be expressed naively, as the following if-statement:

```
if color[item1] == color[item2] then
   issue[producer[item1]]>death[item2] or
   issue[producer[item2]]>death[item1]
```

The integer linear programming encoding of this if-statement is done by replacing the if-condition with two disjuncts:

```
color[item1] < color[item2] or
    color[item1] > color[item2]
```

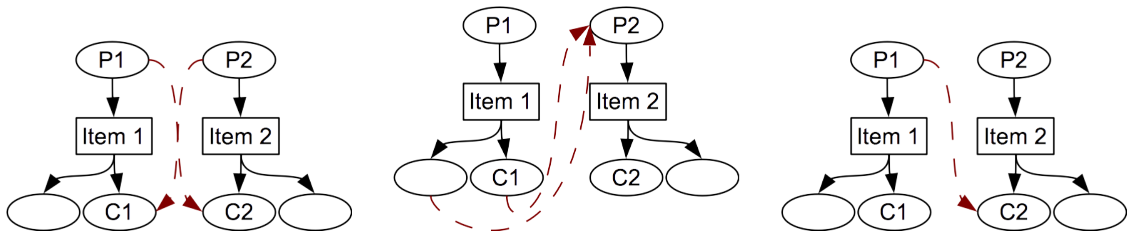
We then transform the if statement `if A then B` into a disjunction $\bar{A}$ *or* B.

Then, we apply the technique of using boolean indicator variables (named a,$b.c$,d) and an additional constraint to represent disjunctions [**Williams:2013**], obtaining the following equations, in which M and N are big constants:

$$\begin{cases} color[item1] - color[item2] + M \times a \leq M - 1 \\ color[item2] - color[item1] + M \times b \leq M - 1 \\ death[item1] - issue[producer[item2]] + N \times c \leq N - 1 \\ death[item2] - issue[producer[item1]] + N \times d \leq N - 1 \\ a + b + c + d \geq 1 \end{cases}$$

This set of constraints is correct, but inefficient, adding 4 variables and 5 constraints for each pair of items. Decreasing the number of constraints and variables added is essential for efficient execution. To do this, we analyze the possible relations of the lifetime of items as shown in Figure 4.2. For items that must-overlap, we can elide the third and fourth constraints and corresponding variables. For items that may-overlap we elide either the third or fourth constraints if there is a path as in Figure 4.2c. For items that cannot overlap, we elide all constraints and associated indicator variables.

Another constraint that can be optimized is the one that defines the time of death (constraint number 2). These constraints restrict the time of death of each item to happen after all the consumers of that item have been issued. In some cases, consumers of the item are ordered by other data dependence edges, so we can omit the time of death constraints corresponding to all but the last consumer.



(a) Item lifetimes must overlap. (b) Item lifetimes cannot overlap. (c) Item lifetimes may overlap, but one item must be created first.

Figure 4.2 : Several item patterns enable more efficient encoding of the color assignment constraint. The dotted edges are paths in the graph that enforce the must-overlap, cannot-overlap and may-overlap relations.

4.5.2 Tight lower bounds to speed up ILP BMS

Often, the tightest possible schedule is found by heuristic BMS, but the ILP solver takes a long time to prove its optimality since it needs to search through many schedules for a possibly better solution. Adding tight lower bounds on the minimum memory possible is important, since the search stops if the heuristic BMS solution equals the lower bound. We propose using two lower bounds, each of which works best for a different type of graphs.

The first lower bound is the memory requirement of the step with the largest number of inputs and outputs. In some cases, this step is the environment-out node and we can improve the bound further by using the following observation: after all but one of the output items are produced, in order to produce the last item, the inputs from which this last item is computed must also be live, and included in the lower bound. For Cholesky factorization, for example, this bound is equal to the minimum memory footprint of the application.

The second bound we propose is useful for applications where, even though each step requires a modest amount of memory, the total footprint is large. This pattern occurs, for example, in divide-and-conquer applications where the memory pressure is proportional to the height of the graph. To handle these cases, we build a tree that is subsumed by the computation graph (the tree identification is done by ignoring all but one of the edges that connect an item to its consumers), and use the Strahler number* of the tree as a lower bound. For applications such as merge sort, this happens to be the minimum memory footprint of the application.

4.5.3 Hot start ILP: Using heuristic BMS to speed up ILP BMS

To decrease the time and memory costs associated with ILP, we combine linear programming with the heuristic approach presented in Section 4.4. Since the optimal approach is only used when the heuristic algorithm does not find a schedule that fits the desired memory bound, this means that the minimum footprint schedule found by the heuristic can be used as initial solution for the the ILP solver. If the heuristic already found the minimum footprint possible, but the desired footprint is smaller, the ILP will need to confirm the lack of better solutions by solving the linear programming relaxation and checking that the objective value matches the one provided by the heuristic. In this case, using the initial solution, the solver will finish early with the optimal solution being the heuristic one.

If the heuristic does not find the minimum footprint possible, its resulting schedule is still used by the ILP solver in the branch and cut stage, since the existence of a close-to-optimal solution helps to avoid the exploration of areas of the search space that can only offer solutions with worse memory bounds.

*The Strahler number [**Flajolet:1979**] is the minimum number of registers required to evaluate an expression tree and can be computed in linear time.

4.6 Schedule reuse

Traditional inspector/executor systems amortize the inspection cost by reusing the inspector results, for example by executing multiple times a loop that has been inspected once. Since the BMS executor runs only once, we amortize the inspector cost across multiple executions of the application by caching the inspector results. To the best of our knowledge, our approach is the first to reuse inspector-executor results across different runs of an application. The proposed approach can be applied for even if the input parameters differ between runs with the same desired memory bound, as long as the computation graph structure remains unchanged. This requirement is mitigated our model's ability to express applications as serial sequences of parallel kernels that are modeled independently as separate BMS problems. Because schedule reuse is performed at the kernel granularity instead of the application granularity, as long as any kernel has the same computation graph, then that kernel's schedule can be reused.

Note that having a compact representation of the computation graph and of the inspector output is critical for efficiency of schedule reuse. A fast matching operation of the current computation graph to the graph of past executions is also key to making the schedule reuse efficient, so we focused our efforts on improving these three aspects.

To determine if the BMS schedule of a previous run fits the current one, one option is to generate the computation graph and compare it with the graph of the previous executions; this can be costly in both time and memory. Instead, we use only a small root set of graph edges and vertices that uniquely identifies the graph[*]. This root set contains two types of edges. First, it contains the edges whose tail

[*]This assumes the determinacy of the control flow (step tags) in the program, since BMS-CnC can only express this kind of programs.

vertex is the *environment-in* node. These edges lead to item keys produced by the environment and the tags of the tasks spawned by the environment which uniquely characterize all the items and tasks that will be spawned during the computation. Second, the root set also includes edges whose tail is the *environment-out* node. The tail of these edges are the keys of items read after the computation completes (i.e. the application result); they affect the minimum footprint of the execution because, for the same computation graph, more output items lead to larger minimum memory requirements.

The schedule reuse works as follows. First, we identify the root set. If it does not match with the root set of a previous execution, we expand the whole computation graph, run the BMS algorithm and save the resulting (*serialization edges*, *root set*) pair on disk, in a schedule library, along with a MD5 hash of the root set. For subsequent runs of the application, the inspector will compare the MD5 hashes of the current root set with the root sets from the schedule library. If it finds a matching root set, the inspector loads the serialization edges, avoiding the graph expansion and BMS computation. If there is no match, the only additional work performed is the hashing, since the root set expansion is done anyway during the graph expansion.

Because the root set consists of keys and tags only (no data), matching the root set to the root set of a previous program is fast. The schedule loading consists of reading the set of serialization edges from the schedule library.

As an example, take Cholesky factorization, whose 2×2 tiling is shown in Figure 4.1. The root set consists of seven edges (four starting from the *env-in* node and three ending in the *env-out* node). In general, for Cholesky factorization tiled $k \times k$, the root set will have an order of $k^2/2$ edges. Since each vertex is identified by 3 integers, the whole root set will have $3 \times k^2$ integers. This is much smaller than the input matrix which is also read from disk. Since the computation graph depends on the matrix size and tile size only and the tile size is usually tuned for the machine the serialization edges can be reused for any input matrix of the same size. Similarly, for image processing applications, the input is usually the same size and the schedules should be reusable all the time.

In applications with irregular graphs, such as the sparse Cholesky factorization as implemented in HSL [**HSL:2013**], the root set consists of the keys of non-zero tiles, which is still smaller than the sparse input matrix. The schedule cache consists of the corresponding serialization edges, whose number is inversely proportional to the memory bound.

To conclude, the schedule reuse approach relies on the combination of root sets, hashing and the intrinsic compactness of serialization edges to amortize the inspector overhead across multiple runs of an application.

4.7 Other features

4.7.1 Automatic garbage collection

Our graph exploration enables automatic memory collection for items that would otherwise need manual collection techniques. Such items are also challenging to collect in traditional programming models because they are pointed to by other objects[*], so a classic garbage collector would not be able to collect them. The mechanism regularly used to collect items is *get-counts* [**Sbirlea:12**], a parameter provided by the programmer that specifies the number times the item will be read. Identifying the

[*]We want to collect when objects will not be used in the future, as opposed to when objects are not referenced anymore.

get-count requires global knowledge about the application, which inhibits modularity, is error-prone and difficult.

The computation graph contains all information required to automatically compute get-counts for items, making item collection a completely automatic task. The same technique can be applied to programs written in traditional programming languages (and follow the restrictions described in Section 4.2) to collect objects which are still referenced, but will never be used (cleaning up the memory leaks).

4.7.2 Fast debugging of concurrency bugs

Our inspector/executor system accurately identifies all cases of the following problems that arise because of programmer error. We include in parenthesis the name used for these problems in traditional programming models:

- dynamic single assignment violations (data-race)
- cyclic dependence between steps (deadlock)
- waiting for an item that is never produced (blocked thread)
- producing an item that is never read (unused variable)
- tasks that do not produce items (dead tasks) [*]

Finding concurrency bugs traditionally involves being able to reproduce the parallel control flow that lead to them happening, which in itself is a time consuming step. Traditional tools that identify these problems commonly serialize the application **[Feng:97]** and add space overhead linear in the size of the application footprint

[*]Out tool helped discover an instance of this bug that had existed for two years in the Intel CnC implementation of Cholesky.

and in the number of parallel threads [**Flanagan:09**]. Our approach separates the testing phase from the execution phase and outputs the results before the application reaches the executor stage. Each of the bugs, with the exception of deadlock freedom, are identified during a linear pass through the application graph. For deadlocks, we simply test for any tasks not scheduled after the BMS algorithm that have not been reported as other types of bugs. A cycle detection algorithm for directed graphs can identify the complete deadlock cycle.

4.8 BMS Extensions

In this section we describe two extensions to the BMS algorithm: the first one adds support for different item sizes and the second one accounts for memory used by waiting and executing tasks.

4.8.1 Supporting multiple item sizes

To support items of different sizes, one can use the approach of allowing items to be allocated at any memory location. This results in memory fragmentation that requires a global compaction phase to reclaim the free space. The compaction can introduce a barrier during execution of the parallel program, thereby increasing the computation's critical path.

Instead, we observe that applications often have only a few *classes of items*, where all items in a class have the same size (for example the size of the input matrix, the size of a tile)[*]. Memory is initially divided into slots the size of the largest items, each of which can be split into multiple sub-slots suitable for smaller items. When all

[*]A similar approach is used by memory allocators of operating systems which have pools of memory objects of different sizes.

sub-slots become unused, they are merged into a larger slot. We refer to the colors used for items of the largest size as *root colors*.

Algorithm 4 shows how colors are assigned to items of different sizes. The `Pop (freeColors)` function from the BMS algorithm is replaced by a `PopFromClass( freeColors, crtItem)` function which takes the item that needs space as an additional parameter. The `freeColors` parameter now contains only free root colors. The `PopFromClass` first identifies the class (size) of the item (line 3) and looks for an available color in the list of free colors that is specific to that item class (line 5). If a color is available, we return it (line 21); otherwise, we have to split a root color `freeColors` into sub-colors of size that matches the current item. The number of new colors is determined in line 12 and they are added to the list of free colors for that class (line 12). Note that, for each new color, we need to find (line 11) and propagate (line 16) the correct consumers for the item which was last stored in it — this information is needed when inserting ordering edges.

To prevent fragmentation, BMS reassembles sub-colors into root colors. This happens when all sub-colors that are splinters of a root color become available again; we use the function AddFreeColorToClass (line 24), which replaces the union operation on line 28 in the BMS algorithm. When allocating a new item to a reclaimed root color, we need to ensure that the lifetimes of the items previously stored in the sub-colors do not overlap with the item later assigned to the root color. This is done by adding ordering edges, but the previously stored items must be recorded by the SetStoredItem and GetStoredItem functions which work with sets of items instead of single items.

