BMS-CnC: Bounded Memory Scheduling of Dynamic Task Graphs

Technical Report TR13-07, Department of Computer Science, Rice University, November 2013
Updated: August 2014

Dragoş Sbîrlea
Rice University
dragos@rice.edu

Zoran Budimlić
Rice University
zoran@rice.edu

Vivek Sarkar
Rice University
vsarkar@rice.edu

ABSTRACT

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

To address this problem we propose *bounded memory scheduling* (BMS) for parallel programs expressed as dynamic task graphs, in which an upper bound is imposed on the program's peak memory. Using the inspector/executor model, BMS tailors 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.

Since solving BMS is NP-hard, we propose an approach in which we first use our heuristic algorithm, and if it fails we fall back on a more expensive optimal approach which is sped up by the best-effort result of the heuristic.

Through evaluation on seven benchmarks, we show that BMS gracefully spans the spectrum between fully parallel and serial execution with decreasing memory bounds. Comparison with OpenMP shows that BMS-CnC can execute in 53% of the memory required by OpenMP while running at 90% (or more) of OpenMP's performance.

Keywords

task graphs; task scheduling; inspector/executor

1. INTRODUCTION

Multicore, with its increasing levels of parallelism, has arrived at a time when memory capacity has already stopped scaling [34]. Currently, memory per core is decreasing by 30% every two years [35] and projections state that it will soon drop to megabytes in extreme scale systems [37]. 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." [51]. But this problem is not only an obstacle for future supercomputers; for the embedded multicore processors, memory is already at a premium today.

Unfortunately, parallel execution is known to increase memory requirements compared to a serial baseline [9]. The community has been aware of this problem since 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" [13]. Without mitigation techniques, the increased memory consumption can lead to an increased occurrence of out-of-memory errors [20].

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 program will not run out of memory[1]. We believe this lack of control over peak memory usage stems from a more in-depth challenge: programming systems do not have access to the dependence structure (task communication and creation relationships) of a program.

In light of these problems, 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 [44] 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 knowl-

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

edge, 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 [28], 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 [44] system. Our parallel programming model (see Section 3) 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 paper 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.

2. BACKGROUND: THE CONCURRENT COLLECTIONS PROGRAMMING MODEL

The programming model used in this work is an extension of the Concurrent Collections (CnC) [12] 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*.

Once a step is spawned, it 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, which is performed through `item_collection.put(key, value)`. As long as the dynamic single assignment rule is respected and since `get`s are blocking [2], there are no data-races on items. Once steps read their input items, they perform computation and then may produce items (through `put` operations) and/or start new steps through `step_collection.spawn`.

CnC has been shown to offer especially good performance for applications with asynchronous medium-grained tasks [17], an attractive target for our BMS system.

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 [45].

3. 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

[2]With the BMS variant of CnC in Section 3, the logic behind gets remains the same, but since the task does not start until all input items are available, blocking is not needed anymore.

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. [3]

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 [46]), 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[4], the stack memory consumption is constant during execution.

3.1 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 [12], enabling BMS schedule reuse across multiple runs (Section 7).
- 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 [46]. 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.

[3]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 3.2 for a discussion.

[4]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.

3.2 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 the `step_collection.get_inputs(t)` call 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. [46].

4. BOUNDED MEMORY SCHEDULING

This section describes the computation graph used by the algorithm; Section 5 describes our heuristic BMS algorithm and Section 6 presents the optimal approach. Section 7 describes the technique for reusing the schedules, while

Section 8 describes improvements to memory management and bug finding that are enabled by the inspector/executor approach.

4.1 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[5]. 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 1.

5. 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 [28] which is well-known to be NP-Complete can be reduced to BMS[6]. 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 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 9 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).

Algorithm 1 The BMS Algorithm.

```
1: function BMS(G, M, α)
2:                                          ▷ G is the computation graph
3:                                        ▷ M is the desired memory bound
4:                ▷ α affects the task priority queue (see Section 5.1)
5:   noColors ← M/G.itemsize
6:   freeColors ← InitializeSet(noColors)
7:   freeTasks ← PriorityQueue(α)
8:   push(freeTasks, G.env)
9:   while freeTasks ≠ ∅ do
10:    crtTask ← Pop(freeTasks)
11:    for all crtItem ∈ ProducedItems(crtTask) do
12:      MarkAsProduced(crtItem)
13:      color ← Pop(freeColors, crtItem)
14:      if color = null then
15:        return false              ▷ Failed to find BMS schedule
16:      else
17:        prevItem ← GetStoredItem(color)
18:        for prev ∈ ConsumersOf(prevItem) do
19:          AddEdge(prev, crtTask)
20:        for all cTask ∈ ConsumersOf(prevItem) do
21:          MarkInputAvailable(cTask, crtItem)
22:          if ReadyToRun(cTask) then
23:            Push(freeTasks, cTask)
24:        SetStoredItem(color, crtItem)
25:    for all crtItem ∈ ConsumedItems(crtTask) do
26:      if UnexecutedConsumers(crtItem) == 0 then
27:        availableColor ← ColorOf(crtItem)
28:        freeColors ← freeColors ∪ availableColor
29:    for all spawn ∈ SpawnedTasks(crtTask) do
30:      MarkPrescribed(spawn)
31:  return true                       ▷ Found BMS schedule
32: end function
```

5.0.1 The algorithm

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-

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

[6]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.

