Bounded memory scheduling of dynamic task graphs

Dragoș Sbirlea
Zoran Budimlić
Vivek Sarkar
“Capacity increases will continue to slow down .... **DRAM may stop scaling** before the end of this decade.” [ZC13]
The end of memory scaling

"Our ability to .... calculate on data is growing faster than our ability to access .... that data.”

[DT05]
Memory per core decreases by 30% every two years.

Source: [LCM09]
“This projection indicates that the average memory per core drops to megabytes in extreme scale systems.”

[LKT12]
Memory requirements of parallel programs

“The amount of memory required by a parallel program may be **spectacularly larger** than ... an equivalent sequential program.”

“Parallel memory requirements **may be both large** (relative to memory requirements of an equivalent sequential program) and **unpredictable**.”

[WB96]  
[WB96]
Motivation

Finding memory-efficient ways of executing parallel programs will be even more important in the future.
• **Tasks** are computational units
  – can create other tasks.
  – can allocate memory (data items)

• **The scheduler** is the component that
  – assigns tasks to cores
  – decides task execution order
Claim

Task schedulers should consider the footprint implications of the dynamic schedules they create.

Getting out-of-memory errors should not be a game of chance!
Advantages of memory-aware scheduling

• State of the art schedulers do not adapt to available memory limitations.

• This would enable us to:
  – Avoid out of memory errors.
  – Limit disk swapping, by using smaller footprints.
  – Run programs of larger input sizes.
What do we need?

• Information is needed about the task and data structure of the program
  – How many tasks?
  – Who spawns who?
  – Who produces what data?
  – What is the data size?
  – Who consumes what data?
The problem with current task schedulers

• Current task schedulers cannot inform about tasks at runtime until they run the tasks... which is too late for bounding the memory consumption.

• In general, static program analysis cannot find this information.

We need an oracle.
Our solution:
The use of inspector/executor

• The solution is executing programs in two stages:

1. **Inspector:**
   Query the computation for its runtime memory behavior.

2. **Executor:**
   Based on the inspector, execute the program
Bounded memory scheduling (BMS) problem

Given a program $P$ with input $I$ and a memory bound $M$, find a set of task ordering relations $TO$, such that every schedule that is legal for $P$ and also respects $TO$ has to fit in $M$.

• Goal: Keep the schedule dynamic
  – The frequent case should be when $TO$ is small, in which case the BMS schedules should be the same set of schedules that traditional schedulers (work stealing) offer
The inspector

• For each step that would run (starting with those prescribed by environment):
  – Call BMS-CnC function that returns the tags of its input and output items.
  – Call BMS-CnC function that returns steps spawned.

• **Build the dynamic computation graph**
  This step unfolds of the computation

• **Apply a *bounded memory scheduling algorithm***
  This finds *TO* (the set of task ordering relations)
• Our programming model creates the dynamic computation graph without running the computation itself.

• We add functions mapping tags to:
  • Input items
  • Output items
  • Prescribed tasks

• Only supports applications without data-dependent or get or put operations
BMS algorithm

• Simplifications:
  – Only items consume memory.
  – Items are a fixed size ITEM_SIZE.

\[
|colors| = \frac{memory}{item\_size}
\]
BMS algorithm: Task ordering edges

• Task ordering edges ensure only the schedules that fit the memory bound are valid.

• Iff. A and B have the same color, we add ordering edges:
  – From the consumers of A
  – To the producer of B

• These edges enforce non-overlapping lifetimes for A and B.
BMS algorithm

Input:
- Computation graph
- Number of colors

Output:
- Set of task ordering edges

Schedule fits the memory bound.

Build topological task order

Task list empty?