Algorithm 4 BMS Extension for items of different sizes.

```
// This function assigns a (sub)color for item crtItem
function PopFromClass(freeColors, crtItem)
  $crtClass \leftarrow$ GetClass($crtItem$)
  $freeSubcolors \leftarrow$ GetFreeColors($crtClass$)
  $color \leftarrow$ Pop($freeSubcolors, crtItem$)
  if $color = null$ then
    // Call Pop function from the BMS algorithm
    $pageColor \leftarrow$ Pop($freeColors, crtItem$)
    if $pageColor \neq null$ then
      $prevIt \leftarrow$ GetStoredItem($pageColor$)
      $prevConsumers \leftarrow$ ConsumersOf($prevItem$)
      $noSubcolors \leftarrow rootSize/class.itemSize$
      for $i = 1 \rightarrow noSubcolors$ do
        $newColor \leftarrow new$color()
        PushToClass($freeSubcolors, newColor$)
        SetStoredItem($newColor, crtItem$)
      $color \leftarrow$ Pop($G, freeColorsInClass, crtItem$)
  if $color \neq null$ then
    $rootColor \leftarrow$ GetRootColor($color$)
    $rootColor.uses \leftarrow rootColor.uses + 1$
  return $color$
end function
// This function reclaims a (sub)color
function AddFreeColorToClass(
      freeColors, itemColor)
  $crtClass \leftarrow$ GetClass($itemColor$)
  $rootColor \leftarrow$ GetRootColor($color$)
  $rootColor.uses \leftarrow rootColor.uses - 1$
  if $rootColor.uses = 0$ then
    AddFreeColor($freeColors, rootColor$)
    $itemsSet \leftarrow$ SubColorsOf($rootColor$)
    SetStoredItem($rootColor, itemsSet$)
end function
```

4.8.2 Bounding task memory

In BMS-CnC, tasks have two states: prescribed or executing. This section looks at the memory taken up by tasks. *Executing tasks* use the stacks of the worker threads executing them, so they do not consume additional memory beyond the worker stack space allocated at the start of program execution. BMS can consider worker thread memory starting with a memory bound parameter M_1 that is lower than the total memory M in the system: $M_1 = M - no\_workers * worker\_stack\_size$. Note that since BMS-CnC tasks have fixed-size stacks, large levels of recursion can only be performed by spawning new tasks.

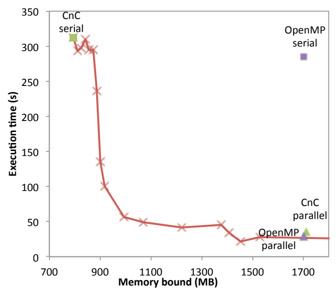
Prescribed tasks are the tasks that have been `spawned` but have not yet started running. In our implementation, these waiting tasks consist only of the task tag and the task function pointer, so their size can be computed by the BMS scheduler. This memory is needed from the moment tasks are created by a `spawn` operation to the moment the task finishes so they are similar to items whose lifetime extends between the moment they are `put` up to the moment their last consumer task finishes execution. The same mechanism used to handle items of different sizes (described in Section 4.8.1) also handles prescribed tasks.

4.9 Evaluation

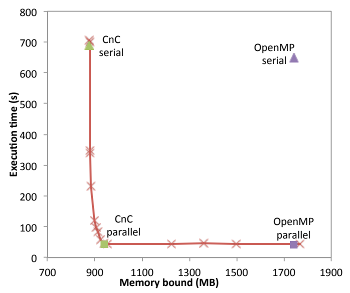
4.9.1 Implementation and experimental setup

The BMS-CnC system was implemented on top of Qt-CnC [**Sbirlea:13**], an open-source CnC implementation[*] based on the Qthreads task library [**Wheeler:08**]. The evaluation was performed on an Intel Xeon E7330 system with 32GB RAM and 16

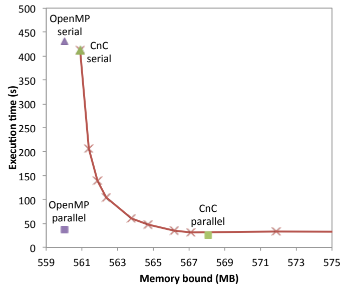
[*]https://code.google.com/p/qthreads/wiki/qtCnC



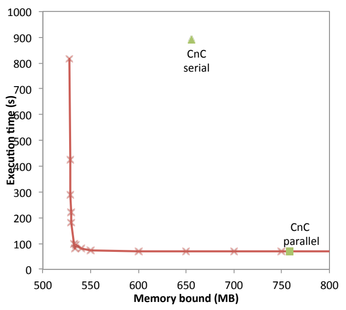
(a) Smith Waterman



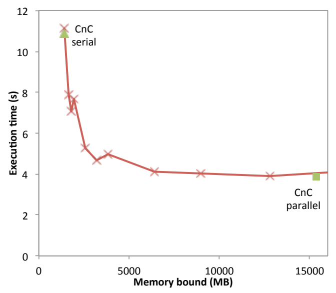
(b) Blackscholes



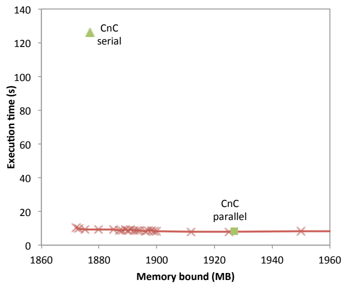
(c) Cholesky



(d) Gauss-Jordan



(e) Merge Sort



(f) Standard Task Graph (STG) 59

Figure 4.3 : BMS-CnC executor run-time (the red line) as a function of memory bound for each of the benchmarks. BMS-CnC is able to enforce memory bounds down to the serial execution and even lower for Gauss-Jordan and STG 59. OpenMP results included where available.

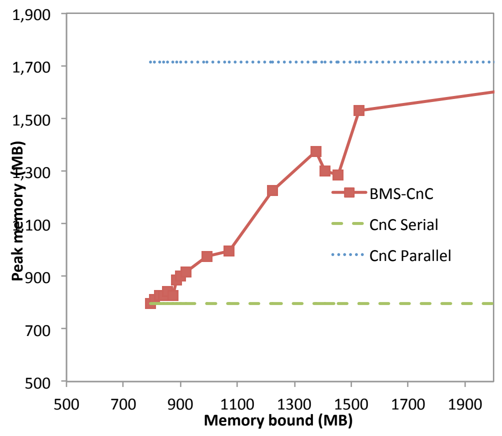
cores. We instrumented the runtime to keep track of the item memory allocations and item deallocations performed. Because CnC is implicitly parallel and there is no separate CnC serial implementation, we obtain serial execution times by using Qt-CnC (not BMS-CnC) with a single worker thread.

For each application, we present the BMS executor time as a function of the memory bound (see Figure 4.3). To evaluate the performance of BMS, we note that the minimum memory bound for which BMS finds a schedule should at least match the serial execution memory. When the bound is large enough to fit a normal (CnC) parallel execution, BMS should not lead to performance degradation.

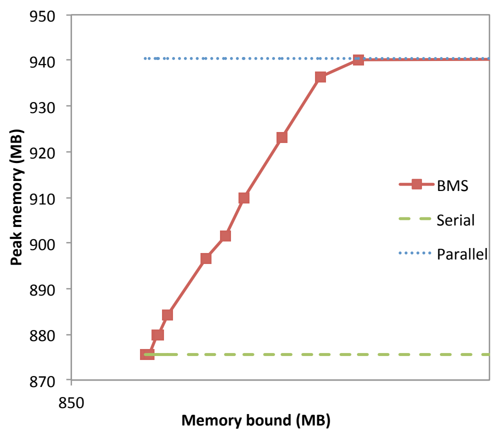
One possible concern related to the bounded memory scheduling algorithm is if accurately enforced a desired memory bound, unnecessarily decreasing the memory footprint may lead to a corresponding decrease in parallelism. To address this concern, we present Figure 4.4 which shows the actual peak memory encountered as a function of the memory bound. On this graph, the peak memory for single-threaded CnC and parallel CnC are horizontal lines, since they are are constant. Because both axes have the same scale and origin points, the performance of the BMS algorithm can be assessed visually by checking that the peak BMS memory varies between the peak memory corresponding to serial and parallel executions - this ensures the range of bounds imposed by BMS is good. To estimate how accurate is the BMS bound, one can check that the peak memory series follows the graph diagonal (x=y) between the serial and parallel execution series. Having a peak memory smaller than the memory bound is not necessarily a weakness of the algorithm; as long as the execution time for that memory bound does not suffer. A good BMS algorithm should enable us to get a reduction in memory footprint and only showing a slowdown if the memory bound is tight; this criteria can be analyzed by looking at the execution time graph and at the memory footprint graph for the same value of the memory bound.

4.9.2 Benchmarks

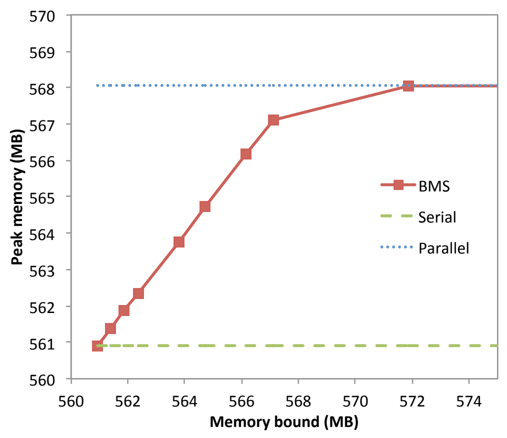
Applications are usually implemented as sequences of parallel computation kernels invoked with different parameters. To maximize the benefits of schedule reuse for such applications, it makes sense to model each parallel kernel independently as a BMS problem, since this enables schedule reuse at the kernel granularity instead of the application granularity. For this reason the evaluation includes several computational kernels instead of fewer large applications. By themselves, the kernels reach footprints which can be satisfied without BMS on today's machines; they will require BMS when



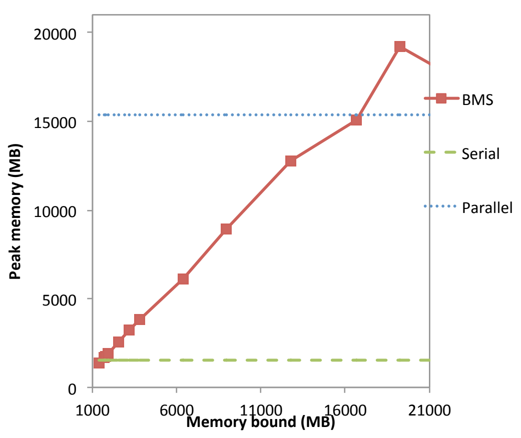
(a) Smith Waterman



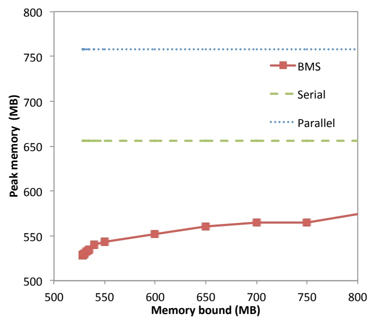
(b) Blackscholes



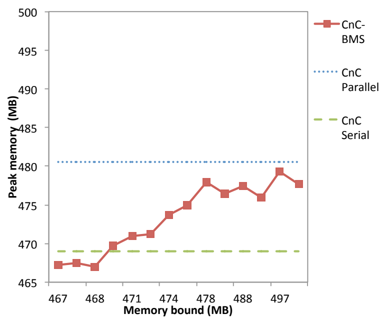
(c) Cholesky



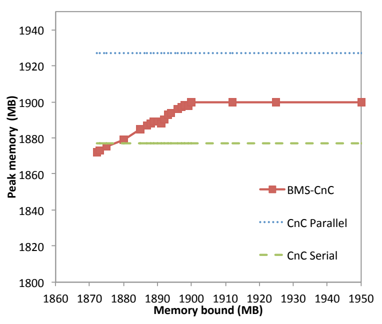
(d) Merge Sort



(e) Gauss-Jordan Elimination



(f) Standard Task Graph (STG) 58



(g) Standard Task Graph (STG) 59

Figure 4.4 : BMS-CnC actual peak memory as a function of memory bound for each of the benchmarks.

Benchmark		**Type**	**Graph vertices**	**Input parameters**	**Schedule reuse conditions**
Smith Water-man		biomedical	5002	2 sequences of 70000 length and tile size (2000×2000)	same tile sizes & same sequence sizes
Blackscholes		financial	6730	number of options (25.6M) and option data	same number of options
Cholesky		dense algebra	41558	input matrix (12000×12000) and tile size (125×125)	same tile sizes & same matrix sizes
Gauss-Jordan		dense algebra	8450	input matrix (4096×4096) and tile size (256×256)	same tile sizes & same matrix sizes
Merge Sort		recursive	3582	(2^{25} integers)	same input array sizes
STG	sparse	task graph	198	graph shape	same graph shape
	fpppp	task graph	647	graph shape	same graph shape
	58, 59	task graph	5402	graph shape	same graph shape

Table 4.3 : Benchmarks, their inputs, computation graph sizes and the schedule reuse conditions. The corresponding results are shown in Figure 4.3.

used in the context of larger applications containing multiple kernels as well as on future many-core systems with smaller amounts of available memory per core. Table 4.3 contains a short summary of the benchmarks, their input parameters, computation graph size and conditions for schedule reuse.

Smith-Waterman is a dense linear algebra kernel from the Intel CnC distribution. The results in Figure 4.3a show that BMS gracefully spans the range between large memory-high performance to low memory with lower performance.

The results for **Blackscholes** (in Figure 4.3b) show that BMS-CnC is able to control the peak memory from the largest values obtained with CnC parallel execution, to the smallest (serial execution).

The **Cholesky factorization** (Figure 4.3c) shows BMS enables a trade-off similar to the one in Smith-Waterman, between large memory consumption and high performance.

For **Gauss-Jordan elimination** (see Figure 4.3d), BMS-CnC is able to enforce a footprint 18% lower than the serial footprint of CnC, with minimal loss of parallelism. This is the result of the abundant parallelism, as well as good coloring heuristics.