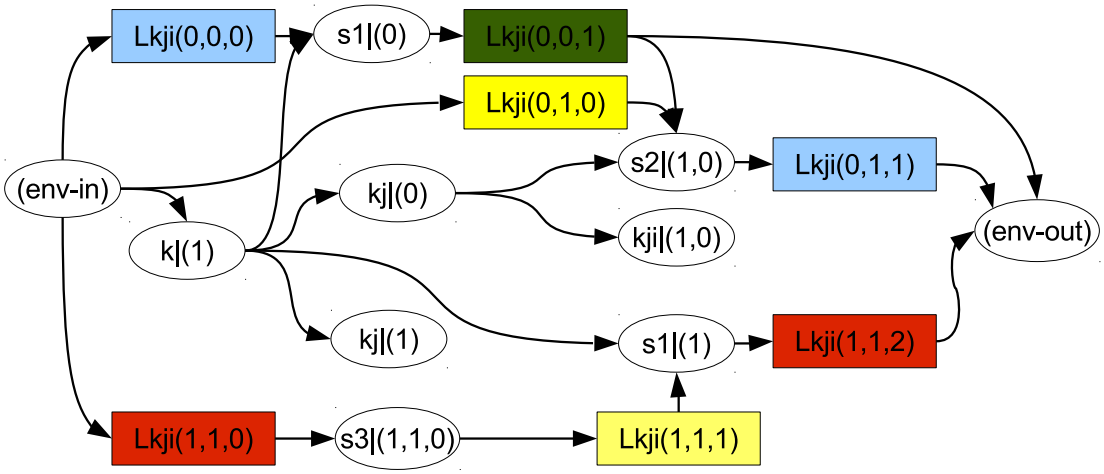


Figure 1: The BMS-CnC computation graph for Cholesky factorization tiled 2×2. Data items are represented as rectangles and Circles represent steps. Nodes are labeled with the collection name followed by the key or tag. Item colors are assigned by the BMS algorithm (Section 5).

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 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 [16, 11] with a worst-case space complexity of $\mathcal{O}(n^2)$, we opted for the more memory-efficient linear scan register allocator [43].

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.

5.1 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. [7] Because the default parallel sched-

[7] Depth-first and breath-first ordering of tasks are done on a graph where items are treated as direct edges from pro-

ule 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:   α ← 1
 3:   while α ≠ 0 do
 4:     success ← BMS(G, M, α)
 5:     if success then
 6:       return true
 7:     α ← α − Δα
 8:   return false
 9: end function
10:
11: // Used for the task priority queue:
12: function PRIORITYQUEUE.COMPARE( task1, task2)
13:   rank1 ← α × RANKBF(task1) + (1 − α) × RANKDF(task1)
14:   rank2 ← α × RANKBF(task2) + (1 − α) × RANKDF(task2)
15:   return rank1 − rank2
16: end function
```

5.2 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 [48] 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 5.1. *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.

```
 1: function POP(freeColors, crtItem)
 2:   producer ← GETPRODUCER(crtItem)
 3:   minColor ← null
 4:   minEdges ← MAX_INT
 5:   for color ∈ freeColors do
 6:     prevItem ← GETSTOREDITEM(color)
 7:     edges ← 0
 8:     for consum ∈ ConsumersOf(prevItem) do
 9:       if !ISPREDECESSOR(consum, producer) then
10:         if BFRANK(prod) ≥ BFRANK(consum) then
11:           edges ← edges + CONSUMERSCOUNT(prevItem)
12:         edges ← edges + 1
13:     if edges < minEdges then
14:       minEdges ← edges
15:       minColor ← color
16:   return minColor
17: end function
```

6. 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 6.1. Section 6.2 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 6.3.

ducer 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.

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 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 2: Constraints used in the ILP formulation.

We propose a disjunctive formulation with variables and constraints shown in Tables 1 and 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 10, 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.

6.1 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 [54], obtaining the following equations, in which M and N are big constants:

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

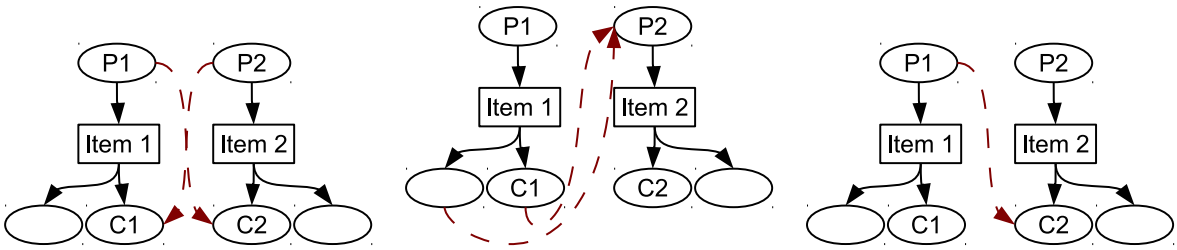
This set of constraints is correct, but inefficient, adding 4 variables and 5 constraints for each pair of items. Decreasing the number of constraints and variables added is essential for efficient execution. To do this, we analyze the possible relations of the lifetime of items as shown in Figure 2. 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 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.

6.2 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),



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

Figure 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.

and use the Strahler number[8] of the tree as a lower bound. For applications such as merge sort, this happens to be the minimum memory footprint of the application.

6.3 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 5. 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.

7. 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[9]. This root set contains two types of edges. First, it contains the edges whose tail 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.

[8]The Strahler number [23] is the minimum number of registers required to evaluate an expression tree and can be computed in linear time.

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

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 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 [32], 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.

8. OTHER FEATURES

8.1 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[10], so a classic garbage collector would not be able to collect them. The mechanism regularly used to collect items is *get-counts* [45], a parameter provided by the programmer that specifies the number times the item will be read. Identifying the 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 3) to collect objects which are still referenced, but will never be used (cleaning up the memory leaks).

8.2 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) [11]

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 [22] and add space overhead linear in the size of the application footprint and in the number of parallel threads [24]. 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.

9. 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.

9.1 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)[12]. 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 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 basic 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

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

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

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

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.

