Trade-Offs Between Synchronization, Communication, and Computation in Parallel Linear Algebra Computations

EDGAR SOLOMONIK, ERIN CARSON, NICHOLAS KNIGHT, and JAMES DEMMEL,
University of California, Berkeley

This article derives trade-offs between three basic costs of a parallel algorithm: synchronization, data movement, and computational cost. These trade-offs are lower bounds on the execution time of the algorithm that are independent of the number of processors but dependent on the problem size. Therefore, they provide lower bounds on the execution time of any parallel schedule of an algorithm computed by a system composed of any number of homogeneous processors, each with associated computational, communication, and synchronization costs. We employ a theoretical model that measures the amount of work and data movement as a maximum over that incurred along any execution path during the parallel computation. By considering this metric rather than the total communication volume over the whole machine, we obtain new insights into the characteristics of parallel schedules for algorithms with nontrivial dependency structures. We also present reductions from BSP and LogGP algorithms to our execution model, extending our lower bounds to these two models of parallel computation. We first develop our results for general dependency graphs and hypergraphs based on their expansion properties, and then we apply the theorem to a number of specific algorithms in numerical linear algebra, namely triangular substitution, Cholesky factorization, and stencil computations. We represent some of these algorithms as families of dependency graphs. We derive their communication lower bounds by studying the communication requirements of the hypergraph structures shared by these dependency graphs. In addition to these lower bounds, we introduce a new communication-efficient parallelization for stencil computation algorithms, which is motivated by results of our lower bound analysis and the properties of previously existing parallelizations of the algorithms.

CCS Concepts: • **Mathematics of computing** → **Computations on matrices**; • **Theory of computation** → *Communication complexity*; *Massively parallel algorithms;*

Additional Key Words and Phrases: Communication lower bounds, graph expansion, numerical linear algebra, stencil computations

ACM Reference Format:
Edgar Solomonik, Erin Carson, Nicholas Knight, and James Demmel. 2016. Trade-offs between synchronization, communication, and computation in parallel linear algebra computations. ACM Trans. Parallel Comput. 3, 1, Article 3 (June 2016), 47 pages.
DOI: http://dx.doi.org/10.1145/2897188

The first author was supported by a DOE computational science graduate fellowship (DE-FG02-97ER25308) and an ETH Zurich postdoctoral fellowship. This material is based on work supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under awards DE-SC0004938, DE-SC0003959, and DE-SC0010200; by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research, X-Stack program under awards DE-SC0005136, DE-SC0008700, and AC02-05CH11231; and by DARPA award HR0011-12-2-0016.
Authors' addresses: E. Solomonik, Department of Computer Science, ETH Zurich, CAB, Universitätsstrasse 6, 8092 Zürich, Switzerland; email: solomonik@inf.ethz.ch; E. Carson and N. Knight, Soda Hall, Department of Computer Science, University of California at Berkeley, Berkeley CA 94720-1776; emails: {ecc2Z, knight}@cs.berkeley.edu; J. Demmel, Soda Hall, Department of Mathematics and Department of Computer Science, University of California at Berkeley, Berkeley CA 94720-1776; email: demmel@cs.berkeley.edu.

DOI: http://dx.doi.org/10.1145/2897188

1. INTRODUCTION

The parallelization of algorithms requires the partitioning and scheduling of the work and coordination between processors via data movement and synchronization. To exploit more parallelism, it is often necessary to communicate more data per unit of work or perform more frequent synchronizations between processors. This article formally analyzes the trade-offs between the computation, communication, and synchronization costs in algorithms by representing the algorithms via dependency graphs and establishing lower bounds for the cost of possible schedules for these dependency graphs. We present a general technique for deriving cost trade-offs associated with the parallel execution of a dependency graph by analyzing the rate of expansion of the size of dependency intervals and of the minimum vertex separator size of the subgraphs induced by the dependency intervals. We apply the lower bounds to algorithms in dense linear algebra as well as stencil computations. We employ hypergraphs to simultaneously treat families of dependency graphs associated with linear algebra algorithms. This allows us to obtain lower bounds on vertex separators of the dependency graphs of all algorithms that we consider in a general fashion.

We model a parallel machine as a network of processors, each with a main memory of size M and a cache of size $\hat{M}$, which communicate via asynchronous point-to-point messages and collective (although not necessarily global) synchronizations. This model has four basic architectural parameters:

—α = network latency, time for a synchronization between two or more processors,
—β = time to inject a word of data into (or extract it from) the network,
—γ = time to perform a floating point operation on in-cache data, and
—ν = time to perform a transfer of a word of data between cache and main memory.

We define a schedule to be a directed acyclic graph (DAG), $\bar{G}$, in which each vertex is a computation, communication, or synchronization task with an associated cost in terms of these coefficients. The overall cost of the schedule, $T(\bar{G})$ is given by the maximal sum of costs of tasks on some path in the schedule DAG. The sequence of executions done locally by any processor corresponds to one such execution path in the schedule, so the schedule cost is at least as large as that incurred by any single processor during the execution of the schedule.

We can decompose the execution time in terms of the four coefficients as

$$T(\bar{G}) = \alpha \cdot S + \beta \cdot W + \gamma \cdot F + \nu \cdot \hat{W},$$

where we introduced four algorithmic cost components:

—S = number of synchronizations (network latency cost),
—W = number of words of data moved across the network (interprocessor bandwidth cost/communication cost),
—F = number of in-cache floating point operations performed (computational cost), and
—$\hat{W}$ = number of words of data moved between main memory and cache (memory bandwidth cost).

If communication is overlapped with computation, the execution time may correspond to a maximum rather than a sum of these four components; however, this does not bear relevance for our asymptotic analysis. In Section 3, we describe our execution schedule model and show how S, W, F, and $\hat{W}$ are measured in more detail. We also show that both LogGP [Alexandrov et al. 1995] and BSP [Valiant 1990] schedules may be efficiently translated to our scheduling model (under some assumptions on the LogGP cost parameters).

We represent algorithms as dependency graphs (or as families of dependency graphs) with vertices corresponding to inputs, intermediate elements, and outputs. We represent a parallelization of the algorithm as a coloring of its dependency graph. This representation precludes recomputation of intermediate values by different processors, as each intermediate value is assigned a single color. For the same reason, inputs to the algorithm exist on only a single starting processor. The use of replicated inputs or recomputed intermediates therefore yields a different algorithm rather than a different parallelization in our representation. For consistency with the literature, when referring to previous work, we will sometimes use the term *algorithm* in reference to schedules/parallelizations.

In Section 4, we introduce the formalism of dependency chains and dependency intervals, which are essential to our analysis of dependency graphs. Dependency chains correspond to noncontiguous subsequences of nodes in a path in the dependency graph. Dependency intervals are closed intervals in the partial order defined by the dependency graph. They are defined by a starting vertex and an end vertex, and they correspond to the set of vertices that are dependent on the starting vertex and are dependencies of the end vertex.[1] Our main lower bound result is for dependency graphs that contain a dependency chain of length n, for which there exists a constant $\omega \in \{2, 3, \ldots\}$ such that for all pairs of vertices that are at distance b from one another along the dependency chain,

—the dependency interval between these two vertices is of size $\Theta(b^\omega)$, and
—the subgraph induced by this dependency interval has a minimum balanced vertex separator of size $\Omega(b^{\omega-1})$.

If the dependency graph satisfies the preceding conditions, we show that the execution costs of any schedule for this dependency graph must satisfy

$$F \cdot S^{\omega-1} = \Omega\left(n^\omega\right), \qquad W \cdot S^{\omega-2} = \Omega(n^{\omega-1}).$$

A key requirement on the dependency graphs to which the cost lower bounds of Section 4 apply is the existence of a lower bound on the size of the minimum vertex separator of the dependency graph and some of its subgraphs. Therefore, to apply the lower bounds to a variety of algorithms with different dependency graphs, we introduce additional machinery for deriving lower bounds on minimum vertex separator size. We discuss volumetric inequalities in Section 5, which yield lower bounds on the size of projections of vertex sets. These projections are closely associated with the sets of dependencies of subgraphs in the dependency graphs of numerous algorithms. We primarily rely on a generalized version of the Loomis-Whitney inequality [Loomis and Whitney 1949] but give our own specialization to vertex sets enumerated as tuples of increasing positive integers.

The volumetric inequalities are then used in Section 6.1 to derive lower bounds on the minimum hyperedge cut of lattice hypergraphs. Hypergraphs are a generalization of graphs where each hyperedge can contain more than two vertices. The lattice hypergraphs we analyze are defined on a set of vertices that are enumerable on a lattice of order m and a set of hyperedges, each of which contains all vertices that share $r < m$ indices. The lower bound on the minimum hyperedge cuts of these hypergraphs allow derivation of lower bounds of the minimum vertex separator of families of dependency graphs, which share the same "parent hypergraph," a notion defined in Section 6.2.

We then use this machinery to derive lower bounds on S, W, F for several algorithms in numerical linear algebra that correspond to a set (family) of dependency graphs

[1]When A depends on B, we refer to A as dependent and B as dependency, contrary to some definitions of *dependency*.

rather than a single dependency graph. In particular, we demonstrate lower bounds on standard algorithms for the following problems:

(1) solving a dense n-by-n triangular system by substitution (TRSV),

$$F_{\text{TRSV}} \cdot S_{\text{TRSV}} = \Omega(n^2),$$

which is attainable as discussed in Section 7.1.2,

(2) computing the Cholesky factorization of a dense symmetric positive-definite n-by-n matrix,

$$F_{\text{Ch}} \cdot S_{\text{Ch}}^2 = \Omega(n^3), \qquad W_{\text{Ch}} \cdot S_{\text{Ch}} = \Omega(n^2),$$

which are also attainable as shown in Section 7.2.2, and

(3) computing s applications of a $(2m+1)^d$-point stencil (defined in Section 8.1),

$$F_{\text{St}} \cdot S_{\text{St}}^d = \Omega(m^{2d} \cdot s^{d+1}), \qquad W_{\text{St}} \cdot S_{\text{St}}^{d-1} = \Omega(m^d \cdot s^d),$$

which are again attainable (Section 9).

The lower bound analysis for iterative stencil computations suggests a trade-off between synchronization and interprocessor bandwidth cost. However, known algorithms that lower the synchronization cost for this problem additionally obtain a lower memory bandwidth cost [Demmel et al. 2008]. In Section 9.1, we demonstrate that analogous improvements in memory bandwidth cost may be attained while also keeping a low interprocessor bandwidth cost (and consequently a high synchronization cost).

The Cholesky factorization lower bounds that we derive in this article suggest the communication optimality of the parallel algorithms for LU factorization given by Tiskin [2002] and Solomonik and Demmel [2011]. However, the two LU algorithms given by Tiskin [2002] employ either triangular matrix inversion or pairwise pivoting, which lead to dependency graphs that are not in the family that we consider. Further, the use of Strassen's algorithm within LU factorization as proposed in Tiskin [2002] (while being outside of the domain of the algorithms that we consider) yields a cost that is tight with respect to our lower bound on communication cost, $W_{\text{Ch}} \cdot S_{\text{Ch}} = \Omega(n^2)$.

In Solomonik and Demmel [2011], a lower bound proof was given that demonstrated this trade-off between communication and synchronization costs for LU factorization. However, the proof argument in Solomonik and Demmel [2011] overlooked the possibility of concurrent communication of the dependencies for different blocks and therefore was incorrect. This article extends the idea of this trade-off to a more general theoretical context and presents a fixed and strengthened proof in this new framework. We show that Cholesky factorization is just one of many problems for which standard algorithms have a dependency structure that necessitates the trade-off. We conjecture that our results extend to algorithms for other problems, such as dense matrix factorizations like those for LU and QR factorization; graph algorithms as in the work of Floyd [1962] and Warshall [1962] for all-pairs shortest paths; and Bellman [1956] and Ford [1956] for single-source shortest paths, as well as some dynamic programming algorithms.

2. PREVIOUS WORK

Theoretical lower bounds on communication volume and synchronization are often parameterized by the size of the cache $\hat{M}$ in the sequential setting or the size of the main memory M in the parallel setting. Most previous work has considered the total sequential or parallel communication volume Q, which corresponds to the amount of data movement across the network (by all processors) or through the memory hierarchy. Hong and Kung [1981] introduced sequential communication volume lower bounds for

computations including n-by-n matrix multiplication, $Q_{\text{MM}} = \Omega(n^3/\sqrt{\hat{M}})$, the n-point FFT, $Q_{\text{FFT}} = \Omega(n\log(n)/\log(\hat{M}))$, and the order ω diamond DAG (a Cartesian product of path graphs of length n), $Q_{\text{dmd}} = \Omega(n^\omega/\hat{M}^{1/(\omega-1)})$. Parallel lower bounds for the FFT were investigated for the BSP model by Bilardi et al. [2012]. Irony et al. [2004] introduced memory-dependent lower bounds for distributed-memory matrix multiplication on p processors, obtaining the bound $W_{\text{MM}} = \Omega(n^3/(p\sqrt{M}))$. Aggarwal et al. [1990] proved a version of the memory-independent lower bound $W_{\text{MM}} = \Omega(n^2/p^{2/3})$. Scquizzato and Silvestri [2014] also gave similar memory-independent communication lower bounds for the BSP model for matrix multiplication and stencil computations. Ballard et al. [2012] explored the relationship between these memory-dependent and memory-independent lower bounds. Ballard et al. [2011] extended the results for matrix multiplication to Gaussian elimination of n-by-n matrices and many other matrix algorithms with similar structure, finding

$$W_{\text{Ch}} = \Omega\left(\frac{n^3}{p\sqrt{M}} + \frac{n^2}{p^{2/3}}\right).$$

Bender et al. [2010] extended the sequential communication lower bounds introduced in Hong and Kung [1981] to sparse matrix-vector multiplication. This lower bound is relevant to our analysis of iterative stencil computation algorithms, which correspond to iterative sparse matrix-vector multiplications. However, Bender et al. [2010] used a sequential memory hierarchy model and established bounds in terms of main memory size and track (cacheline) size, whereas we focus on interprocessor communication.

Papadimitriou and Ullman [1987] demonstrated trade-offs for the order 2 diamond DAG (a slight variant of that considered in Hong and Kung [1981]). They proved that the amount of computational work F_{dmd} along some execution path (in their terminology, *execution time*) is related to the communication volume Q_{dmd} and synchronization cost S_{dmd} as

$$F_{\text{dmd}} \cdot Q_{\text{dmd}} = \Omega(n^3) \text{ and } F_{\text{dmd}} \cdot S_{\text{dmd}} = \Omega(n^2).$$

These trade-offs imply that to decrease the amount of computation done along the critical path of execution, more communication and synchronization must be performed. For instance, if a schedule has an "execution time" cost of $F_{\text{dmd}} = \Omega(nb)$, it requires $S_{\text{dmd}} = \Omega(n/b)$ synchronizations and a communication volume of $Q_{\text{dmd}} = \Omega(n^2/b)$. The lower bound on $F_{\text{dmd}} \cdot S_{\text{dmd}}$ is a special case of the order ω dependency interval latency lower bound trade-off that we derive in the next section, with $\omega = 2$. These diamond DAG trade-offs were also demonstrated by Tiskin [1998].

Bampis et al. [1996] considered finding the optimal schedule (and number of processors) for computing order ω grid graphs, similar in structure to those that we consider in Section 6. Their work was motivated by Papadimitriou and Ullman [1987] and took into account dependency graph structure and communication, modeling the cost of sending a word between processors as equal to the cost of a computation. Trade-offs in parallel schedules have also been studied in the context of data locality on mesh network topologies by Bilardi and Preparata [1999].

We will introduce lower bounds that relate synchronization to computation and data movement along sequences of dependent execution tasks. Our work is most similar to the approach in Papadimitriou and Ullman [1987]; however, we obtain bounds on W (the parallel communication volume along some dependency path) rather than Q (the total communication volume). Whereas bounds on Q translate to bounds on the energy necessary to perform the computation, bounds on W translate to bounds on execution time. Our theory also obtains trade-off lower bounds for a more general

set of dependency graphs, which allows us to develop lower bounds for a wider set of computations.

3. THEORETICAL COST MODEL

We first introduce some notational conventions that we employ throughout the article: sets are denoted by uppercase letters (S, V); vectors are denoted by lowercase boldface letters ($\mathbf{v}$), and the ith element of $\mathbf{v}$ is indexed as v_i; accordingly, matrices and tensors are denoted by uppercase boldface letters ($\mathbf{A}$, $\mathbf{T}$), with elements A_{ij}, T_{ijk}. We denote a discrete integer interval $\{1, \ldots, n\}$ by $[1, n]$.

The *dependency graph* of an algorithm is a DAG, $G = (V, E)$. The vertices $V = I \cup Z \cup O$ correspond to either input values I (the vertices with indegree zero), or the results of (distinct) operations, in which case they are either temporary (or intermediate) values Z, or outputs O (including all vertices with outdegree zero). There is an edge $(u, v) \in E \subset V \times (Z \cup O)$ if v is computed from u (so a k-ary operation resulting in v would correspond to a vertex with indegree k). These edges represent data dependencies and impose limits on the parallelism available within the computation. For instance, if the dependency graph $G = (V, E)$ is a path graph with $V = \{v_1, \ldots, v_n\}$ and $E = \{(v_1, v_2), \ldots, (v_{n-1}, v_n)\}$, the computation is entirely sequential, and a lower bound on the execution time is the time that it takes a single processor to compute $F = n - 1$ operations.

The dependency graph encodes an algorithm's execution independently from the particular input values. Not all algorithms are fully representable as a single such dependency graph; however, all algorithms that we consider may be represented as a family of dependency graphs. Using volumetric inequalities and hypergraph analysis, we will derive lower bounds on the computation and communication cost of any execution of any dependency graph in these families. In the following sections, we develop a formal model of a parallel schedule in detail and show that our construction is closely related to the BSP and the LogGP models.

3.1. Parallel Execution Model

A *parallelization* of a dependency graph $G = (V, E)$ corresponds to a coloring—that is, a partition of the vertices into p disjoint sets $C_1, \ldots, C_p$ where $V = \bigcup_{i=1}^{p} C_i$ and processor i computes $C_i \cap (Z \cup O)$. We require that in any parallel execution among p processors, at least two processors compute at least $\lfloor |Z \cup O|/p \rfloor$ elements each; this minor assumption is necessary to avoid the case of a single processor executing the dependency graph sequentially (without parallel communication). We will later make further problem-specific restrictions that require that no processor computes more than some (problem-dependent) constant fraction of $Z \cup O$. Any vertex v of color i ($v \in C_i$) must be communicated to a different processor j if there is an edge from v to a vertex in C_j, although there need not necessarily be a message going directly between processor i and j, as the data can move through intermediate processors. We define each processor's *communicated set* as

$$T_i = \Xi(V, C_i, E) \equiv \left\{u : (u, w) \in \left[(C_i \times (V \backslash C_i)) \cup ((V \backslash C_i) \times C_i)\right] \cap E\right\}.$$

We note that each T_i is a vertex separator in G between $C_i \backslash T_i$ and $V \backslash (C_i \cup T_i)$.

We define a (*parallel*) *schedule* a dependency graph $G = (V, E)$ to be a DAG $\bar{G} = (\bar{V}, \bar{E})$, which consists of a set of p edge-disjoint (but not necessarily vertex disjoint) execution paths, Π, where the vertices in each $\pi \in \Pi$ correspond to the tasks executed by a certain processor. The edges along each processor's execution path carry the state of the cache and main memory of that processor. Along with $\bar{G}$, a schedule is associated with a partition $\{\bar{V}_{\text{comp}}, \bar{V}_{\text{sync}}, \bar{V}_{\text{send}}, \bar{V}_{\text{recv}}, \bar{V}_{\text{trsf}}\}$ of $\bar{V}$ and functions $\hat{f}, \hat{s}, \hat{r}, \hat{h}, \hat{c}, \tilde{m}, \tilde{s}, \tilde{r}, \tilde{h}, \tilde{c}$, defined later. (In our notation, functions with hats have domain

$\bar{V}$, whereas functions with tildes have domain $\bar{E}$.) Each vertex $\bar{v} \in \bar{V}$ corresponds to a unique type within the following types of tasks:

- $\bar{v} \in \bar{V}_{\text{comp}}$: the computation of $\hat{f}(\bar{v}) \subset V$,
- $\bar{v} \in \bar{V}_{\text{sync}}$: a synchronization point,
- $\bar{v} \in \bar{V}_{\text{send}}$: the sending of a message $\hat{s}(\bar{v}) \subset V$,
- $\bar{v} \in \bar{V}_{\text{recv}}$: the reception of a message $\hat{r}(\bar{v}) \subset V$, and
- $\bar{v} \in \bar{V}_{\text{trsf}}$: the transmission of a transfer buffer $\hat{h}(\bar{v}) \subset V$ between cache and main memory (in either direction).

We assign a unique color (processor number) i using function $\tilde{c}$ to the edges along each execution path π so that for every edge $\bar{e} \in (\pi \times \pi) \cap \bar{E}$, $\tilde{c}(\bar{e}) = i$. Each vertex $\bar{u} \in \bar{V}_{\text{comp}} \cup \bar{V}_{\text{send}} \cup \bar{V}_{\text{recv}}$ should be adjacent to at most one incoming edge $\bar{e}_i$ and at most one outgoing edge $\bar{e}_o$. We assign $\bar{u}$ the same color $\hat{c}(\bar{u}) = \tilde{c}(\bar{e}_i) = \tilde{c}(\bar{e}_o)$ as the processor that executes it (the single execution path that goes through it). Each vertex $\bar{v} \in \bar{V}_{\text{sync}}$ corresponds to a synchronization of $k \in [2, p]$ processors and should have k incoming as well as k outgoing edges (each of the k on a different execution path). With every edge $\bar{e} \in \bar{E}$, we associate

—the data kept by processor $\tilde{c}(\bar{e})$ in main memory as $\tilde{m}(\bar{e}) \subset V$,
—the data a processor keeps in cache between tasks as $\tilde{h}(\bar{e}) \subset V$,
—the data kept in the send buffer as a collection of sets $\tilde{s}(\bar{e}) = \{W_1, W_2, \ldots\}$ where each $W_i \subset V$ denotes the contents of a sent point-to-point message posted at some node $\bar{u}$ with $\hat{s}(\bar{u}) = W_i$, and
—the data kept in the receive buffer as a collection of sets $\tilde{r}(\bar{e}) = \{W'_1, W'_2, \ldots\}$ where each $W'_i \subset V$ denotes the contents of a point-to-point message received at some node $\bar{v}$ with $\hat{r}(\bar{v}) = W'_i$.

The schedule has p indegree-zero vertices corresponding to startup tasks $\bar{I} \subset \bar{V}_{\text{comp}}$—that is, "no-op" computations that we assume have no cost. The single outgoing edge of each startup task is assigned a disjoint part of the input data so that $\bigcup_{\bar{e} \in (\bar{I} \times \bar{V}) \cap \bar{E}} \tilde{m}(\bar{e}) = I$. We assume that the cache starts out empty, so $\tilde{h}(\bar{e}) = \emptyset$ for any $\bar{e} \in (\bar{I} \times \bar{V}) \cap \bar{E}$.

In our model, each message is point-to-point in the sense that it has a single originating and destination processor. However, multiple messages may be transferred via the same synchronization vertex. We say that the schedule $\bar{G}$ is *point-to-point* if each synchronization vertex has no more than two execution paths going through it. Each message is sent asynchronously, but a synchronization is required before the communicated data may be used by the receiving processor. We illustrate this by an example schedule (Figure 1), which demonstrates the messaging behavior within our model. Reads from main memory must be done to move data into cache prior to computation tasks, and writes of computed data to main memory are necessary prior to interprocessor communication of this data.

