PDC Algorithmics that "Stays with You":
Using Dependency DAGs to Teach Parallel Algorithms

Arnold L. Rosenberg
Northeastern University, Boston, MA 02115, USA
rsnbrg@ccs.neu.edu

***Abstract*—Parallel and distributed computing (*PDC*) pervades modern living. It is essential that we imbue all aspiring computer professionals with PDC-related skills early in their educations. How, however, can we incorporate such material throughout the curriculum *incrementally*, so that advanced material builds upon earlier material? We present a proposal for teaching the *algorithmic* aspects of PDC in the core undergraduate curriculum in a way that can carry through to more advanced courses. The essence of the proposal is to expose students to the intertask dependency structure of each algorithm as part of its specification. This structure explains why the algorithm parallelizes—thereby exposing unexpected kinships; examples: *mergesort* and *numerical integration via Simpson's rule; carry-lookahead addition* and the *discrete Laplace transform; odd-even mergesort* and the *discrete Fourier transform*. The elementary algorithm in each pairing prepares the student for the advanced one. Teaching PDC Algorithmics in this way also helps teach PDC concepts in other areas.**

I. Introduction

It is an incontrovertible fact that *parallel and distributed computing* (*PDC*) has insinuated itself into almost all corners of modern living—telephones, televisions, home appliances, etc. This fact mandates teaching PDC-related material to all aspiring computer professionals, and doing so as early in the computing-related curriculum as possible. An important step toward incorporating PDC into computing-related curricula, from the low-level core undergraduate courses to more advanced, specialized ones, has been the development of the curricular guidelines in [20], [21]. These guidelines enumerate a long list of PDC-related topics and suggest the level at which each should be covered in various courses at various stages of the curriculum. One challenge that largely remains to be addressed is how to incorporate PDC-related material throughout the curriculum in an *incremental* manner, i.e., in a way that has more advanced coverage build upon earlier coverage rather than replicate it. This paper presents—and illustrates—a modest proposal for teaching the *algorithmic* aspects of PDC in the early, core, undergraduate curriculum in a way that can carry through more advanced courses. The proposal aims to teach PDC Algorithmics in a way that will benefit the teaching not only of Algorithms courses, but also of "client" subjects such as Architecture, Compilers, and Operating Systems. (The latter are "client" subjects because many of the concepts they build on have important algorithmic components.) In common with the philosophy underlying the guidelines in [20], [21], our aim is to teach PDC-related aspects of the design and analysis of algorithms (which we dub *algorithmics* for short) in a manner that

1) is as *unobtrusive* as possible, in the sense that it does not displace extant algorithmic material from the curriculum (even though it does advocate teaching some of this material somehat differently than one finds in standard texts);
2) plants seeds that will sprout in subsequent courses so that the teaching of PDC-related algorithmics "flows" throughout the curriculum.

A. The Proposal in a Nutshell

We elaborate on our second point: The essence of the proposal is to expose students to the dependency structure of an algorithm/algorithmic paradigm as one teaches specific algorithms. In detail, this would involve present algorithms via their *intertask dependency graphs*, as well as via detailed operational specifications. The justifying rationale is that this structure often explains why/how the algorithm admits parallelization—thereby exposing kinships among apparently unrelated algorithms. Examples of such kinships abound and include: (1) *mergesort* [9] and *numerical integration via the trapezoid method* [6]; (2) *carry-lookahead addition* [16] and the *discrete Laplace transform* [4]; (3) *odd-even mergesort* [17] and the *discrete Fourier transform* [9]. Note that each of these pairings involves one elementary algorithm and one more advanced one—so, presenting the structural basis for the kinships when covering an elementary algorithm prepares the student for the algorithm's more advanced kin. While this proposal adds a conceptually nontrivial prerequisite—viz, dependency graphs—to introductory algorithmic material, this really amounts only to a *repositioning* of these concepts within the curriculum. The recognition (and exploitation) of intertask dependencies impacts many aspects of dealing with a process, including (*inter alia*) the potential for parallelizing the process and for rearranging its specification when preparing to execute it. Thus, this concept must appear early in a computer-related curriculum. Our proposal may only move its introduction a bit earlier and into a different introductory course.

Expanding on the main benefit of the proposed approach: While a specific algorithm that appears early in the curriculum may not recur in later courses, its underlying structure may be shared by many algorithms that do occur later. When this underlying structure influences important computational aspects of the algorithm that share it, such as parallelization, having

the student understand the structure will enhance his/her access to the more advanced material—and may even heighten awareness when new algorithmic problems are encountered.