```
1: // This function assigns a (sub)color for item crtItem
2: function PopFromClass(freeColors, crtItem)
3:   crtClass ← GetClass(crtItem)
4:   freeSubcolors ← GetFreeColors(crtClass)
5:   color ← Pop(freeSubcolors, crtItem)
6:   if color = null then
7:     // Call Pop function from the BMS algorithm
8:     pageColor ← Pop(freeColors, crtItem)
9:     if pageColor ≠ null then
10:      prevIt ← GetStoredItem(pageColor)
11:      prevConsumers ← ConsumersOf(prevItem)
12:      noSubcolors ← rootSize/class.itemSize
13:      for i = 1 → noSubcolors do
14:        newColor ← newcolor()
15:        PushToClass(freeSubcolors, newColor)
16:        SetStoredItem(newColor, crtItem)
17:      color ← Pop(G, freeColorsInClass, crtItem)
18:  if color ≠ null then
19:    rootColor ← GetRootColor(color)
20:    rootColor.uses ← rootColor.uses + 1
21:  return color
22: end function
23: // This function reclaims a (sub)color
24: function AddFreeColorToClass(
        freeColors, itemColor)
25:  crtClass ← GetClass(itemColor)
26:  rootColor ← GetRootColor(color)
27:  rootColor.uses ← rootColor.uses − 1
28:  if rootColor.uses = 0 then
29:    AddFreeColor(freeColors, rootColor)
30:    itemsSet ← SubColorsOf(rootColor)
31:    SetStoredItem(rootColor, itemsSet)
32: end function
```

9.2 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 9.1) also handles prescribed tasks.

9.3 Bounding the inspector memory consumption

The BMS algorithm relies on the existence of the application graph which it queries through the ProducedItems, ConsumedItems and SpawnedTasks functions. Storing the whole computation graph in memory can be avoided by using two techniques: *on-demand expansion* of the graph and computational *epochs*. Combined, these two techniques allow us to only store a slice of the graph. Similarly to the BMS algorithm, we use a best-effort approach to bounding the inspector footprint: if the memory required to store the graph slice outweighs the memory bound imposed on the BMS execution, the algorithm aborts.

On-demand graph expansion is used when a `pop` is performed from the `freeTasks` list; at this point any items produced by the task must be colored. With on-demand expansion, the complete set of consumers for items may not be known (since the graph has not been completely expanded), but their number is (specified by get-count). The expansion stops when the number of consumers identified for the item is equal to the *get-count* of the item [13]

The *epoch* technique summarizes the portion of the graph that has already been inspected and colored (the portion between the environment and the items currently stored in colors), using a representation in which the environment produces all tasks in the *freeTasks* list and all currently live items, as well as all steps marked produced by BMS, but not executed. All dead items and all steps which have been symbolically executed are removed.

Creating an epoch after each on-demand expansion reduces the graph to a sliding window. The memory footprint of the inspector is computed on the fly during inspection as the sum of the color assignment table (which records the consumers of the item stored in each color) and the graph slice consisting of the stored edges and nodes. If after expansion, the inspector footprint is still larger then the memory bound, the algorithm aborts for that schedule.

[13]Programmer-specified get-counts are needed for on-demand expansion, but not for the BMS algorithm which instead automatically computes them.

10. EVALUATION

10.1 Implementation and experimental setup

The BMS-CnC system was implemented on top of Qt-CnC [46], an open-source CnC implementation[14] based on the Qthreads task library [52]. The evaluation was performed on an Intel Xeon E7330 system with 32GB RAM and 16 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 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 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.