We enforce the validity of the schedule via the following constraints. For every task $\bar{v}$ executed by processor i, with incoming edge $\bar{e}_{\text{in}}$ and outgoing edge $\bar{e}_{\text{out}}$,

—if $\bar{v} \in \bar{V}_{\text{comp}} \backslash \bar{I}$ (for all computation tasks),
  (1) $(\forall f \in \hat{f}(\bar{v}))(\forall (g, f) \in E)$, $g \in \hat{f}(\bar{v})$ or $g \in \tilde{h}(\bar{e}_{\text{in}})$ (meaning that all dependencies must either be in cache when the task is executed or must be computed by the task),
  (2) $\tilde{h}(\bar{e}_{\text{out}}) \subset \hat{f}(\bar{v}) \cup \tilde{h}(\bar{e}_{\text{in}})$ (meaning that the cache can carry data computed during the preceding task or the data previously existing in the cache), and

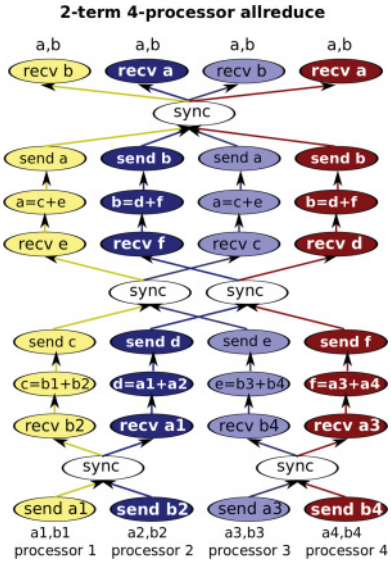

Fig. 1. Depiction of a sample four-processor execution schedule in our model for an all-reduction, which computes $a = \sum_i a_i$ and $b = \sum_i b_i$. Vertices for transfer of data between main memory and cache would also be needed between each send or receive task and a computation task. Not shown are the startup vertices, which would immediately precede each of the botsttom (send) vertices.

(3) $\tilde{m}(\bar{e}_{\text{out}}) = \tilde{m}(\bar{e}_{\text{in}})$ (meaning that the contents of main memory are the same before and after the execution of the task);

—if $\bar{v} \in \bar{V}_{\text{send}}$ (for all tasks that send messages),

(1) $\exists \bar{w} \in \bar{V}_{\text{recv}}$ with $\hat{s}(\bar{v}) = \hat{r}(\bar{w})$ and an execution path from $\bar{v}$ to $\bar{w}$ in $\bar{G}$ (meaning that the task sending the message is connected by an execution path to the task that receives the message),

(2) $\hat{s}(\bar{v}) \subset \tilde{m}(\bar{e}_{\text{in}})$ and $\tilde{s}(\bar{e}_{\text{out}}) = \tilde{s}(\bar{e}_{\text{in}}) \cup \{\hat{s}(\bar{v})\}$ (meaning that what is sent must be a subset of what is stored in main memory when the task is executed and becomes a part of what is stored in the send buffer after the task's execution),

(3) $\tilde{m}(\bar{e}_{\text{out}}) \subset \tilde{m}(\bar{e}_{\text{in}})$ (meaning that what is contained in main memory after the task is executed must be a subset of what was in main memory prior to its execution), and

(4) $\tilde{h}(\bar{e}_{\text{out}}) = \tilde{h}(\bar{e}_{\text{in}})$ (meaning that the contents of the cache are the same before and after the execution of the task);

—if $\bar{v} \in \bar{V}_{\text{recv}}$ (for all tasks that receive messages),

(1) $\hat{r}(\bar{v}) \in \tilde{r}(\bar{e}_{\text{in}})$ and $\tilde{r}(\bar{e}_{\text{out}}) = \tilde{r}(\bar{e}_{\text{in}}) \backslash \{\hat{r}(\bar{v})\}$ (meaning that what is received must be a subset of the receive buffer when the task is executed, but not after its execution),

(2) $\tilde{m}(\bar{e}_{\text{out}}) \subset \tilde{m}(\bar{e}_{\text{in}}) \cup \hat{r}(\bar{v})$ (meaning that the contents of main memory after the execution of the task may contain the data received by the task), and

(3) $\tilde{h}(\bar{e}_{\text{out}}) = \tilde{h}(\bar{e}_{\text{in}})$ (meaning that the contents of the cache are the same before and after the execution of the task);

—if $\bar{v} \in \bar{V}_{\text{trsf}}$ (for all tasks that transfer data in either direction between cache and main memory),

(1) $\hat{h}(\bar{v}) \subset \tilde{m}(\bar{e}_{\text{in}}) \cup \tilde{h}(\bar{e}_{\text{in}})$ (meaning that all elements of the transfer buffer are either in main memory or in cache when the task is executed),

(2) $\tilde{m}(\bar{e}_{\text{out}}) \subset \tilde{m}(\bar{e}_{\text{in}}) \cup \hat{h}(\bar{v})$ (meaning that the main memory may contain any elements from the transfer buffer after the execution of the task),

(3) $\tilde{h}(\bar{e}_{\text{out}}) \subset \tilde{h}(\bar{e}_{\text{in}}) \cup \hat{h}(\bar{v})$ (meaning that the cache may contain any elements from the transfer buffer after the execution of the task); and

—if $\bar{v} \in \bar{V}_{\text{sync}}$ (for all synchronization tasks), messages may pass from send to receive buffers under the following rules:

(1) if processor i posted a send $\bar{u}_i \in \bar{V}_{\text{send}}$ and processor j posted the matching receive $\bar{u}_j \in \bar{V}_{\text{recv}}$ (i.e., $\hat{s}(\bar{u}_i) = \hat{r}(\bar{u}_j)$), and if there exist execution paths $\pi_i = \{\bar{u}_i, \ldots, \bar{w}_i, \bar{v}\}$ and $\pi_j = \{\bar{v}, \bar{w}_j, \ldots, \bar{u}_j\}$ in $\bar{G}$, they may exchange the message during the execution of task $\bar{v}$ by moving the data $\hat{s}(\bar{u}_i)$ from $\tilde{s}((\bar{w}_i, \bar{v}))$ to $\tilde{r}((\bar{v}, \bar{w}_j))$,

(2) $\bigcup_{(\bar{u},\bar{v})\in\bar{E}} \tilde{s}((\bar{u}, \bar{v}))\setminus\bigcup_{(\bar{v},\bar{w})\in\bar{E}} \tilde{s}((\bar{v}, \bar{w})) = \bigcup_{(\bar{v},\bar{w})\in\bar{E}} \tilde{r}((\bar{v}, \bar{w}))\setminus\bigcup_{(\bar{u},\bar{v})\in\bar{E}} \tilde{r}((\bar{v}, \bar{u}))$ (meaning that if the data on one processor's execution path is exchanged in a synchronization node, it should be removed from the send buffer after the task's execution), and

(3) $\tilde{c}((\bar{u}, \bar{v})) = \tilde{c}((\bar{v}, \bar{w})) \Rightarrow \tilde{m}((\bar{u}, \bar{v})) = \tilde{m}((\bar{v}, \bar{w}))$ and $\tilde{h}((\bar{u}, \bar{v})) = \tilde{h}((\bar{v}, \bar{w}))$ (meaning that the contents of main memory and cache are the same before and after the execution of the task).

For all edges $\bar{e} \in \bar{E}$, the cache contents must not exceed the cache size, $|\tilde{h}(\bar{e})| \leq \hat{M}$, and the contents of main memory must not exceed the main memory size, $|m(\bar{e})| \leq M$. Any set of elements may be discarded from cache during a computation or memory transfer task and may be discarded from main memory during a message reception or memory transfer task.

The runtime of the schedule

$$T(\bar{G}) = \max_{\pi\in\Pi} \sum_{\bar{v}\in\pi} \hat{t}(\bar{v})$$

is maximum total weight of any execution path in the schedule, where each vertex is weighted according to its task's cost,

$$\hat{t}(\bar{v}) = \begin{cases} \gamma \cdot |\hat{f}(\bar{v})| & \bar{v} \in \bar{V}_{\text{comp}} \\ \alpha & \bar{v} \in \bar{V}_{\text{sync}} \\ (\nu + \beta) \cdot |\hat{s}(\bar{v})| & \bar{v} \in \bar{V}_{\text{send}} \\ (\nu + \beta) \cdot |\hat{r}(\bar{v})| & \bar{v} \in \bar{V}_{\text{recv}} \\ \nu \cdot |\hat{h}(\bar{v})| & \bar{v} \in \bar{V}_{\text{trsf}} \end{cases}.$$

These definitions of T and $\hat{t}$ clarify how we compute the cost measures $S, W, F, \hat{W}$ introduced in Section 1: the cost $\sum_{\bar{v}\in\pi} \hat{t}(\bar{v})$ along any $\pi \in \Pi$ can be written as $\alpha \cdot S + \beta \cdot W + \gamma \cdot F + \nu \cdot \hat{W}$ by matching coefficients. The runtime $T(\bar{G})$ is by construction greater than the time that it takes for any processor to execute its computation and communication operations, as these form an execution path through the schedule. We require all sent and received data to incur a cost of $\nu + \beta$ per element of the message transferred from main memory to the send buffer or from the receive buffer to main memory. Since all data from the receive buffer arrives from remote processors and all data from the sent buffer is forwarded to remote processors, at least β network cost is incurred by the sending and receiving processor per word of data in the message. Whether this data also needs to go through the memory hierarchy depends on the particular network card and memory architecture; however, in general, we can expect that $\beta > \nu$—that is, interprocessor communication is more expensive than memory traffic. Therefore, the $\nu + \beta$ cost does not significantly overestimate the β cost and allows us to make the simplifying assumption that $\hat{W} \geq W$. Thus, from here on, we specifically consider $\hat{W}$ only when $\hat{W} > W$. In addition, note that our construction allows us to ignore idle time, as a processor can only be idle at a message node if there exists a costlier execution path to that node.

3.2. Relation to Existing Models

Our theoretical model is closest to, and can efficiently simulate, the LogGP model [Alexandrov et al. 1995], a generalization of the LogP model [Culler et al. 1993]. The LogGP model differs from our model most notably by inclusion of parameters that allow overlap of communication and computation by measuring the overhead of each processor as well as the time for the message to travel through the network. The cost parameters in the LogGP model are

—L = interprocessor latency cost incurred on network,
—o = messaging overhead incurred by sending and receiving processes,
—g = injection rate of different messages, and
—$\hat{G}$ = inverse bandwidth (per byte cost) of a message to a single processor.

Both models are asynchronous and measure the cost of a schedule along the longest execution path, so a close relationship is expected. Since our model only has a single latency parameter α, whereas the LogGP model considers three parameters that affect latency—overhead o, injection rate g, and interprocessor latency L—we will consider the special case where $g \leq L = o$ (i.e., the sequential overhead of sending a message is equivalent to the network latency of message delivery, and the injection rate is trivial (no greater than the overhead)). The β parameter in our model corresponds to the $\hat{G}$ parameter in the LogGP model (we use $\hat{G}$ rather than the standard G to disambiguate from graphs).

THEOREM 3.1. *If there exists a schedule for dependency graph G with LogGP latency cost $L \cdot S$ and bandwidth cost $\hat{G} \cdot W$, then there exists a point-to-point parallel schedule $\bar{G}$ with synchronization cost $O(\alpha \cdot S)$ and bandwidth cost $O(\beta \cdot W)$.*

PROOF. Consider the given schedule for G, which encodes a time line of LogGP actions for each processor. We now construct an equivalent schedule $\bar{G} = (\bar{V}, \bar{E})$. Consider the kth (LogGP) action of the ith processor, which is either a local computation, a sent message, or a received message. For each computation, add a sequence of computation vertices to $\bar{V}_{\text{comp}} \subset \bar{V}$ and memory transfer vertices (the cost of which was not bounded by the LogGP schedule and so is also not bounded in $\bar{G}$) as necessary to move dependencies between cache and main memory. If the kth action of the ith processor is a send of dataset U that is received as the lth action of processor j, add nodes $\bar{v}_1$, $\bar{s}$, and $\bar{v}_2$ to $\bar{V}$, where

—$\bar{v}_1 \in \bar{V}_{\text{send}}$, $\hat{s}(\bar{v}_1) = U$, $\hat{c}(v_1) = i$, and $\bar{v}_1$ succeeds the $(k-1)$th action of processor i and precedes $\bar{s}$;
—$\bar{s} \in \bar{V}_{\text{sync}}$ and $\bar{s}$ has an incoming edge from $\bar{v}_1$ as well as from the $(l-1)$th action of processor j, and outgoing edges to $\bar{v}_2$ and the $(k+1)$th action of processor i; and
—$\bar{v}_2 \in \bar{V}_{\text{recv}}$, $\hat{r}(\bar{v}_2) = U$, and $\bar{v}_2$ is of color j with an incoming edge from $\bar{s}$.

Since each synchronization node has two execution paths through it, $\bar{G}$ is a point-to-point schedule. A runtime bound on the LogGP schedule provides us with a bound on the cost of any execution path in $\bar{G}$, as any execution path in the LogGP schedule exists in $\bar{G}$ and vice versa. Every adjacent pair of actions on this execution path will be done consecutively on a single processor or the execution path will follow some message of size $\bar{M}$. In the former case, the cost of the LogGP message on the execution path is $o + \hat{G} \cdot \bar{M}$, whereas the cost of this portion of the execution path in $\bar{G}$ is $\alpha + \beta \cdot \bar{M}$. In the latter case, the cost of the LogGP message is $o \cdot 2 + L + \hat{G} \cdot \bar{M}$, whereas an execution path in $\bar{G}$ that goes from the sending to the receiving node will incur cost $\alpha + \beta \cdot 2\bar{M}$.

Since we limit our analysis to $L = o$, this means that each LogGP interprocessor execution path message cost of $O(L + \hat{G} \cdot \bar{M})$ and cost incurred by a single processor for sending or receiving a message, $O(o + \hat{G} \cdot \bar{M})$, translates to a cost of $O(\alpha + \beta \cdot \bar{M})$ in the constructed schedule. Since we know that for any execution path in the LogGP schedule the bandwidth cost is bounded by $O(\beta \cdot W)$, and since $\bar{G}$ has the same execution paths, the bandwidth cost of any execution path in $\bar{G}$ is also bounded by $O(\beta \cdot W)$. Since the cost of the LogGP schedule is bounded by S, the longest execution path will be of length S and latency cost $O(\alpha \cdot S)$, giving a total cost bound of $O(\alpha \cdot S + \beta \cdot W)$. □

Our theoretical model is also a generalization of the BSP model [Valiant 1990], which allows global synchronization at each BSP superstep. At every superstep, a BSP schedule executes an h-relation, in which processors communicate arbitrary datasets between each other and h is the largest total amount of data sent or received by any processor during the step. The communication cost of each superstep is the maximum number of words sent and received over all processors. We refer to the overall communication cost of a BSP schedule as W.

We show a reduction from any BSP schedule to a schedule in our model by adding a global synchronization vertex for every superstep. Each such synchronization vertex has incoming edges from the last local computations or posted sends of all processors in the superstep associated with the vertex and outgoing edges to the first operation of each processor in the next superstep.

THEOREM 3.2. *Given a BSP schedule for a dependency graph G with S synchronizations and communication cost W, there exists a parallel schedule $\bar{G}$ with synchronization cost $O(\alpha \cdot S)$ and bandwidth cost $O(\beta \cdot W)$.*

PROOF. We construct $\bar{G} = (\bar{V}, \bar{E})$ from the BSP schedule by adding a single synchronization vertex to $\bar{V}_{\text{sync}}$ for each superstep and connecting all processors' execution paths to this vertex. Thus, for the jth of the S supersteps, we define a global synchronization vertex $\bar{s}_j$, as well as sequences of vertices $\bar{F}_{1j}, \ldots, \bar{F}_{pj}$ in $\bar{V}_{\text{comp}}$ corresponding to the local computations and memory transfers done by each processor during that superstep (BSP only bounds the computation cost, so we have no bound on memory transfers). For the kth message sent by processor i during superstep j, we add a vertex $\bar{w}_{ijk}$ to $\bar{V}_{\text{send}}$, and for the lth message received by processor i, we add a vertex $\bar{r}_{ijl}$ to $\bar{V}_{\text{recv}}$. The execution path of processor i will then take the following form:

$$\pi_i = \{\ldots, \bar{F}_{ij}, \bar{w}_{ij1}, \bar{w}_{ij2}, \ldots, \bar{s}_j, \bar{r}_{ij1}, \bar{r}_{ij2}, \ldots, \bar{F}_{i,j+1}, \ldots\}.$$

For every superstep, which executes an h-relation with cost $\beta \cdot h$ in BSP, the cost of $\bar{G}$ includes not only all execution paths that correspond to the data sent and received by some processor but also execution paths that correspond to the data sent on one processor (no greater than h) and received on another processor (also no greater than h), for a total cost of at most $2h$. The total communication cost of $\bar{G}$ constructed in this way will therefore be at least $\alpha \cdot S + \beta \cdot W$ but no more than $\alpha \cdot S + \beta \cdot 2W$, where W, the total bandwidth cost of the BSP schedule, is the sum of h over all S supersteps. □

4. LOWER BOUNDS ON COST TRADE-OFFS BASED ON DEPENDENCY PATH EXPANSION

In this section, we introduce the concept of dependency intervals and then derive communication and synchronization lower bounds as functions of the rate of expansion of these intervals. Each dependency interval represents a set of interdependent computations positioned between two bounding vertices in the dependency graph. Dependency interval expansion refers to the growth of the dependency interval size with respect to the distance between the two bounding vertices on some dependency chain. When a

dependency chain with a high dependency interval expansion is present in a dependency graph, the parallelization of this dependency graph must incur synchronization cost or, alternatively, incur higher computation and communication costs. In particular, the computation cost scales with the total size of the dependency intervals, whereas the communication cost scales with the cross-section size (minimum vertex separator size) of the subgraphs connecting the vertices in the dependency intervals.

4.1. Dependency Interval Expansion

If there exists a path $\mathcal{P} = \{v_1, \ldots, v_n\}$ in a given DAG $G = (V, E)$ (meaning that $\{(v_1, v_2), \ldots, (v_{n-1}, v_n)\} \subset E$), we say that v_n *depends on* v_1. These dependency relations yield a partial ordering on the dependency graph, namely for any $v_1, v_2 \in V$, $v_1 < v_2$ if and only if v_2 is dependent on v_1 in G. Accordingly, we denote a sequence of vertices $\{w_1, \ldots, w_n\}$ a *dependency chain*, if for $i \in [1, n-1]$, w_{i+1} depends on w_i (i.e., $w_1 < w_2 < \cdots < w_n$). Any subsequence of a dependency chain is also a dependency chain, which we will sometimes call a *dependency subchain*.

We define a *dependency interval* from vertex v_1 to v_2 (with $v_1 < v_2$) to be the union of all dependency chains between v_1 and v_2,

$$\langle v_1, v_2 \rangle \equiv \{v_1, v_2\} \cup \{w : v_1 < w < v_2\}.$$

The dependency interval contains all vertices that are dependent on v_1 and a dependency of v_2. The dependency interval is a closed interval in the partial order (i.e., it also contains v_1 and v_2). We use the notation $\langle v_1, v_2 \rangle$ to disambiguate dependency intervals from integer intervals.

We define a *dependency interval subgraph* in graph $G = (V, E)$ from vertex v_1 to v_2 to be a subgraph induced by the union of all dependency chains between v_1 and v_2, as well as the edges between these vertices, which are given by

$$G\langle v_1, v_2 \rangle \equiv (\langle v_1, v_2 \rangle, (\langle v_1, v_2 \rangle \times \langle v_1, v_2 \rangle) \cap E).$$

4.2. Lower Bounds Based on Dependency Interval Expansion

To lower bound the communication necessary to compute a dependency interval, we obtain a lower bound on the size of the minimum vertex separator. To obtain nontrivial vertex separators, we place constraints on the sizes of the induced vertex partitions.

For any $q \geq x \geq 2$, a $\frac{1}{q}$-$\frac{1}{x}$-*balanced vertex separator* of $\hat{V} \subset V$ in a directed graph $G = (V, E)$ (if $\hat{V}$ is unspecified, then $\hat{V} = V$) is a set of vertices $Q \subset V$, which splits the vertices V in G into two disconnected partitions, $V \backslash Q = V_1 \cup V_2$ so that $E \subset (V_1 \times V_1) \cup (V_2 \times V_2) \cup (Q \times V) \cup (V \times Q)$. Further, the smaller of the two vertex partitions is bounded in size by the constants q and x as follows, for $\hat{V}_1 = \hat{V} \cap V_1$ and $\hat{V}_2 = \hat{V} \cap V_2$, $\lfloor |\hat{V}|/q \rfloor - |Q| \leq \min(|\hat{V}_1|, |\hat{V}_2|) \leq \lfloor |\hat{V}|/x \rfloor$. We denote the minimum size $|Q|$ of a $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator Q of G as $\chi_{q,x}(G)$. The expression $\lfloor |\hat{V}|/q \rfloor - |Q|$ can be negative, but not when Q is a minimal separator, as any $Q' \subset \hat{V}$ of size $\lfloor |\hat{V}|/q \rfloor$ is a $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator with $V_1 = \hat{V}_1 = \emptyset$, so $\chi_{q,x}(G) \leq \lfloor |\hat{V}|/q \rfloor$. For most graphs, we will consider $\chi_{q,x}(G) < \lfloor |\hat{V}|/q \rfloor$. For a dependency interval subgraph $G\langle v_1, v_2 \rangle$, we call $\chi_{q,x}(G\langle v_1, v_2 \rangle)$ the *cross-section expansion* of this dependency interval subgraph.

We now introduce the notion of a (ϵ, σ)-*path-expander* graph G, which is the key characterization of dependency graphs to which our lower bound analysis is applicable. These dependency graphs correspond to algorithms that contain a sequence of dependent operations that is expensive for one processor to compute without communication. In particular, vertices on this sequence must have many dependents that are themselves dependencies of later vertices in the sequence. Quantitatively, the overlap

between the dependents of one vertex in the sequence and the dependencies of a later vertex must grow in size as a function σ of the distance between the two vertices along the sequence. Whereas σ provides a bound on the amount of computation that must be done between the computation of vertices, the ϵ function provides a bound on the vertex separator size and can consequently be used to derive communication lower bounds.

In asymptotic terms, such a (ϵ, σ)-path-expander G needs to contain a dependency chain, $(v_1, \ldots, v_n)$, such that for any two vertices, v_{i+1}, v_{i+b}, if b is large enough, the dependency interval $\langle v_{i+1}, v_{i+b}\rangle$ has size $|\langle v_{i+1}, v_{i+b}\rangle| = \Omega(\sigma(b))$ and the dependency interval subgraph $G\langle v_{i+1}, v_{i+b}\rangle$ has cross-section size $\chi_{q,x}(G\langle v_{i+1}, v_{i+b}\rangle) = \Omega(\epsilon(b))$ for some q, x.

Now we define this notion more precisely. We call a DAG G a (ϵ, σ)-path-expander if for some $n \in [1, |V|]$ there exists a dependency chain $\mathcal{P} = (p_1, \ldots, p_n)$ in G, where $G\langle p_1, p_n\rangle = G$, and positive real constants $k, c_\epsilon, c_\sigma, \hat{c}_\sigma, q, x \ll n$ (meaning that n asymptotically dominates these constants), where $k \in \mathbb{Z}$, $c_\epsilon, c_\sigma > 1$, $x \geq 2$, and $q = \max(x\hat{c}_\sigma^2 c_\sigma + 1, \hat{c}_\sigma \cdot \sigma(k))$ such that

—for every pair of vertices $p_{i+1}, p_{i+b} \in \mathcal{P}$ with $b \geq k$,
(1) the dependency interval $\langle p_{i+1}, p_{i+b}\rangle$ has size $|\langle p_{i+1}, p_{i+b}\rangle| = \Theta(\sigma(b))$, more precisely, $\sigma(b)/\hat{c}_\sigma \leq |\langle p_{i+1}, p_{i+b}\rangle| \leq \hat{c}_\sigma \cdot \sigma(b)$;
(2) the dependency interval subgraph $G\langle p_{i+1}, p_{i+b}\rangle$ has cross-section expansion $\chi_{q,x}(G\langle p_{i+1}, p_{i+b}\rangle)) = \Omega(\epsilon(b))$,
—where the given real-valued functions ϵ, σ are
(1) positive and convex and
(2) increasing with $\epsilon(h+1) \leq c_\epsilon \epsilon(h)$ and $\sigma(h+1) \leq c_\sigma \sigma(h)$ for all real numbers $h \geq k$.

