Extending the Nested Parallel Model to the Nested Dataflow Model with Provably Efficient Schedulers

David Dinh
Computer Science Division
Univ. of California, Berkeley,
Berkeley, CA 94720, USA
dinh@cs.berkeley.edu

Harsha Vardhan Simhadri
Computer Science Dept.,
Lawrence Berkeley National Lab.,
Berkeley, CA 94720, USA
harshas@lbl.gov

Yuan Tang [*]
School of Software, Fudan Univ.,
Shanghai Key Lab. of Intelligent
Information Processing,
Shanghai 200433, P. R. China
yuantang@fudan.edu.cn

ABSTRACT

The nested parallel (a.k.a. fork-join) model is widely used for writing parallel programs. However, the two composition constructs, i.e. "$\|$" (parallel) and "$;$" (serial), that comprise the nested-parallel model are insufficient in expressing "partial dependencies" in a program. We propose a new dataflow composition construct "$\leadsto$" to express partial dependencies in algorithms in a processor- and cache-oblivious way, thus extending the Nested Parallel (NP) model to the *Nested Dataflow* (ND) model. We redesign several divide-and-conquer algorithms ranging from dense linear algebra to dynamic-programming in the ND model and prove that they all have optimal span while retaining optimal cache complexity. We propose the design of runtime schedulers that map ND programs to multicore processors with multiple levels of possibly shared caches (i.e, Parallel Memory Hierarchies) and prove guarantees on their ability to balance nodes across processors and preserve locality. For this, we adapt space-bounded (SB) schedulers for the ND model. We show that our algorithms have increased "parallelizability" in the ND model, and that SB schedulers can use the extra parallelizability to achieve asymptotically optimal bounds on cache misses and running time on a greater number of processors than in the NP model. The running time for the algorithms in this paper is $O\left(\frac{\sum_{i=0}^{h-1} Q^*(\mathrm{t};\sigma\cdot M_i)\cdot C_i}{p}\right)$ on a p-processor machine, where Q^* is the parallel cache complexity of task t, C_i is the cost of cache miss at level-i cache which is of size M_i, and $\sigma \in (0,1)$ is a constant.

CCS Concepts

•**Software and its engineering** $\rightarrow$ **Parallel programming languages;** •**Theory of computation** $\rightarrow$ *Shared memory algorithms; Control primitives;*

Keywords

Parallel Programming Models, Fork-Join, Data-Flow, Nested Parallelism, Space-Bounded Scheduler, Cache-Oblivious Algorithms, Cache-Oblivious Wavefront, Numerical Algorithms, Dynamic Programming, Shared-memory multicore processors.

[*]All the coauthors contributed equally to this paper. Yuan Tang is the corresponding author. Part of the work was done when the author was a visiting scientist at MIT CSAIL.

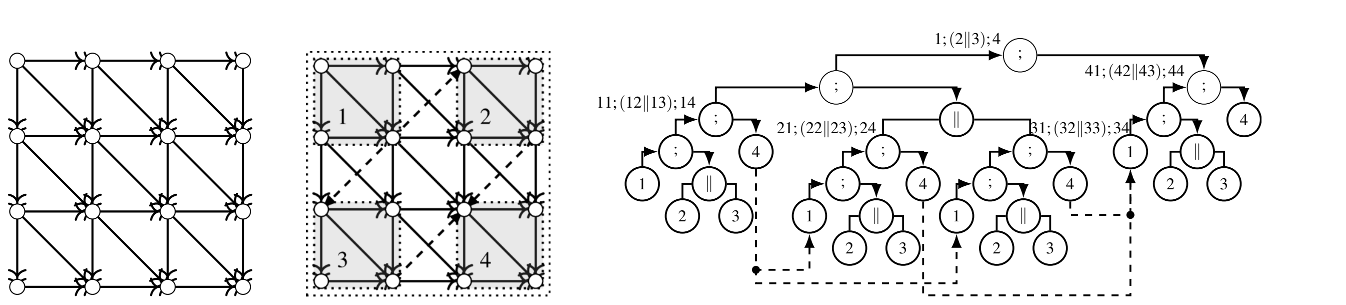
SPAA '16, July 11 - 13, 2016, Pacific Grove, CA, USA
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-4210-0/16/07…$15.00
DOI: http://dx.doi.org/10.1145/2935764.2935797

1. INTRODUCTION

A parallel algorithm can be represented by a directed acyclic graph (DAG) that contains only ***data dependencies***, without reference to the control dependencies induced by any particular programming model. We call this the ***algorithm DAG***. In an algorithm DAG, each vertex represents a piece of computation without any parallel constructs and each directed edge represents a data dependency from its source to the sink vertex. For example, Figure 1a is the algorithm DAG of the dynamic programming algorithm for the Longest Common Subsequence (LCS) problem. This DAG is a 2D array of vertices labeled $X(i,j)$, where the values with coordinates $i=0$ or $j=0$ are given. For all $i,j>0$, vertex $X(i,j)$ depends on vertices $X(i-1,j-1), X(i,j-1)$ and $X(i-1,j)$. In an algorithm DAG, there are two possible relations between any pair of vertices x and y. If there is a path from x to y or from y to x, one of them must be executed before the other, i.e. they have to be serialized; otherwise, the two vertices can run concurrently.

It is often tedious to specify the algorithm DAG by listing individual vertices and edges, and in many cases the DAG is not fully known until the computation has finished. Therefore, higher level programming models are used to provide a description of a possibly dynamic DAG. One such model is the ***nested parallel programming model*** (a.k.a. the fork-join model), in which DAGs can be described via ***spawn trees***: recursive compositions based on two constructs, "$\|$" ("parallel") and "$;$" ("serial"). The internal nodes of the spawn tree are serial and parallel composition constructs while the leaves are strands — segments of serial code that contain no function calls, returns, or *spawn* operations. The notation $a\ ;\ b$ is infix shorthand for a subtree of the spawn tree rooted at a node with a "$;$" construct with left child a and right child b; this indicates that b has a dependence on a and cannot start until a finishes. Likewise, $a \,\|\, b$ is an analogous construct that indicates that a and b can run concurrently. Expressing algorithms recursively in the NP model exposes their locality at various scales, enabling good schedulers to reduce the communication costs of their execution on a variety of machine configurations [1, 12, 11].

For instance, one might express the LCS algorithm in the NP model by decomposing the 2D array of vertices in the algorithm DAG into four smaller blocks, recursively solving the smaller instances of the LCS algorithms on these blocks, and composing them by specifying the dependencies between them using $;$ or $\|$ constructs. Figure 1 illustrates the resulting spawn tree up to two levels of recursion. For correctness, the NP model demands a serial composition between two subtrees of the spawn tree even if there is a ***partial dependency*** (or equivalently, ***partial parallelism***) between them: that is, a subset of vertices in the DAG corresponding to one of the subtrees depends on a subset of vertices corresponding to the



(a) Algorithm DAG (b) Divide-and-Conquer. (c) Spawn tree in the NP model

Figure 1: Algorithm DAG and the spawn tree of the LCS algorithm in the NP model. The labels $1,2,3,4$ correspond to the four quadrants in the recursive decomposition of the dynamic programming table. The leaves of the spawn tree are smaller LCS tasks while the internal nodes are composition constructs. Solid arrows in the algorithms DAG represents data dependencies in the algorithm DAG and control dependencies implied by the NP model in the spawn tree. The dashed arrows represent artificial dependencies induced by the NP model.

other. As a result, while the spawn tree in the NP programming model can accurately retain the data dependencies of the algorithm DAG, it also introduces many ***artificial dependencies*** that are not necessary to maintain algorithm correctness. Artificial dependencies induced by the NP programming model between subtrees of the spawn tree in Figure 1 are shown overlaid by dashed arrows onto the algorithm DAG in Figure 1b. Many parallel algorithms, including dynamic programming and direct numerical algorithms, incur an asymptotic increase in span due to the artificial dependencies introduced by the NP programming model. For example, the algorithm DAG for the LCS problem has $O(n)$ span. When expressed as a spawn tree in the NP model, its span follows the recurrence $T_\infty(n) = 3T_\infty(n/2)$, i.e., $T_\infty(n) = O(n^{1.5})$, which indicates a significant reduction in parallelism. *The insufficiency of the NP programming model in expressing partial dependencies in a spawn tree* is the fundamental reason that causes artificial dependencies between subtrees of the spawn tree. This deficiency not only limits the parallelism of such algorithms exposed to schedulers, but also makes it difficult to simultaneously optimize for multiple program complexity measures such as span and cache complexity [45]. In practice, while schedulers can take advantage of the NP model's ability to expose locality, they are unable to load balance many classes of algorithms due to reduced parallelism [43].

Our Contributions:

- **Nested Dataflow model.** We introduce a new ***fire*** construct, denoted "$\rightsquigarrow$" (pronounced "fire"), to compose subtrees in a spawn tree. This construct, in addition to the $\|$ and ; constructs, forms the ***nested dataflow (ND)*** model, an extension of the nested parallel programming model. The "$\rightsquigarrow$" construct allows us to precisely specify the partial dependence patterns in many algorithms that $\|$ and ; constructs cannot. One of the design goals of the ND programming model is to allow runtime schedulers to execute inter-processor work like a dataflow model, while retaining the locality advantages of the nested parallel model by following the depth-first order of spawn tree for intra-processor execution.
- **DAG Rewriting System (DRS).** We provide a *DAG Rewriting System* that defines the semantics of the "$\rightsquigarrow$" construct by specifying the algorithm DAG that is equivalent to a dynamic spawn tree in the ND model (see Section 2).
- **Re-designed divide-and-conquer algorithms.** We re-design several typical divide-and-conquer algorithms in the ND model eliminating artificial dependencies, thus minimizing span. The set of algorithms include dense linear algebra and dynamic programming. Section 3 presents algorithm for solving triangular systems and the LCS problem; the associated technical report [25] presents more such examples. Our critical insight is that the data dependencies in all these algorithm DAGs can be precisely described with a small set of recursive partial dependency patterns (which we formalize as *sets of fire rules*) that allows us

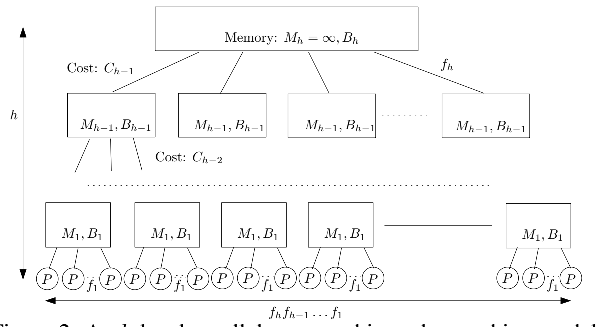


Figure 2: An h-level parallel memory hierarchy machine model.

to specify them compactly without losing any locality or parallelism. Other algorithms such as stencils and fast matrix multiplication can also be effectively described in this model.
- **Provably Efficient Runtime Schedulers.** The NP model has robust schedulers that map programs to shared-memory multicore systems, including those with hierarchical caches [17, 22, 11]. These schedulers have strong performance bounds for many programs based on complexity measures such as *work, span*, and *cache complexity* [16, 5, 10, 38, 1, 12, 11, 42]. In Section 4, we propose an extension of one such class of schedulers called the space-bounded (SB) schedulers for the ND model and provide provable performance guarantees on its performance on the Parallel Memory Hierarchy machine model (see Figure 2) — a symmetric "trees of caches" that models the multiple levels of possibly shared caches in shared memory multicore processors. We show that the algorithms in Section 3 have greater "parallelizability" in the ND model than in the NP model, and that space-bounded schedulers can use the extra parallelizability to achieve asymptotically optimal bounds on total running time on a greater number of processors than in the NP model for "reasonably regular" algorithms. Qunatitatively, suppose that the tree of caches, rooted at the DRAM, has h levels and any cache at level i has size M_i. If the input (which initially resides in DRAM) has size $N > M_{h-1}$, the SB scheduler for the ND model can efficiently use all the processors attached to up to a maximum of N^{1-c}/M_{h-1} level-$(h-1)$ caches for all the algorithms in this paper, where c is an arbitrarily small constant. When each level-i cache has at most $(M_i/M_{i-1})^{1-c}$ level-$(i-1)$ caches attached to it (and level-1 cache at most M_1^{1-c} processors), the running time is asymptotically optimal: $O\left(\left(\sum_{i=0}^{h-1} Q^*(\mathrm{t}; \sigma \cdot M_i) \cdot C_i\right)/p\right)$, where Q^* is the *parallel cache complexity* of the algorithm t, C_i is the cost of a cache miss at level-i cache, $\sigma \in (0,1)$ is a constant, and p is the number of processors. This compares favorably with the SB scheduler for the NP model [11] which, for the algorithms in the paper, requires a large input size of at least M_{h-1}^2 before it can asymptotically match the efficiency of the ND version.