For **MergeSort** (Figure 4.3e) we notice an unusual trend when the desired memory bound is larger than 15MB - the execution time of BMS-CnC in these cases becomes smaller than the CnC parallel execution, even though the actual program footprint is the same. We believe that the performance benefit comes from improved cache locality in the BMS schedules.

The **Standard Task Graph (STG) Set** [**Takao:02**] provides a set of random task graphs from which we picked the largest (STG 59), shown in Figure 4.3f. Since STG graphs do not contain any work, we used a fixed amount of computation for each task and a fixed size for each item. In both cases, there is sufficient parallelism

Benchmark	**BMS-CnC memory (%) when Speedup is**		
	90%	50%	10%
Smith-Waterman	48.8	10.6	0.0
Blackscholes	93.2	10.8	1.4
Cholesky	84.6	46.2	0.0
Gauss-Jordan	0.0	0.0	0.0
Merge Sort	12.0	0.9	0.0
STG 58	12.0	0.0	0.0
STG 59	22.2	0.0	0.0

Table 4.4 : Memory consumption for BMS-CnC when it has 90%, 50% and 10% of the parallel CnC speedup. Values are percentages of the additional memory required by parallel execution - 0% means no increase in footprint, 100 % means maximum increase (same footprint as parallel execution).

to hide the BMS constraints up to the boundary condition where BMS cannot find a valid schedule. There is no loss of performance from using BMS with the tightest memory bound, which is lower than serial execution memory. For these graphs, BMS is able to offer the best of both worlds - the footprint of serial execution with the performance of parallel execution.

In summary, BMS shows the ability to control the trade-off between parallelism and memory usage. Furthermore, this trade-off is not linear — there is a ”sweet spot” in the memory bound space where BMS enables most of the performance of the unbounded memory parallel execution with only a small increase in memory relative to the serial execution. To further illustrate this, Table 4.4, shows the memory requirements of BMS-CnC when its speedup is 90%, 50% and 10% of the parallel CnC speedup. The values are percentages of the memory difference between parallel and serial executions. For example, 0% means the BMS-CnC program does not require more memory than serial execution, while 100% would mean that the memory use matches the memory utilization of parallel execution (maximum memory increase).

4.9.3 OpenMP comparison

OpenMP results have been included in Figure 4.3 where external implementations of the same benchmarks were available. One interesting pattern is that the OpenMP memory footprint does not vary between the serial and parallel executions because OpenMP encourages programmers to parallelize computation loops while the memory allocation and de-allocation are usually performed outside parallel regions. In BMS-CnC, item lifetime is minimized by allocating items only when needed and by automatically collecting them after their last use.For Smith-Waterman and Blackscholes, BMS-CnC offers similar performance with OpenMP while enabling considerable memory savings. For Blackscholes, for example, OpenMP has a performance advantage of under 10%, but requires twice the memory of CnC, since it pre-allocates all the memory to reduce overhead.

Because the OpenMP implementation of Cholesky exploits less parallelism (barrier style versus dataflow) so so it has a lower memory footprint and lower performance than CnC.

4.9.4 Minimum memory evaluation

To identify how close the BMS heuristic approach can be to the absolute minimum memory footprint possible, we fed the ILP formulation of the problem to the commercial Gurobi solver which finds find the minimum possible footprint. The results are shown in Table 4.5. For small and medium problem sizes, both the ILP and BMS approaches can enforce the minimum memory footprint possible, but there are some examples, such as Gauss Jordan, where ILP can obtain a better bound that heuristic BMS.

On larger graphs, the ILP solver may run out of memory or not finish before the

Benchmark	**Input**	**Graph Nodes**	**Min. mem. (MB)**		**Bounds (MB)**	
			BMS	**ILP**	**Strahler**	**Local**
Smith Waterman	small	52	26.6	26.6	15.2	15.2
	med	100	34.2	*34.2	19.0	15.2
	large	2452	141.0	*141.0	22.8	15.2
Cholesky	small	315	0.6	0.6	0.1	0.6
	med	1907	8.2	8.2	0.2	8.2
	large	4555	403.8	NA	1.4	403.8
Blackscholes	small	402	63.2	63.2	1.1	63.2
	med	802	125.6	125.6	1.2	125.6
	large	1602	250.4	250.4	1.2	250.4
Gauss Jordan	small	22	62.5	62.5	25.0	62.5
	med	65	150.0	125.0	37.5	125.0
	large	146	250.0	*225.0	50.0	212.5
Merge Sort	small	222	0.9	0.9	0.9	0.4
	med	7166	1.5	1.5	1.5	0.4
	large	14334	1.6	1.6	1.6	0.4
STG	sparse	198	17.6	14.9	1.2	14.9
	fpppp	647	57.4	*57.4	1.2	24.9
	59	5406	468.0	NA	1.8	83.8

Table 4.5 : The minimum memory with heuristic BMS and with ILP and the lower bounds fed to ILP. Cells are marked with * when ILP timeouts.

5 hour cutoff. This happens in cases where the two lower bounds are much smaller than the actual feasible minimum memory. We analytically discovered that in 3 out of 4 cases when this happened, the ILP had already found the minimum memory schedule, but had not proved its optimality before running out of time. BMS is capable of finding a schedule with minimum bound in all but 4 out of the 18 cases.

Benchmark	**Input**	**Graph nodes**	**Time (s)**		
			BMS	**ILP**	**hot ILP**
Smith Waterman	small	52	0.2	1.0	3.6
	med	100	0.7	148	189.51
	large	2452	3.59	NA	NA
Cholesky	small	315	0.0	0.7	0.4
	med	1907	0.1	7920	281
	large	4555	3.6	NA	NA
Blackscholes	small	402	0.1	5403	5
	med	802	0.1	NA	16
	large	1602	250.4	NA	1189
Gauss Jordan	small	22	0.0	0.0	0.0
	med	65	0.1	155	44
	large	146	0.1	NA	NA
Merge Sort	small	222	0.1	40	10
	med	7166	4.1	336	18.61
	large	14334	8.7	NA	28.22
STG	sparse	198	1.1	200	37
	fppp	647	2.5	NA	NA
	59	5406	69.4	NA	NA

Table 4.6 : Performance evaluation of heuristic BMS and ILP with and without hot start. Even with hot start, the ILP approach cannot handle large graphs.

4.9.5 Runtime comparison of ILP and heuristic BMS

Table 4.6 shows the run time of the BMS inspector. The ILP approach can handle graphs of up to tens of thousands of vertices, but there are some examples where it either runs out of memory or reaches the 5 hour timeout. However, the hot start optimization in which we provide the heuristic BMS schedule as initial solution for the ILP solver along with the ILP formulation, leads to a considerable speedup and in some cases, such as Blackscholes for medium and large inputs, this avoids a timeout. Heuristic BMS is fast for all graph sizes, but for tight bounds may need to be followed by the hot ILP execution if it cannot find a suitable schedule.

The most closely related previously published results are for finding the minimum numbers of registers needed to execute instruction graphs whose size is in general much smaller than the computation graph sizes. The only public graph and ILP solving time we could find is from the work of Chang et al. [**Chang:97**] and has only 12 vertices. On this graph, their ILP formulation takes one minute (on their 1997 machine), while both the heuristic BMS and ILP BMS finish in under a second (on our system).

4.9.6 Inspector phase time evaluation

The inspector phase consists of building the computation graph and running the BMS algorithm. Schedule caching removes the overhead associated with both these stages and adds some overhead of its own (for hashing the schedules and loading them from disk). Table 4.7 shows the execution time of the inspector relative to the serial execution. For the BMS runtime we include the smallest and largest time encountered. The reason for this variation is that BMS may take more time for tighter bounds, since the first schedules attempted will fail to observe the memory bound.

Benchmark	**Graph creation(%)**	**BMS Algorithm Min(%)**	**BMS Algorithm Max(%)**
Smith-Waterman	0.5	17.8	98.1
Blackscholes	3.3	2.1	29.0
Cholesky	2.9	3.8	99.4
Gauss Jordan	20.3	6.6	94.0
Merge Sort	19.8	20.0	310.2
STG 58	0.5	1.0	109.2
STG 59	0.1	0.7	42.5

Table 4.7 : Timing results for the inspector (graph creation and BMS scheduling), as percentages of the serial execution time.

From the table, we see that graph construction can take up to 20% of execution time and the maximum time needed to run the BMS algorithm can be 3× larger than the serial execution time. Schedule caching is therefore valuable in amortizing the potentially large overhead of the inspector.

4.9.7 Large memory experiment

For systems without support for paging to disk, BMS enables the execution of programs that would otherwise crash attempting to use more than the available memory, but how does the paging mechanism affect the BMS results?

We analyze application behavior on workloads that require disk paging by using a larger input size for the Smith Waterman application. The results in Figure 4.5 include the BMS performance for 270, 280 and 310 tiles of the same size, and the graphs show interesting changes relative to Figure 4.3a. For very tight memory bounds, the BMS-CnC performance is close to serial, because sequential execution is needed to reach the desired memory bounds. As the bound gets larger, performance increases due to more parallelism, until it reaches a performance sweet-spot. This sweet-spot is generally close to the physical memory size (32GB), but its exact location depends on how close the enforced maximum memory bound matches the actual memory used at run-time.

Increasing the memory bound even more leads to a performance degradation because disk swapping starts being used. The last part of the graph shows constant time because the program has already reached its parallel footprint and giving a larger bound does not affect the schedule any more. The sweet-spot enabled by BMS leads to 39% faster execution compared to parallel CnC, showing that BMS can increase performance and lower the memory footprint of applications with large memory requirements.

Comparing the results for the three runs which use inputs of increasingly large sizes (270, 280 and 310 input tiles), we notice that all three have similar curves. Interestingly, the fraction of memory saved by using the BMS sweet-spot instead of parallel execution increases with the input size. The memory savings reach 34% for 310 tiles.

4.10 Conclusion

This chapter proposes a new scheduling technique to find memory-efficient parallel schedules for programs expressed as dynamic task graphs. Our technique, called bounded memory scheduling, enforces user-specified memory bounds by restricting schedules and trading off parallelism when necessary. The evaluation on several benchmarks illustrates its ability to accurately control the memory footprint while exploiting the parallelism allowed by the memory bound.

To make use of an inspector/executor approach in the context of dynamic task scheduling, we presented an efficient schedule reuse mechanism. This technique amortizes the inspector overhead by reusing schedules across executions of the application

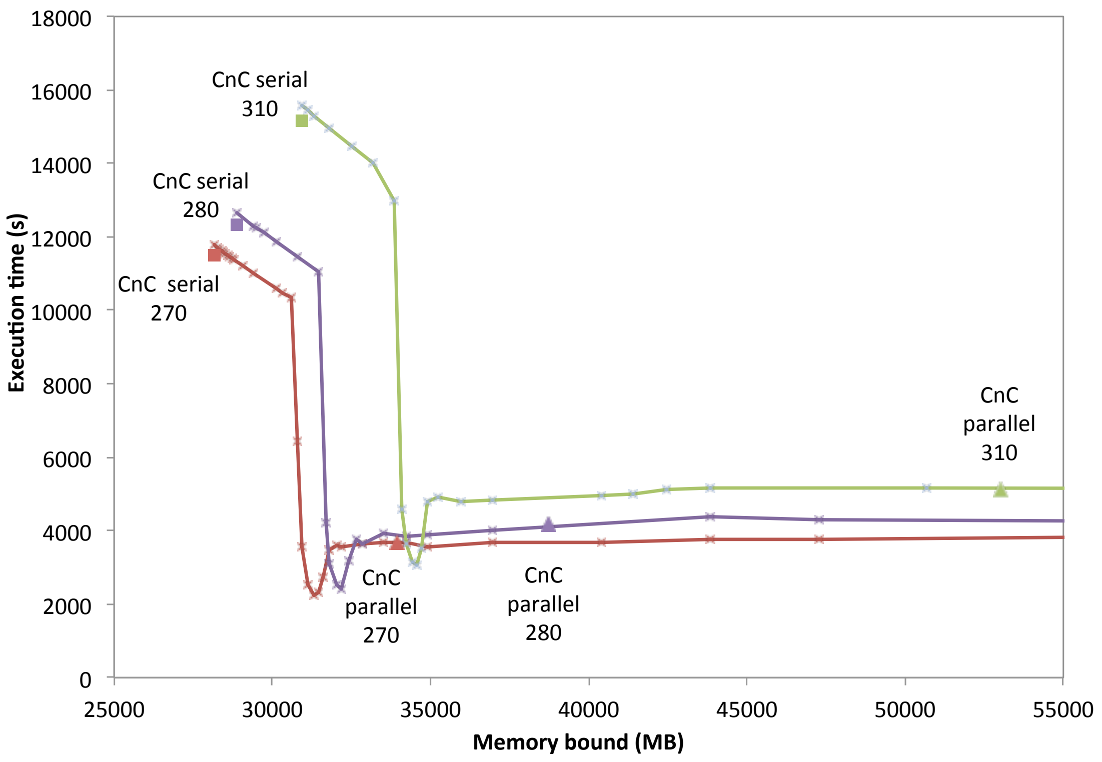


Figure 4.5 : Smith-Waterman results on large inputs (270, 280 and 310 tiles). BMS enables the use of a sweet-spot with good performance and low footprint at the same time because it avoids swapping. The physical memory size is 32 GB and the computation graph for 310 input tiles has 192,202 nodes.

that exhibit the same computation graph — even when the input parameters change.