10.2 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 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 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 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 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 3c) shows BMS enables a trade-off similar to the one in Smith-Waterman, between large memory consumption and high performance.

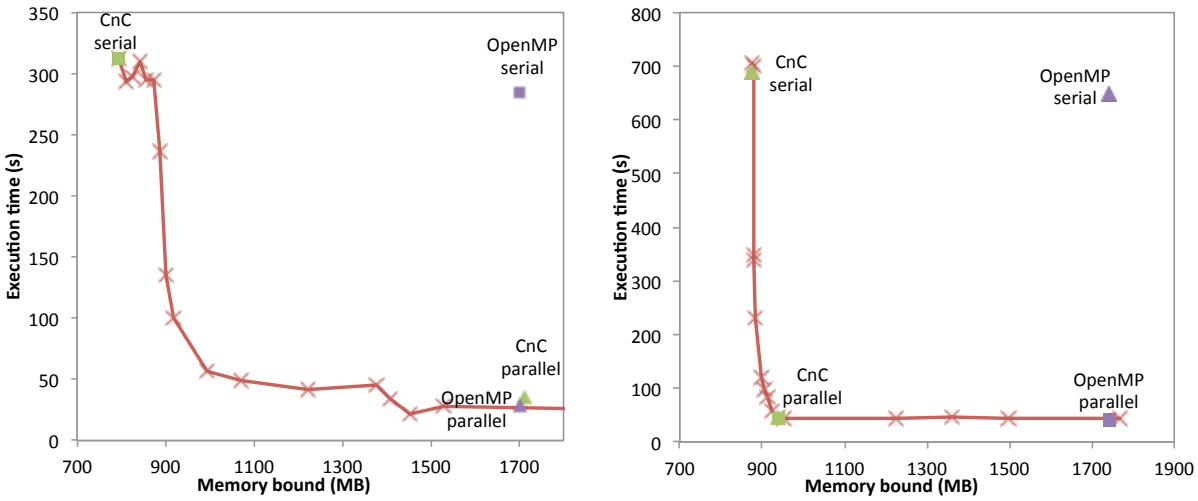
For **Gauss-Jordan elimination** (see Figure 3e), BMS-CnC is able to enforce a footprint 18% lower than the serial footprint of CnC, with minimal loss of parallelism. This is the result of the abundant parallelism, as well as good coloring heuristics.

For **MergeSort** (Figure 3d) 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** [49] provides a set of random task graphs from which we picked the largest (STG 59), shown in Figure 3g. 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 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.

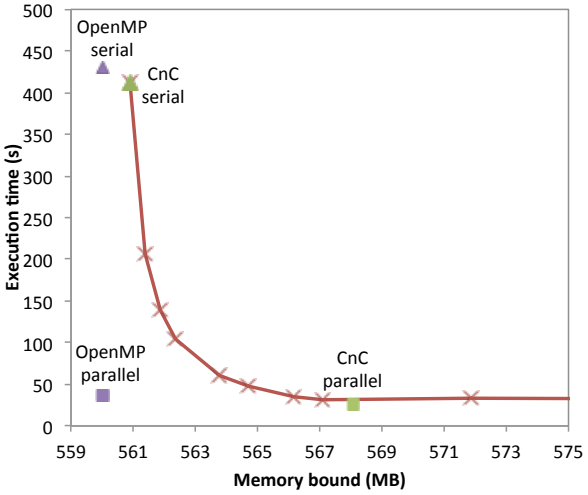
In summary, BMS shows the ability to control the trade-off between parallelism and memory usage. Furthermore, this trade-off is not linear — there is a "sweet spot" in the memory bound space where BMS enables most of the performance of the unbounded memory parallel execution with only a small increase in memory relative to the serial execution. To further illustrate this, Table 4, 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).

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

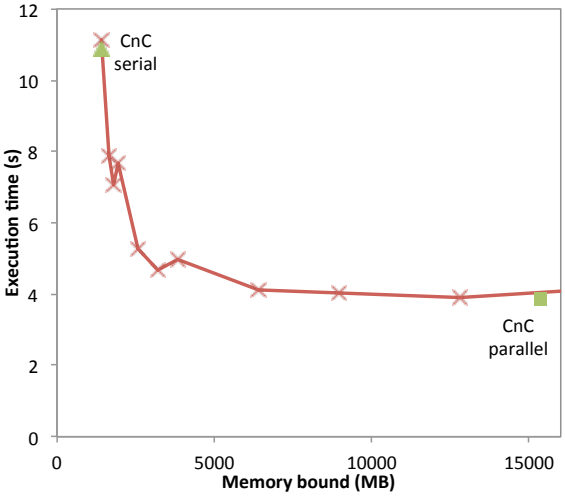


(a) Smith Waterman

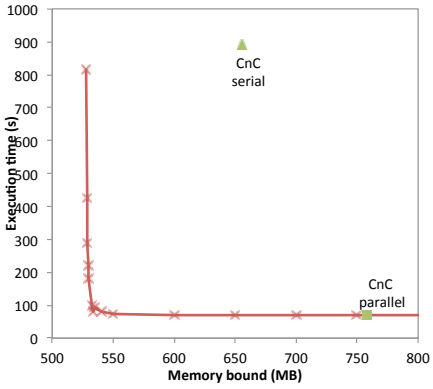
(b) Blackscholes



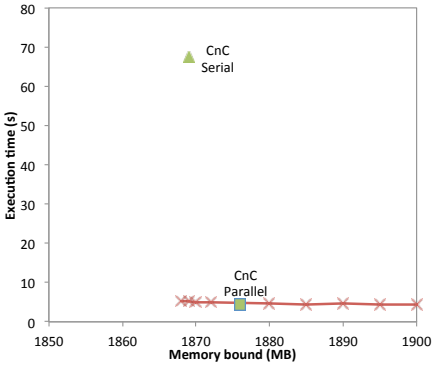
(c) Cholesky Factorization



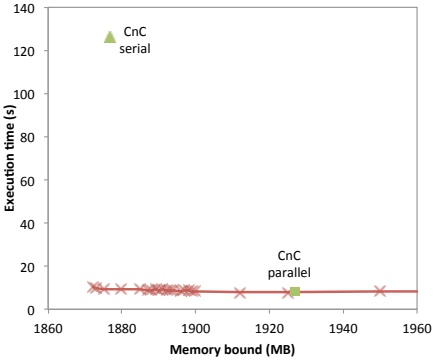
(d) Merge Sort



(e) Gauss-Jordan Elimination

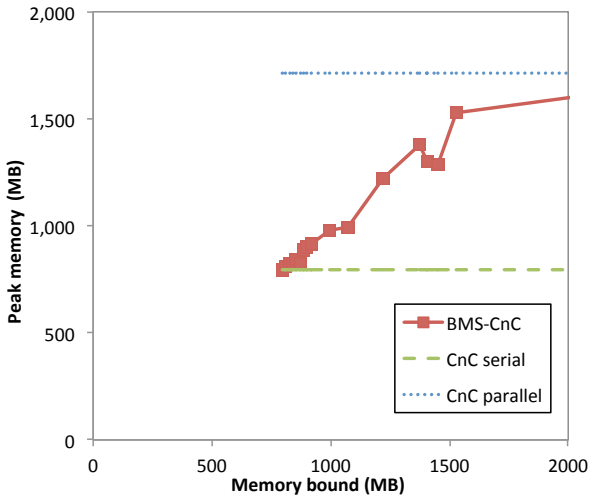


(f) Standard Task Graph (STG) 58

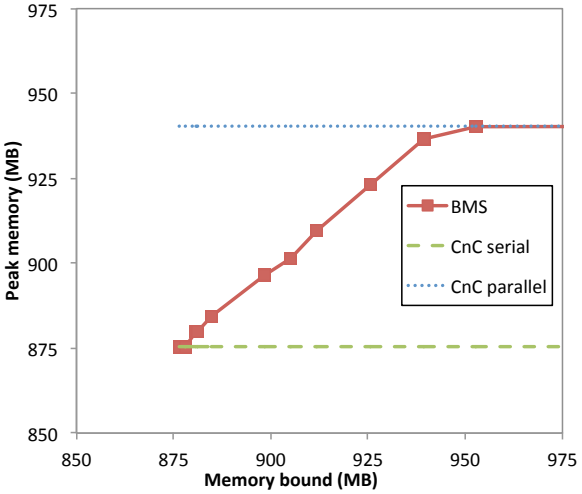


(g) Standard Task Graph (STG) 59

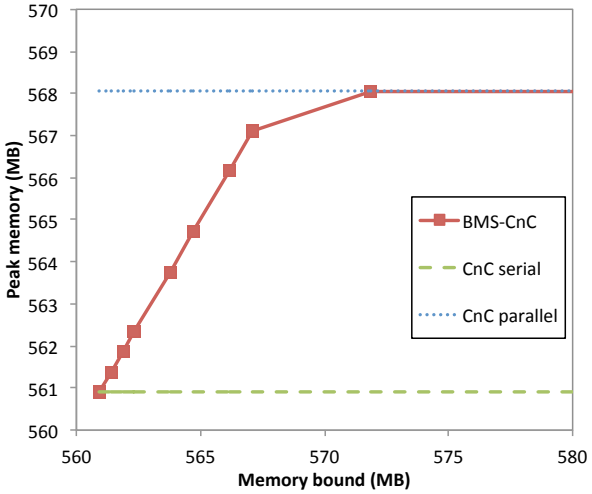
Figure 3: BMS-CnC executor run-time (the red line) as a function of memory bound for each of the benchmarks. BMS-CnC is able to enforce memory bounds down to the serial execution footprint and even lower for Gauss-Jordan and STG 58 and 59. OpenMP results included where available.



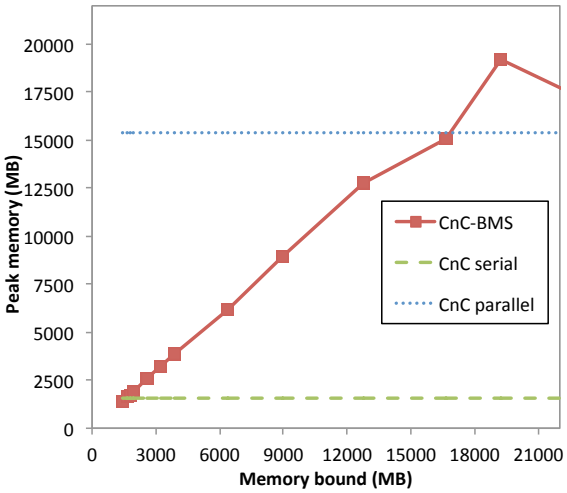
(a) Smith Waterman



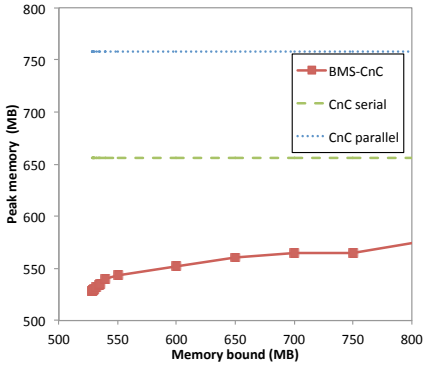
(b) Blackscholes



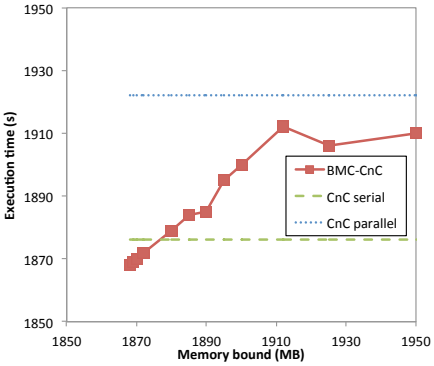
(c) Cholesky



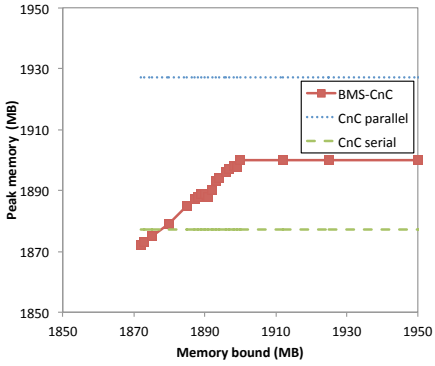
(d) Merge Sort



(e) Gauss-Jordan Elimination



(f) Standard Task Graph (STG) 58



(g) Standard Task Graph (STG) 59

Figure 4: Actual peak memory as a function of memory bound for each of the benchmarks.

Benchmark		Type	Graph vertices	Source	Input parameters	Schedule reuse conditions
Smith Waterman		biomedical	5002	in-house	2 sequences of 70000 length and tile size (2000 × 2000)	identical tile sizes identical sequence sizes
Blackscholes		financial	6730	Intel CnC [33] Parsec [6]	number of options (25.6M) and option data	identical number of options
Cholesky		dense algebra	41558	Intel CnC [33] Buttari [14]	input matrix (12000 × 12000) and tile size (125 × 125)	identical tile sizes identical matrix sizes
Gauss-Jordan		dense algebra	8450	Intel CnC [33]	input matrix (4096 × 4096) and tile size (256 × 256)	identical tile sizes identical matrix sizes
Merge Sort		recursive	3582	in-house	vector data (2^{25} integers)	identical input array sizes
STG	sparse	task graph	198	STG [49]	graph shape	identical graph shape
	fpppp	task graph	647	STG [49]	graph shape	identical graph shape