2. NESTED DATAFLOW MODEL

The nested dataflow model extends the NP model by introducing an additional composition construct, “⇝”, which generalizes the existing “∥” and “;” constructs. Programs in both the NP and ND models are expressed as spawn trees, where the internal nodes are the composition constructs and the leaf nodes are strands. We refer to subtrees of the spawn tree as ***tasks*** or ***function calls***. We refer to the subtree rooted at the i-th child of an internal node as its ***i*-th *subtask***. In both the models, larger tasks can be defined by composing smaller tasks with the “;” and “∥” constructs. The ND model allows tasks to be defined as a composition using the additional **binary** construct, “⇝”, which enables the specification of “partial dependencies” between subtasks. This represents an arbitrary middle-point between the “;” construct (full dependency) and the “∥” construct (zero dependency).

For example, consider the program in Figure 3 represented by the spawn tree in Figure 4. The entire program, MAIN, is comprised of two tasks F and G. Task F is the serial composition of tasks A and C. Similarly, task G is the serial composition of B and D. Task C depends on A, which creates a partial dependency from F to G. Instead of using a “;” construct, which would block D until the completion of F (including both A and B), we denote the partial dependency with the “⇝” construct in Figure 4.

$$\begin{array}{llll} \text{MAIN}()\{ & \text{F}()\{ & \text{G}()\{ & \oplus \overset{FG}{\rightsquigarrow} \ominus = \{ \\ \quad \text{F}() \overset{FG}{\rightsquigarrow} \text{G}() & \quad \text{A}() \,;\, \text{B}() & \quad \text{C}() \,;\, \text{D}() & \quad \oplus\textcircled{1} \,;\, \ominus\textcircled{1} \\ \} & \} & \} & \} \end{array}$$

Figure 3: Code for MAIN, F, G, and a fire rule.

The partial dependency from F to G is specified with the rule $\overset{FG}{\rightsquigarrow}$. To specify that the only dependence is from A, the first subtask of F, to C, the first subtask of G, we write $\oplus \overset{FG}{\rightsquigarrow} \ominus = \{\oplus\textcircled{1} \,;\, \ominus\textcircled{1}\}$. The circled values denote ***relative pedigree***, or ***pedigree*** in short, which represents the position of a nested function call in a spawn tree with respect to its ancestor [36]. We use wildcards ⊕ and ⊖ to represent the ***source*** and ***sink*** of the partial dependency. We then specify ***a set of fire rules*** to describe the partial dependence pattern of the “⇝” construct between the source and the sink nodes. In the above case, we used ⊕① to denote the first subtask of the source, ⊕; similarly, ⊖① denotes the first subtask of the sink. The semicolon indicates a full dependency between them. In the context of MAIN in Figure 3, ⊕ is bound to F and ⊖ to G, implying that there is a full dependency from Ⓕ①, which refers to A, to Ⓖ①, which refers to C. In the general case, we allow multiple rewriting rules in the definition of a fire construct, and “multilevel” pedigrees (e.g. ⊕②① denotes the first subtask of the second subtask of the source) in each rule.

In the previous example, the dependency from A to C is a full dependency; that is, the entirety of A must be completed before C can start. However, this dependency itself may be artificial. Therefore, we allow the “⇝” construct to be recursively defined using fire rules that themselves represent partial dependencies.

Consider the following divide-and-conquer algorithm for computing the matrix product $C+=A\times B$, which we denote $\text{MM}(A,B,C)$. Let C_{00},C_{01},C_{10} and C_{11} denote the top left, bottom left, top right, and the bottom right quadrants of C respectively. In the ND model, we can define $\text{MM}(A,B,C)$ to be

$$\begin{array}{ll} ((\text{MM}(A_{00},B_{00},C_{00}) \parallel \text{MM}(A_{00},B_{01},C_{01})) & //\textcircled{1}\textcircled{1}\textcircled{1}\|\textcircled{1}\textcircled{1}\textcircled{2} \\ \parallel (\text{MM}(A_{10},B_{00},C_{10}) \parallel \text{MM}(A_{10},B_{01},C_{11}))) & //\textcircled{1}\textcircled{2}\textcircled{1}\|\textcircled{1}\textcircled{2}\textcircled{2} \\ \overset{MM}{\rightsquigarrow} ((\text{MM}(A_{01},B_{10},C_{00}) \parallel \text{MM}(A_{01},B_{11},C_{01})) & //\textcircled{2}\textcircled{1}\textcircled{1}\|\textcircled{2}\textcircled{1}\textcircled{2} \\ \parallel (\text{MM}(A_{11},B_{10},C_{10}) \parallel \text{MM}(A_{11},B_{11},C_{11}))). & //\textcircled{2}\textcircled{2}\textcircled{1}\|\textcircled{2}\textcircled{2}\textcircled{2} \end{array}$$

Each quadrant of C is written to by two of the eight subtasks; each

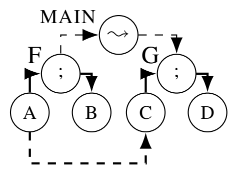


Figure 4: Spawn tree corresponding to the code in Figure 3.

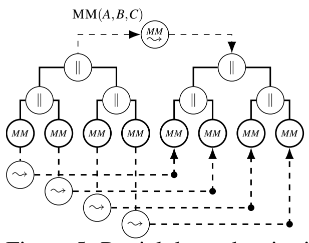


Figure 5: Partial dependencies in the recursive matrix multiply algorithm.

such pair of subtasks must be serialized to avoid a data race. For this, we might naively define the fire construct “$\overset{MM}{\rightsquigarrow}$” between the immediate subtasks of $\text{MM}(A,B,C)$ with a pair of fire rules:

$$\oplus \overset{MM}{\rightsquigarrow} \ominus \quad \{\oplus\textcircled{1} \,;\, \ominus\textcircled{1}, \quad \oplus\textcircled{2} \,;\, \ominus\textcircled{2}\}$$

However, notice that the dependency between first subtasks (as well as second subtasks), which is expressed with “;” in the code above, is in reality a partial dependency. Furthermore, each of these partial dependencies has the same pattern as “$\overset{MM}{\rightsquigarrow}$”. Since this pattern repeats recursively down an arbitrary number of levels, the “$\overset{MM}{\rightsquigarrow}$” construct should have been described by the fire rules:

$$\oplus \overset{MM}{\rightsquigarrow} \ominus \quad \{\oplus\textcircled{1} \overset{MM}{\rightsquigarrow} \ominus\textcircled{1}, \quad \oplus\textcircled{2} \overset{MM}{\rightsquigarrow} \ominus\textcircled{2}\}, \tag{1}$$

wherein “;” is replaced by “$\overset{MM}{\rightsquigarrow}$”.

If the recursion terminates at the level indicated in Figure 5, the four instances of “$\overset{MM}{\rightsquigarrow}$” between leaves of the spawn tree will be interpreted as four full dependencies between the corresponding strands. If the recursion continues, the fire rules are used to further refine the dependencies. Whereas this algorithm has only one set of dependence patterns (fire rules), we will see algorithms with multiple types of fire rules in the next section.

DAG Rewriting System (DRS). We formalize the semantics of the “⇝” construct with a DRS that defines the algorithm DAG corresponding to the spawn tree given at runtime. The spawn tree can unfold dynamically at runtime by incrementally ***spawning*** new tasks – a spawn operation rewrites a leaf of the spawn tree into an internal node by adding two new leaves below. The composition construct in the internal nodes of the spawn tree imply dependencies between its subtrees. We represent these dependencies as directed ***dataflow arrows*** in the spawn tree. The equivalent algorithm DAG implied by the spawn tree is the DAG with the leaves of the spawn tree as vertices, and edges representing dataflow edges implied by both the serial and fire constructs that are incident to the leaves of the spawn tree. The DAG also grows with the spawn tree; new vertices are added to the DAG whenever new tasks are spawned, and the construct used in the spawn operation defines the edges between these new vertices in the algorithm DAG. Note that maintaining a full algorithm DAG at runtime is not necessary. To save space, one can carefully design the order of the execution of the spawn tree, and recycle the memory used to represent parts of the spawn tree that have finished executing as in [17, 34]. We will leave this for future work. Instead, we focus here on the algorithm DAG to clarify the semantics of the fire construct.

The DRS iteratively constructs the dataflow edges, and equivalently the algorithm DAG, by starting with a single vertex representing the root of the spawn tree and successively applying ***DAG rewriting rules***. Given a DAG G, a rewriting rule replaces a subgraph that is isomorphic to L with a copy of sub-graph $R=\langle V,E\rangle$, resulting in a new DAG G'. There are two rewriting rules:

1. ***Spawn Rule***: A spawn rule corresponds to a spawn operation. Any current leaf of the spawn tree corresponds to a single-vertex no-edge subgraph $L=\langle\{\text{A}\},\emptyset\rangle$ of the DAG. If it spawns, we

rewrite the leaf as a (sub)tree rooted by either a "$;$", "$\|$" or "$\rightsquigarrow$" in the spawn tree.[1] The root of the newly spawned (sub)tree inherits all incoming and outgoing dataflow arrows of the old leaf. For instance, if task A spawns B and C in serial, we rewrite the single-vertex, no-edge DAG L to $R = \langle\{\text{B},\text{“;”},\text{C}\},\overrightarrow{\text{BC}}\rangle$, where $\overrightarrow{\text{BC}}$ is a ***solid dataflow arrow*** (directed edge) from B to C ($\overrightarrow{\text{BC}}$ is actually a shorthand for all-to-all dataflow arrows from all possible descendants of B to those of C, i.e. $\text{B} \times \text{C}$). If task A calls B and C in parallel, we rewrite it as $R = \langle\{\text{B},\text{“}\|\text{”},\text{C}\},\emptyset\rangle$. While the parallel construct introduces no dataflow arrows between B and C, a rewriting rule from its closest ancestor that is a "$\rightsquigarrow$" construct can introduce dataflow arrows to these two nodes according to the *fire rule*. Similar semantics apply for non-binary serial and parallel constructs. rewrite to $R = \langle\{\text{B},\text{“}\rightsquigarrow\text{”},\text{C}\},E' \subseteq \text{B} \times \text{C}\rangle$, where E' is a ***dashed dataflow arrow*** and is a subset of all possible arrows from descendants of B to descendants of C to be defined by the fire rule as follows.