THEOREM 4.1 (GENERAL DEPENDENCY INTERVAL LOWER BOUNDS). *Suppose that a dependency graph G is a (ϵ, σ)-path-expander about dependency chain $\mathcal{P} = \{p_1, \ldots, p_n\}$. Then, for any schedule of G corresponding to a p-processor parallelization (of G) in which no processor computes more than $\frac{1}{x}$ of the vertices of G for some $x \in [2, p]$, there exists an integer $b \in [k, n]$ such that the computation (F), bandwidth (W), and latency (S) costs incurred are*

$$F = \Omega(\sigma(b) \cdot n/b), \quad W = \Omega(\epsilon(b) \cdot n/b), \quad S = \Omega(n/b).$$

PROOF. We consider any possible parallelization, which implies a coloring of the vertices of $G = (V, E)$, $V = \bigcup_{i=1}^{p} C_i$, and show that any schedule $\bar{G} = (\bar{V}, \bar{E})$, as defined in Section 3, incurs the desired computation, bandwidth, and latency costs. Our proof technique works by defining a set of dependency intervals within G between subsequent vertices on a dependency subchain of $\mathcal{P}$, which allows us to accumulate the costs of these dependency intervals.

The lower bounds on work and synchronization, $F = \Omega(\sigma(b) \cdot n/b)$ and $S = \Omega(n/b)$, can be derived by considering a sequence of dependency intervals between consecutive pairs of vertices along a dependency subchain of $\mathcal{P}$, with each dependency interval corresponding to a set of computations performed sequentially by some processor (see middle picture of a diamond dependency graph in Figure 2). However, to obtain the bandwidth lower bound, we must instead show that there exists some sequence of $c \geq 1$ dependency intervals, where for $i \in [1, c]$, some processor l_i computes a constant fraction (i.e., $1/w$, where $2 \leq x \leq w \leq q$ and x, q are constants) of the vertices of dependency interval i. The bandwidth cost of the schedule is then at least the sum over all $i \in [1, c]$ of the bandwidth cost needed by processor l_i to compute the ith interval in the sequence. We show such a set of multicolored dependency intervals for a diamond dependency graph in Figure 2. The intervals for this two-processor schedule are selected so that communication of vertices computed inside each interval is required for the full computation of the interval. Since one interval must be fully

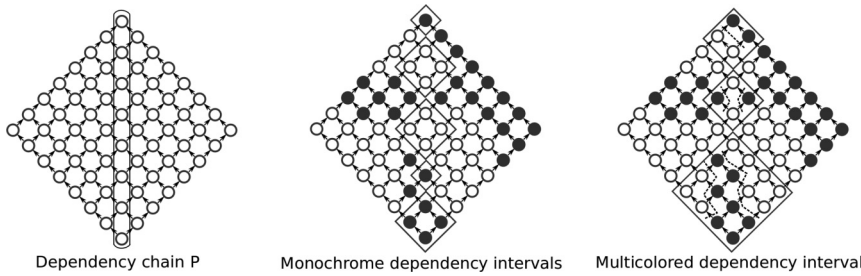

Fig. 2. Illustration of the construction in the proof of Theorem 4.1 in the case of a diamond DAG (e.g., Papadimitriou and Ullman [1987]), depicting a dependency chain, along which a set of dependency intervals are defined based on the vertex coloring of a two-processor parallelization.

computed before the computation of the next can begin, the communication of such vertices may not be done concurrently for multiple intervals.

Our proof works by following these steps:

(1) A procedure generates a sequence of disjoint dependency intervals between vertices comprising a dependency chain in $\mathcal{P}$:
   (a) Each generated dependency interval is defined so that some processor computes a constant fraction of the dependency interval.
   (b) The procedure terminates when no further dependency intervals in which a processor does a small enough portion of the work can be found.
(2) Case (I)—the last vertex in the generated dependency chain, p_z, is in the first half of $\mathcal{P}$ (the procedure terminates early):
   (a) We show that some processor must have computed at least a constant fraction of the dependency interval $\langle p_{z+1}, p_n \rangle$.
   (b) By expanding this dependency interval backward, we show that there is an $m \leq z + 1$ such that the same processor computes a constant fraction of the dependency interval $\langle p_m, p_n \rangle$.
   (c) We show that the existence of this dependency interval implies the theorem holds for $b = n/2$ (with $\Omega(1)$ synchronizations but a lot of work and communication done by some processor).
(3) Case (II)—the last vertex in the generated dependency chain is in the second half of $\mathcal{P}$ (the procedure outputs enough dependency intervals for the desired analysis):
   (a) Since some processor l_i computes at least a constant fraction of the ith dependency interval, processor l_i must have done work proportional to the size of the dependency interval.
   (b) Additionally, since processor l_i also computes at most a constant fraction of each dependency interval, processor l_i must have communicated at least a number of vertices equal to the minimum vertex separator (communicated set) size of the dependency interval.
   (c) Each dependency interval must be computed wholly before the computation of the next one begins, so there exists an execution path that includes all communication and computation tasks of processor l_i in the ith generated dependency interval and of processor l_{i+1} in the $(i + 1)$th dependency interval.
   (d) The sum of the communication and computation costs over all c dependency intervals is minimized when the distance in $\mathcal{P}$ between subsequent vertices in the dependency chain is the same size $b = \Theta(n/c)$ for each consecutive vertex pair.

(e) Each dependency interval requires a synchronization, and when each consecutive pair of vertices in the dependency chain are equidistant, $\Omega(\sigma(b))$ work as well as $\Omega(\epsilon(b))$ communication.
(f) Summing the preceding costs over $\Theta(n/b)$ intervals yields the costs in the theorem.

We now formally define the set of dependency intervals via the following procedure, which defines a dependency subchain of $\mathcal{P}$ by outputting a set of integers corresponding to distances between each subsequent pair of vertices in this dependency subchain. We define the procedure inductively, assuming that sequences of integers $(b_1, \ldots, b_{i-1})$ for $i \geq 1$ with $\bar{b}_i = \sum_{j=1}^{i-1} b_j$ have been produced by the procedure and defining the rules for whether the procedure terminates or produces a new integer b_i, which will correspond to the dependency interval $V_i = \langle p_{\bar{b}_i+1}, p_{\bar{b}_i+b_i} \rangle$ and continues. We let l be the index of the processor that computes $p_{\bar{b}_i+1}$. The integer b_i is selected to be the smallest integer greater or equal to k such that the following two conditions are simultaneously satisfied for the dependency interval V_i:

Condition 1. The subset of the dependency interval V_i that processor l computes, $V_i \cap C_l$, is of size

$$|V_i \cap C_l| \geq \lfloor |V_i|/q \rfloor.$$

Condition 2. The subset of the dependency interval V_i that processor l does not compute, $V_i \backslash C_l$, is of size

$$|V_i \backslash C_l| \geq |V_i| - \lfloor |V_i|/x \rfloor.$$

If no $b_i \geq k$ exists such that the preceding conditions are simultaneously satisfied, the procedure terminates, outputting the sequence $(b_1, \ldots, b_{i-1})$.

For some $c \geq 0$, the procedure outputs the sequence $(b_1, \ldots, b_c)$. We denote $z = \bar{b}_{c+1} = \sum_{j=1}^{c} b_j$. We consider two cases, $z < n/2$ and $z \geq n/2$, one of which must hold, and show that the theorem holds in either case.

Case (I)—($z < n/2$). We show that Condition 1 must be satisfied for the dependency interval $\langle p_{z+1}, p_n \rangle$ and further that both Condition 1 and Condition 2 are simultaneously satisfied for some dependency interval $\langle p_m, p_n \rangle$ with $m \leq z + 1$; we then show that this implies the theorem holds. We demonstrate that Condition 1 is satisfied by induction on r, which defines the dependency interval $W_r = \langle p_{z+1}, p_{z+r} \rangle$ for $k \leq r \leq n - z$. At the base case, $r = k$, and Condition 1 is satisfied because $|W_r|/q \leq \hat{c}_\sigma \cdot \sigma(k)/q \leq 1$ and processor l computes at least one element, p_{z+1}.

For $r > k$, we have

$$|W_r| \leq \hat{c}_\sigma \cdot \sigma(r) \leq \hat{c}_\sigma c_\sigma \cdot \sigma(r-1) \leq \frac{(q-1)}{x\hat{c}_\sigma} \cdot \sigma(r-1).$$

Further, by induction, Condition 1 was satisfied for W_{r-1}, which implies that Condition 2 was not satisfied for W_{r-1} (otherwise, the procedure would have output another integer corresponding to the dependency interval $\langle p_{z+1}, p_{z+r-1} \rangle$). Now, using bounds on dependency interval growth, we show that since Condition 2 was not satisfied for W_{r-1}, Condition 1 has to be satisfied for the subsequent dependency interval, W_r,

$$|C_l \cap W_r| \geq |C_l \cap W_{r-1}| \geq |W_{r-1}| - (|W_{r-1}| - \lfloor |W_{r-1}|/x \rfloor) = \lfloor |W_{r-1}|/x \rfloor,$$

applying the lower bound on dependency interval size $|W_{r-1}| \geq \sigma(r-1)/\hat{c}_\sigma$,

$$|C_l \cap W_r| \geq \lfloor \sigma(r-1)/(x\hat{c}_\sigma) \rfloor,$$

and applying the bound $|W_r| \leq \sigma(r-1) \cdot (q-1)/(\hat{c}_\sigma x)$, which implies that $\sigma(r-1) \geq |W_r| \cdot (x\hat{c}_\sigma)/(q-1)$, we obtain

$$|C_l \cap W_r| \geq \lfloor (|W_r| \cdot (x\hat{c}_\sigma)/(q-1))/(x\hat{c}_\sigma) \rfloor = \lfloor |W_r|/(q-1) \rfloor \geq \lfloor |W_r|/q \rfloor,$$

and thus Condition 1 holds for $W_{n-z} = \langle p_{z+1}, p_n \rangle$. Due to Condition 1, processor l must compute $F \geq \lfloor |W_{n-z}|/q \rfloor = \Omega(\sigma(n-z))$ vertices. Since, by assumption, no processor can compute more than $\frac{1}{x}$ of $\langle p_1, p_n \rangle$, we claim for some $1 \leq m \leq z$ that there exists a dependency interval $\langle p_m, p_n \rangle$ in which Condition 1 and Condition 2 are satisfied for processor l. Condition 2 has to be satisfied for some $1 \leq m \leq z$, as no processor computes more than $\lfloor |V|/x \rfloor = \lfloor |\langle p_1, p_n \rangle|/x \rfloor$ vertices. Further, we can find an m for which both Condition 1 and Condition 2 are satisfied since we can apply the preceding inductive argument backward by defining $Z_r = \langle p_{n-r+1}, p_n \rangle$, starting in the base case with $r = n - z$ (which corresponds to the dependency interval $Z_{n-z} = W_{n-z}$ and for which we just showed that Condition 1 holds), and increasing r until Condition 1 and Condition 2 are both satisfied for Z_r, then picking $m = n-r+1$. The inductive argument stating that Condition 1 will remain satisfied until Condition 2 is satisfied still holds, as we can apply growth bounds on the size of the dependency interval in the same way to lower bound the number of entries that processor l computes in Z_r based on the number of entries processor l computed in Z_{r-1}. Thus, the size of processor l's communicated set is at least the size of a $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator of $G\langle p_m, p_n \rangle$, namely $\chi_{q,x}(G\langle p_m, p_n \rangle) = \Omega(\epsilon(n-m+1))$, as the communicated set

$$R = \Xi(\langle p_m, p_n \rangle \cap C_l, C_l, E)$$

separates two partitions, one of size at least

$$|(\langle p_m, p_n \rangle \cap C_l) \backslash R| \geq \lfloor |\langle p_m, p_n \rangle|/q \rfloor - |R|$$

due to Condition 1 and at most

$$|\langle p_m, p_n \rangle \cap C_l| \leq \lfloor |\langle p_m, p_n \rangle|/x \rfloor$$

due to Condition 2 on $\langle p_m, p_n \rangle$, as well as another partition of size at least

$$\begin{aligned} |(\langle p_m, p_n \rangle \backslash C_l) \backslash R| &\geq |\langle p_m, p_n \rangle| - \lfloor |\langle p_m, p_n \rangle|/x \rfloor - |R| \geq \lfloor |\langle p_m, p_n \rangle|/x \rfloor - |R| \\ &\geq \lfloor |\langle p_m, p_n \rangle|/q \rfloor - |R| \end{aligned}$$

due to Condition 2 on $\langle p_m, p_n \rangle$ and the fact that $q \geq x \geq 2$. Applying the assumed lower bound on the $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator of $G\langle p_m, p_n \rangle$, we obtain a lower bound on the size of the communicated set R, which is also a lower bound on interprocessor communication cost, $W = \Omega(\epsilon(n-m+1)) = \Omega(\epsilon(n/2))$ since $m \leq z < n/2$. Since these costs are incurred along a dependency path in the schedule consisting of the work and communication done only by processor l, the bounds hold for $b = n/2$.

Case (II)—($z \geq n/2$). In this case, we work with the c intervals generated by the procedure. We define a dependency chain $\{s_1, t_1, s_2, t_2, \ldots, s_c, t_c\}$ (a dependency subchain of $\mathcal{P}$) whose elements s_i, t_i correspond to the first vertex of the ith interval, $s_i = p_{\bar{b}_i+1}$ and the last vertex $t_i = p_{\bar{b}_{i+1}}$ where as before, $\bar{b}_i = \sum_{j=1}^{i-1} b_j$. Thus, the ith interval generated is $V_i = \langle p_{s_i}, p_{t_i} \rangle$. For each $i \in [1, c]$, consider

—the task $\bar{u}_i \in \bar{V}$, during which some processor l_i computed the first vertex in the ith interval s_i (i.e., $s_i \in \hat{f}(\bar{u}_i)$);
—the task $\bar{v}_i \in \bar{V}$, during which processor l_i computed its last vertex within the ith interval, so $\hat{f}(\bar{v}_i) \cap V_i \cap C_{l_i} \neq \emptyset$; and

—the task $\bar{w}_i \in \bar{V}$, during which the last vertex in the ith dependency interval, t_i, was computed (i.e., $t_i \in \hat{f}(\bar{w}_i)$).

Since t_i depends on all other vertices in V_i, there will be an execution path $\pi_i = \{\bar{u}_i, \ldots, \bar{v}_i, \ldots, \bar{w}_i\} \subset \bar{V}$ in the schedule $\bar{G}$. Since $\hat{f}(\bar{u}_i)$ contains the first vertex of $\mathcal{R}_i$, all communication necessary to satisfy the dependencies of processor l_i (dependencies of $V_i \cap C_{l_i}$) within dependency interval V_i must be incurred along π_i. This communicated set is given by

$$\hat{T}_i = \Xi(V_i, V_i \cap C_{l_i}, E),$$

which is a separator of $G\langle p_{s_i}, p_{t_i}\rangle$ and is $\frac{1}{q}$-$\frac{1}{x}$-balanced by the same argument as given earlier in case (I) for V_i, since in both cases Conditions 1 and 2 both hold. Any separator of $G\langle p_{s_i}, p_{t_i}\rangle$ must create two partitions, one of size at least

$$|(V_i \cap C_{l_i})\backslash\hat{T}_i| \geq \lfloor |V_i|/q \rfloor - |\hat{T}_i|$$

due to Condition 1 and at most

$$|V_i \cap C_{l_i}| \leq \lfloor |V_i|/x \rfloor$$

due to Condition 2 on V_i, as well as another partition of size at least

$$\begin{aligned} |(V_i\backslash C_{l_i})\backslash\hat{T}_i| &\geq |V_i| - \lfloor |V_i|/x \rfloor - |\hat{T}_i| \\ &\geq \lfloor |V_i|/x \rfloor - |\hat{T}_i| \geq \lfloor |V_i|/q \rfloor - |\hat{T}_i| \end{aligned}$$

due to Condition 2 on V_i and the fact that $q \geq x \geq 2$.

We use the lower bound on the minimum separator of each dependency interval subgraph to obtain a lower bound on the size of the communicated set for processor l_i in the ith dependency interval,

$$|\hat{T}_i| \geq \chi_{q,x}(G\langle p_{s_i}, p_{t_i}\rangle) = \Omega(\epsilon(b_i)),$$

where we are able to bound the cross-section expansion of $G\langle p_{s_i}, p_{t_i}\rangle$ since $|b_i| \geq k$. Every dependency interval $V_i = \langle p_{s_i}, p_{t_i}\rangle$ must be computed entirely before $V_{i+1} = \langle p_{s_{i+1}}, p_{t_{i+1}}\rangle$ since s_{i+1} is dependent on t_i (they are subsequent elements of dependency chain $\mathcal{P}$) and all elements of $\langle p_{s_i}, p_{t_i}\rangle$ except t_i itself are dependencies of t_i. Therefore, there is an execution path π_{critical} in the schedule $\bar{G}$ that contains $\pi_i \subset \pi_{\text{critical}}$ as an execution subpath for every $i \in [1, c]$. The execution cost of $\bar{G}$ is bounded below by the cost of π_{critical},

$$T(\bar{G}) \geq \sum_{\bar{v} \in \pi_{\text{critical}}} \hat{t}(\bar{v}) = \alpha \cdot S + \beta \cdot W + \gamma \cdot F,$$

which we can bound below component-wise,

$$F = \sum_{\bar{v} \in \pi_{\text{critical}} \cap \bar{V}_{\text{comp}}} |\hat{f}(\bar{v})| \geq \sum_{i=1}^{c} \frac{1}{q}|V_i| = \Omega\left(\sum_{i=1}^{c} \sigma(b_i)\right),$$

$$W = \sum_{\bar{v} \in \pi_{\text{critical}} \cap \bar{V}_{\text{send}}} |\hat{s}(\bar{v})| + \sum_{\bar{v} \in \pi_{\text{critical}} \cap \bar{V}_{\text{recv}}} |\hat{r}(\bar{v})| \geq \sum_{i=1}^{c} \chi_{q,x}(G\langle p_{s_i}, p_{t_i}\rangle) = \Omega\left(\sum_{i=1}^{c} \epsilon(b_i)\right).$$

Further, since each dependency interval contains vertices computed by multiple processors, each π_i must go through at least one synchronization vertex; therefore, we also

have a lower bound on latency cost,

$$S \geq \sum_{\tilde{v} \in \pi_{\text{critical}} \cap \tilde{V}_{\text{sync}}} 1 \geq c.$$

Because each $b_i \geq k$, σ as well as ϵ are assumed convex, and $\sum_i b_i \leq n$, the preceding lower bounds for F and W are minimized when all values $|\mathcal{R}_i|$ are equal.[2] Thus, we can replace b_i for each i by $b = \lfloor (\sum_j b_j)/c \rfloor \geq \lfloor n/(2c) \rfloor = \Theta(n/c)$ simplifying the bounds to obtain the conclusion. □

COROLLARY 4.2 (ORDER-ω DEPENDENCY INTERVAL LOWER BOUNDS). *Suppose that there exists a dependency chain $\mathcal{P} = \{p_1, \ldots, p_n\}$ in a dependency graph G and integer constants $2 \leq \omega \ll k \ll n$ such that for every pair of vertices p_{i+1}, p_{i+b} with $b \geq k$, the dependency interval $\langle p_{i+1}, p_{i+b} \rangle$ has size $|\langle p_{i+1}, p_{i+b} \rangle| = \Theta(b^\omega)$ and the dependency interval subgraph $G\langle p_{i+1}, p_{i+b} \rangle$ has cross-section expansion $\chi_{q,x}(G\langle p_{i+1}, p_b \rangle) = \Omega(b^{\omega-1}/q^{(\omega-1)/\omega})$ for all real numbers $k \leq q \ll n$. Then, the computation (F), bandwidth (W), and latency (S) costs incurred by any schedule of G corresponding to a parallelization (of G) in which no processor computes more than $\frac{1}{x}$ of the vertices in $\langle p_1, p_n \rangle$ (for $2 \leq x \ll n$) must be at least*

$$F = \Omega(b^{\omega-1} \cdot n), \quad W = \Omega(b^{\omega-2} \cdot n), \quad S = \Omega(n/b)$$

for some $b \in [k, n]$. Further, the costs need to obey the following trade-offs:

$$F \cdot S^{\omega-1} = \Omega(n^\omega), \quad W \cdot S^{\omega-2} = \Omega(n^{\omega-1}).$$

PROOF. This is an application of Theorem 4.1 since $G\langle p_1, p_n \rangle$ is a (ϵ, σ)-path-expander graph with $\epsilon(b) = b^{\omega-1}/q^{(\omega-1)/\omega}$ and $\sigma(b) = b^\omega$. The particular constants defining the path-expander graph being k and x as given, as well as $c_\sigma = \frac{(k+1)^\omega}{k^\omega}$, $c_\epsilon = \frac{(k+1)^{\omega-1}}{k^{\omega-1}}$, and since for $b \geq k$, $|\langle p_{i+1}, p_{i+b} \rangle| = \Theta(b^\omega) = \Theta(\sigma(b))$ there exists a constant $\hat{c}_\sigma$ such that $\sigma(b)/\hat{c}_\sigma \leq |\langle p_{i+1}, p_{i+b} \rangle| \leq \hat{c}_\sigma \cdot \sigma(b)$. These constants all make up q, which is $O(x\hat{c}_\sigma^2 k^\omega) = O(1)$, so we can disregard the factor of $q^{(\omega-1)/\omega}$ inside ϵ and obtain the desired lower bounds,

$$F = \Omega(b^{\omega-1} \cdot n), \quad W = \Omega(b^{\omega-2} \cdot n), \quad S = \Omega(n/b).$$

These equations can be manipulated algebraically to obtain the trade-offs in the corollary. □

5. VOLUMETRIC INEQUALITIES

A key step in the lower bound proofs in Irony et al. [2004] and Ballard et al. [2011] was the use of an inequality introduced by Loomis and Whitney [1949]. We state this inequality in Theorem 5.1. This inequality states that the size of a set of d-tuples cannot be greater than a fractional power of the product of all of the projections of vertices in the set onto $(d-1)$-tuples. In the context of communication lower bounds, the Loomis-Whitney theorem is used in the other direction—that is, the fractional power of the product of the projections is no less than the size of the set of d-tuples, which allows one to obtain a lower bound on the total size of the projections (typically associated with the amount of communication necessary to compute a set of vertices enumerated by the set of d-tuples).

[2]This mathematical relation can be demonstrated by a basic application of convexity.

THEOREM 5.1 (DISCRETE LOOMIS-WHITNEY INEQUALITY). *Let V be a finite, nonempty set of d-tuples $(i_1, \ldots, i_d) \in [1, n]^d$. Then we have*

$$|V| \leq \left(\prod_{j=1}^{d} |\pi_j(V)| \right)^{1/(d-1)},$$

where for $j \in [1, d]$, $\pi_j : [1, n]^d \to [1, n]^{d-1}$ is the projection

$$\pi_j(i_1, \ldots, i_d) = (i_1, \ldots, i_{j-1}, i_{j+1}, \ldots, i_d).$$

We use a generalized discrete version of the inequality, which was given by Tiskin [1998] and is a special case of the theory in Christ et al. [2013]. We state this generalized inequality in Theorem 5.2. This generalized inequality considers m-tuples and their projections onto r tuples, where $r < m$. Thus, Theorem 5.1 is a special case of Theorem 5.2 with $m = d$ and $r = d - 1$.