	58, 59	task graph	5402	STG [49]	graph shape	identical graph shape

Table 3: Benchmarks, their inputs, computation graph sizes and the schedule reuse conditions. The corresponding results are shown in Figure 3.

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: 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).

10.3 OpenMP comparison

OpenMP results have been included in Figure 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.

10.4 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 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.

Bench	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
Black scholes	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 5: The minimum memory with heuristic BMS and with ILP and the lower bounds fed to ILP. Cells are marked with * when ILP timeouts.

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

Bench	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 6: Performance evaluation of heuristic BMS and ILP with and without hot start. Even with hot start, the ILP approach cannot handle large graphs.

10.5 Runtime comparison of ILP and heuristic BMS

Table 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. [18] 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).

10.6 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 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. 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.

Benchmark	Graph creation(%)	BMS Algorithm Min(%)	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 7: Timing results for the inspector (graph creation and BMS scheduling), as percentages of the serial execution time.

10.7 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 5 include the BMS performance for 270, 280 and 310 tiles of the same size, and the graphs show interesting changes relative to Figure 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.

11. RELATED WORK

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

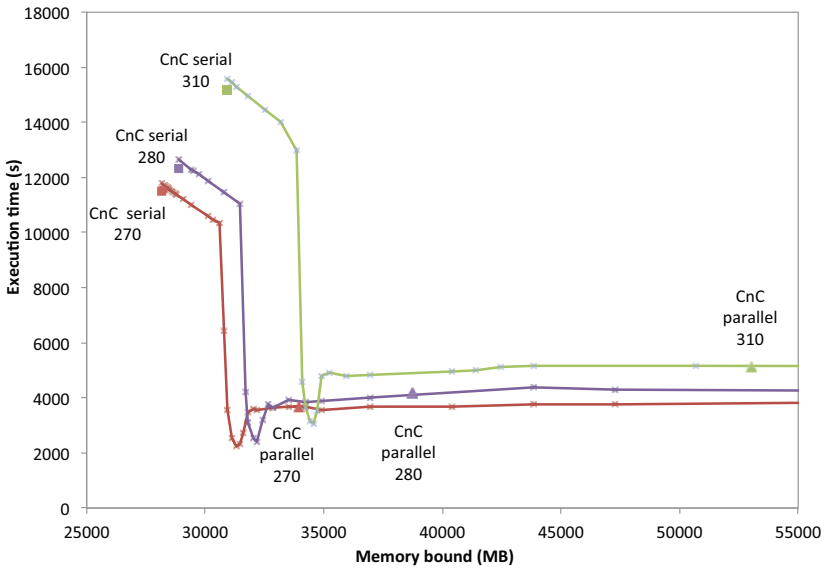


Figure 5: Smith-Waterman results on large inputs (270, 280 and 310 tiles). BMS enables the use of a sweet-spot with good performance and low footprint at the same time. The physical memory size is 32 GB and the computation graph for 310 input tiles has 192,202 nodes.

for parallel programs. Burton [13] was the first to propose bounds on the memory complexity of dynamically scheduled parallel computations. The asymptotic memory complexity obtained was $\mathcal{O}(s \times p)$, where s is the serial memory footprint and p is the number of processors. Simpson and Warren [47] present a survey of work in this area. Blelloch et al. [8], Narlikar and Blelloch [40], Blelloch et al. [7] and Fatourou [21] 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 [25] 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 [30, 10] analyze the memory consumption of serial programs. Other projects [30, 10, 31, 15] 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.

BMS is a novel application of the inspector/executor system proposed by Salz [44] 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 [26] 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 [27] inspector/executor is used to bound the memory assigned to copies of data in the distributed environment, but not the total footprint of a paral-

lel 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.