Chapter 5

Automatic Selection of Data and Task Distributions for Tagged Dynamic Task Graphs

5.1 Introduction

Distributed systems are increasingly common, but writing programs for them is still notoriously challenging [**Sookoor:09**, **Conway:14**, **Murphy:06**]. Much of this difficulty arises because programmers are required to manually manage the data partitioning and communication which are known to the community as critical performance factors: "for distributed-memory multicomputers, the quality of data partitioning is crucial to obtaining high performance" [**Palermo:01**].

Automating data and computation distribution is important for several reasons. First, it considerably eases programming, by abstracting away these low level details which developers are rarely required to consider when programming the more common shared-memory systems. Second, it improves code maintainability because otherwise, changes to program code may require corresponding changes to the data distribution. Third, code modularity makes it difficult to identify the best distribution, since it depends on how the module is used in other applications. Finally, portability is difficult to achieve with hand-written data distribution because different hardware platforms require the use of different data distributions.

Mainstream parallel programming environments provide little or no help with selection of data distribution functions. While many projects have proposed automatic partitioning and data distribution, these approaches have not yet reached mainstream programming. One possible reason for this is that compiler-based approaches are not a good fit for the dynamic nature in which many applications are written [**Wholey:92**, **Reddy:14**]. In this chapter, we propose an automated approach for run-time data and task distribution. Compared to traditional approaches, our system targets task-based applications which lack the regularity of traditional distributed-memory algorithms and instead supports dynamic task-based applications. Instead of using heuristics to determine a possibly suboptimal data distributions, we investigate methods to determine optimal distributions (under certain constraints) through integer linear programming; this approach is particularly attractive because the performance of integer solvers has increased by an order of magnitude in just the last decade [**Bixby:13**].

Our work described in Chapter 4 on the use of inspector/executor (I/E) for dynamic task graphs opens up new possibilities for program optimization in the shared-memory context. On the other hand, distributed systems will be increasingly important in the future, since they allow scaling to much larger input sizes. Extending the techniques presented to support distributed execution is attractive because it could have profound effects on what input sizes are solvable or are not solvable. This chapter focuses on enabling the application of I/E in a distributed memory context through flexible reuse of the results and shows that I/E-based auto-generation of distribution functions leads to results that are on par with or better than hand-coded distribution functions.

Our technique automatically identifies optimal data distribution and task placement for distributed programs based on the data access patterns obtained from the dynamic computation graph of the application. The problem is formulated as an integer linear programming model and fed to a state-of-the-art solver. The optimality of the distribution is relative to the computational model described in Section 5.2 and to the parametrized distribution function presented in Section 5.3.

In distributed execution, the dynamic computation graph can grow to sizes for which the inspection cost becomes unreasonable. Traditionally, this overhead would be amortized by only inspecting code in loops, so amortization happens by running multiple iterations, but this is not a frequent case when applying I/E for inspecting task graphs rather than loops. A major contribution of this work is an approach to enable amortization of the inspector cost which we accomplish through two technique.

First, we limit the inspection overhead by only inspecting runs with small to medium scale inputs. Even on small inputs, the problem of finding optimal distribution functions with linear programming has an exceedingly large cost, so we show an formulation of this problem that is polynomial in the number of processors used, rather than in the size of the dynamic computation graph of the application. This has the advantage that the upper bound is constant rather than growing with program input size, as described in Section 5.4.

Second, we amortize the inspection cost by reusing the resulting distribution functions on graphs of large scale. Section 5.5 discusses our strategy for distribution function reuse.

Experimental results discussed in Section 5.6 show that the runtime performance loss incurred by not generating the placement on the same dynamic graph used when executing the application is small.

5.1.1 Using dataflow for automatic task and data distribution

The execution model used in this work is based on macro-dataflow. It is an extension of Concurrent Collections [**Budimlic:10**] which uses a work stealing [**Blumofe:99**[redacted]] task scheduler inside each node of a distributed system, but tasks are assigned to individual nodes and cannot migrate or be stolen across nodes.

Our programming model's use of implicit data communication offers a critical advantage in writing distributed software because it enables the separation between the distribution function that governs task and data placement on one hand and the application code on the other. As a result, each algorithm does not need to have hard-coded support for a number of specified distribution patterns, but can instead focus only on the computation to be performed and leaves the choice of distribution to the runtime. This desirable feature of the model is known as *decomposition independence* [**Dongarra:92**].

Our dataflow model (as described in Section 4.2) is an especially good match for distributed execution because it enables transparent overlapping of computation and communication. As soon as each new data item is created, the runtime start copying it to the nodes on which it will be consumed. The dataflow nature of the model is key because it enables us to inspect the computation and find the consumers of each item without performing the computation of the tasks themselves. It is these data transfers that enable communication-computation overlap because copying happens in parallel with the execution of other tasks. They start eagerly, so that consumer tasks become ready to run [*] as soon as their inputs are available. This eager approach minimizes the number of tasks delayed because their input data needs to be requested and fetched from another node. Compared to lazy transfers in which tasks have to

[*]The moment they start running is controlled by the work-stealing algorithm.

explicitly request their data to be brought in, this approach also avoids sending request messages and the associated latency.

While eager data transfers have performance advantages, their use may increase the memory pressure - a problem for which we already developed a solution through BMS (see Chapter 4). However, we did not find this to be a problem in practice.

5.1.2 The need for automatic selection of data and task distributions

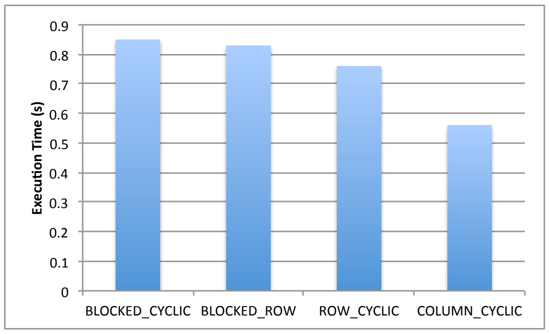
Leaving the problem of finding good distributions in the hands of the programmers is not a general solution because of the large search space involved. An even bigger problem is that there may not exist a single distribution that performs well across all runtime parameters. Figure 5.1 shows results[*] on how the relative performance of various distributions changes when using two different inputs for the same application (Cholesky factorization[†]): the best distribution on one input performs the worst on the other. Similarly, Figure 5.2 illustrates how the performance of a distribution depends critically on the number of nodes on which the program is executed.

These figures show how the best distribution changes drastically with the input size and the number of nodes. Unless we expect the programmer to experimentally evaluate a large combination of these parameters, the use of hand–picked distributions is unfeasible. Instead, we propose a system that automatically finds the best distribution function for task-based programs, taking into account the dynamic graph of the application and the number of nodes on which it runs.

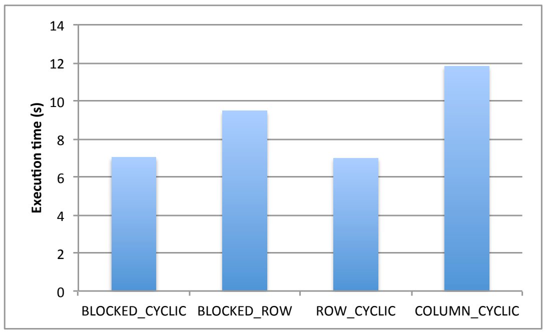
The CnC model allows the mapping of tasks and data to nodes to be done at runtime through arbitrary programmer-written distribution functions. Its ability to

[*]These results were obtained by us, using Intel CnC as described in Section 5.6.

[†]The implementation of Cholesky factorization is the task-based one which is included the Intel CnC distribution.

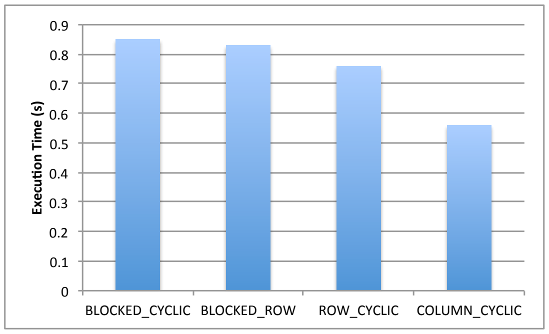


(a) Execution time with an input of $5k \times 5k$.

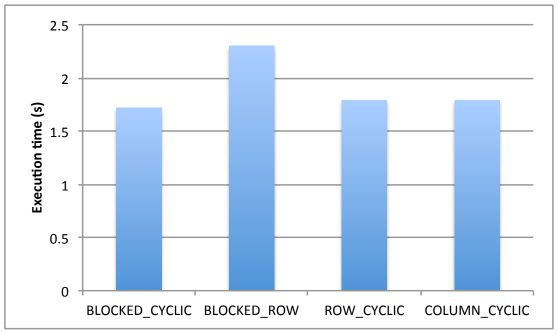


(b) Execution time with an input of $12k \times 12k$.

Figure 5.1 : Execution time of hand-picked distribution functions on Cholesky factorization on 8 nodes. The distribution that performs best for the small inputs is the worst for the larger one.



(a) Execution time with 8 nodes.



(b) Execution time with 2 nodes.

Figure 5.2 : Execution time of hand-picked distributions on Cholesky factorization with an input size of $5k \times 5k$. The distribution that performs the worst on 8 nodes is the best performing when using only 2 nodes.

decide the data and task distribution at runtime instead of compile-time allows for more flexibility, but it does not ease the programmer's burden, since they are still required to find good distributions which remains a challenge.

Because we use a dynamic-single-assignment model, it is useful to understand how the distribution section problem for such a model compares to the distribution problem for programming models with multiple assignment. Having dynamic-single-assignment leads to better precision than traditional task distributions based on owner-computes[*] because the data and task placement is now made for a value, rather than for a variable which can be written to multiple times. The result is that the distribution problem for dynamic single assignment languages also is equivalent to the distribution with re-distributon at arbitrary points for traditional programming models.

5.2 An efficient performance model for choosing data distributions

Software that guarantees optimality is only useful if the guarantees correspond to the improvements in observed characteristics of the program such as execution time. To ensure this, such systems use models of the program which need to match program behavior as close as possible, but at the same time should be easy to compute. Modeling parallel programs to get accurate performance prediction is challenging, but a large amount of research on the topic already exists [**Al-Tawil:01**, **Grove:04**, **Adhianto:06**, **Hoefler:10**]. Good modeling of task-based programs requires identifying the characteristics of the task DAG that lead to good performance. While some of these characteristics can be drawn from previous work on static and dynamic scheduling (such as the critical path length, load balancing) the choice of these parameters is difficult because some of them are not clearly defined for distributed execution. For example, the notion of critical path is usually understood as the longest chain of tasks from the start task to the end task, where length is defined as the sum of execution time of tasks (vertices).

[*]The owner-computes rule for task distribution is widely used and postulates that the processor that owns the left-hand side of an assignment will perform the computation on the right hand size.

In our case, selecting parameters for the model is even more challenging for two reasons. First, many of the program characteristics useful in modeling are unavailable during the inspection because the tasks are not fully executed during inspection. These parameters are those related to the execution time of individual tasks which are essential in computing many characteristics of the dynamic computation DAG, such as the critical path length. Second, because we find the optimal distribution function though integer linear programming (ILP), whose runtime increases with the number of variables and inequalities, any program characteristics we choose must be expressible in ILP with few variables and inequalities.

With these two constraints in mind, we settled on using two parameters whose importance in performance modeling is agreed-upon [**Lee:97**]: the total inter-node communication and the load imbalance of the nodes. Assuming a linear contribution of the two parameters, the objective function that we minimize to obtain the distribution functions has the following form:

$$minimize(\alpha \times WorkImbalanceCost + (1-\alpha) \times CommunicationCost)$$

where $\alpha \in [0, 1]$ is a programmer-defined value expressing the relative importance of the two parameters/costs.

The two costs involved deserve more explanation regarding their meaning and how they are computed. An important factor in controlling the relative importance of the two parameters is finding a baseline to normalize against. The baseline we use is the communication and computation of the average task in the program, so normalize the cost of communication to the average size of items. Ideally, tasks could be normalized based on their relative execution times, but since this information is not

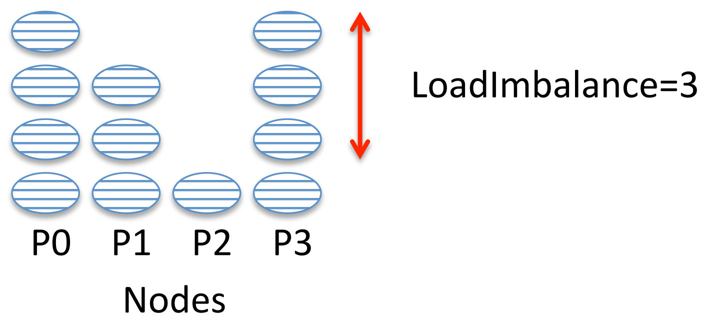


Figure 5.3 : The load imbalance cost is defined as the difference in the number of normalized tasks between the node with highest load and the lowest load.

available during inspection, we instead assume a linear relation between the amount of data touched by a task and its execution time.

The definition of work imbalance cost is chosen by taking into consideration that the ILP formulation needs to be reasonably efficient, while still allowing for an accurate model of the program performance. It is computed as the difference in number of normalized tasks between the node with most work and the node with the least amount of work, as shown in Figure 5.3. This cost can thus take values between zero and the number of tasks spawned.

For similar reasons, the communication cost is computed as the number of normalized[*] items that are only needed on more than one node. With this in mind, and assuming items are the same size, the communication cost can take values between zero and the number of items.