THEOREM 5.2 (GENERALIZED DISCRETE LOOMIS-WHITNEY INEQUALITY). *Let V be a finite, nonempty set of m-tuples $(i_1, \ldots, i_m) \in [1, n]^m$. Then we have*

$$|V| \leq \left(\prod_{1 \leq s_1 < \cdots < s_r \leq m} |\pi_{s_1,\ldots,s_r}(V)| \right)^{1/\binom{m-1}{r-1}},$$

where there are $\binom{m}{r}$ projections, defined for all $1 \leq s_1 < \cdots < s_r \leq m$ as

$$\pi_{s_1,\ldots,s_r}(i_1, \ldots, i_m) = (i_{s_1}, \ldots, i_{s_r}).$$

We now derive a version of the Loomis-Whitney inequality that is specialized for sets of strictly increasing tuples (i.e., any tuple of positive integers $(a_1, \ldots, a_m)$, where $a_i < a_{i+1}$ for all $i \in [1, m-1]$). In the new version, we also rearrange the form of the inequality so that it directly bounds the size of the projections. In particular, we achieve this by proving a bound on the size of the set V with respect to the union of the projections rather than a fractional power of a product of each projection. We will use this inequality to prove Theorem 6.1 (lower bound on the hyperedge cut of a lattice), where it yields a higher lower bound on the size of the union of the projections (the hyperedges) for any given set of vertices by a constant factor (this factor is needed in the proof). We state the inequality in Theorem 5.3 and prove its correctness.

THEOREM 5.3. *Let V be a finite, nonempty set of strictly increasing m-tuples $(i_1, \ldots, i_m) \in [1, n]^m$, $i_1 < \cdots < i_m$. Consider $\binom{m}{r}$ projections $\pi_{s_1,\ldots,s_r}(i_1, \ldots, i_m)$ defined as in Theorem 5.2. Define the union of these projections of V as*

$$\bar{\Pi} = \bigcup_{1 \leq s_1 < \cdots < s_r \leq m} \pi_{s_1,\ldots,s_r}(V).$$

Now the size of V may be bounded as a function of the number of projections in $\bar{\Pi}$,

$$|V| \leq \left(r! \cdot |\bar{\Pi}|\right)^{m/r} / m!.$$

PROOF. The proof works by

(1) considering any possible set of projections $\bar{\Pi}$ that are all strictly increasing tuples and expanding it to contain all $r!$ permutations of each of these projections into the set $\hat{\Pi}$;
(2) constructing the maximal set of vertices $\hat{V}$ (not necessarily strictly increasing tuples) whose projections are in $\hat{\Pi}$;

(3) bounding $|\hat{V}|$ via Theorem 5.2;
(4) selecting all strictly increasing tuples as part $\bar{V} \subset \hat{V}$, which is smaller by a factor of $m!$; and
(5) asserting that $V \subset \bar{V}$, which yields the necessary bound in Theorem 5.3.

For any V, consider the projected sets $\pi_{s_1,\ldots,s_r}(V)$ for $1 \leq s_1 < \cdots < s_r \leq m$. We now construct set $\bar{V}$, with $V \subset \bar{V}$, and obtain an upper bound on the size of $\bar{V}$, yielding an upper bound on the size of V. We expand the union of all projections, $\bar{\Pi}$, into larger set $\hat{\Pi} \supset \bar{\Pi}$, which contains all permutations of the tuples in $\bar{\Pi}$. In particular, for any r-tuple permutation p, if $w \in \bar{\Pi}$ then $p(w) \in \hat{\Pi}$. Since $\bar{\Pi}$ consisted only of r-tuples whose indices are strictly increasing (since they are ordered projections of strictly increasing d-tuples),

$$|\hat{\Pi}| = r! \cdot |\bar{\Pi}|.$$

We now construct the set $\hat{V}$ of all (not necessarily strictly increasing) tuples whose projections are in $\hat{\Pi}$—that is,

$$\hat{V} = \{(w_1, \ldots, w_m) \in [1, n]^m : \text{ such that } \forall \{q_1, \ldots, q_r\} \subset \{w_1, \ldots, w_m\}, \ \ (q_1, \ldots, q_r) \in \hat{\Pi}\}.$$

We note that the original m-tuple set is a subset of the new one, $V \subset \hat{V}$, since

$$\forall (q_1, \ldots, q_r) \in \pi_{s_1,\ldots,s_r}(V) \subset \hat{\Pi}, \ \exists (v_1, \ldots, v_m) \in V \text{ such that } \{q_1, \ldots, q_r\} \subset \{v_1, \ldots, v_m\}.$$

Further, $\hat{V}$ includes all m-tuples such that all arbitrarily ordered r-sized subsets of these tuples are in $\hat{\Pi}$. Since subsets of all possible orderings are considered, the ordering of the m-tuple does not matter, so $\hat{V}$ should contain all orderings of each tuple and is therefore a symmetric subset of $[1, n]^m$.

Now by application of the Loomis-Whitney inequality (Theorem 5.2), we know that the size of $\hat{V}$ is

$$|\hat{V}| \leq |\hat{\Pi}|^{\binom{m}{r}/\binom{m-1}{r-1}}.$$

We now let $\bar{V}$ be the set of all strictly increasing tuples inside $\hat{V}$. We have that $V \subset \bar{V}$ since $V \subset \hat{V}$ and V contains only strictly increasing tuples. Since there are $m!$ symmetrically equivalent tuples to each of the strictly increasing tuples inside $\hat{V}$, $|\bar{V}| = |\hat{V}|/m!$. Combining this bound with the bound on Loomis-Whitney bound on W, we obtain

$$\begin{aligned} |V| \leq |\bar{V}| \leq |\hat{V}|/m! &\leq |\hat{\Pi}|^{\binom{m}{r}/\binom{m-1}{r-1}}/m! \\ &= (r! \cdot |\bar{\Pi}|)^{\binom{m}{r}/\binom{m-1}{r-1}}/m! = (r! \cdot |\bar{\Pi}|)^{m/r}/m!. \quad \square \end{aligned}$$

6. LOWER BOUNDS ON LATTICE HYPERGRAPH CUTS

A hypergraph $H = (V, E)$ is defined by a set of vertices V and a set of hyperedges E, where each hyperedge $e_i \in E$ is a subset of V, $e_i \subseteq V$. We say two vertices in a hypergraph $v_1, v_2 \in V$ are connected if there exists a path between v_1 and v_2, namely if there exist a sequence of hyperedges $\mathcal{P} = \{e_1, \ldots, e_k\}$ such that $v_1 \in e_1$, $v_2 \in e_k$, and $e_i \cap e_{i+1} \neq \emptyset$ for all $i \in [1, k-1]$. We say that a hyperedge $e \in E$ is *internal* to some $V' \subset V$ if $e \subset V'$. If no $e \in E$ is adjacent to (i.e., contains) a $v \in V' \subset V$, then we say that V' is *disconnected* from the rest of H. A $\frac{1}{q}$-$\frac{1}{x}$-*balanced hyperedge cut* of a hypergraph is a subset of E whose removal from H partitions $V = V_1 \cup V_2$ with $\lfloor |V|/q \rfloor \leq \min(|V_1|, |V_2|) \leq \lfloor |V|/x \rfloor$ such that all remaining (uncut) hyperedges are internal to one of the two parts.

For any $q \geq x \geq 2$, a $\frac{1}{q}$-$\frac{1}{x}$-*balanced vertex separator* of vertex set $\hat{V} \subset V$ in a hypergraph $H = (V, E)$ (if $\hat{V}$ is unspecified then $\hat{V} = V$) is a set of vertices $Q \subset V$ whose removal from V and from the hyperedges E in which the vertices appear splits the vertices V in H into two disconnected parts $V \setminus Q = V_1 \cup V_2$. Further, the subsets of these vertices inside $\hat{V}$ must be balanced, meaning that for $\hat{V}_1 = \hat{V} \cap V_1$ and $\hat{V}_2 = \hat{V} \cap V_2$, $\lfloor |\hat{V}|/q \rfloor - |Q| \leq \min(|\hat{V}_1|, |\hat{V}_2|) \leq \lfloor |\hat{V}|/x \rfloor$. The smaller partition may be empty if the separator Q contains at least $\lfloor |\hat{V}|/q \rfloor$ vertices.

6.1. Lattice Hypergraphs

For any $m > r > 0$, we define a (m, r)-*lattice hypergraph* $H = (V, E)$ of dimension n with $|V| = \binom{n}{m}$ vertices and $|E| = \binom{n}{r}$ hyperedges. Each vertex is represented as $v_{i_1,\ldots,i_m} = (i_1, \ldots, i_m)$ for $(i_1, \ldots, i_m) \in [1, n]^m$ with $i_1 < \cdots < i_m$. Each hyperedge connects all vertices that share r indices—that is, $e_{j_1,\ldots,j_r}$ for $(j_1, \ldots, j_r) \in [1, n]^r$ with $j_1 < \cdots < j_r$ includes all vertices $v_{i_1,\ldots,i_m}$ for which $\{j_1, \ldots, j_r\} \subset \{i_1, \ldots, i_m\}$. Each vertex appears in $\binom{m}{r}$ hyperedges, and therefore each hyperedge contains $\binom{n}{m}\binom{m}{r}\binom{n}{r}^{-1} = \binom{n-r}{m-r}$ vertices.

THEOREM 6.1. *For $1 \leq r < m \ll n$ and $2 \leq x \leq q \ll n$, the minimum $\frac{1}{q}$-$\frac{1}{x}$-balanced hyperedge cut of a (m, r)-lattice hypergraph $H = (V, E)$ of dimension n is of size $\epsilon_q(H) = \Omega(n^r/q^{r/m})$.*

PROOF. The proof considers any $\frac{1}{q}$-$\frac{1}{x}$-balanced hyperedge cut and obtains a lower bound on its size as follows:

(1) Case (I)—most of the vertices in the smaller partition are contained in hyperedges internal to this partition, W_1:
   (a) Define *fibers*, each of which corresponds to $n-(m-1)$ vertices, and *hyperplanes*, each of which corresponds to $n-(r-1)$ hyperedges.
   (b) Let F_1 be the set of fibers internal to W_1.
   (c) Define $\binom{m-1}{r-1}$ projections from fibers F_1 to hyperplanes Z_1.
   (d) Apply Theorem 5.3 to lower bound the number of hyperplanes $|Z_1|$ adjacent to F_1 (and subsequently to W_1).
   (e) Note that all hyperedges within each hyperplane in Z_1 must be part of the cut or in W_1.
   (f) Show that the number of hyperedges in Z_1 is larger than $|W_1|$ by a factor of
   $$\rho = 2^{1-(r-1)/(m-1)} > 1,$$
   showing that this small factor difference is made possible by use of Theorem 5.3 rather than Theorem 5.2.
   (g) Assert the lower bound on the size of the hyperedge cut as the number of hyperedges in Z_1 but not in W_1,
   $$\frac{(n-(r-1))}{r} \cdot |Z_1| \cdot (1 - 1/\rho) = \Omega\big(n^r/q^{(r-1)/(m-1)}\big),$$
   which is better than the desired bound.
(2) Case (II)—most of the vertices in one of the partitions are disconnected entirely by the hyperedge cut:
   (a) Define $\binom{m}{r}$ projections that give the $\binom{m}{r}$ hyperedges adjacent to each vertex.
   (b) Either Theorem 5.2 or Theorem 5.3 suffices to show the number of hyperedges adjacent to the disconnected set of vertices (and therefore part of the hyperedge cut) is $\Omega(n^r/q^{r/m})$.

A previous version of this work [Solomonik et al. 2014], which considered the special case of $m = d$ and $r = d - 1$, was erroneous in its proof of case (I). That version attempted to prove the theorem by induction on d, using the inductive assumption that the theorem holds for $d-1$ to obtain a lower bound on the hyperedge cut of a new lattice hypergraph in which the vertices are fibers of the original lattice (with $r = d-1$, these are just the hyperedges) and the hyperedges are hyperplanes of the original lattice. However, it is not the case that all hyperedges in some hyperplane need to be in the hyperedge cut to disconnect two internal hyperedges, as asserted in that version of the work. The following proof uses a more complicated counting argument for case (I) instead, as summarized previously.

Consider any $\frac{1}{q}$-$\frac{1}{x}$-balanced hyperedge cut $Q \subset E$. Since all hyperedges that contain at least one vertex in both V_1 and V_2 must be part of the cut Q, all vertices are either disconnected by the cut or remain in hyperedges that are all internal to either V_1 or V_2. Let $U_1 \subset V_1$ be the vertices contained in a hyperedge internal to V_1, and let $U_2 \subset V_2$ be the vertices contained in a hyperedge internal to V_2. Since both V_1 and V_2 contain at least $\lfloor |V|/q \rfloor$ vertices, either $\lfloor |V|/(2q) \rfloor$ vertices must be in internal hyperedges within both V_1 as well as V_2—that is,

$$\text{case (I): } |U_1| \geq \lfloor |V|/(2q) \rfloor \text{ and } |U_2| \geq \lfloor |V|/(2q) \rfloor,$$

or there must be $\lfloor |V|/(2q) \rfloor$ vertices that are disconnected by the cut,

$$\text{case (II): } |(V_1 \backslash U_1) \cup (V_2 \backslash U_2)| \geq \lfloor |V|/(2q) \rfloor.$$

First we note that when $r \leq \lfloor m/2 \rfloor$, every hyperedge intersects with every other hyperedge at some vertex, since for any two r-tuples (hyperedges), the union of their entries has no more than m entries and so must correspond to at least one vertex. Since we assume that $q < n$, case (I) implies that U_1 and U_2 each have an internal hyperedge. This is not possible for $r \leq \lfloor m/2 \rfloor$, as any pair of hyperedges intersects at some vertex. We note that this precludes case (I) for $r = 1$, $m = 2$ (the most basic version of the (m, r)-lattice hypergraph), and furthermore it precludes any scenario with $r = 1$.

In case (I), for $r \geq 2$ and $r > m/2$ (the other cases were ruled out in the previous paragraph), we start by assuming without loss of generality that $|U_1| \leq |U_2|$, and therefore $|U_1| \leq \lfloor |V|/x \rfloor \leq \lfloor |V|/2 \rfloor$. We consider the hyperedges W_1 internal to U_1. Each vertex may appear in at most $\binom{m}{r}$ hyperedges, and each hyperedge has $\binom{n-r}{m-r}$ vertices, so the number of hyperedges is at most

$$|W_1| \leq \frac{\binom{m}{r}|U_1|}{\binom{n-r}{m-r}} \leq \frac{\binom{m}{r}\binom{n}{m}}{2\binom{n-r}{m-r}} = \frac{1}{2}\binom{n}{r}. \tag{6.1.1}$$

Since both U_1 and U_2 have at least $\lfloor |V|/(2q) \rfloor$ vertices and each hyperedge has $\binom{n-r}{m-r}$ vertices, the number of hyperedges is at least

$$|W_1| \geq \frac{|U_1|}{\binom{n-r}{m-r}} \geq \left\lfloor \frac{\binom{n}{m}}{2q} \right\rfloor \cdot \binom{n-r}{m-r}^{-1} \geq \left\lfloor \frac{\binom{n}{r}}{2q\binom{m}{r}} \right\rfloor. \tag{6.1.2}$$

We define a *fiber* $f_{k_1,\ldots,k_{m-1}}$ for each $(k_1, \ldots, k_{m-1}) \in [1, n]^{m-1}$ with $k_1 < \cdots < k_{m-1}$ as the set of vertices in the hypergraph H that satisfy $\{k_1, \ldots, k_{m-1}\} \subset \{j_1, \ldots, j_m\}$. We note that each fiber contains $n - (m-1)$ vertices and is internal to at least one hyperedge (which is defined by $r \leq m-1$ indices). We define a *hyperplane* $x_{k_1,\ldots,k_{r-1}}$ for each $(k_1, \ldots, k_{r-1}) \in [1, n]^{r-1}$ with $k_1 < \cdots < k_{r-1}$ as the set of all hyperedges $e_{j_1,\ldots,j_r}$ that satisfy $\{k_1, \ldots, k_{r-1}\} \subset \{j_1, \ldots, j_r\}$. Thus, each of the $|X| = \binom{n}{r-1}$ hyperplanes contains $n-(r-1)$ hyperedges, and each hyperedge is in r hyperplanes. Note that each hyperplane shares a unique hyperedge with $(r-1)(n-(r-1))$ other hyperplanes.

We now obtain a bound on the number of hyperplanes to which the hyperedges W_1 are adjacent. The number of fibers internal to each hyperedge is $\binom{n-r}{m-r-1}$ (all ways to choose the other $m-r-1$ indices of the $m-1$ indices in the fiber, r of which come from the hyperedge). We denote the fibers internal to W_1 as F_1 and determine the number of hyperplanes adjacent to these fibers. The total number of fibers in the set W_1 is then $|F_1| = \binom{n-r}{m-r-1}|W_1|$. We now apply the generalized modified Loomis-Whitney inequality (Theorem 5.3) for $\binom{m-1}{r-1}$ projections of a set of $(m-1)$-tuples (fibers) onto $(r-1)$-tuples (hyperplanes),

$$\pi_{s_1,\ldots,s_{r-1}}(e_{i_1,\ldots,i_{m-1}}) = (i_{s_1},\ldots,i_{s_{r-1}}),$$

for $0 < s_1 < s_2 < \cdots < s_{r-1} < m$ corresponding to each of $\binom{m-1}{r-1}$ hyperplanes adjacent to each fiber. The inequality (Theorem 5.3) yields the following lower bound on the product of the size of the projections with respect to the union of the projections $Z_1 = \cup_j \pi_j(F_1)$—that is,

$$|F_1| \leq ((r-1)! \cdot |Z_1|)^{(m-1)/(r-1)}/(m-1)!.$$

The cardinality of the union of the projections, $|Z_1|$, is the number of unique hyperplanes adjacent to F_1, which we seek to lower bound, so we reverse the preceding inequality,

$$|Z_1| \geq \left((m-1)! \cdot |F_1|\right)^{(r-1)/(m-1)}/(r-1)! \equiv L(|F_1|). \tag{6.1.3}$$

We now argue that when $|W_1|$ (and therefore $|F_1|$) is at its maximum, $|F_1| = w \equiv \frac{1}{2}\binom{n-r}{m-r-1}\binom{n}{r}$ from Equation 6.1.1, and using the fact that $|F_1| = \binom{n-r}{m-r-1}|W_1|$, the right side of the Equation 6.1.3 inequality becomes $L(w) \geq \frac{1}{2^{(r-1)/(m-1)}}\binom{n}{r-1}$ (with the assumption that $n \gg m, r$) since

$$\begin{aligned}
L(w) &= ((m-1)! \cdot w)^{(r-1)/(m-1)}/(r-1)! \\
&= \left((m-1)! \cdot \frac{1}{2}\binom{n-r}{m-r-1}\binom{n}{r}\right)^{(r-1)/(m-1)}/(r-1)! \\
&\geq \left(\frac{1}{2}n!/(n-(m-1))!\right)^{(r-1)/(m-1)}/(r-1)! \\
&= \frac{1}{2^{(r-1)/(m-1)}}\left(n!/(n-(m-1))!\right)^{(r-1)/(m-1)}/(r-1)!.
\end{aligned}$$

We now note that $k = n!/(n-(m-1))!$ is a product of $m-1$ terms, the largest $m-r$ of which are the product $n!/(n-(m-r))!$. This implies that we can lower bound $k^{(r-1)/(m-1)}$ as the product of the lowest $r-1$ terms, $k^{(r-1)/(m-1)} \geq (n-(m-r))!/(n-(m-1))!$. This implies that

$$\begin{aligned}
L(w) &= \frac{1}{2^{(r-1)/(m-1)}}k^{(r-1)/(m-1)}/(r-1)! \\
&\geq \frac{1}{2^{(r-1)/(m-1)}}[(n-(m-r))!/(n-(m-1))!]/(r-1)! \\
&= \frac{1}{2^{(r-1)/(m-1)}}\binom{n-(m-r)}{r-1}.
\end{aligned}$$

Applying Pascal's rule repeatedly, we lower bound the binomial coefficient as

$$\begin{aligned}\binom{n-(m-r)}{r-1} &= \binom{n}{r-1} - \binom{n-1}{r-2} - \binom{n-2}{r-2} - \cdots - \binom{n-(m-r)}{r-2} \\ &\geq (1-(m-r)(r-1)/n) \cdot \binom{n}{r-1}.\end{aligned}$$

Plugging this back into the bound for $L(w)$, we obtain

$$L(w) \geq \frac{1}{2^{(r-1)/(m-1)}}\left[\left(1-\frac{(m-r)(r-1)}{n}\right)\binom{n}{r-1}\right],$$

and applying our assumption that $n \gg m, r$, we approximate the inequality

$$L(w) \geq \frac{1}{2^{(r-1)/(m-1)}}\binom{n}{r-1}.$$

Now, the number of hyperedges R_1 contained inside the hyperplanes Z_1 is at least

$$|R_1| \geq \frac{(n-(r-1))|Z_1|}{r},$$

since each hyperplane contains $n-(r-1)$ hyperedges and each hyperedge can appear in at most r hyperplanes. Noting that each hyperedge is adjacent to every other hyperedge within any of its hyperplanes, we can conclude that Z_1 contains either hyperedges that are in W_1 or hyperedges that are cut (it cannot contain hyperedges that are internal to V_2). Therefore, we can lower bound the number of hyperedges in the cut as

$$\begin{aligned}|Q| \geq |R_1| - |W_1| &\geq \frac{(n-(r-1))|Z_1|}{r} - |W_1| \\ &\equiv T_1 - T_2.\end{aligned}$$

We would like to show that $T_1/T_2 > 1$ and drop $T_2 = |W_1|$. To do so, we first recall our $|F_1|$-dependent bound on $|Z_1|$ in Equation 6.1.3 and simplify its form (to what the generalized Loomis-Whitney inequality would yield, which is weaker),

$$\begin{aligned}|Z_1| &\geq ((m-1)! \cdot |F_1|)^{(r-1)/(m-1)}/(r-1)! \\ &\geq |F_1|^{(r-1)/(m-1)}.\end{aligned}$$

Since $T_1 = \frac{n-(r-1)}{r}|Z_1|$ has a fractional (less than one) power of $|F_1|$ while $T_2 = |W_1|$, the ratio of T_1/T_2 is minimized when $|W_1|$ is maximized, namely when $|W_1| = \frac{1}{2}\binom{n}{r}$. In this case, we have shown that $|Z_1| \geq L(w) \geq \frac{1}{2^{(r-1)/(m-1)}}\binom{n-r}{m-r-1}\binom{n}{r-1}$ (with the assumption that $n \gg m, r$), so we see that the ratio is at least

$$\begin{aligned}T_1/T_2 &= \frac{(n-(r-1))|Z_1|}{r} \cdot \frac{1}{|W_1|} \\ &\geq \frac{(n-(r-1))\frac{1}{2^{(r-1)/(m-1)}}\binom{n}{r-1}}{r} \cdot \frac{1}{\frac{1}{2}\binom{n}{r}} \\ &= \frac{\binom{n}{r}}{2^{(r-1)/(m-1)}} \cdot \frac{2}{\binom{n}{r}} \\ &= 2^{1-(r-1)/(m-1)} > 1.\end{aligned}$$

This ratio is greater than one by a small factor. Our approximation of $L(w)$ that used $n \gg m, r$ implies that the ratio is greater than one in the limit when $n \gg m, r$ and

both m and r are small constants (in application scenarios m never exceeds four for our algorithms). This lower bound on the ratio implies that

$$\begin{aligned}|Q| \geq T_1 - T_2 &\geq \left(1 - \frac{1}{2^{1-(r-1)/(m-1)}}\right) T_1 \\ &= \left(1 - \frac{1}{2^{1-(r-1)/(m-1)}}\right) \frac{(n-(r-1))|Z_1|}{r}.\end{aligned}$$

Remembering that we have $|Z_1| \geq |F_1|^{(r-1)/(m-1)}$ and that $|F_1| = \binom{n-r}{m-r-1}|W_1| \geq \binom{n-r}{m-r-1}\left\lfloor \frac{\binom{n}{r}}{2q\binom{m}{r}} \right\rfloor$ from Equation 6.1.2, we obtain

$$\begin{aligned}|Q| &\geq \left(1 - \frac{1}{2^{1-(r-1)/(m-1)}}\right) \frac{(n-(r-1))|Z_1|}{r} \\ &\geq \left(1 - \frac{1}{2^{1-(r-1)/(m-1)}}\right) \frac{(n-(r-1))\left[\binom{n-r}{m-r-1}\left\lfloor \frac{\binom{n}{r}}{2q\binom{m}{r}} \right\rfloor\right]^{(r-1)/(m-1)}}{r} \\ &= \Omega\left(n \cdot \left[n^{m-r-1} \cdot \frac{n^r}{q}\right]^{(r-1)/(m-1)}\right) = \Omega(n^r/q^{(r-1)/(m-1)}),\end{aligned}$$

which is greater (and so better) than the desired bound $|Q| = \Omega(n^r/q^{r/m})$ since $r/m > (r-1)/(m-1)$.