2. ***Fire Rule***: Given a dashed dataflow arrow between arbitrary source and sink nodes, including those from the left child of a fire construct to its right child, we (recursively) rewrite the arrow using the set of fire rules associated with it. These rules specify how the "$\rightsquigarrow$" construct is rewritten to a set of dataflow arrows between the descendants of the source and the sink nodes. There are two possible cases for rewriting:
   - If both operands A and B are strands, the dataflow arrow between them is rewritten as either "A ; B" or, if the fire construct has no rewriting rules, "A $\|$ B".
   - If the source task A of a "$\rightsquigarrow$" construct is rewritten by a spawn rule into a tree containing k subtasks, we add dataflow arrows $E' \subseteq \{\text{A}_1,\ldots\text{A}_k\} \times \text{B}$ to the resulting DAG, i.e. $R = \langle V, E \cup E'\rangle$, where the arrows in E' and their labels are determined based on the set of fire rules as follows: for a fire rule of the form $\oplus \text{ⓘ} p \overset{T}{\rightsquigarrow} \ominus q$ (where p and q are some pedigrees) from A to B, we add a dataflow arrow $\oplus p \overset{T}{\rightsquigarrow} \ominus q$ from A_i to B. An analogous rule applies when the sink spawns.

From the DRS, it is evident that the binary "$;$" and "$\|$" constructs are special cases of the "$\rightsquigarrow$" construct. Four fire rules that recursively refine between both pairs of subtasks of $\oplus$ and $\ominus$ define the ; construct, and an empty set of rules defines "$\|$". Higher-degree "$;$" and "$\|$" constructs are also easily replaced by "$\rightsquigarrow$".

Work-Span Analysis. *Work-Span analysis* is commonly used to analyze the complexity of an algorithm DAG. We use T_1 to denote a task's ***work***, that is, the total number of instructions it contains. We use T_∞ to denote its ***span***, that is, the length of the critical path of its DAG. The composition rule to calculate work T_1 for all three constructs of the ND model is always a simple summation. In principle, the composition to calculate the span T_∞ for all three constructs is the maximum length of all possible paths from source to sink, i.e. the critical path. Since "$;$" and "$\|$" primitives have fixed semantics in all contexts, the span of tasks constructed with them can be simplified as follows: for $c = a\ ;\ b$, $T_{\infty,c} = T_{\infty,a} + T_{\infty,b}$; for $c = a \ \|\ b$, $T_{\infty,c} = \max\{T_{\infty,a}, T_{\infty,b}\}$. On the other hand, since the semantics of a "$\rightsquigarrow$" construct are parameterized by its set of fire rules, we have to calculate the depth of the task constructed with it on a case-by-case basis. For instance, for the code in Figure 3, we have $T_{\infty,\text{MAIN}} = T_{\infty,\text{F}\overset{\text{FG}}{\rightsquigarrow}\text{G}} = \max\{T_{\infty,\text{A}} + T_{\infty,\text{B}}, T_{\infty,\text{A};\text{C}} + T_{\infty,\text{D}}\}$, where $T_{\infty,\text{A};\text{C}}$ is $T_{\infty,\text{A}} + T_{\infty,\text{C}}$. If the rule "$\overset{\text{FG}}{\rightsquigarrow}$" were to place a partial dependence "$\overset{\text{AC}}{\rightsquigarrow}$" from A to C, calculating span would require further recursive analysis.

[1]A leaf with a non-constant degree parallel construct such as a parallel for loop must be rewritten as an binary tree in our model.

3. ALGORITHMS IN THE ND MODEL

We now demonstrate the advantages of the ND model by expressing typical 2-way divide-and-conquer classical linear algebra and dynamic programming algorithms in it, and deriving the necessary fire rules. We present algorithms for solving triangular systems and the LCS problem. The associated technical report [25] also presents algorithms for Cholesky factorization, LU factorization with partial pivoting and Floyd-Warshall algorithm for All-Pairs-Shortest-Paths. We demonstrate that these algorithms have improved parallelism in the ND model by proving that their span in the ND model is smaller than in the NP model and in fact matches the span of their algorithm DAG. Furthermore, the arrangement of tasks in the spawn tree for these algorithms does not change between the NP and the ND model, so that the cache complexity of the depth-first traversal is the same in both ND and NP models. In Section 4, we present more sophisticated metrics to quantify parallelism in the presence of caches; the algorithms presented here have improved parallelism according to those metrics as well.

Triangular System Solver. A Triangular System Solver $\text{TRS}(T,B)$ takes as input a lower triangular $n \times n$ matrix T and a square matrix B and outputs a square matrix X such that $TX = B$. A triangular system can be recursively decomposed as follows:

$$\begin{aligned}\begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix} &= \begin{bmatrix} T_{00} & 0 \\ T_{10} & T_{11} \end{bmatrix} \begin{bmatrix} X_{00} & X_{01} \\ X_{10} & X_{11} \end{bmatrix} \\ &= \begin{bmatrix} T_{00}X_{00} & T_{00}X_{01} \\ T_{10}X_{00} + T_{11}X_{10} & T_{10}X_{01} + T_{11}X_{11} \end{bmatrix} \qquad (2)\end{aligned}$$

Equation (2) recursively solves TRS on four equally sized sub-quadrants X_{00}, X_{01}, X_{10}, and X_{11}, as graphically depicted in Figure 7. It can be expressed in the NP model as shown in Equation (3), where $\text{MMS}(A,B,C)$ represents a cache-oblivious matrix multiplication and subtraction (identical to the one presented in Section 2, except instead of computing $C += AB$ it computes $C -= AB$) with span $O(n)$ and using $O(n^2)$ space.[2]

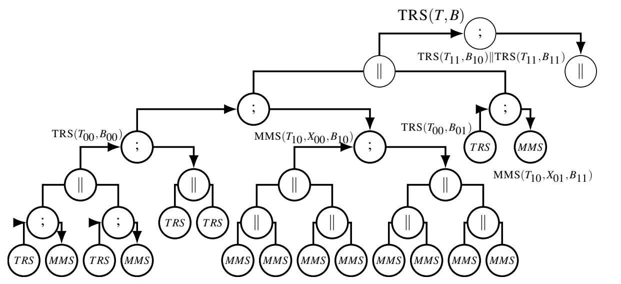
$$\begin{aligned} X \leftarrow \text{TRS}(T,B) = &((X_{00} \leftarrow \text{TRS}(T_{00},B_{00})\ ;\ \text{MMS}(T_{10},X_{00},B_{10})) \\ &\| \ (X_{01} \leftarrow \text{TRS}(T_{00},B_{01})\ ;\ \text{MMS}(T_{10},X_{01},B_{11}))) \\ &;(X_{10} \leftarrow \text{TRS}(T_{11},B_{10}) \ \|\ X_{11} \leftarrow \text{TRS}(T_{11},B_{11})) \quad (3)\end{aligned}$$

The span of the TRS algorithm, expressed in the NP model is given by the recurrence $T_{\infty,\text{TRS}}(n) = 2T_{\infty,\text{TRS}}(n/2) + T_{\infty,\text{MMS}}(n/2)$, which evaluates to $O(n \log n)$. This is not optimal; a straightforward right-looking algorithm has a span of $O(n)$.

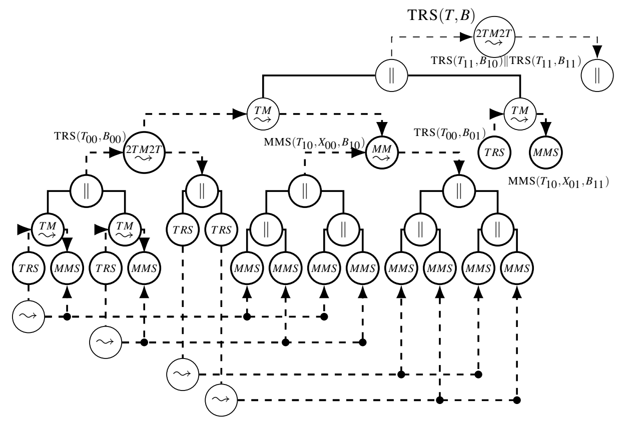
In Equation (4), we replace the "$;$" constructs from the original spawn tree with "$\rightsquigarrow$" constructs, in order to remove artificial dependencies. Because the two "$\rightsquigarrow$" constructs join different types of tasks, they have distinct types, which we denote "$\overset{TM}{\rightsquigarrow}$" and "$\overset{2TM2T}{\rightsquigarrow}$". Note that there are algorithms, e.g. Cholesky factorization, where two types of subtasks, say, TRS and MMS, have more than one kind of partial dependency pattern between them based on where they occur. Each type of fire construct has a different set of fire rules; in order to determine what these rules are, we expand an additional level of recursion to examine finer-grained data dependencies.

$$\begin{aligned} &X \leftarrow \text{TRS}(T,B) = \\ &((X_{00} \leftarrow \text{TRS}(T_{00},B_{00}) \overset{TM}{\rightsquigarrow} \text{MMS}(T_{10},X_{00},B_{10})) \\ &\quad \| \ (X_{01} \leftarrow \text{TRS}(T_{00},B_{01}) \overset{TM}{\rightsquigarrow} \text{MMS}(T_{10},X_{01},B_{11}))) \\ &\overset{2TM2T}{\rightsquigarrow}(X_{10} \leftarrow \text{TRS}(T_{11},B_{10}) \ \|\ X_{11} \leftarrow \text{TRS}(T_{11},B_{11})) \qquad (4)\end{aligned}$$

[2]There is also an 8-way divide-and-conquer cache-oblivious parallel algorithm of MMS that has an optimal span of $O(\log^2 n)$ but uses $O(n^3)$ space which can be used to trade off span for space complexity.



(a) Spawn tree of TRS with only "$\|$" and ";" constructs in NP model



(b) Spawn Tree of TRS with "$\leadsto$", "$\|$", and ";" constructs in ND model

Figure 6: Spawn trees of TRS in the NP and ND models. The shape of the tree and the leaves are the same between the two models, except that some of the ; constructs in NP model are relaxed with $\leadsto$ constructs and their dataflow arrows in the ND model. Dashed arrows corresponding to "$\leadsto$" constructs are recursively rewritten until both source and sink subtrees are leaves, where they are treated as solid arrows. For simplicity, the figure illustrates only dataflow arrows of type TM between the leaves, and omits dataflow arrows of other types.

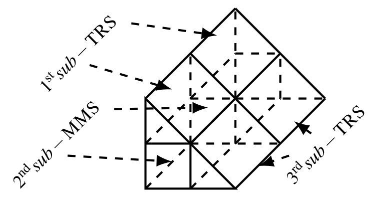


Figure 7: Geometric picture of a 2-way divide-and-conquer TRS algorithm.

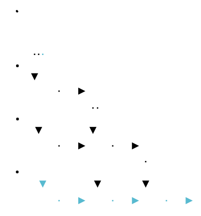


Figure 8: Cross-section of the TRS algorithm DAG.

Notice that the source task of $\overset{2TM2T}{\leadsto}$ is $((X_{00} \leftarrow \mathrm{TRS}(T_{00},B_{00}) \overset{TM}{\leadsto} \mathrm{MMS}(T_{10},X_{00},B_{10})) \parallel (X_{01} \leftarrow \mathrm{TRS}(T_{00},B_{01}) \overset{TM}{\leadsto} \mathrm{MMS}(T_{10},X_{01},B_{11})))$, and the its sink is $(X_{10} \leftarrow \mathrm{TRS}(T_{11},B_{10}) \parallel X_{11} \leftarrow \mathrm{TRS}(T_{11},B_{11}))$. Since the left (analogously, right) subtask of the sink can start as soon as the matrix multiply updating B_{10} (B_{11}), which is the right subtask of the left (right) subtask of the source, is completed:

$$\oplus \overset{2TM2T}{\leadsto} \ominus = \{\oplus\textcircled{1}\textcircled{2} \overset{MT}{\leadsto} \ominus\textcircled{1}, \oplus\textcircled{2}\textcircled{2} \overset{MT}{\leadsto} \ominus\textcircled{2}\}. \quad (5)$$

Both fire constructs in the fire rules are of type "$\overset{MT}{\leadsto}$" since the dependency structure is identical: the matrix updated in the source MMS task is used as the second argument of the TRS task.