Pop task

Not enough colors

Assign colors to produced items

Colors assigned

Add ordering edges for the current task

If unused, reclaim color of consumed items
BMS algorithm: Coloring heuristic

- Ordering edges
  - transitive
  - serialization

- Goal: Minimize number of serialization edges
BMS algorithm: Example coloring

7 colors:

6 colors:

Fewer colors means less parallelism.
BMS example

0. Topologically sort the tasks.

Memory: 384 MB
Item size: 64 MB
# Colors: 6
BMS example

1. Assign color to output item $\text{data}(0,0)$. 
2. Assign **color** to output item \( \text{data}(1,0) \).
3. Assign **color** to output item $\text{data} | (1,0)$.
BMS example

4. Free color of item data\( (0,0) \).
5. Assign color to output item data\( (2,0) \).
6. Assign color to output item data|(2,1).
BMS example

7. Free color of item \texttt{data}(1,0).
8. Assign color to output item \texttt{data\(\mid \)2,2).
BMS example

9. Add ordering edges for \text{data}(2,3).
BMS example

Repeat until all items have been colored.
Picking a topological sort

• The success of applying the basic BMS algorithm depends on the choice of topological sort.

• **Problem:** Find a colorable topological sort.

• **Solution:** Sample topological sort orders until the BMS algorithm succeeds.
  – Sampling based on changing task rank between breadth-first and depth-first.

\[
\text{RANK}(\alpha) = \alpha \times \text{BF}_\text{RANK} + (1-\alpha) \times \text{DF}_\text{RANK}, \quad \alpha \in [0,1]
\]
BMS Schedule reuse

• The inspector cost is large...
  – Exploring the graph
  – Building the graph
  – Running the BMS algorithm

• ... so it has to be amortized
  – Traditional inspector-executor is used for loops, so the inspector cost is amortized across iterations.
  – Proposal: schedule caching.
BMS Schedule caching

• We use a root set of edges that identifies the graph completely:
  – Edges \((\text{env-in}) \rightarrow (X)\)
  – Edges \((X) \rightarrow (\text{env-out})\)
  – Root node sizes

• Runs with the same root set avoid:
  – building the computation graph
  – applying the BMS algorithm
Experimental setup

• 16 core Intel Xeon with 32GB RAM

• Using the Qthreads CnC runtime [SSW13]

• Benchmarks
  – Blackscholes
  – Cholesky
  – Matrix Inverse
  – Merge Sort
  – Standard Task Graph 58
  – Standard Task Graph 59
Results for Blackscholes

Figure: BMS-CnC executor time as a function of memory bound for Blackscholes (input 25.6M)
Figure: BMS-CnC executor time as a function of memory bound for STG Graph 59
Extensions

• Support items of different sizes
• Bound for task stack size
• Bound for the inspector memory
Take-away

• The inspector-executor model can be used for scheduling of task graphs.
  – Enabled by efficient schedule caching across runs.

• Our bounded memory scheduling algorithm enforces a user-specified bound on peak memory usage.
References


Backup slides
Parallelism and memory

• Task scheduling traditionally emphasizes parallelism (load balancing, reducing overhead, reducing contention)
  – Chunking
  – Guided self scheduling
  – Work sharing
  – Work stealing

• State of the art bounds for parallel dynamic scheduling do not consider the available memory:
  – $S_1 + O(C \times D \times P)$ [1]
    • $S_1$ – serial execution memory usage
    • $C$ – constant factor (user-adjustable)
    • $D$ – computation depth
    • $P$ – number of processors
  – $S_1 \times P$ [2]

[1] Narlikar & Blelloch “Space efficient scheduling of nested parallelism”
[2] Blumofe & Leiserson “Scheduling multithreaded computations by work stealing”
Other uses for the inspector-executor CnC system

- Garbage collection (used for BMS)
  - Computes get-counts automatically
    - No measurable overhead
- Visualization
- Synchronization checking:
  - Single assignment violations
  - Useless steps/items
  - Items that are never produced
  - Steps that never complete
Cost analysis

• Assuming $O(1)$ color assignment
  – The basic BMS algorithm:
    • Single pass over schedule – $O(|\text{edges}|)$
  – Successive schedule relaxation
    • Up to $\alpha_{\text{steps}}$ times the cost of basic BMS

• With color assignment heuristic
  – For each item, need to find the color that adds the minimum number of constraints – $O(|\text{colors}|)$
  – To find the number of constraints, we need to maintain a list of all predecessors for each step:
    • For k-th step, this can cost up to $O(k)$, for a total of $O(|\text{edges}|^2)$

• Total of $O(|\text{edges}|^3 \times |\text{colors}| \times \alpha_{\text{steps}})$
CPU and memory cycle trends