One may wonder why the two costs have seemingly different ranges, but because of the dynamic single assignment, each task creates at least one item and in practice,

[*]As mentioned above, this number is normalized to the average size of an item, so items that are double the size of an average item are considered as two items.

usually exactly one item, in which case the two ranges are identical.

5.3 Choosing the parametrized distribution function

The parametrized distribution function is the family of distribution functions from which we select the one which minimizes the total cost. Our system can select instances of a generic type of checkered distribution pattern, as follows:

$$\mathit{NodeOf}(\langle i, j, \ldots \rangle) = (\lceil \frac{i}{blk_i} + \frac{j}{blk_j} + c \rceil) \bmod P.$$

Here, P is the number of processors(nodes) on which the computation runs. The blk_i and blk_j are parameters are called *block sizes* and the c parameter is the *constant factor*. The problem of selecting the optimal distribution consists of finding the values of blk_i, blk_j and c to minimize the value of the objective described in Section 5.2.

The importance of the constant term is revealed when input programs have multiple item collections because each one gets its own distribution function and the constant enables us to obtain lower costs by "rotating" the allocation of different item collection. For example, the node with the most tasks from one item collection can be assigned a minimum amount of tasks from another, so that the overall load balancing cost is lower, as shown in Figure 5.4. For the simple case where there is a single item collection, the constant factor makes no difference.

An important question arises: *What are the possible values from which the block size parameters should be selected?* Traditionally, block sizes larger than one are used to exploit locality between "consecutive" tasks (consecutive tasks are those whose tags differ by one on exactly one dimension). This makes sense when working with fine grained data because the cost of communication dominates the execution time. How-

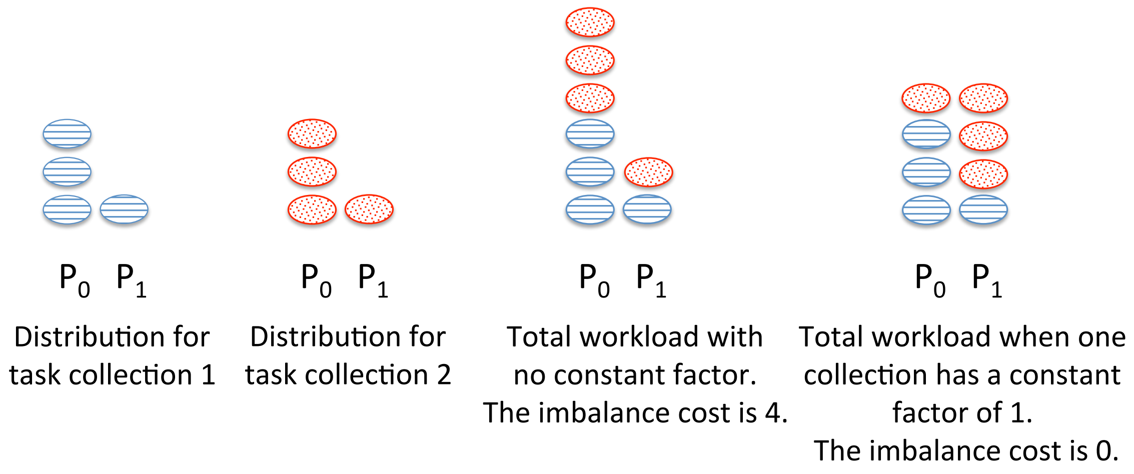

P_0 P_1
Distribution for task collection 1
P_0 P_1
Distribution for task collection 2
P_0 P_1
Total workload with no constant factor. The imbalance cost is 4.
P_0 P_1
Total workload when one collection has a constant factor of 1. The imbalance cost is 0.

Figure 5.4 : The importance of the constant term of the distribution function: it allows us to “rotate” the arrangement of tasks on nodes, to minimize the load imbalance or communication costs. The example above illustrates this for the case of two task collections for which the block sizes used in the distribution functions are fixed.

ever, since CnC applications are tiled, the block sizes that offer the best performance are between 0 and 1. Such values create a “separation” effect by allowing elements of each item collection to be assigned to only a subset of the processor nodes rather than being distributed on all of them, leading to lower levels of communication compared to block-cyclic distributions. This can be done without decreasing the load balancing if another item collection is assigned to the complementary subset of nodes, as shown in Figure 5.5.

Initially, we allowed block sizes to span freely the 0-1 interval, but this led to extremely long ILP solution time. Because of this, we decided to only allow block sizes of the form $\frac{1}{n}$ where $n \in \mathbb{N}$. This key decision that enabled profound simplification of the ILP formulation, as discussed in Section 5.4. Even with this restriction, the parametrized distribution function remains general enough to select the hand-coded

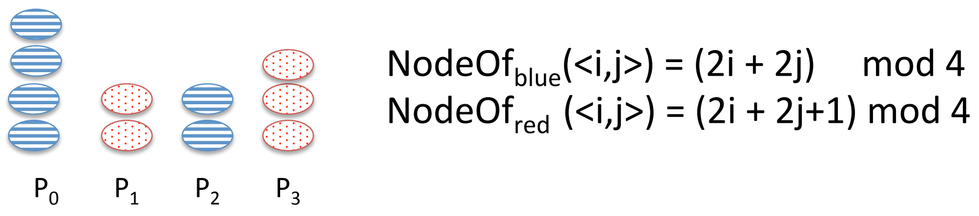

$NodeOf_{blue}(<i,j>) = (2i + 2j) \mod 4$
$NodeOf_{red}(<i,j>) = (2i + 2j+1) \mod 4$
P_0
P_1
P_2
P_3

Figure 5.5 : Distributions with block sizes smaller than one can be used to achieve a "separation effect" in which items from the each item collection are assigned to a subset of nodes to improve communication in the case when much of the communication is between items of the same item collection. Load balancing is then achieved by using different item collections, such as the red (dotted) and blue (lines) item collections in the figure.

distributions included in the Intel CnC distribution. This version of the checkered distribution is as follows:

$$NodeOf(\langle i, j, \dots \rangle) = (RB_i \times i + RB_j \times j + c) \bmod P$$

where $RB_i, RB_j \in \mathbb{N}$ are the reverse of the block parameters and P is the number of nodes.

Our distributed execution system obtains, by inspecting the program, the dynamic computation graph and the number of nodes used and the user specified the value of the α factor characterizes the ratio of computation to communication of the program (see section 5.2). With these two pieces of information, the system expresses the problem of task distribution on nodes for the dynamic graph as an ILP problem and selects the distribution function parameters that minimize the ILP objective. The function with these parameters is then used to run the application.

The main reason for choosing this pattern for distribution function was that it allows us to generate many of the distribution functions hand-picked by Intel for our test applications, enabling us to compare the auto-generated functions against the Intel distribution. However, it is important to know how our pattern compares to the traditional functions, such as block cyclic.

As described in Section 2.4, the $2D$ block cyclic distribution usually views the nodes as a grid and the distribution on each dimension of the grid is controlled by a single dimension of the input data. The checkered distribution function takes a linear view of the nodes, but makes node choice a function of both dimensions of the input data. We make this choice because solving an ILP model that corresponds to the $2D$ block-cyclic distribution is computationally intensive since we would need to express two modulo operations (which are expensive to express as ILP inequalities).

Note that, because the CnC model is already tiled, so instead of block-cyclic the distribution function that mirrors block-cyclic for fine-grained distribution is simply the cyclic one.

Definition 5. *Two distribution functions $f : \mathbb{N}^d \rightarrowtail P_f$ and $g : \mathbb{N}^d \rightarrowtail P_g$ where $|P_f| = |P_g|$ are equivalent if and only if there is a bijective function $t : P_f \rightarrowtail P_g$ such that $t \circ f = g$.*

Theorem 3. *There is no checkered distribution with integer parameters such that it and the $2D$ cyclic distribution with square grid are equivalent.*

Proof. The $2D$ cyclic distribution with square grid and the parametrized checkered distribution are expressed formally as:

$$\text{Cyclic distribution: } f(\langle i, j\rangle) = (B_i \times i + B_j \times j) \bmod P^2$$

$$\text{Checkered distribution: } g(\langle i, j\rangle) = \langle i \bmod P, j \bmod P\rangle$$

In the above expressions, P is the number of processors on one dimension of the square grid with to a total of P^2 processors.

From the definition of the *modulo* operation and the definition of g, we notice that $g([i,j]) = g([i,j+P]), \forall i,j \in \mathbb{N}$. If there is a bijective function b such that $b \circ f = g$, then:

$$\begin{aligned} b(f(\langle i,j\rangle)) &= g(\langle i,j\rangle) \\ &= g(\langle i,j+P\rangle) \\ &= b(f(\langle i,j+P\rangle)) \end{aligned}$$

which in turn implies $f(\langle i,j\rangle) = f(\langle i,j+P\rangle), \forall i,j \in \mathbb{N}$ because of the bijective nature of b. But this means that

$$\begin{aligned} f(\langle i,j+P\rangle] &= (B_i \times i + B_j \times (j+P)) \bmod P^2 \\ &= (B_i \times i + B_j \times j + B_j \times P) \bmod P^2 \end{aligned}$$

for $\forall i,j \in \mathbb{N}$.

This means $\exists k_1 \in \mathbb{N}$ s.t. $B_j = k \times P$.

A similar process for the j axis leads to $\exists k_j \in \mathbb{N}$ s.t. $B_j = k_j \times P$. But this means that f can be rewritten as:

$$\begin{aligned} f(\langle i,j\rangle) &= (B_i \times i + B_j \times j) \bmod P^2 \\ &= (P \times (k_i \times i + k_j \times j)) \bmod P^2 \\ &= (k_i \times i + k_j \times j) \bmod P \end{aligned}$$

Then, the codomain of f must have a cardinality of P rather than P^2 which contradicts the assumption that b is bijective since the cardinality of its domain is P^2. $\square$

5.4 Efficient ILP formulation of the model

The time it takes to solve an ILP problem instance depends critically on the number of variables and equations in that instance. For this reason, the straightforward formulation of the task and data distribution problem cannot be used; in practice, the unoptimized formulation uses more memory than reasonable (32 GB) and hits the timeout of 24 hours of computation. To solve this issue, we propose an approach in which the problem is transformed by the inspector into an equivalent one which is optimized for efficient solving by the ILP solver. The key principle is that we are trading off a little extra computation time during the inspector for a large reduction in ILP solution time.

The straightforward ILP formulation consists of assigning one ILP variable per task to represent the node on which that task runs. Then, we find the workload of each node by counting how many of these variables equal each node id. The communication cost can be computed similarly based on the number of communication edges that connect tasks assigned to different nodes.

The key to building a more efficient ILP formulation is that instead of using the formulation described above which is proportional to the size of the input graph, we can reduce it to one proportional to the number of nodes (processors) by using. Instead of assuming that each task can be assigned to an arbitrary node, we can observe that a task with tag $\langle i, j, \ldots \rangle$ must be assigned to the same node as $\langle i + n \times NO\_NODES, j + m \times NO\_NODES, \ldots \rangle$ no matter what integer parameters to the checkered distribution function are used (in accordance with Section 5.3). This observation allows us to group together these tasks and use of a single ILP variable for the node they are assigned to. When computing the load balancing cost, this change significantly reduces the number of variables, and adds only a constant coefficient equal to the number of original tasks associated with each compacted task. The number of variables becomes $\mathcal{O}(NO\_NODES^2)$ instead of being proportional to the size of the computation graph.

To compute the communication cost, the naive formulation would count, for all items, the number of producer-consumer pairs whose assigned nodes are different - meaning that the data needs to be communicated to a different node. We can apply a similar compaction as described above by aggregating the number of edges whose end-points must be assigned to the same pair of nodes. This reduces the number of variables used for recording the local or remote status of each communication to $\mathcal{O}(NO\_NODES^4)$.

Since the ILP objective is a sum of the load imbalance and communication costs and their ILP encoding was described above, this means we have just reduced the size of the problem to a value whose upper bound is independent of the computation graph and of the program input size. This reduction is essential since computation graphs used in distributed executions are large and ILPs are computationally expensive to solve.

5.5 Distribution function reuse

Because ILP solving is part of the inspector stage, the inspector time is a large fraction of the total application execution time. Traditionally, I/E systems would only inspect code located inside a loop, so they could amortize the inspection cost across multiple loop iterations, but this is not possible in our case. Previous work showed that, for shared-memory execution of task graphs, the inspector cost can be amortized by reusing the inspector results across application runs, when the computation graph is identical, even when most data is different. This is only part of the solution for the reuse of distribution functions in a distributed execution scenario, because running the inspector on large input sizes becomes impossible because of the high cost of generating the computation graph. In this work, we propose and evaluate the reuse of inspector-selected distribution functions across different inputs without requiring that they have the same dynamic computation graph. The main advantage of this approach is that it allows us to inspect small to medium inputs — doable in reasonable time — and then to reuse the results for larger inputs commonly used in distributed systems. The question that arises is: *Will distribution functions generated on an execution lead to good performance when applied on a different one?* To answer this we need to look at what data is used as an input parameter to the distribution function and differs between the medium and large inputs.

The first such parameter is the input data of the application since the minimization of the objective is based on the specific computation graph of that input. The reuse of distributions across different inputs is based on the concept of computation-to-communication ratio which is a modeling notion traditionally used for performance evaluation of shared-memory and distributed applications. The computation-to-communication ratio is known to scale sub-linearly with the input size in a majority of applications [**Culler:97**]. We take advantage of this fact to reuse the distribution functions obtained on medium-sized inputs when running large-sized inputs.