The BMS problem is related to the widely-studied problems of register sufficiency and combined instruction scheduling and register allocation. Barany and Krall [2] propose a type of code motion to decrease spills by using integer programming to identify the schedules that reduce overlapping lifetimes. Pinter [42] identified the fact that some variables in the program must have overlapping lifetimes while some don't need to which is an observation that e 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 [41] use the parallelizable interference graph and add data-dependence graph edges (similar to our serialization edges) to remove possible interference. loops. The CRISP project [39] 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. [1] 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. [29] 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 [5] compares various approaches for combined register allocation. Touati [50] proposes the use of serialization arcs to decrease the number of required registers.There are multiple related projects that apply optimal techniques [53, 3, 36, 4, 38]) for scheduling or register allocation, but a direct comparison is difficult since the objective and constraints differ.

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 [19]. When ensuring program correctness through symbolic execution, the principle of schedule memoization is the following: because the schedule space is very large, it may be intractable to test all schedules for correctness. Instead, a few schedules can be sampled, checked for correctness, memoized and then subsequent executions can be forced to follow one of them.

Schedule memoization relies on symbolic execution to find the set of constraints that, applied to the input, are sufficient to match it to a 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.

12. CONCLUSIONS

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

References

[1] Wolfgang Ambrosch et al. "Dependence-Conscious Global Register Allocation". In: PLSA. 1994. ISBN: 3-540-57840-4.

[2] Gergö Barany and Andreas Krall. "Optimal and heuristic global code motion for minimal spilling". In: CC. 2013.

[3] Andrzej Bednarski and Christoph Kessler. "VLIW Code Generation with Integer Programming". In: EuroPar '06.

[4] Peter van Beek and Kent D. Wilken. "Fast Optimal Instruction Scheduling for Single-Issue Processors with Arbitrary Latencies". In: CP '01, pp. 625–639. ISBN: 3-540-42863-1.

[5] David A. Berson et al. "Integrated Instruction Scheduling and Register Allocation Techniques". In: LCPC. 1999.

[6] Christian Bienia. "Benchmarking Modern Multiprocessors". PhD thesis. Princeton University, Jan. 2011.

[7] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. "Provably Efficient Scheduling for Languages with Fine-grained Parallelism". In: *J. ACM* (1999).

[8] Guy Blelloch et al. "Space-efficient scheduling of parallelism with synchronization variables". In: SPAA '97.

[9] Robert D. Blumofe and Charles E. Leiserson. "Scheduling multithreaded computations by work stealing". In: *J. ACM* (1999).

[10] Víctor Braberman et al. "Parametric prediction of heap memory requirements". In: ISMM '08. 2008.

[11] Preston Briggs, Keith D. Cooper, and Linda Torczon. "Improvements to graph coloring register allocation". In: *ACM Trans. Program. Lang. Syst.* ().

[12] Zoran Budimlić et al. "Concurrent Collections". In: *Scientific Programming* (2010).

[13] F. Warren Burton. "Guaranteeing Good Memory Bounds for Parallel Programs". In: *IEEE Trans. Softw. Eng.* (1996).

[14] Alfredo Buttari et al. "A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures". In: *Parallel Comput.* 35.1 (Jan. 2009), pp. 38–53. ISSN: 0167-8191.

[15] Brian Campbell. "Amortised Memory Analysis Using the Depth of Data Structures". In: ESOP '09.

[16] G. J. Chaitin. "Register allocation & spilling via graph coloring". In: *SIGPLAN Not.* (1982).

[17] A. Chandramowlishwaran, K. Knobe, and R. Vuduc. "Performance evaluation of concurrent collections on high-performance multicore computing systems". In: IPDPS. 2010.