In order to determine the set of fire rules for "$\overset{MT}{\leadsto}$", we expand a pair of subtasks connected by the "$\overset{MT}{\leadsto}$" construct to an additional level of recursion. For instance, we will expand the task $\mathrm{MMS}(T_{10},X_{00},B_{10})$ in equation Equation (6), which (as the source) binds to $\oplus$ in $\overset{TM}{\leadsto}$, and $X_{00} \leftarrow \mathrm{TRS}(T_{11},B_{10})$ in Equation (7), which binds to $\ominus$. In the following program, we use $A_{00,11}$ to denote the bottom right quadrant of the top left quadrant of A.

$$\begin{aligned}
&\mathrm{MMS}(T_{10},X_{00},B_{10}) = \qquad\qquad //\oplus\\
&((\mathrm{MMS}(T_{10,00},X_{00,00},B_{10,00}) \parallel \mathrm{MMS}(T_{10,00},X_{00,01},B_{10,01}))\\
&\ \parallel (\mathrm{MMS}(T_{10,10},X_{00,00},B_{10,10}) \parallel \mathrm{MMS}(T_{10,10},X_{00,01},B_{10,11})))\\
&\overset{MM}{\leadsto} ((\mathrm{MMS}(T_{10,01},X_{00,10},B_{10,00}) \parallel \mathrm{MMS}(T_{10,01},X_{00,11},B_{10,01}))\\
&\ \parallel (\mathrm{MMS}(T_{10,11},X_{00,10},B_{10,10}) \parallel \mathrm{MMS}(T_{10,11},X_{00,11},B_{10,11}))). \quad (6)
\end{aligned}$$

$$\begin{aligned}
&X_{10} \leftarrow \mathrm{TRS}(T_{11},B_{10}) = \qquad\qquad //\ominus\\
&((X_{00,00} \leftarrow \mathrm{TRS}(T_{11,00},B_{10,00}) \overset{TM}{\leadsto} \mathrm{MMS}(T_{11,10},X_{00,00},B_{10,10}))\\
&\quad \parallel (X_{00,01} \leftarrow \mathrm{TRS}(T_{11,00},B_{10,01}) \overset{TM}{\leadsto} \mathrm{MMS}(T_{11,10},X_{00,01},B_{10,11})))\\
&\overset{2TM2T}{\leadsto} (X_{00,10} \leftarrow \mathrm{TRS}(T_{11,11},B_{10,10}) \parallel X_{00,11} \leftarrow \mathrm{TRS}(T_{11,11},B_{10,11}))\\
&\qquad\qquad\qquad\qquad (7)
\end{aligned}$$

The dependence of the sink task, $\ominus$, on the source task, $\oplus$, in "$\overset{MT}{\leadsto}$" is a result of requiring the value of matrix B_{10} to be updated by $\oplus$ before $\ominus$ can use it in a computation. At a more fine-grained level, we can examine which quadrant of B_{10} each subtask of $\ominus$ requires (and which subtask of $\oplus$ computes that quadrant) in order to calculate the fine-grained dependencies. For instance, consider the subtask $X_{00,00} \leftarrow \mathrm{TRS}(T_{11,00},B_{10,00})$, whose pedigree is $\ominus\textcircled{1}\textcircled{1}\textcircled{1}$, which requires $B_{10,00}$. This quadrant of B_{10} is updated in $\mathrm{MMS}(T_{10,01},X_{00,10},B_{10,00})$ of the source, whose pedigree is $\oplus\textcircled{2}\textcircled{1}\textcircled{1}$. Furthermore, notice that the dependency from $\oplus\textcircled{2}\textcircled{1}\textcircled{1}$ to $\ominus\textcircled{1}\textcircled{1}\textcircled{1}$ takes the same form as the dependency from $\oplus$ to $\ominus$ itself: the matrix updated by the source is the second argument in the sink. Therefore, the fire rule for this particular dependency is $\oplus\textcircled{2}\textcircled{1}\textcircled{1} \overset{TM}{\leadsto} \ominus\textcircled{1}\textcircled{1}\textcircled{1}$. Similar reasoning gives the remaining fire rules:

$$\begin{aligned}
\oplus \overset{TM}{\leadsto} \ominus = \{&\oplus\textcircled{1}\textcircled{1}\textcircled{1} \overset{TM}{\leadsto} \ominus\textcircled{1}\textcircled{1}\textcircled{1}, \oplus\textcircled{1}\textcircled{1}\textcircled{1} \overset{TM}{\leadsto} \ominus\textcircled{1}\textcircled{2}\textcircled{1},\\
&\oplus\textcircled{1}\textcircled{2}\textcircled{1} \overset{TM}{\leadsto} \ominus\textcircled{1}\textcircled{1}\textcircled{2}, \oplus\textcircled{1}\textcircled{2}\textcircled{1} \overset{TM}{\leadsto} \ominus\textcircled{1}\textcircled{2}\textcircled{2},\\
&\oplus\textcircled{2}\textcircled{1} \overset{TM}{\leadsto} \ominus\textcircled{2}\textcircled{1}\textcircled{1}, \oplus\textcircled{2}\textcircled{1} \overset{TM}{\leadsto} \ominus\textcircled{2}\textcircled{2}\textcircled{1},\\
&\oplus\textcircled{2}\textcircled{2} \overset{TM}{\leadsto} \ominus\textcircled{2}\textcircled{1}\textcircled{2}, \oplus\textcircled{2}\textcircled{2} \overset{TM}{\leadsto} \ominus\textcircled{2}\textcircled{2}\textcircled{2}\} \qquad (8)\\
\oplus \overset{2TM2T}{\leadsto} \ominus = \{&\oplus\textcircled{1}\textcircled{2} \overset{MT}{\leadsto} \ominus\textcircled{1}, \oplus\textcircled{2}\textcircled{2} \overset{MT}{\leadsto} \ominus\textcircled{2}\}\\
\oplus \overset{MT}{\leadsto} \ominus = \{&\oplus\textcircled{2}\textcircled{1}\textcircled{1} \overset{MM}{\leadsto} \ominus\textcircled{1}\textcircled{1}\textcircled{2}, \oplus\textcircled{2}\textcircled{1}\textcircled{2} \overset{MM}{\leadsto} \ominus\textcircled{1}\textcircled{2}\textcircled{2},\\
&\oplus\textcircled{2}\textcircled{2}\textcircled{1} \overset{MT}{\leadsto} \ominus\textcircled{1}\textcircled{1}\textcircled{1}, \oplus\textcircled{2}\textcircled{2}\textcircled{2} \overset{MT}{\leadsto} \ominus\textcircled{1}\textcircled{2}\textcircled{1}\}
\end{aligned}$$

We now argue that the span of TRS in the ND model is $O(n)$, which is the span of the algorithm itself and therefore optimal. The span of an algorithm is the length of the longest path in its DAG. The algorithm DAG defined by TRS expressed in the ND model forms a periodic structure, a cross section of which can be seen in Figure 8, where squares represent matrix multiplications and triangles represent smaller TRS tasks (there are no edges between separate cross sections). The length of the longest path in the DAG, shown in blue, is $O(n)$.

We now formally prove that the algorithm we constructed in the ND model achieves this span. Let $T_{\infty,\mathrm{TRS}}(n)$ denote the span of TRS on a matrix with input size $n \times n$, and let $T_{\infty,\mathrm{TRS}\ p}(n)$ denote the span of the subtask with pedigree p descended from TRS with input size $n \times n$. Furthermore, let $T_{\infty,\overset{TM}{\leadsto}}(n)$ denote the critical path length of a TRS composed with a MMS by a "$\overset{TM}{\leadsto}$" construct, where both tasks are directly descended from a TRS of size $n \times n$. Note that it involves a TRS and a MMS, of size $n/2 \times n/2$ each.

Since replacing a fire construct with a serial construct can only increase the span, it suffices to show that a version of the problem with some fire constructs replaced with serial constructs has optimal span (this replacement can also simplify the algorithm description by allowing us to remove fire rules without making the span asymptotically worse). Replacing the "$\overset{2TM2T}{\rightsquigarrow}$" construct with ";" gives the following upper bound on the span of TRS in the ND model:

$$T_{\infty,\mathrm{TRS}}(n) \le T_{\infty,\mathrm{TRS}\textcircled{1}}(n) + T_{\infty,\mathrm{TRS}\textcircled{2}}(n). \tag{9}$$

Since the right subtask of TRS is merely the parallel composition of two TRS operations, each on a matrix of size $n/2 \times n/2$, the second term on the right reduces to the max of their (identical) spans, which is $T_{\infty,\mathrm{TRS}\textcircled{2}}(n) = T_{\infty,\mathrm{TRS}}(n/2)$.

The left subtask consists of two pairs (connected by a parallel composition), each consisting of a TRS task and a MMS task, connected by "$\overset{TM}{\rightsquigarrow}$" construct and done in parallel, and their spans are identical. Therefore, the first term on the right hand side of inequality 9 reduces to

$$T_{\infty,\mathrm{TRS}\textcircled{1}}(n) = T_{\infty,\mathrm{TRS}\textcircled{1}\textcircled{1}\textcircled{1}\overset{TM}{\rightsquigarrow}\mathrm{TRS}\textcircled{1}\textcircled{1}\textcircled{2}}(n) = T_{\infty,\overset{TM}{\rightsquigarrow}}(n).$$

The term on the right is the maximum length among all possible paths rewritten from ①①① $\overset{\mathrm{TM}}{\rightsquigarrow}$ ①①②. There are two types of paths that could potentially be the longest. An instance of the first type is the "$\overset{TM}{\rightsquigarrow}$" composition of tasks TRS①①①①① ①, a TRS of size $n/4$, with TRS①①②①① ①, a MMS of size $n/4$, followed by a MMS of size $n/4$. This gives the first expression in the max term in the equation below. An instance of the second type is the $\overset{TM}{\rightsquigarrow}$ composition of the task TRS①①①①① ①, a TRS of size $n/4$, with TRS①①①①① ②, a MMS of size $n/4$, followed by the "$\overset{TM}{\rightsquigarrow}$" composition of TRS①①①②①, a TRS of size $n/4$, with TRS①①②②① ①, a MMS of size $n/4$. This results in the second expression in the max term below.

$$T_{\infty,\overset{\mathrm{TM}}{\rightsquigarrow}}(n) \le \max\{T_{\infty,\overset{\mathrm{TM}}{\rightsquigarrow}}(n/2) + T_{\infty,\mathrm{MMS}}(n/4),\ 2T_{\infty,\overset{\mathrm{TM}}{\rightsquigarrow}}(n/2)\}$$

For the base case of the recurrence, we simply run TRS and MM sequentially at the base case size. Therefore, we have

$$T_{\infty,\overset{\mathrm{TM}}{\rightsquigarrow}}(1) = T_{\infty,\mathrm{TRS}}(1) + T_{\infty,\mathrm{MMS}}(1) = O(1).$$

Noting that $T_{\infty,\mathrm{MMS}}(n) = O(n)$, the recurrences can be solved to show that $T_{\infty,\mathrm{TRS}}(n) = O(n)$, which is asymptotically optimal.

One can similarly derive fire rules for composing divide-and-conquer algorithms for Cholesky factorization and LU factorization with partial pivoting in the ND model (see associated technical report [25]). Their depth is the optimal $O(n)$ for matrices of size $n \times n$. These algorithms are also cache-oblivious and have asymptotically optimal cache complexity.