The second parameter that affects the distribution functions is the number of nodes: it is used in the modulo operation of the parametrized distribution function. To ensure that the function works correctly when running on fewer processors, our strategy is to compute distribution functions assuming the largest number of processors, and then reuse them on the number of processors that is desired [*], with the restriction that both the maximum and actual number of processors are powers of 2.

To obtain a quantitative guarantee on the effects of reuse using a different number of processors, let us analyze what are the consequences of having a distribution function computed for 2^n processors and reusing it on s^p processors, with $n > p$.

Remember that the first part of the total cost is the load imbalance cost, which is the difference in normalized task load between the busiest and lightest nodes. The cost can increase, the worst-case (biggest) increase happens when the reuse folds together the nodes with the highest load onto one and the nodes with the lowest load onto another, leading to a worst-case 2^{n-p} increase in the load imbalance cost. In the best case is cost can decrease if using the lower number of nodes leads to a better load balance.

If we analyze the effect of such reuse on the communication part of the cost, in practice we can expect a reduction because some data that previously required communication may become shared-memory data accesses because of the increased number of tasks assigned to each node. There can also be cases where the cost increases because of the particular communication pattern of the application.

Figure 5.6 shows how distribution function reuse works in our system both regarding the input size and the number of processors.

With the approach for reuse described above, we can detail the inspection process which is as shown in Figure 5.7.

Recall from Section 4.3 that the inspector can build the dynamic computation

[*]The reused distribution function uses the correct number of processors, but its parameter values are those identified on the original run.

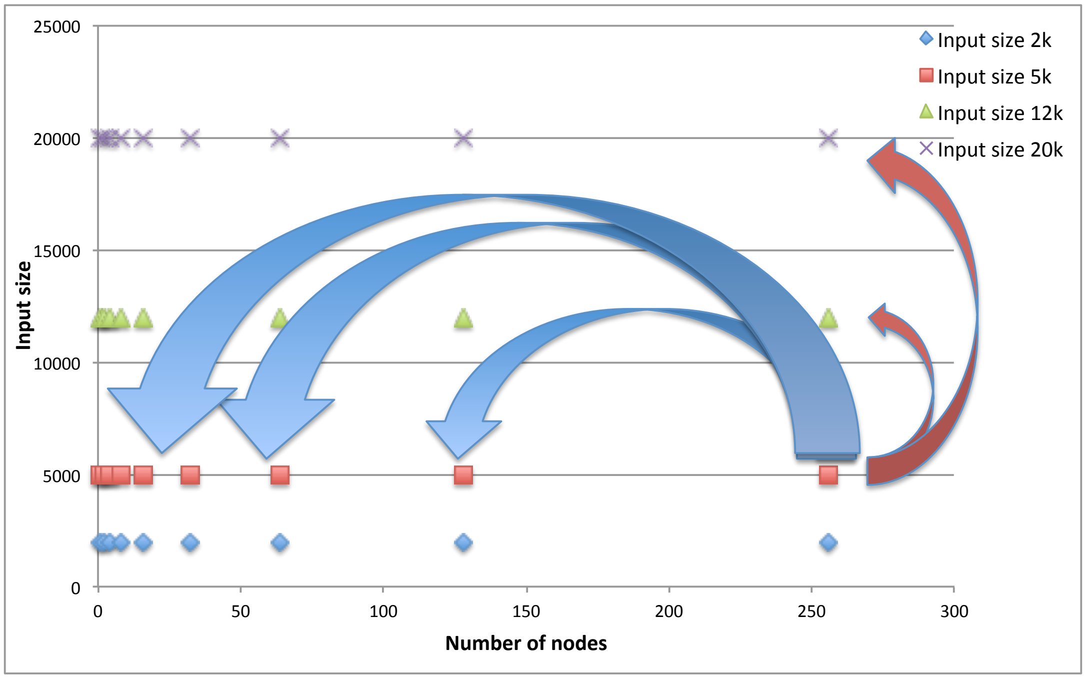


Figure 5.6 : Distributions computed on a particular input size and number of nodes are reused on fewer nodes (lower values on the X axis) and on larger input sizes (higher values on the Y axis).

graph without performing the actual computation of the tasks. In Chapter 4 we defined *the root set* as the set of items consumed or produced by the environment and the set of steps prescribed by the environment. Because of the deterministic nature of CnC, two computation graphs with the same root set must be identical.

During the inspector we use the root set to identify if an execution with the same computation graph has been previously encountered [*] and if so, we just apply the stored distribution function associated with it. If this is the first time that root set has been seen, we must expand the full computation graph to identify its size. Once the graph size is known, we can decide if the graph size is sufficiently close to that of

[*]Note that we do not expand the full computation graph, so for this common case the algorithm is fast.

a smaller graph encountered earlier to warrant the reuse of that graphs distribution function. We use a simple policy that reuses the distribution of the smaller graph if the size difference is less than an order of magnitude. If reuse cannot be applied, we need to express the distribution function generation as an ILP problem and solve it. Finally, we execute with the newly computed distribution function.

The most costly parts of this process (the ILP solving) is only done if the two optimization criteria are not met: no graph of the same order of magnitude has ever been executed.

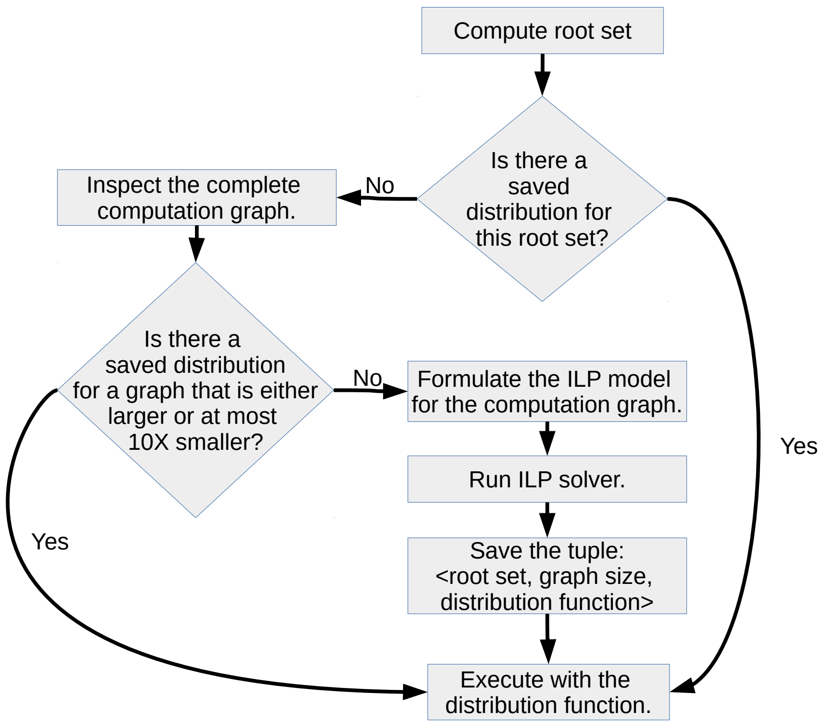

Compute root set
Is there a saved distribution for this root set?
No
Inspect the complete computation graph.
Yes
Is there a saved distribution for a graph that is either larger or at most 10X smaller?
No
Formulate the ILP model for the computation graph.
Run ILP solver.
Save the tuple: <root set, graph size, distribution function>
Yes
Execute with the distribution function.

Figure 5.7 : The operation diagram of the inspector phase.

5.6 Case study: Cholesky factorization

To evaluate the system we need to answer the following questions: *How do the best automatically selected distribution compare to the best hand picked ones?* and *How do the selected distributions perform when reused on larger inputs?* The next two subsections answer these questions.

The results shown in this section have been obtained on the *Davinci* cluster at Rice University, which contains Westmere processors with 12 cores per processor which are clocked at 2.83 GHz. Each node has 48 GB of RAM and the nodes are connected with QDR InfiniBand (40 Gb/s). The results have been obtained using the Intel Concurrent Collections runtime.

5.6.1 Distribution function performance

Figure 5.8 shows how the performance of the automatically selected distributions compares with that of those hand-picked by Intel CnC. Since the hand-selected distribution functions were tuned by Intel manually, we expected that matching the performance of these to be a success. The auto-generated function is able to not only match, but outperform by up to 5% the best hand-coded distribution function, the reason being that it is specific to that particular input graph shape.

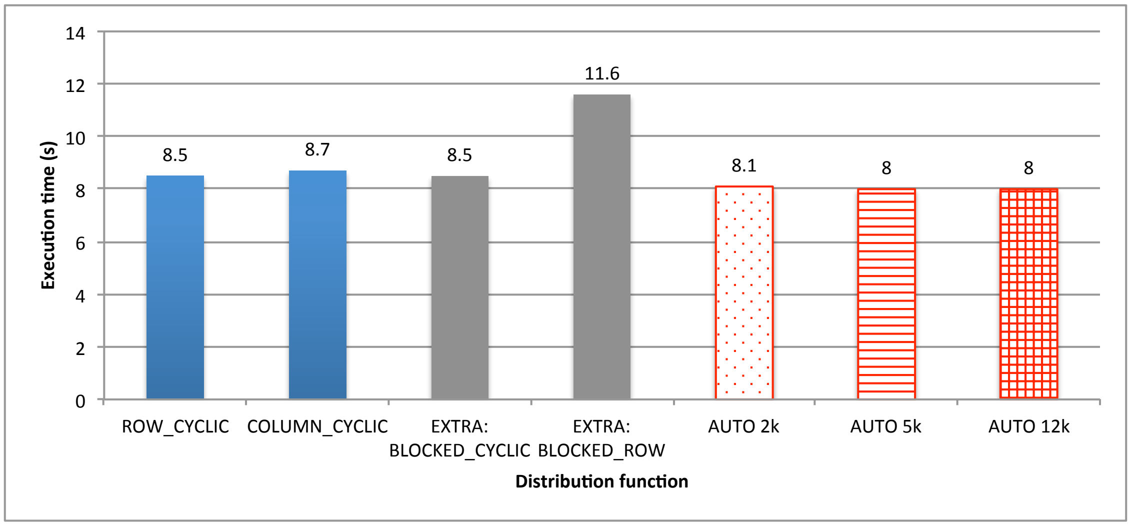
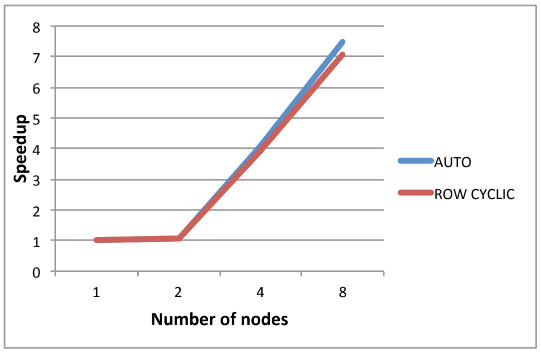


Figure 5.8 : The performance of various distribution functions on Cholesky factorization with an input of size 12k. The auto-generated distribution functions (shown in red) were obtained on inputs of 2k, 5k and 12k respectively with the α value that leads to the best performance. The 2k and 5k show good performance when applied on the larger input size when compared with hand-picked distribution from Intel CnC and with the automatically selected distribution for that input size (12k). The hand-coded functions perform worse, whether they are expressible with our parametrized distribution function (blue bars) or outside of it (gray bars). There is a small difference that shows that generating the distribution function on larger inputs leads to a small performance improvement.

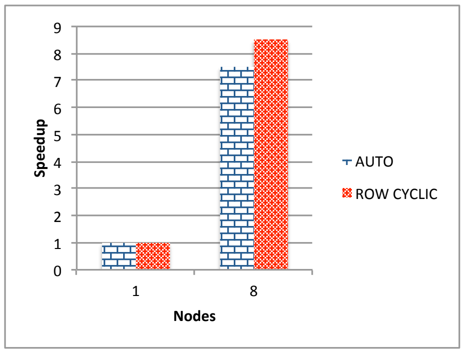
Figure 5.9b shows how the speedup of the auto-selected distribution functions compares with that of the hand-selected ones for different number of nodes. Auto-selected functions perform better than the best hand-picked one on any number of nodes, even though it was generated specifically for 8 nodes.

5.6.2 Distribution function reuse

Since in practice distribution functions will be selected on small to medium inputs but reused on large ones, we are particularly interested in how the performance varies when using reuse instead of selecting new functions. Figure 5.8 shows that, even



(a) The best hand-picked function and the auto-generated one show almost perfect speedup with exception of the case when using 2 nodes (12k input size).



(b) The relative speedup obtained with the automatically selected distribution, relative to the speedup obtained with the best distribution function hand-picked by Intel.

Figure 5.9 : The speedup of the auto-generated distribution function relative to the best hand-picked distribution function on Cholesky (12k input size).

when used on larger inputs (with much larger computation graphs) these functions perform well. The figure shows how the two functions - one obtained on an input matrix of $2k \times 2k$ (a graph with 1750) tasks and one obtained on an input matrix of $5k \times 5000$ (a graph with 23,375 tasks) - perform when used on graphs of a larger size ($12k \times 12k$, a graph with 302,500 tasks).

5.6.3 The effect of alpha coefficient

The only parameter whose value is controlled by the programmer is α, which controls the relative importance of load imbalance cost and communication cost in the overall objective that needs to be minimized. Figure 5.10 shows how the performance of generated distribution functions change with the value of alpha and how the generated functions compare with the hand-coded ones. For α values close to 1 (meaning communication cost is very important) the performance of the generated distribution functions drops visibly. This is because extremely low communication cost can only be achieved by co-locating tasks on fewer and fewer nodes - leading to sequential execution. On the other hand, when α values are between 0 and 0.5, the performance of generated functions is competitive with the hand-coded ones. Note that this is a large range of good values, meaning the programmer does not have to put a lot of effort into tweaking this parameter to get good performance.