In case (II), we know that $\lfloor |V|/(2q) \rfloor = \lfloor \binom{n}{m}/(2q) \rfloor$ vertices $\bar{U} \subset V$ are disconnected by the cut (before the cut, every vertex was adjacent to d hyperedges). We define $\binom{m}{r}$ projections,

$$\pi_{s_1,\ldots,s_r}(e_{i_1,\ldots,i_m}) = (i_{s_1}, \ldots, i_{s_r}),$$

for $0 < s_1 < s_2 < \cdots < s_r \leq m$ corresponding to each of $\binom{m}{r}$ hyperedges adjacent to $v_{i_1,\ldots,i_m}$. In particular, the set of hyperedges adjacent to $\bar{U}$ is

$$\bar{\Pi} = \bigcup_{1 \leq s_1 < \cdots < s_r \leq m} \pi_{s_1,\ldots,s_r}(\bar{U}).$$

We apply the generalized Loomis-Whitney inequality (Theorem 5.3) to obtain a lower bound on the the size of the projections $|\bar{\Pi}|$,

$$\begin{aligned}|\bar{U}| &\leq \left(r! \cdot |\bar{\Pi}|\right)^{m/r}/m! \\ |\bar{\Pi}| &\geq (m! \cdot |\bar{U}|)^{r/m}/r!.\end{aligned}$$

Plugging in the lower bound on $|\bar{U}|$ corresponding to case (II) and discarding asymptotically small factors, we obtain the desired lower bound on the minimum number of hyperedges that must be cut,

$$\begin{aligned}\epsilon_q(H) &\geq \left(m! \cdot \left\lfloor \binom{n}{m}/(2q) \right\rfloor\right)^{r/m}/r! \\ &= \Omega\left(n^r/q^{r/m}\right). \quad \square\end{aligned}$$

6.2. Parent Hypergraphs

To apply our lower bounds to algorithms, we consider how their dependency graphs may be transformed into a more convenient form. In particular, we want to apply hypergraph transformations that do not increase the size of the minimum hyperedge cut.

We employ the idea that it is possible to merge sets of hyperedges that define connected components without increasing the size of any cut. Additionally, we employ the fact that discarding vertices in a hypergraph cannot increase the size of the minimum hyperedge cut that is balanced. A partition of the hyperedges E of any hypergraph $H = (V, E)$ defines a collection R of sub-hypergraphs of H: each part $Y \subset E$ in the partition of E and the corresponding set $W \subset V$ of vertices adjacent to Y in H defines a sub-hypergraph $(W, Y) \in R$. We say that a hypergraph $\hat{H} = (\hat{V}, \hat{E})$ with $\hat{V} \subset V$ is a *parent hypergraph* of $H = (V, E)$ if there exists such a partition R such that for each $(W, Y) \in R$, (W, Y) is connected and there is a unique hyperedge in the parent hypergraph $e \in \hat{E}$ corresponding to the hyperedge part, $e \subset W$. We will derive parent hypergraphs $\hat{H}$ for directed (dependency) graphs via the notion of weak connectivity in the directed graphs—that is, we will treat the directed edges as if they were undirected. This is consistent with our definitions of vertex separators, which do not allow edges in any direction between the two disconnected parts V_1 and V_2, meaning that $E \cap ((V_1 \times V_2) \cup (V_2 \times V_1)) = \emptyset$. Finding common parent hypergraphs will allow us to treat algorithms that are executable using multiple potential dependency graphs.

For example, employing parent hypergraphs allows us to consider all possible summation trees for a reduction algorithm. In particular, let $T = (V, E)$ be a tree in a dependency graph that sums a set of vertices $S \subset V$ into vertex $\hat{s} \in V$ via intermediate partial sums $V \backslash (S \cup \{\hat{s}\})$. Since T must be connected, we can define a hyperedge in a parent hypergraph $H = (S \cup \{\hat{s}\}, \{S \cup \{\hat{s}\}\})$, corresponding to this reduction tree, whose vertices contain all summands of H and the output, and which has a single hyperedge containing all of these vertices. We will then use Theorem 6.2, given next, to assert that the smallest vertex separator of T is no smaller than the hyperedge cut of H, the minimum size of which for a single reduction tree is simply one, as there is only one hyperedge in H. This lower bound will then apply to any reduction tree (any set of intermediate partial sums), as it considers only the summands and the output.

We obtain lower bounds on the vertex separator size of a directed or undirected graph by constructing a parent hypergraph in which every vertex is adjacent to a constant number of hyperedges. The following theorem states this result for vertex separators of (undirected) hypergraphs, which immediately generalizes the case of undirected and directed graphs.

THEOREM 6.2. *Given a hypergraph $H = (V, E)$, consider any parent hypergraph $\hat{H} = (\hat{V}, \hat{E})$ defined by a collection of sub-hypergraphs R as defined earlier. If the degree of each vertex in $\hat{H}$ is at most k, the minimum $\frac{1}{q}$-$\frac{1}{x}$-balanced hyperedge cut of $\hat{H}$, where $q \geq x \geq 4$, is no larger than k times the minimum size of a $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator of $\hat{V}$ in hypergraph H.*

PROOF. Consider any $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator $S \subset V$ of $\hat{V}$ inside H, for $q \geq x \geq 2$, that yields two disjoint vertex sets $\hat{V}_1$ and $\hat{V}_2$. We construct a hyperedge cut $\hat{X}$ of $\hat{H}$ consisting of all hyperedges corresponding to parts (elements of R) in which a vertex in the separator S appears. $\hat{X}$ is a cut of $\hat{H}$, since for any path consisting of hyperedges $\mathcal{P} = \{y_1, y_2, \ldots\}$ in $\hat{H}$ corresponding to parts $\{r_1, r_2, \ldots\}$, there exists a path in H consisting of hyperedges $\mathcal{Q} = \{e_{11}, e_{12}, \ldots\} \cup \{e_{21}, e_{22}, \ldots\} \cup \cdots$, where for each i, $\{e_{i1}, e_{i2}, \ldots\} \subset r_i$. Therefore, since S disconnects every path through H between $\hat{V}_1$ and $\hat{V}_2$, the cut $\hat{X}$ must also disconnect all such paths. Assuming without loss of generality that $|\hat{V}_1| \leq |\hat{V}_2|$, we let the vertex parts defined by the cut $\hat{X}$ of $\hat{H}$ be $\hat{V}_1 \cup S$ and $\hat{V}_2$, since $\hat{X}$ disconnects all hyperedges connected to S, so S can be included in either part.

Making this choice, we ensure that the size of the smaller part

$$|\hat{V}_1 \cup S| = |\hat{V}_1| + |S| \geq \lfloor |\hat{V}|/q \rfloor - |S| + |S| = \lfloor |\hat{V}|/q \rfloor,$$

which ensures the balance of the partition created by the cut $\hat{X}$. We can assert that $|\hat{V}_1 \cup S| \leq |\hat{V}_2|$, since we have assumed that $x \geq 4$, which implies that $|\hat{V}_1| \leq \lfloor |\hat{V}|/4 \rfloor$ and $|S| \leq \lfloor |\hat{V}|/4 \rfloor$. The bound on separator size, $|S| \leq \lfloor |\hat{V}|/4 \rfloor$, holds because any quarter of the vertices would lead to a valid vertex separator of any H by choosing S to be any subset of $\lfloor |\hat{V}|/4 \rfloor$ vertices and $\hat{V}_1 = \emptyset$. In this case, removing the hyperedges in Q leaves $S \cup \hat{V}_1 = S$ disconnected. Further, since the maximum degree of any vertex in $\hat{H}$ is k and each hyperedge in H appears in at most one part in R, which corresponds to a unique hyperedge in $\hat{H}$, the cut $\hat{X}$ of $\hat{H}$ is of size at most $k \cdot |S|$. □

7. LOWER BOUNDS FOR DENSE LINEAR ALGEBRA

We now apply the dependency expansion analysis from Section 4 to obtain lower bounds on the costs associated with algorithms for a few specific dense numerical linear algebra problems. Our analysis proceeds by obtaining lower bounds on the minimum-cut size of a parent lattice hypergraph for the family of dependency graphs representing the algorithm.

7.1. Triangular Solve

First we consider a family of dependency graphs G_{TRSV} for the standard algorithm computing the triangular solve (TRSV) problem. In TRSV, we are interested in computing a vector $\mathbf{x}$ of length n, given a dense nonsingular lower-triangular matrix $\mathbf{L}$ and a vector $\mathbf{y}$, satisfying $\mathbf{L} \cdot \mathbf{x} = \mathbf{y}$—that is, $\sum_{j=1}^{i} L_{ij} \cdot x_j = y_i$ for $i \in [1, n]$. Algorithm 1 describes the standard approach for computing TRSV. For analysis, we introduce the intermediate matrix $\mathbf{Z}$ (which need not be formed explicitly in practice), and corresponding intermediate "update" vertices $\{Z_{ij} : i \in [1, n], j \in [1, n], j < i\}$. We see that the computation of Z_{ij} for $i = [2, n]$ and some $j < i$ is a dependency of x_i, which is itself a dependency of the computation of Z_{ki} for all $k > i$. We note that recursive and blocked algorithms for the triangular solve both compute the same operations in Algorithm 1; however, they execute them in a different order. Further, changing the order of the summation on line 4 changes the dependency graph. Our lower bounds apply to the family of dependency graphs consisting of all orders of this summation, but exclude dependency graphs of algorithms that avoid computing $\mathbf{Z}$ via algebraic reorganization, such as by computing the inverse of $\mathbf{L}$.

ALGORITHM 1: $[\mathbf{x}] \leftarrow \text{TRSV}(\mathbf{L}, \mathbf{y}, n)$

1: **for** $i = 1$ to n **do**
2: **for** $j = 1$ to $i - 1$ **do**
3: $Z_{ij} = L_{ij} \cdot x_j$
4: $x_i \leftarrow \left(y_i - \sum_{j=1}^{i-1} Z_{ij}\right) / L_{ii}$

7.1.1. Lower Bounds. For fixed n, alternative orders exist for the summation on line 4, leading to a family of dependency graphs $G_{\text{TRSV}} = (V'_{\text{TRSV}}, E'_{\text{TRSV}})$. However, the set of vertices, V'_{TRSV}, defined by any dependency graph for this algorithm, must contain a vertex for each entry of $\mathbf{Z}$ and each entry of $\mathbf{x}$ (the variability comes from the partial sums computed within the summation). Further, any order of this summation must eventually combine all partial sums; therefore, the vertices corresponding to the computation of each x_i (i.e., Z_{ij} for all $j \in [1, i - 1]$) must all be weakly connected by some

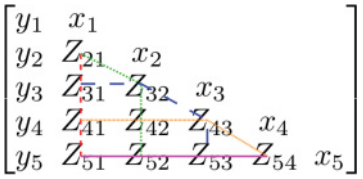

Fig. 3. Depiction of the hypergraph $H_{\text{TRSV}}(5)$ along with the inputs and outputs; each line of a different style corresponds to a hyperedge.

reduction tree. We will define a (2, 1)-lattice hypergraph (with indices in reverse order with respect to the definition in Section 6) of dimension n, $H_{\text{TRSV}} = (V_{\text{TRSV}}, E_{\text{TRSV}})$, which is a parent hypergraph for any possible G_{TRSV}, as we demonstrate next. This hypergraph contains the following vertices and hyperedges:

$$V_{\text{TRSV}} = \{Z_{ij} : i \in [2, n], j \in [1, i-1]\}$$
$$E_{\text{TRSV}} = \{e_i : i \in [1, n]\}, \quad \text{where } e_i = \{Z_{ij} : j \in [1, i-1]\} \cup \{Z_{ki} : k \in [i+1, n]\}.$$

We note that the vertices V_{TRSV} must be present in any G_{TRSV}; however, the vertices corresponding to $\mathbf{x}$, $\mathbf{y}$, and any intermediate partial summations done in reduction trees are omitted in H_{TRSV}. To show that H_{TRSV} is a parent hypergraph of any G_{TRSV}, we need to demonstrate that each hyperedge in H_{TRSV} corresponds to a unique part (sub-hypergraph) of any G_{TRSV}. The hyperedges E_{TRSV} can be enumerated with respect to either vector $\mathbf{x}$ or $\mathbf{y}$; the ith hyperedge $e_i \in E_{\text{TRSV}}$ includes all intermediate values on which x_i depends (Z_{ij} for $j \in [1, i-1]$), or which depend on x_i (Z_{ki} for $k \in [i+1, n]$). Therefore, each hyperedge e_i consists of vertices that are part of a connected component in any G_{TRSV} that corresponds to a unique set of edges: those in the reduction tree needed to form x_i from each Z_{ij} and those between x_i and each Z_{ki}. This hypergraph is depicted in Figure 3.

LEMMA 7.1. *Any $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator of $\mathbf{Z}$ (V_{TRSV}) in any dependency graph G_{TRSV} that subdivides the $n(n-1)/2$ vertices V_{TRSV} for any real numbers $4 \le x \le q \ll n$ must have size at least*

$$\chi_{q,x}(G_{\text{TRSV}}) = \Omega(n/\sqrt{q}).$$

PROOF. Any G_{TRSV} has the parent hypergraph H_{TRSV}, which we defined earlier. The maximum vertex degree of the parent hypergraph H_{TRSV} of G_{TRSV} is two, so by application of Theorem 6.2, the minimum vertex separator of G_{TRSV} is at least half the size of the minimum hyperedge cut of H_{TRSV}. By Theorem 6.1, any $\frac{1}{q}$-$\frac{1}{x}$-balanced hyperedge cut of a (2, 1)-lattice hypergraph of dimension n with $2 \le x \le q \ll n$ is of size $\Omega(n/\sqrt{q})$. Therefore, any vertex separator must be of size at least $\chi_{q,x}(G_{\text{TRSV}}) = \Omega(n/\sqrt{q})$. □

THEOREM 7.2. *Any parallelization of any dependency graph G_{TRSV} where some processor computes no more than $\frac{1}{x}$ of the elements in $\mathbf{Z}$ and at least $\frac{1}{q}$ elements in $\mathbf{Z}$ for $4 \le x \le q \ll n$ incurs the following computation (F), bandwidth (W), and latency (S) costs for some $b \in [1, n]$,*

$$F_{\text{TRSV}} = \Omega(n \cdot b), \qquad W_{\text{TRSV}} = \Omega(n), \qquad S_{\text{TRSV}} = \Omega(n/b),$$

and furthermore, $F_{\text{TRSV}} \cdot S_{\text{TRSV}} = \Omega(n^2)$.

PROOF. Consider any dependency graph $G_{\text{TRSV}} = (V'_{\text{TRSV}}, E'_{\text{TRSV}})$ for Algorithm 1. We note that the computation of $x_i \subset V'_{\text{TRSV}}$ for $i \in [1, n]$ requires the computation of Z_{jk} for $j, k \in [1, i]$ with $k < j$. Furthermore, no element Z_{lm} for $l, m \in [i+1, n]$ with $l > m$ may

be computed until x_i is computed. Consider any pair of vertices x_{i+1}, x_{i+b} that are on the dependency chain $\mathcal{P} = \{x_1, \ldots, x_n\}$. We recall that the dependency interval $\langle x_{i+1}, x_{i+b}\rangle$ is the set of all computations that depend on x_i and are a dependency of x_{i+b}. This dependency interval includes all vertices corresponding to a subtriangle of $\mathbf{Z}$, namely $Z_{kl} \in \langle x_{i+1}, x_{i+b}\rangle$ for $k, l \in [i+1, i+b]$ with $l < k$, and therefore $|\langle x_{i+1}, x_{i+b}\rangle| = \Theta(b^2)$. Further, $G_{\text{TRSV}}\langle x_{i+1}, x_{i+b}\rangle$ is isomorphic to some G'_{TRSV}, a dependency graph of a size b triangular solve. By Lemma 7.1, we have a lower bound on the vertex separator size of any such G'_{TRSV} and therefore on the dependency interval, $\chi_{q,x}(G_{\text{TRSV}}\langle x_{i+1}, x_{i+b}\rangle) = \Omega(b/\sqrt{q})$. Since the dependency intervals for triangular solve have second-order growth, we apply Corollary 4.2 with order $\omega = 2$ to obtain the conclusion for some $b \in [1, n]$. □

7.1.2. Attainability. The lower bounds presented earlier for triangular solve are attained by the communication-efficient blocked schedule suggested in Papadimitriou and Ullman [1987]. In Algorithm 2, we give a parallel algorithm that uses this blocking schedule with blocking factor b to compute the triangular solve. Our algorithm is similar to the wavefront algorithm given by Heath and Romine [1988].

ALGORITHM 2: $[\mathbf{x}] \leftarrow \text{TRSV}(\mathbf{L}, \mathbf{y}, n)$

```
1:  x = y
2:  for k = 1 to n/b do
3:      parfor l = max(1, 2k − n/b) to k do
4:          if l > 1 then Receive vector x[(2k − l − 1)b + 1 : (2k − l)b] from processor p_{l−1}
5:          for i = (2k − l − 1)b + 1 to (2k − l)b do
6:              for j = (l − 1)b + 1 to min(i − 1, lb) do
7:                  x_i ← (x_i − L_ij · x_j)
8:              if k = l then
9:                  x_i = x_i / L_ii
10:         if l < n/b then Send vector x[(2k − l − 1)b + 1 : (2k − l)b] to processor p_{l+1}
11:     parfor l = max(1, 2k + 1 − n/b) to k do
12:         if l > 1 then Receive vector x[(2k − l)b + 1 : (2k − l + 1)b] from processor p_{l−1}
13:         for i = (2k − l)b + 1 to (2k − l + 1)b do
14:             for j = (l − 1)b + 1 to lb do
15:                 x_i ← (x_i − L_ij · x_j)
16:         if l < n/b then Send vector x[(2k − l)b + 1 : (2k − l + 1)b] to processor p_{l+1}
```

The parallel Algorithm 2 can be executed using $p = n/b$ processors. Let processor p_l for $l \in [1, n/b]$ initially own L_{ij}, y_j for $i \in [1, n]$, $j \in [(l-1)b + 1, lb]$. Processor p_l performs parallel loop iteration l at each step of Algorithm 2. Since it owns the necessary panel of $\mathbf{L}$ and vector part x_j, no communication is required outside the vector send/receive calls listed in the code. Thus, at each iteration of the outer loop, at least one processor performs $O(b^2)$ work, and $2b$ (each sent vector is of length b) data is sent, requiring two messages. Therefore, this algorithm yields schedules that achieve the following costs over the n/b iterations,

$$F_{\text{TRSV}} = O(nb), \quad W_{\text{TRSV}} = O(n), \quad S_{\text{TRSV}} = O(n/b).$$

Thus, this algorithm can yield a schedule for any $b \in [1, n]$ that attains our communication lower bounds in Theorems 7.2 for that value of b. Parallel TRSV implementations in current numerical libraries, such as Elemental [Poulson et al. 2013] and ScaLAPACK [Blackford et al. 1997], employ schedules that attain our lower bound.

7.2. Cholesky Factorization

In this section, we consider Cholesky factorization of symmetric positive-definite matrices and show that the possible dependency graphs for the standard Cholesky factorization algorithm have third-order dependency interval growth and dependency graphs that satisfy the expansion properties necessary for the application of Corollary 4.2 with $\omega = 3$. We show that these factorizations of n-by-n matrices form an intermediate 3D tensor $\mathbf{Z}$ such that $Z_{ijk} \in \mathbf{Z}$ for $i > j > k \in [1, n]$, and Z_{ijk} depends on each Z_{lmn} for $l > m > n \in [1, j-1]$. We note that fast matrix multiplication techniques, such as Strassen's algorithm [Strassen 1969], use algebraic reorganization to compute a different intermediate and are outside the space of this analysis. Algorithms that explicitly invert triangular blocks, such as Tiskin [2002], are also outside the space of dependency graphs that we consider. However, blocked and recursive schedules for the Cholesky factorization algorithm correspond to dependency graphs within the family that we consider.

The Cholesky factorization of a symmetric positive definite matrix $\mathbf{A}$, $\mathbf{A} = \mathbf{L} \cdot \mathbf{L}^T$, results in a lower triangular matrix $\mathbf{L}$. A sequential algorithm for Cholesky factorization is given in Algorithm 3. We introduced an intermediate tensor $\mathbf{Z}$, whose elements must be computed during any execution of the algorithm, although $\mathbf{Z}$ itself need not be stored explicitly in an actual implementation.

ALGORITHM 3: $[\mathbf{L}] \leftarrow \text{Cholesky}(\mathbf{A}, n)$

1: **for** $j \leftarrow 1$ to n **do**
2: $\quad L_{jj} = \sqrt{A_{jj} - \sum_{k=1}^{j-1} L_{jk} \cdot L_{jk}}$
3: $\quad$ **for** $i \leftarrow j+1$ to n **do**
4: $\quad\quad$ **for** $k = 1$ to $j-1$ **do**
5: $\quad\quad\quad Z_{ijk} = L_{ik} \cdot L_{jk}$
6: $\quad\quad L_{ij} = (A_{ij} - \sum_{k=1}^{j-1} Z_{ijk})/L_{jj}$

7.2.1. Lower Bounds. We note that the summations in lines 2 and 6 of Algorithm 3 can be computed via any summation order (and will be computed in different orders in different parallel schedules). This implies that the summed vertices are connected in any dependency graph $G_{\text{Ch}} = (V'_{\text{Ch}}, E'_{\text{Ch}})$, but the connectivity structure may be different. Further, we know that $\mathbf{Z}$ and $\mathbf{L}$ must correspond to vertices in V_{Ch} in any G_{Ch}. To express all possible summation orders, we define a parent hypergraph for any dependency graph G_{Ch} that computes Algorithm 3, $H_{\text{Ch}} = (V_{\text{Ch}}, E_{\text{Ch}})$, a $(3, 2)$-lattice hypergraph (with indices in reverse order with respect to the definition in Section 6),

$$
\begin{aligned}
V_{\text{Ch}} &= \{Z_{ijk} : i, j, k \in [1, n], i > j > k\}, \\
E_{\text{Ch}} &= \{e_{ij} \quad : i, j \in [1, n] \text{ with } i > j\}, \text{ where} \\
e_{ij} &= \{Z_{ijk} : k \in [1, j-1]\} \\
&\quad \cup \{Z_{ikj} : k \in [j+1, i-1]\} \\
&\quad \cup \{Z_{kij} : k \in [i+1, n]\}.
\end{aligned}
$$

We note that the vertices V_{Ch} must be present in any G_{Ch}; however, the vertices corresponding to $\mathbf{L}$ and any intermediate partial summations done in reduction trees are omitted in H_{Ch}. To show that H_{Ch} is a parent hypergraph of any G_{Ch}, we need to demonstrate that each hyperedge in H_{Ch} corresponds to a unique part (sub-hypergraph) of any G_{Ch}. The hyperedges E_{Ch} can be enumerated with respect to $\mathbf{L}$. Hyperedge $e_{ij} \in E_{\text{Ch}}$ includes all intermediate values on which L_{ij} depends (Z_{ijk} for $k \in [1, i-1]$) or that

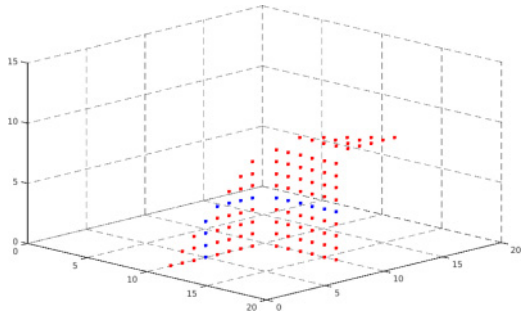

Fig. 4. A hyperplane (red) and a hyperedge (blue) in $H_{\text{Ch}}(16)$.

depend on L_{ij} (Z_{ikj} for $k \in [j+1, i-1]$ and Z_{kij} for $k \in [i+1, n]$). Therefore, each hyperedge e_{ij} consists of vertices that are part of a connected component in any G_{Ch} that corresponds to a unique set of edges: those in the reduction tree needed to form L_{ij} from each Z_{ijk} and those between L_{ij} and each Z_{ikj} as well as Z_{kij}.