LCS (Longest Common Subsequence). We now consider a divide-and-conquer algorithm for the LCS problem. Given two sequences $S = \langle s_1, s_2, \ldots, s_m \rangle$ and $T = \langle t_1, t_2, \ldots, t_n \rangle$, the goal is to find the length of longest common subsequence of S and T. LCS can be computed using the recursion in Equation (10) when $m = n$ [24] (a similar recursion can be written for the general case, $m \neq n$).[3]

$$X(i,j) = \begin{cases} 0 & \text{if } i = 0 \vee j = 0 \\ X(i-1,j-1)+1 & \text{if } i,j > 0 \wedge s_i = t_j \\ \max\{X(i,j-1), X(i-1,j)\} & \text{if } i,j > 0 \wedge s_i \neq t_j \end{cases} \tag{10}$$

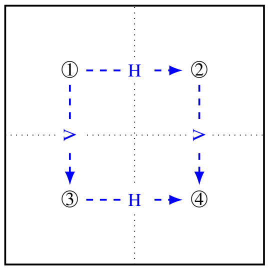
In the ND model, we express this recursion as follows (Figure 9c illustrates the spawn tree this describes):

$$\begin{aligned} X \leftarrow \mathrm{LCS}(X) = ((X_{00} \leftarrow \mathrm{LCS}(X_{00})) &\overset{\mathrm{HV}}{\rightsquigarrow} \\ (X_{01} \leftarrow \mathrm{LCS}(X_{01}) \parallel X_{10} &\leftarrow \mathrm{LCS}(X_{10}))) \\ \overset{\mathrm{VH}}{\rightsquigarrow} (X_{11} \leftarrow \mathrm{LCS}(X_{11})) \end{aligned} \tag{11}$$

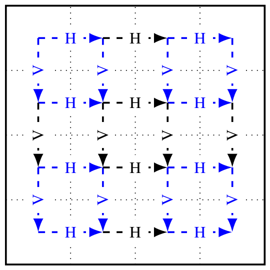
The partial dependencies are given by the following fire rules which are illustrated in Figures 9a and 9b:

$$\oplus \overset{\mathrm{HV}}{\rightsquigarrow} \ominus = \{\oplus \overset{\mathrm{H}}{\rightsquigarrow} \ominus\textcircled{1}, \oplus \overset{\mathrm{V}}{\rightsquigarrow} \ominus\textcircled{2}\} \tag{12}$$

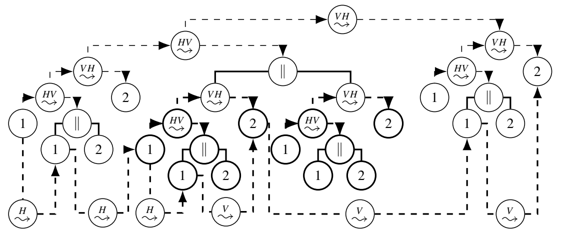
[3]A similar recursion applies to the pairwise sequence alignment with affine gap cost [28].



(a) Dashed arrows are defined by the algorithm in Equation (11) and fire rules in Equations (12) and (13).



(b) Dashed arrows are rewritten by fire rules in Equations (14) and (15).



(c) Spawn tree of LCS in ND model. We only draw one "$\rightsquigarrow$" path in Figure 9b from top-left to bottom-right cell

Figure 9: DAG Rewriting and spawn tree of LCS in ND model.

$$\oplus \overset{\mathrm{VH}}{\rightsquigarrow} \ominus = \{\oplus\textcircled{1} \overset{\mathrm{V}}{\rightsquigarrow} \ominus, \oplus\textcircled{2} \overset{\mathrm{H}}{\rightsquigarrow} \ominus\} \tag{13}$$

$$\oplus \overset{\mathrm{H}}{\rightsquigarrow} \ominus = \{\oplus\textcircled{1}\textcircled{2}\textcircled{1} \overset{\mathrm{H}}{\rightsquigarrow} \ominus\textcircled{1}\textcircled{1}, \oplus\textcircled{2} \overset{\mathrm{H}}{\rightsquigarrow} \ominus\textcircled{1}\textcircled{2}\textcircled{2}\} \tag{14}$$

$$\oplus \overset{\mathrm{V}}{\rightsquigarrow} \ominus = \{\oplus\textcircled{1}\textcircled{2}\textcircled{2} \overset{\mathrm{V}}{\rightsquigarrow} \ominus\textcircled{1}\textcircled{1}, \oplus\textcircled{2} \overset{\mathrm{V}}{\rightsquigarrow} \ominus\textcircled{1}\textcircled{2}\textcircled{1}\} \tag{15}$$

To compute the span of LCS, consider the dynamic programming table. The span is defined by the length of longest path in the DAG which runs from the top left entry to the bottom right entry. We will separately compute the length of the longest horizontal path, $T_h(n)$, and the length of the longest vertical path, $T_v(n)$. Notice that the span, $T_{\infty,LCS}(n)$, is bounded above by $T_h(n) + T_v(n)$.

Since we split an LCS problem whose dynamic programming table is of size $n \times n$ into four LCS problems of size $n/2 \times n/2$ of which the longest horizontal path covers two, we have $T_h(n) = 2T_h(n/2)$. The base case (a 1×1 matrix) only depends on three inputs, so that $T_h(1) = O(1)$. Therefore, $T_h(n) = O(n)$. Similar reasoning shows that $T_v(n) = O(n)$. As a result, $T_{\infty,LCS}(n)$ is bounded above by $O(n)$, which is optimal.

4. SPACE-BOUNDED SCHEDULERS FOR THE ND MODEL

We show that *reasonably regular* programs in the ND model, including the algorithms in Section 3 and [25], can be effectively mapped to Parallel Memory Hierarchies by adapting the space-bounded (SB) schedulers for NP programs. Regularity is a quantifiable property of the algorithm (or spawn tree) that measures how difficult it is to schedule; we will define this for programs in the ND model and show that the algorithm in Section 3 are highly regular. Space-bounded schedulers for programs in the NP model were first proposed for completely regular programs [22], improved upon and rigorously analyzed in [11], and empirically demonstrated to outperform work-stealing based schedulers for many algorithms in [43], but not for TRSM and Cholesky algorithms due to their limited parallelism in the NP model [43]. The key idea in SB schedulers is to map tasks to processors and caches in the hierarchy based on annotations regarding their memory footprint. The main result of this section is Theorem 3, which shows that the SB scheduler is able to exploit the extra parallelism exposed in the ND model.

Machine Model: Parallel Memory Hierarchy. SB schedulers are well suited for the Parallel Memory Hierarchy (PMH) machine model [4] (see Figure 2), which models the multi-level cache hierarchies and cache sharing common in shared memory multi-core architectures. The PMH is represented by a symmetric tree rooted at a main memory of infinite size. The internal nodes are caches and the leaves are processors. We refer to subtrees rooted at some cache as ***subclusters***. Each cache at level i is of the same size M_i, and has the same the number of level-$(i-1)$ caches attached to it. We call this the ***fan-out*** of level-i and denote it by the constant f_i, so that the number of processors in a h-level tree is $p_h = \prod_{i=1}^{h} f_i$. We let the constant M_0 denote the number of registers on a processor. We let C_{i-1} denote the cost parameter representing the cost of servicing a cache miss at level $(i-1)$ from level i. A cache miss that must be serviced from level j requires $C'_j = C_0 + C_1 + \cdots + C_{j-1}$ time steps. For simplicity, we let the cache block be one word long. This limitation can be relaxed and analyzed as in [11].

Terminology. A task is ***done*** when all the leaf nodes (strands) associated with its subtree have been executed. A dataflow arrow originating at a leaf node in the spawn tree is ***satisfied*** when its source node is done. A dataflow arrow originating at an internal node of the spawn tree is ***satisfied*** when all its descendants (rewritings) according to the fire rules have been satisfied. A task is ***fully ready*** or just ***ready*** when all the incoming dependencies (dataflow arrows) originating outside the subtree are satisfied. The ***size***, $s(\cdot)$, of a task or a strand is the number of distinct memory locations accessed by it. We assume that programs are statically allocated, i.e., all necessary heap space is allocated up front and freed at program termination, so that the size function is well defined. The size annotation can be supplied by the programmer or can be obtained from a profiling tool. We call a task ***M-maximal*** if its size is at most M, but its parent in the spawn tree has size $> M$. A task is level-i maximal in a PMH if it is M_i-maximal, M_i being the size of a level-i cache. Note that even though an M_i-maximal task is not ready, a M_j-maximal subtask inside it (where $j < i$) can be ready.

SB Schedulers. We define a space-bounded scheduler to be any scheduler that has the anchoring and boundedness properties [43]:

- **Anchor:** As the spawn tree unfolds dynamically, we assign and ***anchor*** ready tasks to caches in the hierarchy with respect to which they are maximal. Tasks are *allocated* a part of the subcluster rooted at the assigned cache. The anchoring property requires that all the leaves of the spawn tree of a task be executed by processors in the part of the subcluster allocated.
- **Boundedness:** Tasks anchored to a cache of size M have a total size $\leq \sigma M$, where $\sigma \in (0,1)$ is a scheduler chosen constant called ***dilation parameter***.

There are several ways to maintain these properties and operate within its constraints. The approach taken in [11] is to have a task queue with each anchored task that contains its subtasks than can be potentially unrolled and anchored to the caches below it. We adopt the same approach here (outlined below for convenience) for the ND model with the **difference** being that we only anchor and run ready subtasks. In the course of execution, ready tasks are anchored to a suitable cache level (provided there is sufficient space left), and each anchored task is ***allocated*** subclusters beneath the cache, based on the size of the task. Just as in [11], a task of size S anchored at level-i cache is allocated

$$g_i(S) = \min\{f_i, \max\{1, \left\lfloor f_i(3S/M_i)^{\alpha'} \right\rfloor\}\}, \text{where } \alpha' = \min\{\alpha_{max}, 1\}$$

level-$(i-1)$ subclusters[4] where α_{max} is the parallelizability of the task, a term we will define shortly. All processors in the subclusters are required to work exclusively on this task. Initially, the root node of the spawn tree is anchored to the root of the PMH.

[4]The factor 3 in the allocation function helps in proving Thm. 3.

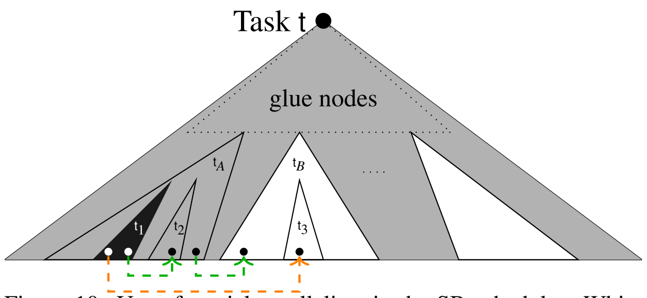


Figure 10: Use of partial parallelism in the SB scheduler. White represents tasks that are yet to start, gray represents running tasks, and black represents complete tasks. Green arrows represent dataflow arrows that may be used to start new tasks by the SB scheduler while orange dataflow arrows are not immediately used. Task t is level-i maximal; tasks t_A and t_B are level-$(i-1)$ maximal; tasks t_1, t_2, and t_3 are level-$(i-2)$ maximal. Although subtask t_1 has completed and has two outgoing dataflow edges, only t_2, which is in the same level-$(i-1)$ maximal subtask (t_A) can be started; t_3 can not immediately started until subtask t_A completes.