[18] Chia-Ming Chang et al. "Using ILP for instruction scheduling and register allocation in multi-issue processors". In: *Computers and Mathematics with Applications* 34.9 (1997).

[19] Heming Cui et al. "Stable Deterministic Multithreading Through Schedule Memoization". In: OSDI'10.

[20] I. Dooley et al. "A study of memory-aware scheduling in message driven parallel programs". In: HiPC. 2010.

[21] Panagiota Fatourou. "Low-contention depth-first scheduling of parallel computations with write-once synchronization variables". In: SPAA '01.

[22] Mingdong Feng and Charles Leiserson. "Efficient detection of determinacy races in Cilk programs". In: SPAA. 1997.

[23] P. Flajolet, J.C. Raoult, and J. Vuillemin. "The number of registers required for evaluating arithmetic expressions". In: *Theoretical Computer Science* 9.1 (1979).

[24] Cormac Flanagan and Stephen N. Freund. "FastTrack: efficient and precise dynamic race detection". In: PLDI. 2009.

[25] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. "The implementation of the Cilk-5 multithreaded language". In: PLDI '98.

[26] Cong Fu and Tao Yang. "Run-time compilation for parallel sparse matrix computations". In: ICS '96.

[27] Cong Fu and Tao Yang. "Space and time efficient execution of parallel irregular computations". In: PPOPP. 1997.

[28] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. 1979. ISBN: 0716710447.

[29] R. Govindarajan et al. "Minimum Register Instruction Sequencing to Reduce Register Spills in Out-of-Order Issue Superscalar Architectures". In: *IEEE Transactions on Computers* (2003).

[30] Martin Hofmann and Steffen Jost. "Static prediction of heap usage for first-order functional programs". In: POPL. 2003.

[31] Martin Hofmann and Dulma Rodriguez. "Efficient type-checking for amortised heap-space analysis". In: CSL. 2009.

[32] *The HSL Library. A collection of Fortran codes for large scale scientific computation.* URL: `http://www.hsl.rl.ac.uk` (visited on 06/25/2014).

[33] *Intel Concurrent Collections*. URL: `http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc` (visited on 06/25/2014).

[34] Randy Isaac. "Influence of Technology Directions on System Architecture". In: PACT 2001 Keynote.

[35] Kevin Lim et al. "Disaggregated Memory for Expansion and Sharing in Blade Servers". In: ISCA '09.

[36] Castañeda Lozano et al. "Constraint-Based Register Allocation and Instruction Scheduling". In: CP'12.

[37] Yin Lu et al. "Memory-conscious Collective I/O for Extreme Scale HPC Systems". In: ROSS '13.

[38] Abid M. Malik, Jim Mcinnes, and Peter van Beek. "Optimal basic block instruction scheduling for multiple-issue processors using constraint programming". In: *IJAITS* (2008).

[39] Rajeev Motwani et al. *Combining Register Allocation and Instruction Scheduling*. Tech. rep. Stanford, CA, USA, 1995.

[40] Girija J. Narlikar and Guy E. Blelloch. "Space-efficient scheduling of nested parallelism". In: *TOPLAS* (1999).

[41] C. Norris and L.L. Pollock. "A scheduler-sensitive global register allocator". In: SC. 1993.

[42] Shlomit S. Pinter. "Register allocation with instruction scheduling". In: PLDI. 1993.

[43] Massimiliano Poletto and Vivek Sarkar. "Linear scan register allocation". In: *ACM Trans. Program. Lang. Syst.* ().

[44] Joel H. Salz et al. "Run-Time Parallelization and Scheduling of Loops". In: *IEEE Trans. Comput.* (1991).

[45] Dragoş Sbîrlea, Kathleen Knobe, and Vivek Sarkar. "Folding of tagged single assignment values for memory-efficient parallelism". In: Euro-Par'12.

[46] Dragoş Sbîrlea et al. "The Flexible Preconditions model for Macro-Dataflow Execution". In: *DFM*. 2013.

[47] David J. Simpson and F. Warren Burton. "Space Efficient Execution of Deterministic Parallel Programs". In: *IEEE Trans. Softw. Eng.* (1999).