We also define hyperplanes x_i for $i \in [1, n]$, where

$$x_i = e_{i,1} \cup e_{i,2} \cup \cdots \cup e_{i,i-1} \cup e_{i+1,i} \cup e_{i+2,i} \cup \cdots \cup e_{n,i}.$$

Figure 4 shows hyperplane x_{10} and hyperedge $e_{10,5}$ on $H_{\text{Ch}}(16)$. These hyperplanes correspond to those in the proof of Theorem 6.1.

LEMMA 7.3. *Any $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator S of $\mathbf{Z}$ within dependency graph G_{Ch} for any real numbers $4 \leq x \leq q \ll n$ must have size at least*

$$\chi_q(G_{\text{Ch}}) = \Omega(n^2/q^{2/3}).$$

PROOF. The maximum vertex degree of the parent hypergraph H_{Ch} of G_{Ch} is 3, so by application of Theorem 6.2, the minimum size of a $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator of G_{Ch} is at least one third that of a $\frac{1}{q}$-$\frac{1}{x}$-balanced hyperedge cut of H_{Ch}. By Theorem 6.1 with $m = 3$ and $r = 2$, any $\frac{1}{q}$-$\frac{1}{x}$-balanced hyperedge cut with $4 \leq x \leq q \ll n$ of H_{Ch} is of size $\Omega(n^2/q^{2/3})$. Therefore, any vertex separator of G_{Ch} must be of size at least $\chi_{q,x}(G_{\text{Ch}}) = \Omega(n^2/q^{2/3})$. □

THEOREM 7.4. *Any parallelization of any dependency graph G_{Ch}, where some processor computes no more than $\frac{1}{x}$ of the elements in $\mathbf{Z}$ (V_{Ch}) and at least $\frac{1}{q}$ elements in $\mathbf{Z}$, for any $4 \leq x \leq q \ll n$, must incur a communication of*

$$W_{\text{Ch}} = \Omega(n^2/q^{2/3}).$$

PROOF. For any G_{Ch}, every vertex that has an outgoing edge to a vertex computed by a different processor (different color) must be communicated (is in the communicated set). Since some processor computes no more than $\frac{1}{x}$ of the elements in $\mathbf{Z}$ (V_{Ch}) and at least $\frac{1}{q}$ elements in $\mathbf{Z}$, for any $4 \leq x \leq q \ll n$, the communicated set can be bounded below by the size of a $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator of $\mathbf{Z}$ in G_{Ch}. By Lemma 7.3, the size of any such separator is $\Omega(n^2/q^{2/3})$. □

THEOREM 7.5. *Any parallelization of any dependency graph G_{Ch} in which some processor computes no more than $\frac{1}{x}$ of the elements in $\mathbf{Z}$ (V_{Ch}) and at least $\frac{1}{q}$ elements in $\mathbf{Z}$, for any $4 \leq x \leq q \ll n$, incurs the following computation (F), bandwidth (W), and*

latency (S) costs for some $b \in [1, n]$*,*

$$F_{\text{Ch}} = \Omega(n \cdot b^2), \qquad W_{\text{Ch}} = \Omega(n \cdot b), \qquad S_{\text{Ch}} = \Omega(n/b),$$

and furthermore, $F_{\text{Ch}} \cdot S_{\text{Ch}}^2 = \Omega(n^3), \quad W_{\text{Ch}} \cdot S_{\text{Ch}} = \Omega(n^2)$.

PROOF. Consider any dependency graph $G_{\text{Ch}} = (V'_{\text{Ch}}, E'_{\text{Ch}})$, which computes Algorithm 3. We note that the computation of $L_{ii} \in V'_{\text{Ch}}$ for $i \in [1, n]$ requires the computation of $Z_{lmk} \in V_{\text{Ch}} \subset V'_{\text{Ch}}$ for $l, m, k \in [1, i]$ with $l > m > k$. Furthermore, no element $Z_{srt} \in V_{\text{Ch}} \subset V'_{\text{Ch}}$ for $s, r, t \in [i + 1, n]$ with $s > r > t$ can be computed until L_{ii} is computed. Consider any pair of vertices $L_{i+1,i+1}$ and $L_{i+b,i+b}$ along the dependency chain $\mathcal{P} = \{L_{11}, \ldots, L_{nn}\}$. The dependency interval $\langle L_{i+1,i+1}, L_{i+b,i+b}\rangle$ includes vertices corresponding to a subcube of $\mathbf{Z}$, namely Z_{klm} for $k, l, m \in [i + 1, i + b]$ with $k > l > m$, and therefore $|\langle L_{i+1,i+1}, L_{i+b,i+b}\rangle| = \Theta(b^3)$. Further, $G\langle L_{i+1,i+1}, L_{i+b,i+b}\rangle$ is isomorphic to some G'_{Ch}, a schedule for a Cholesky factorization of a b-by-b symmetric matrix. By Lemma 7.3, we can lower bound the separator of any such G'_{Ch} and therefore of the dependency interval, $\chi_{q,x}(G\langle L_{i+1,i+1}, L_{i+b,i+b}\rangle) = \Omega(b^2/q^{2/3})$. Since we have third-order growth of dependency intervals with second-order growth of the cross sections of the dependency interval subgraphs, we apply Corollary 4.2 with $\omega = 3$ to obtain the conclusion for some $b \in [1, n]$. □

We conjecture that the lower bounds demonstrated for Cholesky in this section also extend to LU, LDL^T, and QR factorizations, as well as algorithms for computation of the eigenvalue decomposition of a symmetric matrix and the singular value decomposition.

7.2.2. Attainability. The lower bounds presented in the Section 7.2.1 are attained on p processors for $b \approx n/\sqrt{p}$ by "2D algorithms," which utilize a blocked matrix layout and are employed by most standard parallel libraries, including Elemental [Poulson et al. 2013] and ScaLAPACK [Blackford et al. 1997]. The BSP [McColl and Tiskin 1999] algorithms for LU factorization without pivoting in Tiskin [2002], as well as LU with pairwise pivoting ([Sorensen 1985]) along with QR factorization with Givens rotations in Tiskin [2007], match the lower bounds in Theorem 7.4 when $b = n/p^{2/3}$ and Theorem 7.5 whenever $b \in [n/p^{2/3}, n/\sqrt{p}]$. Therefore, Tiskin's algorithms can lower the bandwidth cost with respect to 2D algorithms by a factor of up to $p^{1/6}$, at the cost of increasing the latency cost by the same factor. Similarly, 2.5D algorithms [Solomonik and Demmel 2011] for LU factorization (without pivoting, and with tournament pivoting [Grigori et al. 2011]) also lower bandwidth cost by a factor of up to $p^{1/6}$ by sacrificing latency cost (the nonpivoted 2.5D LU algorithm in Solomonik and Demmel [2011] also have an additional log p factor in the latency cost of collective communication, since the messaging model in Solomonik and Demmel [2011] did not allow bulk synchronization, which is possible in the model in this article and in BSP).

However, the algorithms in Tiskin [2002] (when adapted to Cholesky factorization) are outside the family of dependency graphs that compute Algorithm 3, since they perform triangular matrix inversion as are the algorithms that employ pairwise pivoting in Tiskin [2007]. On the other hand, the algorithms in Solomonik and Demmel [2011] do not employ matrix inversion and (when adapted to Cholesky factorization) are simply schedules of Algorithm 3. The 2.5D algorithms are practical and can improve on the performance of standard 2D algorithms as long as the matrix is large enough to amortize synchronization overheads. Therefore, the latency-bandwidth trade-off is of practical importance for this problem. We note that the 3D parallel LU factorization algorithm given by Irony and Toledo [2002], a major motivation for the 2.5D LU algorithm, is not optimal in our model, as it does not minimize

interprocessor bandwidth cost along the critical path, but only total communication volume.

We also note that the Floyd [1962] and Warshall [1962] all-pairs shortest-paths graph algorithm has the same dependency structure as Cholesky factorization for undirected graphs (and Gaussian elimination for directed graphs), so our lower bounds may be easily extended to this case. However, alternative algorithms, particularly Tiskin's path-doubling algorithm [Tiskin 2001], are capable of solving the all-pairs shortest-paths problem with the same communication and computation costs as matrix multiplication, and a synchronization cost of only $O(\log(p))$. Tiskin's approach exploits the idempotency of the tropical semiring and thus is not immediately extensible to numerical Gaussian elimination.

8. LOWER BOUNDS FOR STENCIL COMPUTATIONS

We now consider stencil computations, analogous to repeated sparse matrix-vector multiplications with a specially structured matrix. Although these computations achieve good asymptotic scalability with respect to problem size by preserving sparsity, the amount of data reuse that these schedules can exploit is small with respect to dense linear algebra algorithms such as Gaussian elimination. One technique that remedies this lack of data reuse is performing blocking across stencil applications (applying multiple stencil applications to only a part of the mesh or, equivalently, repeatedly multiplying by only a part of the vector) rather than doing each complete stencil application in sequence. This technique is also referred to as time skewing. Although this technique reduces synchronization and memory-bandwidth cost, it usually requires extra interprocessor communication costs. In this section, we introduce the dependency graphs of stencil computations and apply our lower bound techniques to derive tradeoffs between synchronization cost and the costs of computation and interprocessor communication.

8.1. Stencil Computations

We consider the s-step $(2m+1)^d$-point (with stencil radius $m \geq 1$) stencil computation

$$\mathbf{x}^{(l)} = \mathbf{A} \cdot \mathbf{x}^{(l-1)},$$

for $l \in [1, s]$, $\mathbf{x}^{(0)}$ given as input, and $\mathbf{A}$ is a symmetric sparse matrix of dimension n^d, which encodes the stencil interactions. In particular, the nonzero structure of $\mathbf{A}$ corresponds to the adjacency matrix of a graph $T = (V_T, E_T)$ with vertices $v_{i_1,\ldots,i_d} \in V_T$ for all $i_1, \ldots, i_d \in [1, n]$ and containing all edges $(v_{i_1,\ldots,i_d}, w_{j_1,\ldots,j_d}) \in E_T$ such that for $k \in [1, d]$, $i_k, j_k \in [1, n]$, $|j_k - i_k| \leq m$. Thus, matrix $\mathbf{A}$ and vectors $\mathbf{x}^{(l)}$, $l \in [0, s]$, have dimension n^d, and $\mathbf{A}$ has $\Theta(m^d)$ nonzeros per row/column. We suppose that the entries of $\mathbf{A}$ are implicit, meaning that they can be computed in cache (i.e., without incurring a memory bandwidth cost). We also assume that $s \leq n/m$ (the lower bound could also be applied to subsets of the computation with $s' = \lfloor n/m \rfloor$). We note that the dependency structure of this computation is analogous to direct force evaluations in particle simulations and the Bellman-Ford shortest-paths algorithm, which may be expressed as sparse matrix-vector multiplication, but on a different algebraic semiring. However, the sparsity structure in these applications may be more general.

We let $G_{\mathrm{St}} = (V_{\mathrm{St}}, E_{\mathrm{St}})$ be the dependency graph of the s-step stencil computation defined earlier. We index the vertices $V_{\mathrm{St}} \ni v_{i_1,\ldots,i_d,l}$ with $d+1$ coordinates, each corresponding to the computation of an intermediate vector element $x_k^{(l)}$ for $l \in [1, s]$ and $k = \sum_{j=1}^{d} (i_j - 1) n^{j-1} + 1$, where the index k corresponds to the lexicographical ordering of $[1, n]^d$. For each edge $(v_{i_1,\ldots,i_d}, w_{j_1,\ldots,j_d})$ in E_T and each $l \in [1, s]$, there is an edge $(v_{i_1,\ldots,i_d,l-1}, w_{j_1,\ldots,j_d,l})$ in E_{St}. By representing the computation of each $x_k^{(l)}$ as a single

vertex, we have precluded the parallelization of the m^d scalar multiplications needed to compute each such vertex (instead, we simply have edges corresponding to these dependencies). Accordingly, the computation costs that we give subsequently will be scaled by m^d for each vertex computed.

8.2. Communication Lower Bounds for Stencil Computations

We now extract a subgraph of G_{St} and lower bound the execution cost of a load-balanced computation of this subgraph. Consider the following dependency chain $\mathcal{P}$,

$$\mathcal{P} = \left\{v_{1,\ldots,1,1}, \ldots, v_{1,\ldots,1,s}\right\}.$$

The dependency interval subgraph $\hat{G}_{\text{St}} \equiv G_{\text{St}}\langle v_{1,\ldots,1,1}, v_{1,\ldots,1,s}\rangle = (\hat{V}_{\text{St}}, \hat{E}_{\text{St}})$ includes

$$\hat{V}_{\text{St}} = \big\{v_{i_1,\ldots,i_d,i_{d+1}} : 1 \le i_{d+1} \le s, \\ \text{and } \forall j \in [1,d],\ i_j \le \max(1, m \cdot \min(i_{d+1}, s - i_{d+1}))\big\},$$

as well as all edges between these vertices $\hat{E}_{\text{St}} = (\hat{V}_{\text{St}} \times \hat{V}_{\text{St}}) \cap E_{\text{St}}$. In the following theorem, we lower bound the communication and synchronization costs required to compute this subset of the computation. The lower bound is independent of n and consequently of the global problem size, as we are only considering a subgraph of the dependency graph. This lower bound is effectively a limit on the available parallelism.

THEOREM 8.1. *Any execution of an s-step $(2m+1)^d$-point stencil computation with $m \ge 1$ on a dth order mesh with $d \ll s$, where some processor computes at most $\frac{1}{x}$ of $\hat{V}_{\text{St}}$ and at least $\frac{1}{q}$ of $\hat{V}_{\text{St}}$, for some $4 \le x \le q \ll s$, requires the following computational, bandwidth, and latency costs for some $b \in [1, s]$,*

$$F_{\text{St}} = \Omega(m^{2d} \cdot b^d \cdot s),\ W_{\text{St}} = \Omega(m^d \cdot b^{d-1} \cdot s),\ S_{\text{St}} = \Omega(s/b),$$

and furthermore,

$$F_{\text{St}} \cdot S_{\text{St}}^d = \Omega(m^{2d} \cdot s^{d+1}), \qquad W_{\text{St}} \cdot S_{\text{St}}^{d-1} = \Omega(m^d \cdot s^d).$$

PROOF. Our proof approach employs the following steps:

(1) A dependency chain in the time (stencil iteration) dimension is defined, and dependency intervals between pairs of vertices on this chain are analyzed.
(2) For each dependency interval subgraph, a "block" dependency graph is defined, where each block corresponds to a set of vertices within half of the stencil radius at the same stencil iteration.
(3) A parent lattice hypergraph of this block dependency interval subgraph is defined by considering a subset of the blocks.
(4) A lower bound on the minimum cut of the lattice hypergraph is obtained via Theorem 6.1 and subsequently for the minimum vertex separator block dependency graph via Theorem 6.2.
(5) A lower bound on the minimum vertex separator (communicated set) of the dependency interval is obtained from the block graph, using the fact that all vertices of each block are dependent on all vertices of each adjacent block of the previous stencil iteration.
(6) The lower bound on the minimum vertex separator allows us to apply Corollary 4.2 with dependency interval order $d+1$ to obtain the desired results.

In the following analysis, we will discard factors of d, as we assume that d is a constant, which is reasonable for most problems of interest ($d \le 4$), but some assumptions

in this analysis may need to be revisited if lower bounds for stencil computations on high-order meshes are desired.

Consider the dependency chain $\mathcal{P}$, which defines $\hat{G}_{\text{St}}$, and any pair of vertices on this dependency chain $v_{1,\ldots,1,h+1}$ and $v_{1,\ldots,1,h+r}$ where $r \geq 3$. The dependency interval between these vertices is

$$\langle v_{1,\ldots,1,h+1}, v_{1,\ldots,1,h+r}\rangle = \{v_{i_1,\ldots,i_d,i_{d+1}} : h+1 \leq i_{d+1} \leq h+r, \text{ and } \forall j \in [1,d],\ i_j \leq \max(1, m \cdot \min(i_{d+1}-h-1, h+r-i_{d+1}))\}.$$

For all $(u_1, \ldots, u_{d+1}) \in [0, r-3]^{d+1}$ (starting at 0 to simplify later transformations and ending at $r-3$, as we exclude the two endpoints of the dependency chain since they do not correspond to blocks as defined later), we define the following blocks of the vertices inside G_{St}

$$B_{u_1,\ldots,u_{d+1}} = \big\{v_{i_1,\ldots,i_{d+1}} \in V_{\text{St}} : \forall j \in [1,d],\ i_j \in \{\lceil m/2\rceil u_j + 1, \ldots, \lceil m/2\rceil (u_j+1)\} \quad i_{d+1} = u_{d+1} + h + 2\big\}.$$

Thus, each block should contain $\lceil m/2\rceil^d$ vertices on the same level, u_{d+1}. We note that because the dimension of each block is $\lceil m/2\rceil$ and the interaction distance (stencil radius) is m, every vertex in $B_{u_1,\ldots,u_{d+1}}$ depends on every vertex in $B_{y_1,\ldots,y_d,u_{d+1}-1}$ such that $\forall i \in [1,d]$, $|y_i - u_i| \leq 1$.

We now construct a DAG $G'_{\text{St}} = (V'_{\text{St}}, E'_{\text{St}})$ corresponding to the connectivity of (a subset of) the blocks within the given dependency interval subgraph $\hat{G}_{\text{St}}\langle v_{1,\ldots,1,h+1}, v_{1,\ldots,1,h+r}\rangle$, enumerating them on a lattice graph of order $d+1$ and dimension $g = \lfloor (r-2)/(d+1)\rfloor$. For each $(\nu_1, \ldots, \nu_{d+1}) \in [0, g-1]^{d+1}$ with $\nu_1 \leq \nu_2 \leq \cdots \leq \nu_{d+1}$, we have a vertex in the lattice $w_{\nu_1,\ldots,\nu_{d+1}} \in V'_{\text{St}}$, which corresponds to block $B_{u_1,\ldots,u_{d+1}} \subset V_{\text{St}}$, where for $i \in [1,d]$, $u_i = \nu_{i+1} - \nu_i$ and $u_{d+1} = \sum_{j=1}^{d+1} \nu_j$. We first show that this mapping of lattice vertices to blocks is injective by considering any pair of two different vertices in the lattice: $w_{\nu_1,\ldots,\nu_{d+1}}, w_{\nu'_1,\ldots,\nu'_{d+1}} \in V'_{\text{St}}$. If they map to the same block $B_{u_1,\ldots,u_{d+1}}$, they must have the same u_{d+1}, which implies that $\sum_{j=1}^{d+1} \nu_j = \sum_{j=1}^{d+1} \nu'_j$. Consider the highest index j in which they differ, $\max\{j : \nu_j - \nu'_j \neq 0\}$, and assume without loss of generality that $\nu_j > \nu'_j$. Now, if $j = 1$, this is the only differing index, and thus the sum of the indices cannot be the same. Otherwise, since $u_{j-1} = \nu_j - \nu_{j-1} = \nu'_j - \nu'_{j-1}$, we must also have $\nu_{j-1} > \nu'_{j-1}$. Repeated application of this argument implies that for any $k \leq j$, we have $\nu_k > \nu'_k$, which yields a contradiction $\sum_{j=1}^{d+1} \nu_j > \sum_{j=1}^{d+1} \nu'_j$. Therefore, the mapping of lattice vertices to blocks is injective.

We now argue that the blocks corresponding to the lattice vertices of G'_{St} have all of their vertices inside the dependency interval $\langle v_{1,\ldots,1,h+1}, v_{1,\ldots,1,h+r}\rangle$. The first lattice vertex $w_{0,\ldots,0} = B_{0,\ldots,0}$, since for $i \in [1,d]$, $u_i = \nu_{i+1} - \nu_i = 0$ and $u_{d+1} = \sum_{j=1}^{d+1} \nu_j = 0$. Therefore, the first lattice vertex (block $B_{0,\ldots,0}$) contains the following vertices of V_{St},

$$B_{0,\ldots,0} = \big\{v_{i_1,\ldots,i_{d+1}} \in V_{\text{St}} : \forall j \in [1,d],\ i_j \in \{1, \ldots, \lceil m/2\rceil\},\ i_{d+1} = h+2\big\},$$

and all of these vertices are dependent on the first vertex in the dependency interval, $v_{1,1,\ldots,1,h+1}$.

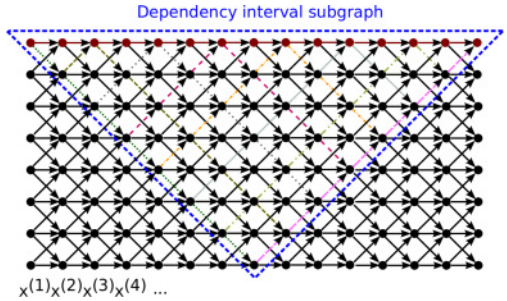

Fig. 5. Depiction of the dependency interval (large blue triangle) from the endpoints of a dependency chain (top solid red path) within a stencil computation on a first-order mesh. Each vertex corresponds to a block as defined in the proof of Theorem 8.1. Edges within the dependency interval subgraph are colored and dashed according to the hypergraph edge to which they correspond in the constructed hypergraph, H'.

Further, the last lattice vertex $w_{g,\ldots,g}$ corresponds to $B_{0,\ldots,0,u_{d+1}}$ with $u_{d+1} \in [r-2-d, r-2]$, since for $i \in [1,d]$, $u_i = v_{i+1} - v_i = 0$ and

$$u_{d+1} = \sum_{j=1}^{d+1} v_j = (d+1)g = (d+1)\lfloor (r-2)/(d+1) \rfloor \in [r-2-d, r-2].$$

The set of vertices of V_{St} in this last block are

$$B_{0,\ldots,0,u_{d+1}} = \left\{ v_{i_1,\ldots,i_{d+1}} \in V_{\mathrm{St}} : \forall j \in [1,d],\ i_j \in \{1,\ldots,\lceil m/2 \rceil\},\ i_{d+1} = u_{d+1} + h + 2 \right\}.$$

Since $u_{d+1} + h + 1 \leq h + r - 1$, all of these vertices appear at least one level (value of index u_{d+1}) before the last vertex on the dependency interval, which is $v_{1,\ldots,1,h+r}$. Further, since this last block contains values at most $m/2$ away from the origin of the mesh, $(i_1,\ldots,i_d) = (1,\ldots,1)$, the last vertex on the dependency interval, $v_{1,\ldots,1,h+r}$ depends on all vertices in the last lattice vertex (block).