To find work, a processor traverses the path from the leaf it represents in the tree towards the root of the PMH until it reaches the lowest anchor it is part of. Here it checks for ready tasks in the queue associated with this anchor, and if empty, re-attempts to find work after a short wait. Otherwise, it pulls out a task from the queue. If the task is from an anchor at the cache immediately above the processor, i.e. **at an L_1 cache**, it executes the subtask by traversing the corresponding spawn tree in depth-first order. If the processor pulled this task out of an allocation at a **cache at level** $i > 1$, it does not try to execute its strands (leaves) immediately. Instead, it unrolls the spawn tree corresponding to the task using the DRS and enqueues those subtasks that are either of size $> M_{i-1}$, or not ready, in the queue corresponding to the anchor. Those subtasks that cannot be immediately worked on due to lack of space in the caches are also enqueued. However, if the processor encounters a ready task that has size less than that of a level-j cache ($j < i$), and is able to find sufficient space for it in the subcluster allocated to the anchor, the task is anchored at the level-j cache, and allocated a suitable number of subclusters below the level-j cache. The processor starts unraveling the spawn tree and finding work repeatedly. When an anchored task is done, the anchor, allocation and the associated resources are released for future tasks. We also borrow other details in the design of the space-bounded schedulers (e.g. how many subclusters are provisioned for making progress on "worst case allocations"? what fraction of cache is reserved for tasks that "skip cache levels"?) from prior work [11].

Roughly speaking, this scheduler uses the partial parallelism between level-$(i-1)$ maximal subtasks within a level-i maximal task. However, it does not use all the partial parallelism across level-$(i-2)$ subtasks, especially those dataflow arrows between level-$(i-2)$ subtasks in different level-$(i-1)$ subtasks (see Figure 10).

Metrics. We now analyze the runtime of the SB scheduler, accounting for the cost of executing the work and load imbalance, but not the overhead of the data structures needed to keep track of anchors, allocations, and the readiness of subtasks; optimizing the overhead is left for a future empirical study. The anchoring and boundedness properties help to preserve locality while trading off some parallelism. Inspired by [11, 42], we develop an analysis for the ND model to argue that the impact of the loss of parallelism due to the anchoring property on load balance is not significant.

A critical consequence of the anchoring property of the SB scheduler is that once a task is anchored to a cache, all the memory locations needed for the task are loaded only once and are not forced to be evicted until the completion of the task. This motivates the following quantification of locality. Given a task t, decompose the spawn tree into M-maximal subtasks, and "glue nodes" that hold these trees together (this decomposition is unique). Define the ***parallel cache complexity (PCC)***, $Q^*(\mathsf{t};M)$, of task t to be the sum of sizes of the maximal subtrees, plus a constant overhead from each glue node. This is motivated by the expectation that a good scheduler (such as SB) should be able to preserve locality within M-sized tasks given an cache of appropriate size, while it might be too cumbersome to preserve locality across maximal subtasks. [5] The PCC metric differs from the another common metric for locality of NP programs: the cache complexity Q_1 of the depth-first traversal in the ideal cache model [1]. Unlike Q_1, Q^* does not depend on the order of traversal, but does not capture data reuse across M-maximal subtasks, which is a smaller order term in our algorithms.

Note that M is a free parameter in this analysis. When the context is clear, we often replace the task t in the Q^* expression with a size parameter corresponding to the task, so that cache complexity is denoted $Q^*(N;M)$. With this notation we have the following bound on the cache complexity of the algorithms in Section 3.

CLAIM 1. *For dense matrices of size $N = n \times n$, the divide and conquer classical matrix multiplication and Triangular System Solve in Section 3, as well as the Cholesky and LU factorizations and the 2D analog of the Floyd-Warshall algorithm in [25] have parallel cache complexity $Q^*(N;M) = O(N^{1.5}/M^{0.5})$, when $N > M$, with the glue nodes contributing an asymptotically smaller term. The LCS algorithm has $Q^*(n;M) = O(n^2/M)$ for input of size $n > \sqrt{M}$. This is true even if the algorithms are expressed in the NP model by replacing fire constructs with the " ; " construct.*

As a direct consequence of the anchoring and boundedness properties, which conservatively provision cache space, the following restatement of [11, Theorem 3] applies to the ND model.

THEOREM 1. *Suppose* t *is a task in ND program that is anchored at a level-i cache of a PMH by a SB scheduler with dilation parameter $0 < \sigma < 1$ (i.e., a SB scheduler that anchors tasks of size at most σM_j at level j). Then for all cache levels $j \leq i$, the sum of cache misses incurred by all caches at level j is at most $Q^*(\mathsf{t};\sigma \cdot M_j)$.*

In conjunction with Claim 1, this gives a bound on the communication cost of the schedulers for ND algorithms. One can verify from results on lower bounds on communication complexity [7] that these bounds are asymptotically optimal. If the scheduler is able to perfectly load balance a program at every cache level on an h level PMH with p processors, we would expect a task t to take

$$\frac{\sum_{i=0}^{h-1} Q^*(\mathsf{t};\sigma \cdot M_i) \cdot C_i}{p} \tag{16}$$

time steps to complete, where $0 < \sigma < 1$ is the dilation parameter.

However, if the algorithm does not have sufficient parallelism for the PMH or is too irregular to load balance, we would expect it take longer. Furthermore, since the number of processors assigned to a task by a SB scheduler depends on its size, unlike in the case of work-stealing scheduler, a work-span analysis of programs may not be an accurate indicator of their running time. A more sophisticated cost model that takes account of locality, parallelism-space imbalances, and lack of parallelism at different levels of recursion is necessary. In the NP model [11, Defn. 3], this was quantified

[5]This definition is a generalization of [11, Defn.2] for the ND model. The full metric measures cache complexity in terms of cache lines to model latency and is also parameterized by a second parameter B: size of a cache line. We set $B = 1$ here for simplicity. This simplification can be reversed.

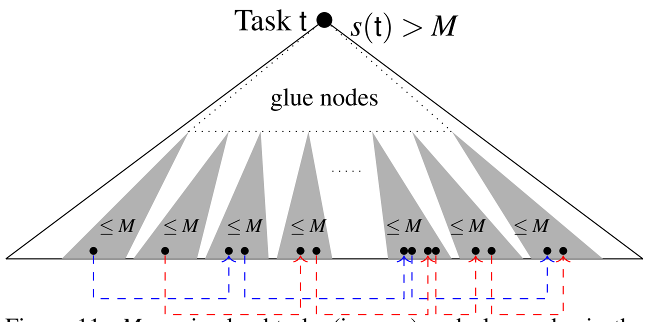


Figure 11: M-maximal subtasks (in gray) and glue nodes in the spawn tree of a task t. The PCC, $Q^*(\mathsf{t};M)$, is the sum of sizes of M-maximal subtasks plus one miss for each glue node. The red and blue sets of arrows represent two chains of dependencies in t. The ECC, $\widehat{Q}_\alpha(\mathsf{t};M)$, is determined by the maximum, among all such chains, of the sum of effective depth of M-maximal subtasks in the chain, and the ratio $Q^*(\mathsf{t};M)/s(\mathsf{t})^\alpha$ for a parameter $\alpha > 0$.

by the **effective cache complexity metric (ECC, $\widehat{Q}_\alpha$)**. We provide a new definition of this metric for the ND model. ECC attempts to capture the cost of load balancing the program on hypothetical machine with ***machine with parallelism at most*** α — a PMH which has at most $f_i \leq (M_i/M_{i-1})^\alpha$ level-$(i-1)$ caches beneath each level-i cache for all $1 \leq i \leq h$.

The metric assigns to each subtree of a spawn tree an estimate of its complexity, measured in cache miss cost equivalents, when mapped to a PMH by a SB scheduler. The estimate is based on its position in the spawn tree and its cache complexity in the PCC metric. The metric has two free parameters: α which represents the parallelism available of a hypothetical machine, and M which represents the size of one of the caches in the hierarchy with respect to which the spawn tree is being analyzed.

DEFINITION 2 (EFFECTIVE CACHE COMPLEXITY (ECC)). *Let* t *be a task in the ND model. Unroll the spawn tree of* t*, applying the DAG rewriting rules until all the leaves of the tree are M-maximal. Regard all dataflow arrows (solid or dashed) between the leaves to be dependencies (see Figure 11).*
The ECC of a M-maximal task t′*s is $\widehat{Q}_\alpha(\mathsf{t}';M) = Q^*(\mathsf{t}';M)$.*

The ECC of t *is $\widehat{Q}_\alpha(\mathsf{t};M)$, where* $\left\lceil \frac{\widehat{Q}_\alpha(\mathsf{t};M)}{s(\mathsf{t})^\alpha} \right\rceil =$

$$\max \begin{cases} \max_{\chi \in chains(\mathsf{t},M)} \left\{ \sum_{\mathsf{t}_i \in \chi} \left\lceil \frac{\widehat{Q}_\alpha(\mathsf{t}_i;M;\kappa)}{s(\mathsf{t}_i)^\alpha} \right\rceil \right\} & \textit{(depth dominated)} \\ \left\lceil \frac{\sum_{\mathsf{t}_i \in maximal(\mathsf{t},M)} \widehat{Q}_\alpha(\mathsf{t}_i;M)}{s(\mathsf{t})^\alpha} \right\rceil & \textit{(work dominated)} \end{cases}$$

where chains(t,M) represents the set of chains of dependence edges between M-maximal tasks, maximal(t,M), in the spawn tree of t*.*

The work dominated term has the same denominator as the left hand side and thus captures the total amount of cache complexity in the spawn tree (summation over leaves). The depth dominated term captures the critical path for the SB scheduler. The term $\lceil \widehat{Q}_\alpha(\mathsf{t};M)/s(\mathsf{t})^\alpha \rceil$ is the proxy for span in our analysis and we call it the ***effective depth*** of the task t. The depth dominated ensures that the effective depth defined by ECC for a task is at least the sum of the effective depths of all M-maximal tasks along any chain between M-maximal tasks induced by DAG rewriting with respect to the fire rules. The definition of ECC is such that:

1. In the range $\alpha \in [0, \alpha_{max})$, for some algorithm-specific constant α_{max}, $\widehat{Q}_\alpha(\mathsf{t};M) \leq c_U Q^*(\mathsf{t};M)$ for all $M > M_U$, for some positive universal constants c_U, M_U.

2. On a machine with parallelism $\beta \leq \alpha_{max} - \varepsilon$ for some arbitrarily small positive constant ε, the running time of the SB scheduler is within a constant factor of the perfectly load balanced scenario in equation 16 (see Theorem 3).
3. For NP programs, it coincides with the definition in [11].

Parallelizability of an Algorithm. For the above reasons, we refer to the α_{max} of an algorithm as its ***parallelizability*** just as in [42]. The greater the parallelizability of the algorithm, the more efficient it is to schedule on larger machines. When the parallelizability of the algorithm asymptotically approaches the difference between the work and the span exponents of the algorithm, we call it ***reasonably regular***. For an input of size $N = n \times n$, TRS, Cholesky and 2D Floyd-Warshall have work exponent 1.5 and span exponent 0.5, and the difference between them is 1. In many divide-and-conquer algorithms, such as in [14], where the NP model does not induce too many artificial dependencies, the parallelizability exceeds that of largest shared memory machines available today. In such algorithms SB schedulers have been empirically shown to be effective at managing locality without compromising load balance, and as a consequence, capable of outperforming work-stealing schedulers [43]. However, this is not the case for algorithms in Section 3, which lose some parallelism when expressed in the NP model.

For example, in the NP model, the parallelizability (w.r.t. cache size M) of the cache-oblivious matrix multiplication is $\alpha_{max,MM} = 1 - \log_M(1 + c_{MM})$ for some small constant c_{MM} (see Claim 2 in Appendix of [25]), which is as high as it can be. We expect the parallelizability of nested parallel TRS algorithm to be less than $\alpha_{max,MM}$. In fact, for an $n \times n$ upper triangular T and a right hand side B of size $N = n \times n$, the parallelizability of the nested parallel TRS algorithm in Equation (3) is $1 - \log_{\min\{N/M,M\}}(1 + c_{TRS})$ (see Claim 3 in Appendix of [25]). This is smaller than the parallelizability of matrix multiplication when $N/M < M$. Since L3 caches are of the order of 10MB, the reduced parallelism adversely affects load balance even in large instances that are of the order of gigabytes (also empirically observed in [43]). When expressed in the ND model, the parallelizability of TRS improves. This can be seen in Figure 8, where the depth dominated term corresponding to the longest chain has effective depth $c(N^{0.5}/M^{0.5})M^{1-\alpha} + c'$, which is less than the work dominated term when $\alpha < 1 - \log_M(1 + c_{TRS})$. This is the parallelizability of TRS in the ND model. This is also the case for other linear algebra algorithms, including Cholesky and LU factorizations. We can similarly show that the parallelizability of LCS in the ND model is 1.