[48] Daniel Spoonhower et al. "Tight Bounds on Work-stealing Overheads for Parallel Futures". In: SPAA '09.

[49] Takao Tobita and Hironori Kasahara. "A standard task graph set for fair evaluation of multiprocessor scheduling algorithms". In: *Journal of Scheduling* (2002).

[50] Sid Ahmed Ali Touati. "Register Saturation in Superscalar and VLIW Codes". In: CC '01.

[51] David Turek. "The Strategic Future: The Push to Exascale". In: *IBM Science and Innovation Summit*. 2009.

[52] Kyle Wheeler et al. "Qthreads: An API for programming with millions of lightweight threads". In: IPDPS. 2008.

[53] Kent Wilken et al. "Optimal Instruction Scheduling Using Integer Programming". In: PLDI '00.

[54] H.P. Williams. *Model Building in Mathematical Programming*. Wiley, 2013. ISBN: 9781118506189.

APPENDIX
A. UNOPTIMIZED ILP FORMULATION FOR THE MEMORY MINIMIZATION PROBLEM

Table 8 shows all the variables used in the unoptimized formulation of the problem and Table 9 shows the constraints.

A.1 Sequential schedule

For a sequential schedule, we need the contraint that no two tasks execute in the same cycle. This constraint will be dropped for the final ILP, but is very helpful in debugging, and may rarely lead to faster solve time.

$\forall task1$ and $\forall task2$ we want $issue^{task1} \neq issue^{task2}$
To express this in ILP, let *indicator1*, and *indicator2* be new binary variables. Then:

$$\begin{cases} issue^{task1} - issue^{task2} - MAX \times indicator1 \leq -1; \\ issue^{task2} - issue^{task1} - MAX \times indicator2 \leq -1; \\ indicator1 + indicator2 \leq 1; \end{cases}$$

A.2 Definition of item birth

Each item has a corresponding variable whose value is the time when the item is created. When *item* is produced by task *producer*, then:

- $birth^{item} = issue^{producer}$

A.3 Definition of item death

Each item has a variable that records its time of death - the time after all its consumers have executed and the memory can safely be reclaimed. Thus, $\forall consumer$ task that reads item *it*, then:

- $death^{it} \geq issue^{consumer}$

A.4 Data dependence

The schedule constraints are data dependences only (CnC does not use synchronization edges other than data dependence). Thus, $\forall producer$ and $\forall consumer$ with ($producer \rightarrow consumer$) then:

- $issue^{producer} < issue^{consumer}$

A.5 Color assignment

What this constraint expresses is that if two items are assigned to the same color, then one must die before the other one is created or the other way around.

- if ($register^{it1} == register^{it2}$) then
  $death^{it1} < birth^{it2} || death^{it2} < birth^{it1}$

A.6 Define max_bandwidth

The maximum bandwidth used by the schedule should be larger than the largest value of any color used. Thus, $\forall it \in items$:

- $max\_bandwidth \geq register^{it}$

Variable	Possible values	Number	Meaning
issue^{task}	integer, 1..*NO_CYCLES*	One per task	In which cycle does task *task* issue?
birth^{item}	integer, 1..*NO_CYCLES*	One per item	In which cycle does item *item* get produced?
death^{item}	integer, 1..*NO_CYCLES*	One per item	In which cycle can item *item* be collected?
indicators group 1	binary	$2 \times NO\_TASKS^2/2$	Auxiliary variables.
register^{item}	integer, 1..*NO_ITEMS*	One per item	To which register is item *item* assigned?
indicators group 2	binary	$4 \times NO\_ITEMS^2/2$	Auxiliary variables.

Table 8: Types of variables used in the ILP formulation, along with their description.

Constraint name	Number of constraints
No two tasks execute in the same cycle	$NO\_TASKS^2/2$
Definition of birth	NO_ITEMS
Definition of death	NO_GETS
Data dependence	NO_GETS
Color assignment	$5 \times NO\_ITEMS^2/2$
Max_bandwidth	NO_ITEMS

Table 9: Types of constraints used in the ILP formulation, along with their description.