Each vertex $w_{v_1,\ldots,v_{d+1}}$ is connected to vertices $w_{v_1,\ldots,v_j+1,\ldots,v_{d+1}}$ for $j \in [1,d+1]$ (so long as either $j = d+1 \wedge v_{d+1} \leq g$ or $j \leq d \wedge v_{j+1} > v_j$) by edges in E'_{St}, since if $w_{v_1,\ldots,v_{d+1}}$ corresponds to $B_{u_1,\ldots,u_{d+1}}$ then $w_{v_1,\ldots,v_j+1,\ldots,v_{d+1}}$ corresponds to $B_{u_1,\ldots,u_{j-1}+1,u_j-1,\ldots,u_d,u_{d+1}+1}$, which is dependent on the former block. Therefore, for any lattice vertex $w_{v_1,\ldots,v_{d+1}}$, there is a dependency path in the lattice from the first lattice vertex $w_{0,\ldots,0}$ to $w_{v_1,\ldots,v_{d+1}}$ and a dependency path from $w_{v_1,\ldots,v_{d+1}}$ to the last lattice vertex $w_{g,\ldots,g}$. Since the first lattice vertex is dependent on the origin of the dependency interval $v_{1,1,\ldots,1,h+1}$ and the last lattice vertex is a dependency of the last vertex $v_{1,1,\ldots,1,h+r}$, this implies that all lattice vertices (blocks) are part of the dependency interval $\langle v_{1,\ldots,1,h+1}, v_{1,\ldots,1,h+r} \rangle$. We give a depiction of the lattice of blocks in the dependency interval with $d = 1$ in Figure 5.

We now transform the block graph G'_{St} into a $(d+1,d)$-lattice hypergraph $H' = (V_H, E_H)$ of dimension g so that for $4 \leq x \leq q \ll s$, a $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator of G'_{St} is proportional to a $\frac{1}{q}$-$\frac{1}{x}$-balanced hyperedge cut of H'. We define $V_H = \{w_{i_1,\ldots,i_{d+1}} \in V'_{\mathrm{St}} : i_1 < i_2 < \cdots < i_{d+1}\}$ (note that V'_{St} included diagonals, but V_H does not), and the hyperedges E_H correspond to a unique member of a disjoint partition of the edges in G'_{St}. In particular, we define sets of hyperedges $e_{i_1,\ldots,i_d} \in E_H$ for $i_1,\ldots,i_d \in [1,g]$ with $i_1 < \cdots < i_d$ to contain all $w_{j_1,\ldots,j_{d+1}}$ that satisfy $j_1 < \cdots < j_{d+1}$ and $\{i_1,\ldots,i_d\} \subset \{j_1,\ldots,j_{d+1}\}$. Each of these hyperedges $e_{i_1,\ldots,i_d}$ corresponds to vertices along an

edge-disjoint dependency path within G'_{St},

$$y_{i_1,\ldots,i_d} \subset \bigcup_{k=1}^{i_1-1}(w_{k,i_1,\ldots,i_d}, w_{k+1,i_1,\ldots,i_d}) \cup \bigcup_{k=i_1}^{i_2-1}(w_{i_1,k,i_2\ldots,i_d}, w_{i_1,k+1,i_2,\ldots,i_d}) \cup \cdots \cup \bigcup_{k=i_d}^{g-1}(w_{i_1,\ldots,i_d,k}, w_{i_1,\ldots,i_d,k+1}),$$

where each pair of vertices in the union corresponds to a unique edge in G'_{St}. Because these hyperedges correspond to disjoint subsets of E'_{St}, any $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator of G'_{St} can be transformed into a $\frac{1}{q}$-$\frac{1}{x}$-balanced hyperedge cut of size no greater than $d+1$ times than the size of some $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator by application of Theorem 6.2, since the degree of any vertex in H' is at most $d+1$ and since H' is a parent hypergraph of G'_{St}.

Since the hypergraph H' is a $(d+1,d)$-lattice hypergraph of dimension $g = \lfloor (r-2)/(d+1) \rfloor$, by Theorem 6.1, its $\frac{1}{q}$-$\frac{1}{x}$-balanced hyperedge cut for $4 \leq x \leq q \ll s$ has size $\epsilon_q(H') = \Omega(g^d/q^{d/(d+1)})$. Furthermore, since the $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator of G'_{St} is at least $\frac{1}{d+1}\epsilon_q(H')$,

$$\chi_{q,x}(G'_{\mathrm{St}}) = \Omega\left(\frac{g^d}{(d+1)q^{d/(d+1)}}\right) = \Omega(r^d),$$

where the last bound follows since we have $d, x, q \ll s$.

This lower bound on edge cut size in the block graph G'_{St} allows us to obtain a lower bound on the size of any $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator of $\hat{G}_{\mathrm{St}}\langle v_{1,\ldots,1,h+1}, v_{1,\ldots,1,h+r}\rangle$ that is larger by a factor of $\Omega(m^d)$. Consider any $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator of $\hat{G}_{\mathrm{St}}\langle v_{1,\ldots,1,h+1}, v_{1,\ldots,1,h+r}\rangle$ that separates the vertices into three disjoint subsets: the separator Q and the parts V_1 and V_2. If two vertices $u, v \in \hat{V}_{\mathrm{St}}$ are in two different parts (are of different color), $u \in V_1$ and $v \in V_2$, and are in the same block, then all vertices in the d dependent adjacent blocks must have all of their vertices entirely in Q, as all vertices in adjacent blocks in G'_{St} are adjacent to u and v in $\hat{G}_{\mathrm{St}}$. Each of these d blocks that neighbors the multicolored block containing u and v may neighbor at most d other multicolored blocks on whose vertices it is dependent, and therefore each multicolored block corresponds to one block whose vertices are entirely in Q. Thus, the number of vertices in blocks of different colors is no greater than the size of the separator, $|Q|$, and therefore is small with respect to the number of blocks in each part, which is at least $|\langle v_{1,\ldots,1,h+1}, v_{1,\ldots,1,h+r}\rangle|/q$. So, most vertices that are not in the separator (e.g., in part V_1) must be in blocks that contain only vertices from V_1 or Q. Thus, Q should yield $\Omega(|V'_{\mathrm{St}}|/q)$ blocks that contain vertices that are either in the separator or in V_1, and similarly for V_2. Now, since two blocks $B_1 \subset (V_1 \cup Q)$ and $B_2 \subset (V_2 \cup Q)$ that contain nonseparator vertices of different color cannot be adjacent, there must exist a separator block $B_3 \subset Q$ for any dependency path on the lattice between B_1 and B_2. Therefore, Q also induces a separator of G'_{St} of size no larger than $|Q| \cdot \lceil m/2 \rceil^d$. So, we obtain the following lower bound on the size of a $\frac{1}{q}$-$\frac{1}{x}$-balanced vertex separator of $\hat{G}_{\mathrm{St}}\langle v_{1,\ldots,1,h+1}, v_{1,\ldots,1,h+r}\rangle$: $\chi_{q,x}(\hat{G}_{\mathrm{St}}\langle v_{1,\ldots,1,h+1}, v_{1,\ldots,1,h+r}\rangle) = \Omega(m^d \cdot \chi_{q,x}(G'_{\mathrm{St}})) = \Omega(m^d r^d)$.

Now, the size of the dependency interval is $|\langle v_{1,\ldots,1,h+1}, v_{1,\ldots,1,h+r}\rangle| = \Omega(m^d r^{d+1})$, and the total length of our main dependency chain in $\hat{G}_{\mathrm{St}}$ is $|\mathcal{P}| = s$. $\hat{G}_{\mathrm{St}}$ is a (ϵ, σ)-path-expander (defined in Section 4.2) with $\epsilon(b) = m^d b^d$ and $\sigma(b) = m^d b^{d+1}$. Therefore, by application of Theorem 4.1 with $k = 3$, we obtain the following lower bounds (scaling by m^d the computation cost for each vertex) for some integer $b \in [3, s]$,

$$F_{\mathrm{St}} = \Omega(m^{2d} \cdot b^d \cdot s), \quad W_{\mathrm{St}} = \Omega(m^d \cdot b^{d-1} \cdot s), \quad S_{\mathrm{St}} = \Omega(s/b). \quad \square$$

9. SCHEDULES FOR STENCIL COMPUTATIONS

Having established lower bounds for stencil computations (defined in Section 8.1) in Section 8.2, we now check their attainability. We first recall the costs of a known technique that attains these bounds. In the case that the matrix A is implicitly represented, we propose an alternate idea that improves memory-bandwidth efficiency while keeping interprocessor bandwidth communication cost low, albeit at the price of increasing network latency cost. We limit our analysis to the case when only the last vector, $x^{(s)}$ is needed, although potentially the algorithms discussed could be extended to the case where a Gram matrix, $[x^{(0)}, \ldots, x^{(s)}]^T \cdot [x^{(0)}, \ldots, x^{(s)}]$ is needed by recomputing these Krylov basis vectors [Carson et al. 2013; Mohiyuddin et al. 2009].

A parallel communication-avoiding schedule for computing a stencil, termed *PA1*, is presented in Demmel et al. [2008]. The idea of applying blocking (tiling) across stencil applications for stencil methods is much older (e.g., see Hong and Kung [1981] and Leiserson et al. [1997]). Computing an s-step $(2m + 1)^d$-point stencil with block size $b \in [1, s]$ can be accomplished by $\lceil s/b \rceil$ invocations of PA1 with basis size parameter b. For each invocation of PA1, every processor imports a volume of the mesh of size $(n/p^{1/d} + bm)^d$ and computes b sparse matrix-vector multiplications on these mesh points without further communication, updating a smaller set at each level so as to keep all dependencies local. Over s/b invocations of PA1, the computation cost, which includes m^d updates to the locally stored and imported vertices, is given by

$$F_{\mathrm{PA1}} = O(s \cdot m^d \cdot (n/p^{1/d} + b \cdot m)^d).$$

The communication cost of each PA1 invocation is proportional to the import volume, $(n/p^{1/d} + bm)^d$, minus the part of the mesh that is initially owned locally, which is of size n^d/p, so

$$W_{\mathrm{PA1}} = O((s/b) \cdot ((n/p^{1/d} + b \cdot m)^d - n^d/p)).$$

Since at every invocation only one round of communication is done, the synchronization cost is

$$S_{\mathrm{PA1}} = O(s/b),$$

as long as $n/p^{1/d} \geq bm$, which means the ghost (import) zones do not extend past the mesh points owned by the nearest-neighbor processors in the processor grid (although the schedule can be extended to import data from processors further away with a proportionally higher synchronization cost). When the problem assigned to each processor is large enough (i.e., $n/p^{1/d} \gg bm$), the costs for PA1 are

$$F_{\mathrm{PA1}} = O(m^d \cdot s \cdot n^d/p), \quad W_{\mathrm{PA1}} = O(m \cdot s \cdot n^{d-1}/p^{(d-1)/d}), \quad S_{\mathrm{PA1}} = O(s/b).$$

However, in the strong scaling limit, $n/p^{1/d} = \Theta(bm)$, the costs become dominated by

$$F_{\mathrm{PA1}} = O(m^{2d} \cdot b^d \cdot s), \quad W_{\mathrm{PA1}} = O(m^d \cdot b^{d-1} \cdot s), \quad S_{\mathrm{PA1}} = O(s/b),$$

limiting the scalability of the schedule. These costs of PA1 asymptotically match the lower bounds and lower bound trade-offs that we proved in Theorem 8.1, demonstrating that the algorithm is optimal in this scaling limit. We also remark that, strictly

speaking, Theorem 8.1 does not apply to PA1 because PA1 performs redundant computation, which changes the dependency graph (albeit it is only a constant factor more work when $n/p^{1/d} = \Omega(bm)$). However, we claim that the costs of PA1 can be achieved by a schedule that avoids redundant computation altogether, following a strategy such as Tang et al. [2011], which avoids recomputed overlapped regions between blocks. Such a schedule should be more expensive than PA1 by at most constant factor increases in bandwidth and latency, thus showing that the lower bounds and trade-offs can be attained without recomputation.

In addition to (interprocessor) latency savings, another advantage of executing PA1 with $b > 1$ is improved memory-bandwidth efficiency. For instance, assuming the matrix $\mathbf{A}$ is implicit and so only the vector entries (mesh points rather than edges) need to be stored in cache, when $b = 1$ (this standard case is referred to as PA0 in Demmel et al. [2008]), and when the local mesh chunk size $n^d/p > \hat{M}$ (as well as $n^{1/d}/p \gg bm$),

$$\hat{W}_{\text{PA0}} = O(s \cdot n^d/p)$$

data is moved between main memory and cache throughout execution. This memory bandwidth cost corresponds to $O(m^d)$ cache reuse per local mesh point and can be much greater than the interprocessor communication cost when m is small relative to $n/p^{1/d}$. However, when PA1 is executed with $b > 1$, an extra factor of b in cache reuse may be achieved by computing up to bm^d updates to each mesh point loaded into cache, yielding a memory bandwidth cost of

$$\hat{W}_{\text{PA1}} = O\left(\frac{s \cdot n^d}{b \cdot p}\right)$$

for $b \leq \hat{M}^{1/d}/m$. This may be accomplished via a second level of blocking, similar to PA1, except with blocks sized to fit in cache, to minimize the number of transfers between cache and main memory. This two-level approach has been studied in Mohiyuddin et al. [2009] in a shared-memory context (see Tang et al. [2011] for a related blocking approach). Analysis in Demmel et al. [2008] for the cases $d \in [1, 3]$ show that cache blocking can reduce memory bandwidth by $O(\hat{M}^{1/d})$, attaining the lower bounds in Hong and Kung [1981], when n is sufficiently large.

Noting that the memory bandwidth cost of this computation can exceed the interprocessor bandwidth cost, we see that PA1 with larger b has lower synchronization cost $S_{\text{PA1}} = O(s/b)$ but higher interprocessor bandwidth cost $W_{\text{PA1}} = O(msn^{d-1}/p^{(d-1)/d})$ (when $d \geq 2$), and also features a lower memory bandwidth cost $\hat{W}_{\text{PA1}} = O(sn^d/(pb))$. It is then natural to ask whether there is a trade-off (in a lower bound sense) between interprocessor and memory bandwidth costs in the computation, or if a different schedule exists, which achieves a low memory bandwidth cost without incurring the extra interprocessor bandwidth cost from which PA1 suffers (when $d \geq 2$). If such a schedule with low or optimal interprocessor communication cost and memory bandwidth cost existed, our lower bounds necessitate that it would have $\Omega(s)$ synchronizations. Indeed, such a schedule exists, and we introduce it in the following section.

9.1. Parallel Bandwidth-Efficient Stencil Computations

Schedule 1, for dependency graph $G_{\text{St}} = (V_{\text{St}}, E_{\text{St}})$, computes the vertices V_{St} needed for the computation of the output $\mathbf{x}^{(s)}$ from the input $\mathbf{x}^{(0)}$ while satisfying the dependencies E_{St}. The schedule assumes that the entries of $\mathbf{A}$ are implicit and thus do not need

Schedule 1 Outer-blocked-Stencil-schedule($V_{\text{St}}, \mathbf{x}^{(0)}, n, m, d, s, \tilde{b}, r, p$)

Require: Restrict V_{St} to the set of computations needed to compute $\mathbf{x}^{(s)}$ from $\mathbf{x}^{(0)}$; assume that $p^{1/d}$ and $s/2\tilde{b}$ are integers and $m \leq \frac{1}{2}r \cdot p^{1/d}$; assume that elements of the input mesh $\mathbf{x}^{(0)}$ may be referenced as $x_{i_1,\ldots,i_d}$.

Let $b = r \cdot p^{1/d}$ and $z = \lceil \frac{n+m\cdot(\tilde{b}-1)}{b} \rceil$.
Augment V_{St} with dummy computations $v_{i_1,\ldots,i_d,l}$ for all $(i_1, \ldots, i_d) \notin [1, n]^d$, the result of which is zero.
Arrange processors into order d grid as $p_{q_1,\ldots,q_d}$ for each $q_j \in [1, p^{1/d}]$.
Let $B^{u_1,\ldots,u_d,0} = \{x_{i_1,\ldots,i_d} \in \mathbf{x}^{(0)} : i_j \in [1, b \cdot u_j]\}$ for each $(u_1, \ldots, u_d) \in [1, z]^d$.
Let subblock $B^{u_1,\ldots,u_d,0}_{q_1,\ldots,q_d} = \{x_{i_1,\ldots,i_d} \in B^{u_1,\ldots,u_d,0} : i_j \in [(q_j - 1) \cdot r + 1, q_j \cdot r]\}$ start in the main memory of processor $p_{q_1,\ldots,q_d}$.
for $w = 1$ to $s/2\tilde{b}$ **do**
  for $(u_1, \ldots, u_d) = (1, \ldots, 1)$ to $(z, \ldots, z)$ **do** in lexicographical ordering
    for $l = 1$ to $\tilde{b}$ **do**
      Let $B^{u_1,\ldots,u_d,l} = \{v_{i_1,\ldots,i_d,(w-1)2\tilde{b}+l} \in V_{\text{St}} :$
        $i_j \in [b \cdot (u_j - 1) - (l - 1) \cdot m + 1, b \cdot u_j - (l - 1) \cdot m]\}$.

      Define subblocks $B^{u_1,\ldots,u_d,l}_{q_1,\ldots,q_d} = \{v_{i_1,\ldots,i_d,l} \in B^{u_1,\ldots,u_d,l} :$
        $i_j \in [(q_j - 1) \cdot r + 1, q_j \cdot r]\}$ for each $(q_j \in [1, p^{1/d}])$.
      Compute $B^{u_1,\ldots,u_d,l}_{q_1,\ldots,q_d}$ with processor $p_{q_1,\ldots q_d}$,
        fetching dependencies from $B^{u_1,\ldots,u_d,l-1}_{s_1,\ldots,s_d}$ for each $(s_j \in \{q_j - 1, q_j\})$.
    % Store subblock from the last loop iteration for use in the next loop
    Store subblock $B^{u_1,\ldots,u_d,\tilde{b}}_{q_1,\ldots,q_d}$ in main memory as $\hat{B}^{\hat{u}_1,\ldots,\hat{u}_d,0}_{q_1,\ldots,q_d}$ with $u_i = z - \hat{u}_i + 1$.
  for $(\hat{u}_1, \ldots, \hat{u}_d) = (1, \ldots, 1)$ to $(z, \ldots, z)$ **do** in lexicographical ordering
    for $l = 1$ to $\tilde{b}$ **do**
      Let $\hat{B}^{\hat{u}_1,\ldots,\hat{u}_d,l} = \{v_{i_1,\ldots,i_d,(w-1)2\tilde{b}+\tilde{b}+l} \in V_{\text{St}} :$
        $i_j \in [b \cdot (z - \hat{u}_j) + (l - \tilde{b} + 1) \cdot m + 1, b \cdot (z - \hat{u}_j + 1) + (l - \tilde{b} + 1) \cdot m]\}$.

      Define subblocks $\hat{B}^{\hat{u}_1,\ldots,\hat{u}_d,l}_{q_1,\ldots,q_d} = \{v_{i_1,\ldots,i_d,l} \in \hat{B}^{\hat{u}_1,\ldots,\hat{u}_d,l} :$
        $i_j \in [(q_j - 1) \cdot r + 1, q_j \cdot r]\}$ for each $(q_j \in [1, p^{1/d}])$.
      Compute $\hat{B}^{\hat{u}_1,\ldots,\hat{u}_d,l}_{q_1,\ldots,q_d}$ with processor $p_{q_1,\ldots q_d}$
        fetching dependencies from $\hat{B}^{\hat{u}_1,\ldots,\hat{u}_d,l-1}_{s_1,\ldots,s_d}$ for each $(s_j \in \{q_j, q_j + 1\})$.
    % Store subblock from the last loop iteration for use in the next loop or the output $\mathbf{x}^{(s)}$
    Store subblock $\hat{B}^{\hat{u}_1,\ldots,\hat{u}_d,\tilde{b}}_{q_1,\ldots,q_d}$ in main memory as $B^{u_1,\ldots,u_d,0}_{q_1,\ldots,q_d}$ with $u_i = z - \hat{u}_i + 1$.

Ensure: $\mathbf{x}^{(s)}$ has been computed and saved into main memory via computation of all of V_{St}.

to be stored in main memory or cache. Schedule 1 also has three execution-related parameters:

—$\tilde{b}$, which is analogous to the parameter b for PA1 in the previous section, as it is the blocking parameter on the $(d+1)$th order (time/nonmesh dimension) of the iteration space that ranges from 1 to s;
—r, which defines the side length of the mesh chunk on which each processor works at a given time; and
—p, which is the number of processors as before.

Like PA1 with $b > 1$, Schedule 1 defines order $(d + 1)$ blocks that enable improved cache reuse when a second level of blocking is imposed. The key difference is that in Schedule 1, the blocking is done on the global problem rather than the local chunk of the work. Each of these global blocks is executed in parallel, with synchronizations done between every consecutive pair of dth order slices of the block. Improved memory

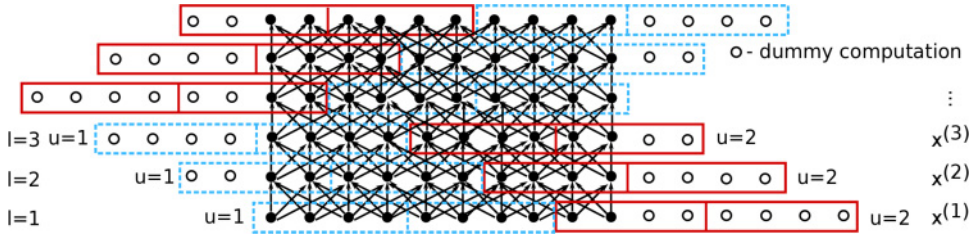

Fig. 6. Depiction of Schedule 1 with $\tilde{b} = 4$, $r = 4$, and $p = 2$ to perform a stencil computation with $n = 10$, $m = 2$, $d = 1$, and $s = 8$. All of the blue (dashed outline) blocks for each of two levels (the first is labeled for $l = 1$ to $l = 4$) are computed prior to all of the red (solid outline) blocks, with the two subblocks composing each blue and red block being computed in parallel by the two processors.

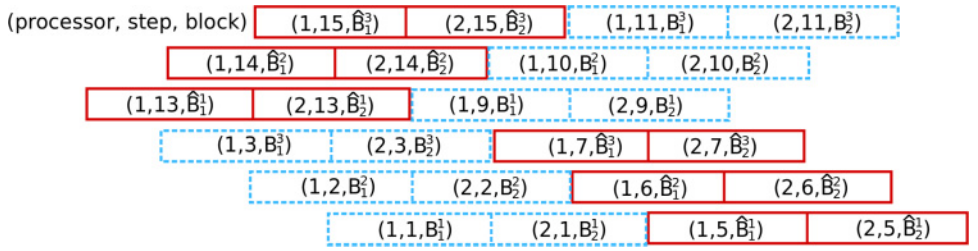

Fig. 7. Work assignment and order of Schedule 1 labeled according to (executing processor number, execution step number, block name corresponding to Schedule 1) for execution in Figure 6.

bandwidth cost is achieved by Schedule 1 by ensuring that each processor executes subslices that, along with their dependencies, are small enough to be stored in cache between synchronizations.

Figure 6 demonstrates the parallelization coloring of Schedule 1 for $s = 8$ stencil applications for a 1D mesh problem with $n = 10$ mesh points and stencil width $m = 2$, using $\tilde{b} = 4$ and $r = 4$. The figure also depicts the dummy computations introduced to simplify Schedule 1. The figure also shows why Schedule 1 contains two alternating loops from $l = 1$ to $\tilde{b}$, as this allows the blocks computed by each processor at the last iteration of the first loop over l to satisfy the dependencies of the block needed by the same processor for the first iteration of the second loop over l. Further, the blocks computed at the last iteration of the second loop over l are subsequently used by the same processors in the first iteration of the first loop over l (in the following iteration of the outermost loop over w). Figure 7 demonstrates at what step the two processors compute each block.