Running time analysis. The main result of this section is Theorem 3 which shows that SB schedulers can make use of the extra parallelizability of programs expressed in the ND model.

THEOREM 3. *Consider an h-level PMH with p_h processors where a level-i cache has size M_i, fanout f_i and cache miss cost C_i. Let* t *be a task such that $S(\mathsf{t};B) > f_h M_{h-1}/3$ (the scheduler allocates the entire hierarchy to such a task) with parallelizability α_{max} in the ND model. Suppose that α_{max} exceeds the parallelism of the machine by a constant. The running time of* t *is no more than:*

$$\frac{\sum_{j=0}^{h-1} \widehat{Q}_\alpha(\mathsf{t}; M_j/3) \cdot C_j}{p_h} \cdot v_h, \quad \textit{where overhead } v_h \textit{ is}$$

$$v_h = 2 \prod_{j=1}^{h-1} \left(\frac{1}{k} + \frac{f_j}{(1-k)(M_j/M_{j-1})^{\alpha'}} \right),$$

for some constant $0 < k < 1$, where $\alpha' = \min\{\alpha_{max}, 1\}$.

When the machine parallelism is no greater than the parallelizability of the algorithm in the ND model, $\widehat{Q}_\alpha(t;M) \leq c_U Q^*(M)$. Therefore, the theorem says that the algorithm runs within a constant factor (v_h) of the perfectly load balanced scenario in Equation (16). Relating this theorem to the definition of machine parallelism, we infer that for highly regular algorithms considered in this paper, the SB scheduler can effectively use up to $O(N^{1-c'}/M_{h-1})$ level-$(h-1)$ subclusters for some arbitrarily small constant $c' < 0$.

We prove this theorem using the notion of effective work, the separation lemma (lemma 5) and a work-span argument based on effective depth as in [11]. The ***latency added effective work*** is similar to the effective cache complexity, but instead of counting just cache misses at one cache level, we add the cost of cache misses at each instruction. The cost $\rho(x)$ of an instruction x accessing location m is $\rho(x) = W(x) + C_i'$, where $W(x)$ is the work, and $C_i' = C_0 + C_1 + \cdots + C_{i-1}$ is the cost of a cache miss if the scheduler causes the instruction x to fetch m from a level-i cache in the PMH. The instruction would need to incur a cache miss at level-i if it is the first instruction within the unique maximal level-i task that accesses a particular memory location. Using this per-instruction cost, we define effective work $\widehat{W}_\alpha^*(.)$ of a task using structural induction in a manner that is deliberately similar to that of $\widehat{Q}_\alpha(.)$.

DEFINITION 4 (LATENCY ADDED COST). *With cost ρ assigned to instructions, the **latency added effective work** of a task* t*, or a strand* s *nested inside a task* t *(from which it inherits space declaration) is:* strand:

$$\widehat{W}_\alpha^*(\mathsf{s}) = s(\mathsf{t})^\alpha \sum_{x \in \mathsf{s}} \rho(x).$$

task: *For task* t *of size between M_i and M_{i+1}, the l.a.e.w. is $\widehat{W}_\alpha^*(\mathsf{t})$, where $\left\lceil \frac{\widehat{W}_\alpha^*(\mathsf{t})}{s(\mathsf{t})^\alpha} \right\rceil =$*

$$\max \begin{cases} \max_{\chi \in chains(\mathsf{t},M)} \left\{ \sum_{\mathsf{t}_i \in \chi} \left\lceil \frac{\widehat{W}_\alpha^*(\mathsf{t}_i)}{s(\mathsf{t}_i)^\alpha} \right\rceil \right\} & \textit{(depth dominated)} \\ \left\lceil \frac{\sum_{\mathsf{t}_i \in maximal(\mathsf{t},M)} \widehat{W}_\alpha^*(\mathsf{t}_i)}{s(\mathsf{t})^\alpha} \right\rceil & \textit{(work dominated)} \end{cases}$$

where $chains(\mathsf{t},M)$ represents the set of chains of dependence edges between M-maximal tasks, $maximal(\mathsf{t},M)$, in the spawn tree of t.

Because of the large number of machine parameters involved ($\{M_i, C_i\}$, $i = 1 \ldots h$ etc.), it is undesirable to compute the latency added work directly for an algorithm. Instead, using induction of the structure of the task in terms of decomposition into strands and maximal tasks, one can show that the latency added effective work can be upper bounded by a sum of per (cache) level machine costs $\widehat{W}_\alpha^{(i)}(\cdot)$ that can, in turn be bounded by machine parameters and ECC of the algorithm. For $i \in [h-1]$, $\widehat{W}_\alpha^{(i)}(\mathsf{t})$ of a task t is computed exactly like $\widehat{W}_\alpha^*(\mathsf{c})$ using a different base case: for each instruction x in c, if the memory access at x costs at least C_i', assign a cost of $\rho_i(x) = C_i$ to that node. Else, assign a cost of $\rho_i(x) = 0$. Further, we set $\rho_0(x) = W(x)$, and define $\widehat{W}_\alpha^{(0)}(\mathsf{c})$ in terms of $\rho_o(\cdot)$. It also follows from these definitions that $\rho(x) = \sum_{i=0}^{h-1} \rho_i(x)$ for all instructions x. With this notation, we have (see [25] for proof):

LEMMA 5. ***Separation Lemma**: On an h-level PMH, and for a parameter $\alpha > 0$, for a task* t *with size at least M_{h-1}, we have:*

$$\left\lceil \frac{\widehat{W}_\alpha^*(\mathsf{b})}{s(\mathsf{t})^\alpha} \right\rceil \leq \left\lceil \frac{\sum_{i=0}^{h-1} \widehat{W}_\alpha^{(i)}(\mathsf{t}).}{s(\mathsf{t})^\alpha} \right\rceil$$

With the separation lemma for the ND model, the proof of Theorem 3 follows from the two lemmas which we adapt from [11]. The first is a bound on the per level latency added effective work term in terms of the effective cache complexity. The second is a bound on the runtime in terms of the latency added effective work using a modified work-span analysis akin to Brent's theorem. The proofs of these two lemmas follow the same arguments as in [11] with minor, but straightforward, modifications that account for the new definition of the ECC in the ND model.

LEMMA 6. *Consider an h-level PMH and a task (or a strand)* t. *If* t *is scheduled on this PMH using a space-bounded scheduler with dilation* $\sigma = 1/3$, *then* $\widehat{W}_{\alpha}^{*}(\mathsf{t}) \leq \sum_{i=0}^{h-1} \widehat{Q}_{\alpha}(\mathsf{t}; M_i/3, B) \cdot C_i$.

LEMMA 7. *Consider an h-level PMH and a task with parallelizability with* α_{max} *that exceeds the parallelism of the PMH by a small constant. Let* $\alpha' = \min\{\alpha_{max}, 1\}$. *Let* N_i *be a task or strand which has been assigned a set* $\mathcal{U}_t$ *of* $q \leq g_i(S(N_i))$ *level-*$(i-1)$ *subclusters by the scheduler. Letting* $\sum_{V \in \mathcal{U}_t}(1-\mu(V)) = r$ *(by definition,* $r \leq |\mathcal{U}_t| = q$*), the running time of* N_i *is at most:*

$$\frac{\widehat{W}_{\alpha}^{*}(N_i)}{rp_{i-1}} \cdot v_i, \textit{where overhead } v_i = 2\prod_{j=1}^{i-1}\left(\frac{1}{k} + \frac{f_i}{(1-k)(M_i/M_{i-1})^{\alpha'}}\right),$$

for some constant $0 < k < 1$.

5. RELATED WORK AND COMPARISON

Nested Parallelism, Complexity and Schedulers. A major advantage of writing algorithms in the NP and ND models is that it exposes its locality (quantified by cache complexity) and parallelism (quantified by work and span) at different scales. This makes it possible to design schedulers that can exploit parallelism and locality in algorithms at different levels of the cache hierarchy. Initial analyses of schedulers for the NP model, such as the randomized work-stealing scheduler [17], were based only on work and span. While such analyses serve as a good indicator of their scalability and load-balancing abilities, better analyses and newer schedulers that minimize both communication costs and load balance in terms of time and cache complexities on various parallel cache configurations have been studied [1, 10, 12, 22, 11]. Many divide-and-conquer parallel cache-oblivious algorithms that can can achieve theoretically optimal bounds on cache complexity, work and span exist [14, 23]. For these NP algorithms, schedulers can achieve optimal bounds on time and communication costs.

Another advantage of the NP (and ND) algorithms is that despite being processor- and cache-oblivious, schedulers execute these algorithms well with minimal tuning; the bounds are fairly robust across cache sizes and processor counts. Tuning of algorithms for time and/or cache complexity has several disadvantages: first, the code structure becomes more complicated; second, the parameter space to explore is usually of exponential size; third, the tuned code is non-portable, i.e., separate tuning is required for different hardware systems; fourth, the tuned code may not be robust to variations and noise in the running environment. Recent work by Bender et al. [8] showed that loop based codes are not cache-adaptive, i.e., when the amount of cache available to an algorithm can fluctuate, which is usually the case in a real-world environment, the performance of tuned loop tiling based can suffer significantly. However, many runtimes and systems (e.g. Halide [41]) that map algorithms such as dense numerical algebra, stencils and memoization algorithms to parallel machines rely heavily on tuning to extract performance.

Futures, Pipelines and other Synchronization Constructs. The limitations of the NP model in expressing parallelism is known in the parallel programming community. Several approaches, such as futures [6, 26] and synchronization variables [13], were proposed to express more general classes of parallel programs.

Conceptually, the *future* construct lets a piece of computation run in parallel with the context containing it. The pointer to future can then be passed to other threads and synchronize at a later point. Several papers have studied the complexity of executing programs with futures. Greiner and Blelloch [29] discuss semantics, cost models and effective evaluations strategies with bounds on the time complexity. Spoonhower et al. [44] calculate tight bounds on the locality of work-stealing in programs with futures. The bounds show that moving from a strict NP model to programs with futures can make WS schedulers pay significant price in terms of locality. To alleviate this problem, Herlihy and Liu [31] suggest that the cache locality of future-parallel programs with work-stealing can be improved by restricting the programs to using "well-structured futures": each future is touched only once, either by the thread that created it, or by a later descendant of the thread that created it. However, it is difficult to express the algorithms in our paper as well-structured futures without losing parallelism or locality. One of the main reasons for this is that the algorithm DAGs we consider have nodes with multiple, even $O(n)$, outgoing dataflow arrows which cannot be easily translated into "touch-once" futures. Even if we were to express such DAGs with touch-once futures, the resultant DAG might be unnecessarily serialized. We seek to eliminate such artificial loss of parallelism with the ND model. Further, the analysis of schedulers for programs with futures is limited to work-stealing, which is a less than ideal candidate for multi-level cache hierarchies. To the best of our knowledge, no provably good hierarchy-aware schedulers for future-parallel programs exist.

Synchronization variables are a more general form of synchronization among threads in "computation DAG" and can be used to implement futures. Blelloch et al. [13] present the *write-once synchronization variable*, which is a variable (memory location) that can be written by one thread and read by any number of other threads. The paper also discusses an online scheduling algorithm for a program with "write-once synchronization variables" with efficient space and time bounds on the CRCW PRAM model with the fetch-and-add primitive.