To get a better understanding of what happens in the 0-0.5 range for α, Figure 5.11 shows a zoomed-in view of the previous graph. We notice that, while using an α value of zero leads to competitive performance, it is not the best value. This is because the communication cost also plays an important role in finding distribution functions that perform well. Functions that are generated on the small ($2k$) and medium ($5k$) input are both able to outperform hand-coded functions when using α values of 0.1-0.2 and they match the performance of the distribution obtained on the native (large) input size.

One issues that arises with the objective function is that performance can be dependent of the finding values of α that perform well. Both figures show that there is a wide range of α values where performance is competitive with the hand-picked distribution functions. When reusing distributions obtained on small inputs ($2k$), it becomes more difficult to have the auto-generated functions outperform the hand-picked, but for medium-sized inputs ($5k$) or when using the native size ($12k$), auto-generated functions outperform all hand-picked ones. This holds true for 80% the possible range of α, meaning that choosing an appropriate value for α is not expected to be a difficult task and is a candidate for successful automation.

Note that the $2k$, $5k$ and $12k$ inputs are separated by an increase of an order of magnitude in the size of the dynamic computation graph. The fact that $5k$ is enough to match the performance of hand-picked functions means that, in practice, generating the functions on inputs 10 times smaller than real inputs is sufficient.

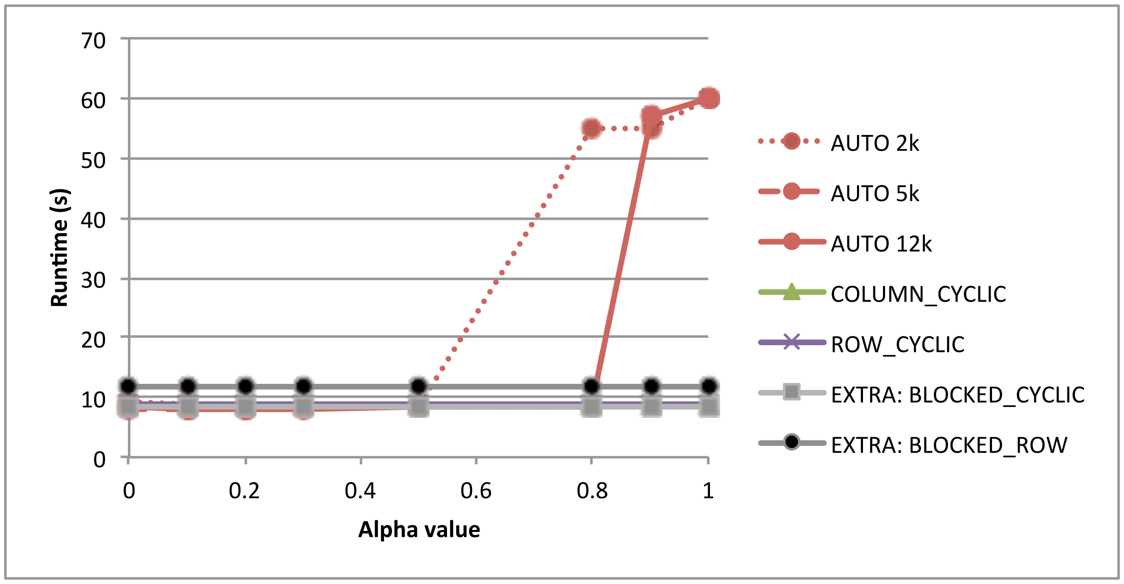


Figure 5.10 : Performance results on Cholesky with input of size 12k and distribution functions obtained on 2k, 5k and 12k inputs, as a function of α. For large values of α ($\alpha \rightarrow 1$), communication cost plays a big role in the function objective, so tasks are aggregated on a few nodes leaving others underutilized and performance suffers.

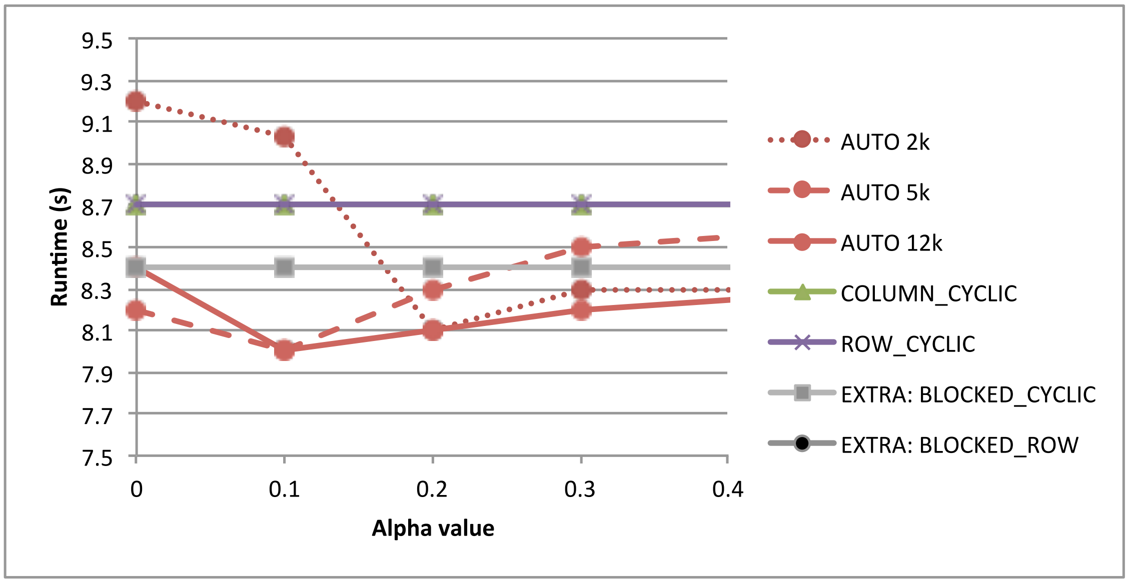


Figure 5.11 : A zoomed-in view of figure 5.10. The performance sweet-spot of the generated functions is achieved for α values in the 0.1-0.2 range. For lower values, the additional load balance improvement is offset by increased communication. The generated functions outperform all hand-coded functions by up to 5%.

5.7 Summary

In this chapter, we proposed techniques for enabling automatic selection of distribution functions which apply to distributed programs expressed as dynamic task graphs, replacing a manual process traditionally performed by programmers in a time-consuming process.

The distribution functions we generate are optimal relative to our cost model which is simple enough to be computationally solvable with integer linear programming, but complex enough to accurately model performance. Our experimental results show that the auto-generated distribution functions perform on par with or up to 5% better than the hand-picked functions included in the Intel CnC distribution, while requiring minimal intervention from the programmer.

To reduce the inspector cost, we propose an efficient formulation of the distribution selection problem whose number of constraints and variables is polynomial in the number of nodes used to execute the program rather than in the size of the problem. To amortize the inspector cost in the context of distributed execution of dynamic task graphs, we propose and evaluate the reuse of distributions across different input sizes and different number of processors.

Chapter 6

Related work

6.1 Dataflow scheduling with bounded space

Because CnC is a macro-dataflow programming model, it makes sense to compare our CnC scheduling approach with those developed for traditional dataflow languages.

While dataflow scheduling has a wide variety of approaches, the most widely known are the static ones. Working in the Ptolemy project, Lee and Messerschmitt [**Lee:1987**] looked at scheduling synchronous dataflow, which differs from traditional dataflow in that the amount of data produced and consumed by a node is specified a priori. Their approach finds space bounds for buffer sizes, based on the existence of cyclic schedules which are in turn computed based on the balance equations of the graph topology matrix. Govindarajan at al. [**Govindarajan:2002**] improve on this work by minimizing buffer storage requirement in constructing rate-optimal compile-time (MBRO) schedules for multi-rate dataflow graphs. They enable overlapping iterations ("rate optimal") though something similar to software pipelining, so the obtained rate is improved. Integer linear programming is used for their optimality claim.

Cyclo-static dataflow is a superset of synchronous dataflow for which close-to-minimal buffer sizes can be identified without converting the graph to single-rate first, as shown by Wiggers [**Wiggers:2007**]. Similar bounds were obtained by Buck and Lee [**BuckLee:1997**] for the boolean dataflow model which is an extension of dataflow in which conditional token consumption and production are allowed.

Streaming systems are modern twists on dataflow systems and share similar scheduling techniques. StreamIt allows the user to control the balance between space consumed by data and code through phased scheduling [**Karczmarek:2003**] which provides a flexible trade-off between code size and buffer size.

6.2 Task scheduling with asymptotic bounds

To the best of our knowledge, this work is the first to tackle the problem of scheduling with a fixed memory bound in the context of dynamic task scheduling, but there is related work on amortized analysis of memory consumption for parallel programs. Burton [**Burton:96**] was the first to propose bounds on the memory complexity of dynamically scheduled parallel computations.

Simpson and Warren [**Simpson:99**] present a survey of work in this area. Blelloch et al. [**Blelloch:97**], Narlikar and Blelloch [**Narlikar:99**], Blelloch et al. [**Blelloch:99**] and Fatourou [**Fatourou:01**] identified increasingly better bounds.

The best memory bounds obtained are directly proportional to the memory consumption of a particular serial schedule and include at least an additive factor proportional to the critical path of the computation. In contrast to these approaches in which bounds are dependent on the memory consumption of the particular serial order of tasks and on the number of processors available, BMS-CnC considers the maximum footprint a hard upper bound for execution. Compared to on-the-fly schedulers with asymptotic memory bounds, we can impose fixed memory bounds and work around the on-the-fly restriction by using the inspector-executor model. This enables us to use the whole computation graph in scheduling, effectively turning the scheduling "offline". Because of this, BMS can handle even the worst case (adversary picks worst task spawn ordering in the input program), that could lead these schedulers to unnecessarily large footprints. Also, the performance of BMS is independent of the order of task spawning in the programmer-written schedule. On the other hand, they can offer performance guarantees and have wider applicability because of their less restrictive programming model, on-the-fly approach and no inspector overhead.

In its view of performance, BMS-CnC relates to work-stealing schedulers such as in Cilk [**Frigo:98**] through its philosophy of starting with an application-defined parallel slack and decreasing it to levels that guarantee bounded-memory execution. Even with this restriction, on systems with good processor-memory balance, the assumption of parallel slack should not be affected by BMS. Once the BMS transformation is done, the application is sent to a work stealing scheduler which ensures provable performance bounds for the modified computation graph.

In traditional work-stealing, once one a task has been executed, it cannot be undone. This may lead to cases where, once a partial schedule has been executed, no remaining scheduling option can fit the memory bound. Fixing this issue while still using work stealing would require backtracking, but BMS achieves the same result, more efficiently, by exploiting the computation graph.

Other projects [**Hofmann:03**, **Braberman:08**] analyze the memory consumption of serial programs. Other projects [**Hofmann:03**, **Braberman:08**, **Hofmann:09**, [redacted] **Campbell:09**] analyze the memory consumption of serial programs, but this is a difficult problem to solve accurately with only static information. The techniques are expensive, based on linear programming, but only need to be computed once per application, compared to the inspector/executor based approach where the valid schedules to be computed once for each computation graph encountered.

6.3 Inspector/executor

BMS is a novel application of the inspector/executor system proposed by Salz [**Salz:91**] who used it to efficiently parallelize irregular computations.

Salz, along with most other inspector/executor works amortize the cost of the inspection across multiple executions of a loop. We use schedule caching instead, as in out case there usually are no iterations.

Based on inspector/executor, Fu and Yang propose the RAPID system for distributed computation [**Fu:96**] which bounds the memory assigned to copies of memory on each node, but does not bound the footprint of the program.

RAPID is similar to BMS-CnC in that it enables inspection of task based programs, but BMS-CnC can take advantage of more scheduling freedom because it lacks anti and output dependences. In a follow-up work [**Fu:97**] inspector/executor is used to bound the memory assigned to copies of data in the distributed environment, but not the total footprint of a parallel program. In their static scheduling approach, each data object is assigned a home node on which the object is persistent, but objects also may be sent on remote nodes where they are volatile. The work proposes algorithms to reduce the footprint of volatile objects on each individual node - a problem specific to the distributed computation model. They separates the computation into slices that access the same data; slices have the characteristic that on each processor only volatile data from the currently executing slice is needed. By scheduling slices to nodes sequentially, they obtain the main result that, for a subset of applications such as sparse LU factorization, only one volatile variable needs to be stored per processor, for a total footprint of $\mathcal{O}(S_1/p+1)$ per processor. The S_1 factor is considered the serial footprint for permanent data that is never collected — the problem of deallocating objects from their home is not considered, which simplifies matters, as is the order of execution of tasks inside a slice which also affects the total memory footprint. Because it is meant for use in cases where the graph size is limited, they do not consider approaches that limit the memory footprint of the inspector.

6.4 Register allocation and instruction scheduling

The BMS problem is related to the widely-studied problems of register sufficiency and combined instruction scheduling and register allocation. Barany and Krall [**Barany:13**] propose a type of code motion to decrease spills by using integer programming to identify the schedules that reduce overlapping lifetimes. Pinter [**Pinter:93**] identified the fact that some variables in the program must have overlapping lifetimes while some don't need to which is an observation that we used in our ILP optimizations; he builds a parallelizable interference graph including "may overlap" edges to ensure that his register allocation does not restrict schedules. In the same context of register allocation and instruction scheduling, Norris and Polloc [**Norris:93**] use the parallelizable interference graph and add data-dependence graph edges (similar to our serialization edges) to remove possible interference. loops. The CRISP project [**Motwani:95**] introduced an analytical cost model for balancing register pressure and instruction parallelism goals in a list scheduler which influenced the schedule relaxation technique we propose.