Schedule 1 satisfies the dependencies E_{St}, as the pieces of the work executed in parallel within the same block in the first loop nest (for-loop on line 7 of Schedule 1) $B^{u_1,\ldots,u_d,l}$ correspond to chunks of a single matrix-vector multiplication with no inter-dependencies for each l. Further, each $B^{u_1,\ldots,u_d,l}$ depends only on $B^{s_1,\ldots,s_d,l-1}$ for each $(s_j \in \{u_j - 1, u_j\})$ due to the fact that the blocks are shifted back by $(l-1)\cdot m$ in all dimensions at inner loop iteration l (and each vertex depends only on vertices at most m away from its index in the block at the previous level) and the fact that every such block $B^{s_1,\ldots,s_d,l-1}$ was computed previously due to the lexicographical order on the iteration index tuples $(u_1, \ldots, u_d)$. The second loop nest (for-loop on line 18 of Schedule 1) reverses the order of the first loop nest, with blocks shifted forward by m with each increment of l, and starting at the opposite end of the mesh, so that each $B^{u_1,\ldots,u_d,l}$ depends only on the vertices from $B^{s_1,\ldots,s_d,l-1}$ for all $(s_j \in \{u_j, u_j + 1\})$. The point of this reversal is to line up the data that each processor owns when computing the block for

$l = \tilde{b}$ with the data that each processor needs when computing the block for $l = 1$ for the next loop over l.

In particular, we argue that every block $B^{u_1,\ldots,u_d,0}_{q_1,\ldots,q_d}$ and $\hat{B}^{\hat{u}_1,\ldots,\hat{u}_d,0}_{q_1,\ldots,q_d}$ accessed by processor $p_{q_1,\ldots,q_d}$ resides in the main memory of the processor. For the first loop nest (line 7) with $w = 1$, this block is the input block defined on line 5 of Schedule 1. For each first inner loop iteration ($l = 1$) of the second loop nest (line 18), the global block defined in the loop iteration,

$$\hat{B}^{\hat{u}_1,\ldots,\hat{u}_d,0} = \{v_{i_1,\ldots,i_d,(w-1)2\tilde{b}+\tilde{b}} \in V_{\mathrm{St}} : i_j \in [b\cdot(z-\hat{u}_j) - (\tilde{b}-1)\cdot m + 1, \\ b\cdot(z-\hat{u}_j+1) - (\tilde{b}-1)\cdot m]\},$$

is the same block as the one computed in the first loop nest at the last inner loop iteration ($l = \tilde{b}$) with $(u_1,\ldots,u_d) = (z-\hat{u}_1+1,\ldots,z-\hat{u}_d+1)$, namely

$$B^{u_1,\ldots,u_d,\tilde{b}} = \{v_{i_1,\ldots,i_d,(w-1)2\tilde{b}+\tilde{b}} \in V_{\mathrm{St}} : i_j \in [b\cdot(u_j-1) - (\tilde{b}-1)\cdot m + 1 \\ b\cdot u_j - (\tilde{b}-1)\cdot m]\},$$

since $u_i = z - \hat{u}_i + 1$ for all $i \in [1,d]$. In both cases, the blocks are subdivided among processors in the exact same fashion, and thus we have demonstrated that all blocks $B^{u_1,\ldots,u_d,l}_{q_1,\ldots,q_d}$ and $\hat{B}^{\hat{u}_1,\ldots,\hat{u}_d,l}_{q_1,\ldots,q_d}$ for $l \in [0,\tilde{b}-1]$ are in main memory of processor $q_1,\ldots,q_d$ each time that they are requested as a dependency.

THEOREM 9.1. *We consider the computation of s-steps of a $(2m+1)^d$-point stencil on a n-by-$\cdots$-by-n mesh, with p processors and cache size $\hat{M}$, and assume that $n \gg \hat{M}^{1/d} > 3m$. Schedule 1 is parameterized by r and $\tilde{b}$, which for any choice of $\lfloor \hat{M}^{1/d} \rfloor - 2m \le r \le n/p^{1/d}$ and $1 \le \tilde{b}/m \ll n$ attain the following costs:*

$$F_{\mathrm{ObKs}} = O(m^d \cdot s \cdot n^d/p), \quad W_{\mathrm{ObKs}} = O\left(\frac{m\cdot s\cdot n^d}{r\cdot p}\right),$$

$$S_{\mathrm{ObKs}} = O(s\cdot n^d/(p\cdot r^d)), \quad \hat{W}_{\mathrm{ObKs}} = O\left(\frac{s\cdot n^d}{\tilde{b}\cdot p} + \frac{m\cdot s\cdot n^d}{r\cdot p}\right).$$

Schedule 1 with $r = \lfloor \hat{M}^{1/d} \rfloor - 2m$ and $\tilde{b} = \lfloor \hat{M}^{1/d}/m \rfloor$ execute this computations with the following costs:

$$F_{\mathrm{ObKs}} = O\left(m^d \cdot s \cdot n^d/p\right), \quad W_{\mathrm{ObKs}} = O\left(\frac{m\cdot s\cdot n^d}{\hat{M}^{1/d}\cdot p}\right),$$

$$S_{\mathrm{ObKs}} = O(s\cdot n^d/(p\cdot \hat{M})), \quad \hat{W}_{\mathrm{ObKs}} = O\left(\frac{m\cdot s\cdot n^d}{\hat{M}^{1/d}\cdot p}\right).$$

PROOF. The amount of extra work done in Schedule 1 is proportional to $sn^{d-1}m\tilde{b}$ and thus is negligible relative to the overall work, $O(sn^d)$, when $n \gg m\tilde{b}$, which has been assumed. Since the computation is entirely load balanced, the floating point cost is

$$F_{\mathrm{ObKs}} = O\left(m^d \cdot s \cdot n^d/p\right).$$

Now, $B^{u_1,\ldots,u_d,l}_{q_1,\ldots,q_d}$ depends on all r^d entries of $B^{\hat{u}_1,\ldots,\hat{u}_d,l-1}_{q_1,\ldots,q_d}$ for all (q_j), and similarly for $\hat{B}$, since each block is shifted by m (the interaction range). Further, there are a total of at most $(2m+r)^d$ entries on which each block depends. Therefore, at most $(2m+r)^d - r^d$ entries need to be communicated from remote processors at each step, and these processors are the nearest neighbors of the recipient, namely $p_{s_1,\ldots s_d}$ for each $(s_j \in \{q_j-1, q_j\})$, for the first loop, over $(u_1,\ldots,u_d)$, and for each $(s_j \in \{q_j, q_j+1\})$,

for the second loop, over $(\hat{u}_1, \ldots, \hat{u}_d)$. Thus, the interprocessor bandwidth cost for each inner loop given a particular iteration $(u_1, \ldots, u_d)$ or $(\hat{u}_1, \ldots, \hat{u}_d)$ of Schedule 1 is (since $r \geq m$)

$$W^u_{\text{ObKs}} \leq \sum_{l=1}^{\tilde{b}} \left[(2m+r)^d - r^d\right] = O(\tilde{b} \cdot m \cdot r^{d-1}).$$

Over $s/(2\tilde{b})$ (assumed an integer) iterations of the outer loop and

$$z^d \approx \frac{(n + m \cdot (\tilde{b}-1))^d}{r^d \cdot p} = O\left(\frac{n^d}{r^d \cdot p}\right)$$

iterations of loops over $(u_1, \ldots, u_d)$ and $(\hat{u}_1, \ldots, \hat{u}_d)$, since we have that $m \cdot \tilde{b} \ll n$, the total interprocessor bandwidth communication cost is then

$$W_{\text{ObKs}} = O\left(\frac{s}{2\tilde{b}} \cdot \frac{n^d}{r^d \cdot p} \cdot W^u_{\text{ObKs}}\right) = O\left(\frac{s}{2\tilde{b}} \cdot \frac{n^d}{r^d \cdot p} \cdot \tilde{b} \cdot m \cdot r^{d-1}\right) = O\left(\frac{s \cdot n^d \cdot m}{r \cdot p}\right).$$

If we pick $r = n/p^{1/d}$, this cost matches the cost of PA0 (equivalently PA1 with $b = 1$); however, we will pick r to allow the local mesh chunks to fit into cache, $r = \lfloor \hat{M}^{1/d} \rfloor - 2m$, which will result in a greater interprocessor bandwidth cost. On the other hand, the quantity $\tilde{b}$ does not appear in this interprocessor bandwidth cost, unlike that of PA1.

To achieve a good memory bandwidth cost, the parameter r should be picked so that the slices $B^{u_1,\ldots,u_d,l}_{q_1,\ldots,q_d}$ (of size r^d) computed by each process, as well as their dependencies (of size at most $(2m+r)^d$), fit into the cache of size $\hat{M}$ (i.e., $\hat{M} \geq (2m+r)^d$). Assuming that the additional space in the cache necessary for the block produced at each iteration is negligible with respect to the size of its dependencies (it can overwrite the old block and its dependencies, as they are not needed at the next iteration), the restriction $r \geq \lfloor \hat{M}^{1/d} \rfloor - 2m$ ensures that the computation of the innermost loop in Schedule 1 can be done entirely in cache. If this is done, the memory bandwidth cost of Schedule 1 is not much greater than the interprocessor bandwidth cost for large enough $\tilde{b}$, since the dependencies from the previous block on a given processor may be kept in cache prior to execution of the next block. However, for each loop iteration over $(u_1, \ldots, u_d)$ and $(\hat{u}_1, \ldots, \hat{u}_d)$, the blocks $B^{u_1,\ldots,u_d,0}_{q_1,\ldots,q_d}$ and $\hat{B}^{\hat{u}_1,\ldots,\hat{u}_d,0}_{q_1,\ldots,q_d}$ must be read from main memory to cache; the memory bandwidth cost for any one such iteration is

$$\hat{W}^u_{\text{ObKs}} \leq r^d + W^u_{\text{ObKs}}.$$

The total memory bandwidth cost of the scheme over $z^d \approx \frac{(n+m\cdot(\tilde{b}-1))^d}{r^d \cdot p}$ iterations is

$$\begin{aligned}\hat{W}_{\text{ObKs}} &= O\left(\frac{s}{2\tilde{b}} \cdot \frac{n^d}{r^d \cdot p} \cdot \hat{W}^u_{\text{ObKs}}\right) + W_{\text{ObKs}} = O\left(\frac{s \cdot n^d}{\tilde{b} \cdot p}\right) + W_{\text{ObKs}} \\ &= O\left(\frac{s \cdot n^d}{\tilde{b} \cdot p} + \frac{s \cdot n^d \cdot m}{r \cdot p}\right) = O\left(\frac{s \cdot n^d}{\tilde{b} \cdot p} + \frac{s \cdot n^d \cdot m}{\hat{M}^{1/d} \cdot p}\right).\end{aligned}$$

Therefore, if we pick $\tilde{b} = \lfloor \hat{M}^{1/d}/m \rfloor$, we obtain a parallel schedule whose memory bandwidth cost will be asymptotically the same as the interprocessor bandwidth cost. Since, as stated earlier, the interprocessor bandwidth cost is optimal when $\hat{M}^{1/d} = \Omega(n/p^{1/d})$, the memory bandwidth cost will be optimal as well. When $\hat{M} \leq (n+2m)^d$, in which case the local mesh chunk and its dependencies do not fit into cache, the memory bandwidth cost of the new scheme, $\hat{W}_{\text{ObKs}}$, will be a factor of $\Theta(1/\tilde{b})$ less than the memory bandwidth cost of doing each matrix-vector multiplication in sequence,

using $\hat{W}_{\text{PA0}}$. Further, the new scheme avoids the increased interprocessor bandwidth cost of PA1 with $b > 1$ (when $d \geq 2$). We can potentially select $\tilde{b}$ up to $\hat{M}^{1/d}/m$ and achieve a factor of up to $\Theta(\frac{m}{\hat{M}^{1/d}})$ reduction in memory bandwidth cost. We see that the schedule corresponding to Schedule 1 with $\tilde{b} = \lfloor \hat{M}^{1/d}/m \rfloor$ has a sweet spot in terms of performance when $\hat{M} < n^d/p$, but not $\hat{M} \ll n^d/p$, so the local mesh chunk does not fit into cache (thus the memory bandwidth efficiency of PA0 deteriorates). However, simultaneously, the cache size is big enough so that the blocking done by Schedule 1 does not significantly increase the interprocessor communication cost (and also the memory bandwidth cost). The total communication volume of this schedule is proportional to the surface area of all mesh blocks multiplied by $s \cdot m$ and therefore grows when the blocks, whose size cannot exceed the cache size, are small.

The synchronization cost of Schedule 1 is

$$S_{\text{ObKs}} = O(s \cdot n^d/(p \cdot r^d)),$$

whereas the synchronization cost of the schedule with $r = \lfloor \hat{M}^{1/d} \rfloor - 2m$ and $\tilde{b} = \lfloor \hat{M}^{1/d}/m \rfloor$ is

$$S_{\text{ObKs}} = O(s \cdot n^d/(p \cdot \hat{M})),$$

as it requires a synchronization for every loop iteration from $l = 1$ to $\tilde{b}$ (over $s/(2\tilde{b})$ invocations), and because the number of outer loop iterations is $O(n^d/(p \cdot \hat{M}))$. This synchronization cost is larger than the s/b synchronizations needed by PA1, especially when the cache size ($\hat{M}$) is small with respect to n^d/p. □

10. CONCLUSION

Our lower bounds show that when computation, communication, and synchronization costs are simultaneously considered, many numerical problems with lattice-like dependency structures require execution costs that are independent of the number of processors but dependent on the problem size. Our lower bound analysis introduces the idea of dependency intervals, which facilitate the derivation of trade-offs between computation and synchronization (by considering dependency interval expansion) as well as between communication and synchronization (by considering the expansion of the minimum vertex separator of the dependency interval). Further, our analysis shows that hypergraphs may be leveraged to obtain communication lower bounds on families of dependency graphs (algorithms) via the notion of parent hypergraphs. We introduce a volumetric inequality and a lower bound on the minimum hyperedge cut for vertex sets enumerated as strictly increasing tuples, which made the derivation of minimum vertex separators possible for our applications. As future work, we hope to derive a similar lower bound on the minimum hyperedge cut of arbitrary vertex sets and employ it to derive communication lower bounds for more applications.

From an architectural perspective, our results give lower bounds on execution time as a function of synchronization latency (α), interprocessor communication throughput (β), memory bandwidth (ν), and arithmetic throughput (γ). The trade-offs that we derive describe the strong scaling limit of Cholesky factorization and stencil computations in terms of these four quantities. In other words, we obtain lower bounds on the parallel execution time needed to solve a system of linear equations via certain numerical algorithms. Our results support observations that algorithms for Cholesky factorization and stencil computations demonstrate trade-offs between communication and synchronization costs. The impact of our lower bound results on implementations

is twofold. First, algorithm designers should not expend energy searching for schedules that simultaneously minimize communication and synchronization for the algorithms we analyzed—we have shown that the trade-off between these two costs is a fundamental aspect of their dependency graphs—rather, to break through this barrier, it is necessary to discover new algorithms for these problems. Second, from the perspective of an algorithm designer, these trade-offs should be viewed as algorithmic tuning parameters, which should be adjusted differently on different architectures to navigate the cost trade-offs and minimize runtime. Last, in the case of stencil computations, we have shown that on some problems and machine configurations, it is possible to minimize memory bandwidth costs at the same time as interprocessor bandwidth costs: previous approaches had only been able to minimize one or the other.

Our research provides a number of potential directions for future work, including the extension of our results for Cholesky factorization to other matrix factorizations and the application of our lower bound analysis to graph algorithms on structured graphs such as expanders and trees.

ACKNOWLEDGMENTS

We thank Harsha Simhadri, Satish Rao, Grey Ballard, and Benjamin Lipshitz for useful discussions on topics related to this work. We also thank the anonymous reviewers for comments that helped to improve the article.

REFERENCES

Alok Aggarwal, Ashok K. Chandra, and Marc Snir. 1990. Communication complexity of PRAMs. *Theoretical Computer Science* 71, 1, 3–28.

Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. 1995. LogGP: Incorporating long messages into the LogP model—one step closer towards a realistic model for parallel computation. In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'95)*. ACM, New York, NY, 95–105. DOI:http://dx.doi.org/10.1145/215399.215427

Grey Ballard, James Demmel, Olga Holtz, Benjamin Lipshitz, and Oded Schwartz. 2012. Brief announcement: Strong scaling of matrix multiplication algorithms and memory-independent communication lower bounds. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'12)*. ACM, New York, NY, 77–79.

Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. 2011. Minimizing communication in linear algebra. *SIAM Journal on Matrix Analysis and Applications* 32, 3, 866–901.

E. Bampis, C. Delorme, and J.-C. König. 1996. Optimal schedules for d-D grid graphs with communication delays. In *STACS 96*. Lecture Notes in Computer Science, Vol. 1046. Springer, 655–666. DOI:http://dx.doi.org/10.1007/3-540-60922-9_53

Richard Bellman. 1956. *On a Routing Problem*. Technical Report. DTIC Document.

Michael A. Bender, Gerth Stølting Brodal, Rolf Fagerberg, Riko Jacob, and Elias Vicari. 2010. Optimal sparse matrix dense vector multiplication in the I/O-model. *Theory of Computing Systems* 47, 4, 934–962. DOI:http://dx.doi.org/10.1007/s00224-010-9285-4

G. Bilardi and F. P. Preparata. 1999. Processor–time tradeoffs under bounded-speed message propagation: Part II, lower bounds. *Theory of Computing Systems* 32, 5, 531–559.

Gianfranco Bilardi, Michele Scquizzato, and Francesco Silvestri. 2012. A lower bound technique for communication on BSP with application to the FFT. In *Euro-Par 2012 Parallel Processing*. Lecture Notes in Computer Science, Vol. 7484. Springer, 676–687. DOI:http://dx.doi.org/10.1007/978-3-642-32820-6_67

L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, et al. 1997. *ScaLAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA.

Erin Carson, Nick Knight, and James Demmel. 2013. Avoiding communication in nonsymmetric Lanczos-based Krylov subspace methods. *SIAM Journal on Scientific Computing* 35, 5, S42–S61. DOI:http://dx.doi.org/10.1137/120881191

Michael Christ, James Demmel, Nicholas Knight, Thomas Scanlon, and Katherine Yelick. 2013. Communication lower bounds and optimal algorithms for programs that reference arrays–Part 1. arXiv:1308.0068.

David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. 1993. LogP: Towards a realistic model of parallel computation. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'93)*. ACM, New York, NY, 1–12.

James Demmel, Mark Hoemmen, Marghoob Mohiyuddin, and Katherine Yelick. 2008. Avoiding communication in sparse matrix computations. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS'08)*. IEEE, Los Alamitos, CA, 1–12.

Robert W. Floyd. 1962. Algorithm 97: Shortest path. *Communications of the ACM* 5, 6, 345. DOI:http://dx.doi.org/10.1145/367766.368168

Lester Randolph Ford. 1956. *Network Flow Theory*. RAND Corporation, Santa Monica, CA.

Laura Grigori, James W. Demmel, and Hua Xiang. 2011. CALU: A communication optimal LU factorization algorithm. *SIAM Journal on Matrix Analysis and Applications* 32, 4, 1317–1350.

Michael T. Heath and Charles H. Romine. 1988. Parallel solution of triangular systems on distributed-memory multiprocessors. *SIAM Journal on Scientific and Statistical Computing* 9, 3, 558–588.

J.-W. Hong and H. T. Kung. 1981. I/O complexity: The red-blue pebble game. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing (STOC'81)*. ACM, New York, NY, 326–333.

Dror Irony and Sivan Toledo. 2002. Trading replication for communication in parallel distributed-memory dense solvers. *Parallel Processing Letters* 71, 3–28.

Dror Irony, Sivan Toledo, and Alexander Tiskin. 2004. Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing* 64, 9, 1017–1026.

Charles E. Leiserson, Satish Rao, and Sivan Toledo. 1997. Efficient out-of-core algorithms for linear relaxation using blocking covers. *Journal of Computer and System Sciences* 54, 2, 332–344. DOI:http://dx.doi.org/10.1006/jcss.1997.1473

Lynn H. Loomis and Hassler Whitney. 1949. An inequality related to the isoperimetric inequality. *Bulletin of the American Mathematical Society* 55, 10, 961–962.

William F. McColl and Alexander Tiskin. 1999. Memory-efficient matrix multiplication in the BSP model. *Algorithmica* 24, 3, 287–297.

Marghoob Mohiyuddin, Mark Hoemmen, James Demmel, and Katherine Yelick. 2009. Minimizing communication in sparse matrix solvers. In *Proceedings of the Conference on High Performance Computing Networking, Storage, and Analysis*. ACM, New York, NY, 36.

Christos H. Papadimitriou and Jeffrey D. Ullman. 1987. A communication-time tradeoff. *SIAM Journal on Computing* 16, 4, 639–646. DOI:http://dx.doi.org/10.1137/0216044

Jack Poulson, Bryan Marker, Robert A. van de Geijn, Jeff R. Hammond, and Nichols A. Romero. 2013. Elemental: A new framework for distributed memory dense matrix computations. *ACM Transactions on Mathematical Software* 39, 2, 13.

Michele Scquizzato and Francesco Silvestri. 2014. Communication lower bounds for distributed-memory computations. In *Proceedings of the 31st International Symposium on Theoretical Aspects of Computer Science (STACS'14)*. 627–638. DOI:http://dx.doi.org/10.4230/LIPIcs.STACS.2014.627

Edgar Solomonik, Erin Carson, Nicholas Knight, and James Demmel. 2014. *Tradeoffs Between Synchronization, Communication, and Computation in Parallel Linear Algebra Computations*. ACM, New York, NY, 307–318. DOI:http://dx.doi.org/10.1145/2612669.2612671

Edgar Solomonik and James Demmel. 2011. Communication-optimal 2.5D matrix multiplication and LU factorization algorithms. In *Proceedings of the 17th International Conference on Parallel Processing (Euro-Par'11)—Volume Part II*. 90–109.

Danny C. Sorensen. 1985. Analysis of pairwise pivoting in Gaussian elimination. *IEEE Transactions on Computers* C-34, 3, 274–278.

Volker Strassen. 1969. Gaussian elimination is not optimal. *Numerische Mathematik* 13, 4, 354–356. DOI:http://dx.doi.org/10.1007/BF02165411

Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. 2011. The Pochoir stencil compiler. In *Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, New York, NY, 117–128.

Alexander Tiskin. 1998. *The Design and Analysis of Bulk-Synchronous Parallel Algorithms*. Ph.D. Dissertation. University of Oxford.

Alexander Tiskin. 2001. All-pairs shortest paths computation in the BSP model. In *Automata, Languages and Programming*. Lecture Notes in Computer Science, Vol. 2076. Springer, 178–189.

Alexander Tiskin. 2002. Bulk-synchronous parallel Gaussian elimination. *Journal of Mathematical Sciences* 108, 6, 977–991. DOI:http://dx.doi.org/10.1023/A:1013588221172

Alexander Tiskin. 2007. Communication-efficient parallel generic pairwise elimination. *Future Generation Computer Systems* 23, 2, 179–188.

Leslie G. Valiant. 1990. A bridging model for parallel computation. *Communications of the ACM* 33, 8, 103–111.

Stephen Warshall. 1962. A theorem on boolean matrices. *Journal of the ACM* 9, 1, 11–12. DOI:http://dx.doi.org/10.1145/321105.321107

Received August 2014; revised December 2014, October 2015, January 2016; accepted January 2016