Though futures or synchronization variables provide a more relaxed form of synchronization among threads in a computation DAG, thus exposing more parallelism, there are some key technical differences between these approaches and the ND model. First, the future construct fails to address the concept of "partial dependencies". A thread computing a future is "parallel", not "partially parallel", to the thread touching the future. The runtime always eagerly creates both threads before the future is computed, thus possibly wasting asymptotically more space and incurring asymptotically more cache misses. In contrast, the "$\rightsquigarrow$" construct allows the runtime the flexibility of creating "sink" tasks as required when partial dependencies are met. Second, there is no existing work on linguistic and runtime support for the recursive construction and refinement of futures over spawn trees. While many dataflow programming models have been studied and deployed in production over the last four decades [33], the automatic recursive construction of dataflow arrows over the spawn tree, which is crucial in achieving locality in a cache- and processor-oblivious fashion, is a new and unique feature of our model. Third, there are algorithms whose maximal parallelism can be easily realized using the "$\rightsquigarrow$" construct but not with futures. In the ND model, it is easy to describe algorithms in which a source can fire multiple sink nodes, and a sink node can depend on multiple sources. Such algorithms with nodes that involve multiple incoming and outgoing dataflow arrows pose problems in future-parallelism models. For instance, programming the LCS algorithm using futures without introducing artificial dependencies is very cumbersome. To eliminate artificial dependencies, this class of problems requires futures to be touched by descendants of the siblings of the node whose descendant created the future; that is, the touching thread may be created before the corresponding future thread is created. To the best of our knowledge, there is no easy scheme to pass the pointer to a future up and down the spawn tree.

Another closely related extension of the nested parallel model is "pipeline parallelism". Pipeline parallelism can be constructed by either futures (as in [15], where it was used to shorten span) or synchronization variables, or by some elegantly defined linguistic constructs [35]. The key idea in pipeline parallelism is to organize a parallel program as a linear sequence of stages. Each stage processes elements of a data stream, passing each processed data element to the next stage, and then taking on a new element before the subsequent stages have necessarily completed their processing. Pipeline parallelism cannot express all the partial dependence patterns described in this paper. To allow the expression of arbitrary DAGs, interfaces for "parallel task graphs" and schedulers for them have been studied [32, 2]. While in principle they can be used to construct computation DAGs that contain arbitrary parallelism, the work flow is more or less similar to dataflow computation without much emphasis on recursion, locality or cache-obliviousness. The same limitation is true of pipeline parallelism as well.

Other algorithms, systems and schedulers. Parallel and cache-efficient algorithms for dynamic programming have been extensively studied (e.g. [22, 27, 37, 45]). These algorithms illustrate algorithms in which it is necessary to have programming constructs that can express multiple (even $O(n)$) dataflows at each node without serialization [27]. The necessity of wavefront scheduling and designs for it have been studied in [37, 45].

Dynamic scheduling in dense numerical linear algebra on shared and distributed memories, as well as GPUs, has been studied in the MAGMA and PLASMA [3], DPLASMA [18], and PaRSEC [19] systems. The programming interface used for these systems is DaGUE [20], which is supported by hierarchical schedulers in runtimes. The DaGUE interface is a slight relaxation of the NP model that allows recursive composition of task DAGs representing dataflow within individual kernels. However, the interface does not capture the notion of partial dependencies. When DAGs of smaller kernels are composed to define larger algorithms, the dependencies are either total or null. The ability to compose kernels with partial dependency patterns is key to the ND model.

The FLAME project [30], and subsequently the Elemental project [40], provides a systematic way of deriving recursions and data dependencies in dense linear algebra from high-level expressions [9], and using them to generate data flow DAG scheduling [21]. The method proposed in these works can be adapted to find the partial dependence patterns derived by hand in this paper.

The Galois system developed at UT Austin [39] proposes a data-centric formulation of algorithms called "operator formulation". This formulation was initiated for handling irregular parallel computation in which data dependencies can change at runtime, and for irregular data structures such as graphs, trees and sets. In contrast, our approach was motivated by more regular parallel computations such as divide-and-conquer algorithms.

Acknowledgements

We thank Prof. James Demmel, Dr. Shachar Itzhaky, Prof. Charles Leiserson, Prof. Armando Solar-Lezama and Prof. Katherine Yelick for valuable discussions and their support in conducting this research. Yuan Tang thanks Prof. Xiaoyang Wang, the dean of Software School and School of Computer Science at Fudan University for general help on research environment. Yuan Tang also thanks Prof. Haibin Kan for generous financial support from grant KRH2301058. We thank the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (DoE ASCR), Applied Mathematics and Computer Science Program, grants DE-SC0010200, DE-SC-0008700, and AC02-05CH11231, for financial support, along with DARPA grant HR0011-12-2-0016, ASPIRE Lab industrial sponsors and affiliates Intel, Google, Huawei, LG, NVIDIA, Oracle, Samsung, and MathWorks.

6. REFERENCES

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. In *Proc. of the 12th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA 2000)*, pages 1–12, 2000.

[2] K. Agrawal, C. E. Leiserson, and J. Sukha. Executing task graphs using work stealing. In *IPDPS*, pages 1–12. IEEE, April 2010.

[3] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The plasma and magma projects. *Journal of Physics: Conference Series*, 180(1):012037, 2009.

[4] B. Alpern, L. Carter, and J. Ferrante. Modeling parallel computers as memory hierarchies. In *Proc. Programming Models for Massively Parallel Computers*, pages 116–123. IEEE Computer Society Press, 1993.

[5] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *SPAA '98*, pages 119–129, June 1998.

[6] H. C. Baker, Jr. and C. Hewitt. The incremental garbage collection of processes. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*, pages 55–59, New York, NY, USA, 1977. ACM.

[7] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Minimizing communication in numerical linear algebra. *SIAM J. Matrix Analysis Applications*, 32(3):866–901, 2011.

[8] M. A. Bender, R. Ebrahimi, J. T. Fineman, G. Ghasemiesfeh, R. Johnson, and S. McCauley. Cache-adaptive algorithms. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pages 958–971, 2014.

[9] P. Bientinesi, J. A. Gunnels, M. E. Myers, E. S. Quintana-Ortí, and R. A. v. d. Geijn. The science of deriving dense linear algebra algorithms. *ACM Trans. Math. Softw.*, 31(1):1–26, Mar. 2005.

[10] G. Blelloch, P. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the ACM*, 46(2):281–321, 1999.

[11] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and H. V. Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 355–366. ACM, 2011.

[12] G. E. Blelloch and P. B. Gibbons. Effectively sharing a cache among threads. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 235–244. ACM, 2004.

[13] G. E. Blelloch, P. B. Gibbons, Y. Matias, and G. J. Narlikar. Space-efficient scheduling of parallelism with synchronization variables. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 12–23, Newport, Rhode Island, June 1997.

[14] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri. Low depth cache-oblivious algorithms. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 189–199, New York, NY, USA, 2010. ACM.

[15] G. E. Blelloch and M. Reid-Miller. Pipelining with futures. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '97, pages 249–259, New York, NY, USA, 1997. ACM.

[16] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. In *Proceedings of the Twenty Fifth Annual ACM Symposium on Theory of Computing*, pages 362–371, San Diego, California, 1993.

[17] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, 46(5):720–748, Sept. 1999.

[18] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. Yarkhan, and J. Dongarra. Distributed dense numerical linear algebra algorithms on massively parallel architectures: Dplasma. Technical Report UT-CS-10-660, University of Tennessee Computer Science, September 2013.

[19] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013.

[20] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. Dague: A generic distributed {DAG} engine for high performance computing. *Parallel Computing*, 38(1-2):37 – 51, 2012. Extensions for Next-Generation Parallel Programming Models.

[21] E. Chan and F. D. Igual. Runtime data flow graph scheduling of matrix computations with multiple hardware accelerators, FLAME Working Note #50, October 2010.

[22] R. Chowdhury, F. Silvestri, B. Blakeley, and V. Ramachandran. Oblivious algorithms for multicores and network of processors. *Journal of Parallel and Distributed Computing (Special issue on best papers from IPDPS 2010, 2011 and 2012)*, 73(7):911–925, 2013. A preliminary version appeared in IPDPS '10.

[23] R. Cole and V. Ramachandran. Efficient resource oblivious algorithms for multicores. *CoRR*, abs/1103.4071, 2011.

[24] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[25] D. Dinh, H. V. Simhadri, and Y. Tang. Extending the nested parallel model to the nested dataflow model with provably efficient schedulers. *CoRR*, abs/1602.04552, 2016.

[26] D. Friedman and D. Wise. Aspects of applicative programming for parallel processing. *Computers, IEEE Transactions on*, C-27(4):289–296, April 1978.

[27] Z. Galil and K. Park. Parallel algorithms for dynamic programming recurrences with more than $O(1)$ dependency. *Journal of Parallel and Distributed Computing*, 21:213–222, 1994.

[28] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162:705–708, 1982.

[29] J. Greiner and G. E. Blelloch. A provably time-efficient parallel implementation of full speculation. *ACM Trans. Program. Lang. Syst.*, 21(2):240–285, Mar. 1999.

[30] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, Dec. 2001.

[31] M. Herlihy and Z. Liu. Well-structured futures and cache locality. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 155–166. ACM, 2014.

[32] T. Johnson, T. A. Davis, and S. M. Hadfield. A concurrent dynamic task graph. *Parallel Comput.*, 22(2):327–333, 1996.

[33] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, Mar. 2004.

[34] I.-T. A. Lee, S. Boyd-Wickizer, Z. Huang, and C. E. Leiserson. Using memory mapping to support cactus stacks in work-stealing runtime systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 411–420, New York, NY, USA, 2010. ACM.

[35] I.-T. A. Lee, C. E. Leiserson, T. B. Schardl, J. Sukha, and Z. Zhang. On-the-fly pipeline parallelism. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 140–151, New York, NY, USA, 2013. ACM.

[36] C. E. Leiserson, T. B. Schardl, and J. Sukha. Deterministic parallel random-number generation for dynamic-multithreading platforms. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 193–204, New York, NY, USA, 2012. ACM.

[37] S. Maleki, M. Musuvathi, and T. Mytkowicz. Parallelizing dynamic programming through rank convergence. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'14, pages 219–232, New York, NY, USA, 2014. ACM.

[38] G. Narlikar. *Space-Efficient Scheduling for Parallel, Multithreaded Computations*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, May 1999.

[39] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 12–25. ACM, 2011.

[40] J. Poulson, B. Marker, R. A. van de Geijn, J. R. Hammond, and N. A. Romero. Elemental: A new framework for distributed memory dense matrix computations. *ACM Trans. Math. Softw.*, 39(2):13:1–13:24, Feb. 2013.

[41] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530. ACM, 2013.

[42] H. V. Simhadri. *Program-Centric Cost Models for Locality and Parallelism*. PhD thesis, CMU, 2013.

[43] H. V. Simhadri, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and A. Kyrola. Experimental analysis of space-bounded schedulers. *Transactions on Parallel Computing*, 3(1), 2016. A preliminary version appeared in SPAA '14.

[44] D. Spoonhower, G. E. Blelloch, P. B. Gibbons, and R. Harper. Beyond nested parallelism: Tight bounds on work-stealing overheads for parallel futures. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 91–100, New York, NY, USA, 2009. ACM.

[45] Y. Tang, R. You, H. Kan, J. J. Tithi, P. Ganapathi, and R. A. Chowdhury. Cache-oblivious wavefront: Improving parallelism of recursive dynamic programming algorithms without losing cache-efficiency. In *PPoPP'15*, San Francisco, CA, USA, Feb.7 – 11 2015.