Ambrosch et al. [**Ambrosch:94**] propose starting from the minimal interference graph which only includes edges between live ranges that must overlap. dependence edges corresponding to ranges assigned the same color, which is the same condition we use when inserting serialization edges. They need to recompute the interference graph when adding such edges, but BMS-CnC does not suffer from this disadvantage.

Govindarajan et al. [**Govindarajan:03**] perform scheduling to minimize the number of registers used by a register DDG, an approach called minimum register instruction sequence (MRIS). BMS and MRIS have considerable differences:

- *different scalability requirements:* MRIS has been targeted to basic block DDGs consisting of tens of nodes, whereas BMS must support tens of thousands of nodes, so the BMS heuristics trade accuracy for performance.
- *different reuse models:* Because BMS works on memory instead of registers, the input and output data of a task cannot share the same memory slot, so lineages cannot be formed. Without lineages, coloring the interference graph of the computation graph of common applications would take more memory than the original footprint of the program.
- *different objectives:* While MRIS simply minimizes the number of registers, BMS the best schedule for a given memory bound. The MRIS minimization objective leads to sacrifices of parallelism that are unnecessary for BMS. For example, value chains are created by inserting sequencing edges that force a particular consumer to execute last; BMS avoids this restriction of parallelism by using multiple serialization edges.

The URSA project [**Berson:98**] compares various approaches for combined register allocation. Touati [**Touati:01**] proposes the use of serialization arcs to decrease the number of required registers.There are multiple related projects that apply optimal techniques [**Wilken:00**, **Bednarski:00**, **Lozano:12**, **Beek:01**, **Malik:08**]) for scheduling or register allocation, but a direct comparison is difficult since the objective and constraints differ.

6.5 Schedule memoization

BMS-CnC is the first system that enables the reuse of inspector/executor results across application runs, but there is related work in the idea of schedule memoization [**Cui:2010**].

Schedule memoization is a technique based on symbolic execution to find the set of constraints that, applied to the input, are sufficient to match the input to a valid schedule and follow that exact schedule for all subsequent runs in order to make the schedule deterministic. On the other hand, schedule reuse does not limit execution to a single schedule, because our schedules are sets of constraints and not total ordering of synchronization operations. Schedule memoization can enforce either a total ordering of synchronization events or a total ordering of both synchronization and data accesses, but enforcing memory access order is expensive; BMS-CnC can cheaply enforce both, since data and synchronization are coupled, but at a coarser granularity which ensures overhead is low.

6.6 Reference counting

Get-counts are similar to a memory management technique called reference counting used in systems such as Microsoft's COM. COM (Common Object Model) is a standard designed to support the notion of distributed object on systems running different operating systems, programming languages or with different hardware. To enable programs written in different languages running on different systems to work together seamlessly, COM opts for automatic memory management which consists of a counter which could be attached to an object itself or to its factory which tracks the number of connections to the object. By inspecting the counter the system can know if the object is or is not in use. Reference counting is error prone because it relies on the programmer to increment or decrement the counter and to free the object memory when the count reaches zero. If a program terminates abruptly, it may leave orphan objects so objects ping their clients periodically, leading to increased network traffic. Certain languages offer automatic reference counters to ease the burden on the programmer.

In garbage collection systems, reference counting is also used to store the number of references to the object that are stored in other objects. The advantage of reference counting over tracing garbage collection is the fact that it is incremental, making it useful in real-time systems. The biggest disadvantages is that it requires additional techniques (such as weak references or a mark and sweep collector) to handle circular references.

6.7 Data distribution for distributed programs

Today, the most widely used programming approach for distributed systems is the combination of sequential C or Fortran with MPI [**MPI**]. MPI represents the *message passing* paradigm for managing communication which is a relatively low level approach which relies on calls to *send* and *receive* functions. It gains the freedom to express any data distribution, but requires the programmer to manage the intricate details of computation and communication and makes it difficult to overlap the two.

Global shared address space languages such as High Performance Fortran [**Fortran:93**] (HPF), Co-Array Fortran [**Coarrays**], Unified Parallel C [**El-Ghazawi:03**], and Titanium [**Titanium**] take a compiler-driven approach for controlling this aspect of execution. The programmer controlled the data distribution, while the task distribution was fixed, computed according to the *owner-computes rule*.

The HPF compiler targets an single program multiple data (SPMD) programming model in which each processor executes the same program, but operates on different data. Each processor allocates and operates on its own local portion of distributed arrays, according to the distributions, array sizes and number of processors as determined at runtime.

HPF data distributions can be specified through two different directives: `ALIGN` which maps from arrays to templates and `DISTRIBUTE` which maps from templates to processors, as shown below. The directive `!HPF$ PROCESSORS proc-name(dim1, ..., dimN))` declares an abstract processor array, with `proc-name` being the name of the abstract processors array and `dim1, ..., dimN` the size and shape of the array.

Aligning such an array with a target array is done through the `ALIGN` directive `!HPF$ ALIGN array WITH target`. A few examples of this directive are given below:

```
!HPF$ ALIGN A(I) WITH B(I)
!HPF$ ALIGN A(:) WITH B(I+2)
!HPF$ ALIGN A(:) WITH B(2*I)
!HPF$ ALIGN A(I,J) WITH B(J,I)
!HPF$ ALIGN A(:,*) WITH B(:)
```

The distribution of arrays onto a processor array is performed though `DISTRIBUTE` directives, where `list-of-arrays` are the arrays to be distributed and `proc-name` is the processor array: `!HPT$ DISTRIBUTE list-of-arrays ONTO proc-name`.

The separation of the two phases serves the purpose of simplifying the change of data distributions which may be needed during the performance tuning phase.

Co-array Fortran (CAF) [**Coarrays**] relies on co-arrays, a data structure that adds a dimension (*codimension*) to traditional arrays, each of the nodes used for distributed execution getting the array allocation. The codimension is used to access the memory space of remote nodes and enables easy performance modeling since it is the only parameter that identifies local versus remote accesses.

Unified Parallel C (UPC) [**UPC**] proposes a more general approach in which array declarations are assumed to be in a memory space partitioned among nodes if they are declared as *shared*. They are distributed cyclically by default and can be distributed block-cyclic by specifying a block size.

Titanium [**Titanium**] is an object-oriented language based on Java which supports an approach similar to UPC. Its support for distributed arrays is limited to allocating an array object in code which results in having an array per processor, which is similar to CAF.

Researchers have recognized the source of the difficulty of programming distributed models as coming from the manual management of data partitioning and communication and have proposed systems which attempt to automatically partition the data. The PARADIGM compiler [**Banerjee:95**] proposed a compiler-based approach. Anderson and Lam [**Anderson:93**] propose an iterative approach which decreases the communication requirements at each iteration. Wholey [**Wholey:92**] shows the importance of performance estimation in selecting a good distribution function. Garcia, Ayguade and Labarta [**Garcia:95**] propose the use of 0-1 integer programming to optimally solve the data distribution problem.

Our approach solves the data partitioning problem, but supports dynamic applications which means the approach is based on new runtime techniques and the performance models are quite different since accurate producer-consumer and controller-controllee relations are available.

Since the best distribution function for a data array may change depending on the phase of the computation, Palermo, Hodges and Banerjee [**Palermo:01**] extend the PARADIGM compiler with support for automatic data partitioning including support for data redistribution. Bixby, Kennedy and Kremer [**Bixby:94**] show that integer linear programming can be fast enough for compiler-based data partitioning with redistribution support.

In our approach, data redistribution does not need to be handled separately because of the dynamic single assignment rule - redistribution of each value can be implicitly performed with fine granularity (at each write operation).

Some parallel programming models offer control of data distribution at runtime rather than being compiler-driven.

The X10 [**Charles:05**] language proposes the notion of *places* as high-productivity [redacted] locality abstraction to which tasks can be assigned. Mapping of places to processors is done by the runtime, but accesses to remote data must be explicit.

Languages based on actors [**Hewitt:73**] such as Charm++ [**Kale:93**] are especially adept to distributed execution because they can handle systems with long latencies very well because of their message queue approach to concurrency. Messages are sent to the actor(‘called ‘chare” in Charm++) that handles them wherever that actor may live and actors are created using a dynamic load balancing strategy, so the programmer does not need to explicitly handle data distribution. Chares can migrate across processors [**Acun:14**] which is similar to the redistribution capabilities of compiler-driven approaches to data distribution.

The RAPID project proposes the use of I/E as an approach for automatic parallelization, included data distribution and computation partitioning for distributed systems. Since this work was published, the size of the distributed systems and of the application graphs executed on them has increased dramatically. We show that even with this increase, better amortization strategies keep I/E execution a valid approach for programming distributed systems. They show that I/E can be used for irregular applications; we add optimal approaches to solve combined data and task distribution problem. RAPID is based on a model that allows anti and output dependences, while we avoid them by using a dynamic single assignment model; RAPID users must specify estimate for task duration while we build automatic estimates through the normalization approach.

Chapter 7

Conclusions

In this dissertation we address challenges in writing parallel software that performs well on modern systems which are often programmed by expressing dynamic task graphs. The dynamic nature of these programs can prevent us from taking advantage of traditional compile-time techniques for program optimization, so we turn to runtime methods, such as inspector/executor, coupled with a programming model based on dataflow that enables our proposed optimizations.

We first target shared-memory machines where we propose folding - an efficient memory management approach for dataflow models. We then focus on the challenge of controlling the balance between the memory consumption of parallel programs and their runtime. Our bounded memory scheduling approach shows that the trade-off between these two essential resources - memory and processing power - can be efficiently controlled. We achieve this balance by combining an inspector/executor runtime with a dataflow programming model which enables separation of the input data from the computation structure. Our approach results in an improved method for reuse of inspector-executor results which enables the use of the technique for programs expressed as dynamic task graphs.

To control the trade-off of memory requirements on one hand and execution time on the other, we developed heuristic algorithms which are shown to allow excellent control of the trade-off at a very low computational cost. They are essential to making our technique work in real-life scenarios because without them, we fall back to an optimal, but slower integer linear programming approach.

Future work in the direction of bounded memory scheduling would be to extend the applicability of the technique towards supporting two common scheduling approaches: process time sharing systems and batch scheduling with multiple jobs assigned to nodes on the fly. To enable the use of our technique in these two scenarios, we need to explore how multiple such graphs can be overlapped in such a way as to maintain a bound on the total memory even if the starting of another graph is unknown at the time the first one starts.

Second, we target distributed systems, where the main challenge is scaling to the increased parallelism required for distributed execution. The key element for good scaling on these systems is data and task distribution, which is traditionally controlled by the programmer who identifies good distribution functions through theoretical analysis or empirical evaluation. We show that it is possible to automatically select distribution functions while getting performance that matches or surpasses that of hand-picked distributions. To achieve this, we turn to inspector/executor and the same programming model used for bounded memory scheduling, but because of the stronger performance constraints we had to develop more efficient techniques for amortizing the inspector cost. We showed that we can inspect execution on small inputs, but reuse the results on medium or large inputs and still obtain good performance.

Another avenue for amortizing the inspector cost is optimizing the inspector execution itself, especially since integer linear programming is an expensive approach to perform during inspection. We showed that the problem of finding the distribution cost which minimizes the execution cost can be formulated in a way that is polynomial in the number of nodes used for execution rather than with the input size, which is key to keeping inspection time in check.

In the future, an interesting extension would be to generate $2D$ block-cyclic distributions instead of checkered distributions. The advantage of using a checkered distribution is that, because of only using a single modulo operation instead of two, it has the advantage that it is more efficiently expressed in integer linear programming. However, the checkered and block-cyclic functions have different expressive power: both can express distributions that the other one cannot, so the experimental results included in this thesis are difficult to compare with those of state-of-the-art block-cyclic distributions. Another important improvement for the evaluation would be to compare against the most flexible distribution function possible - the one which has the freedom to place each individual item on any possible node. This function is not computationally feasible to compute optimally for large input sizes, but it would be useful to know how much performance is lost by choosing to restrict distribution functions to piecewise linear functions such as block-cyclic or checkered. Compared to the inspector time for bounded memory scheduling, the inspector time for distribution selection is higher because we do not include heuristics that could be applied instead of the optimal algorithms. It would be interesting to see if heuristics could achieve results similar to the optimal approaches proposed here.

To conclude, I want to highlight the essential role the programming model plays in enabling our optimizations. Much of the work presented here would not be possible by using any of the programming models that are commonly used to write parallel programs today and several key features of the model are essential to perform our optimizations. First, the dataflow nature of the model enables us to query implicit producer consumer and controller-controllee relations. Second, the separation between control data and input-output data allows for efficient inspection of programs which is essential to make the optimizations fast. Fourth, the dynamic single assignment rule for data improves the scheduling flexibility which leads to better results for the scheduling algorithms proposed. Finally, the tagged nature of the model enables the ILP model to recognize data balance and communication patterns and apply the distribution function to newly created task instances. While dataflow models have been proposed a long time ago and are accepted in the community for their advantages in parallel programming, they have not reached mainstream use on modern systems; this work shows that the use of dataflow models is essential for modern parallel programming in both the shared-memory and distributed memory usecases.