We present and discuss the proposed methodology with the aid of examples. Each example is built on an algorithm/paradigm; each paradigm is illustrated by an elementary algorithm followed by more advanced ones that yield to the same general design and analysis. For each computation/paradigm covered, we provide:

1) an analysis of its intertask dependency structure (expressed as a *directed acyclic graph*; see Section II);
2) a discussion of other computations that share this dependency structure.

Our overriding concern when discussing each covered computation is how the structure of the computation's intertask dependencies exposes *in*-dependencies that enable parallelization. Other issues—even essential ones such as communication load, which may influence one's decision about the computation's suitability for a particular parallel computing environment—are *not* our primary concern here (although instructors may feel compelled to address them, especially in follow-on courses that apply the algorithmics to artifacts such as compilers, operating systems, and digital logic).

B. A Roadmap

After presenting the technical background for our discussion (Section II), we turn to our repertoire of illustrative real computations/paradigms. Section III-A discusses the important *expansion-reduction* computational paradigm, whose constituent expansion-reduction computations notably include computations derived using the *divide-and-conquer paradigm*. Mergesort provides a concrete example of this paradigm, as do several techniques for numerical integration of functions. Section III-C discusses *butterfly-structured* computations, which include several comparator-based sorting algorithms, in addition to the important family of *convolutional computations*, such as polynomial multiplication and the Fast Fourier Transform. Section III-B expands on the theme of expansion-reduction computations, considering more complicated modes of "expansion," such as the important *parallel-prefix* operator (which is also known as the *scan* operator); this section singles out specific computation that include the design of carry-lookahead adders and the Discrete Laplace Transform.

C. Sources

The material in this paper comes from several sources which we recommend to the reader. The philosophy of PDC education that we espouse emerges from the curricular guidelines in [20], [21]. Our general approach to algorithmic material follows the tone of the comprehensive algorithms text [9]. Most of the material related to the expansion-reduction computational paradigm comes from [9]; the specific treatment of numerical integration builds on the foundation in [6]. The material relating to the parallel-prefix/scan operator is inspired by the comprehensive study of the operator in [3] and the examples of its use in [9], [17]; the specific treatment of adder design follows [16], and the formulation of the Discrete Laplace Transform follows [4]. The material relating to butterfly-structured computations largely follows the comprehensive introduction to parallel computation on network-based multiprocessors in [17].

Our interest in computations whose structures can be specified as DAGs follows an extensive literature in pure and applied algorithmics, including [12], [13], [14], [18], [19], [22]. Sources such as [5], [11], [15] emphasize the importance of studying *transformations* of DAGs.

Many of the examples we discuss come from our earlier work on DAG-scheduling [8].

II. Background

Computation-DAGs. A *directed graph* $\mathcal{G}$ is given by a set of *nodes* $N_{\mathcal{G}}$ and a set of *arcs* (or, *directed edges*) $A_{\mathcal{G}}$, each of the form $(u \to v)$, where $u, v \in N_{\mathcal{G}}$. (The arc is *oriented* from u to v.) A *path* in $\mathcal{G}$ is a sequence of arcs that share adjacent endpoints, as in the following path from node u_1 to node u_n: $(u_1 \to u_2),\ (u_2 \to u_3),\ \ldots,\ (u_{n-1} \to u_n)$. A DAG (*directed acyclic graph*) $\mathcal{G}$ is a directed graph that has no cycles—so that no path of the preceding form has $u_1 = u_n$. When a DAG $\mathcal{G}$ is used to model a computation, i.e., is a *computation*-DAG*:*

- each node $v \in N_{\mathcal{G}}$ represents a task in the computation;

  Note: The term "task" in this context is used generically, no matter what the "granularity" of the depicted computation is. Thus, a "task" can represent a single operation when the DAG $\mathcal{G}$ is "fine-grained," or it can represent an entire job when $\mathcal{G}$ is "coarse-grained." To illustrate this point (with an apology for a forward reference), we note that each task of the DAG depicted in Fig. 10 is fine-grained, representing a single real product, whereas each task of the DAG depicted in Fig. 11 is coarse-grained, representing an entire matrix product.
- an arc $(u \to v) \in A_{\mathcal{G}}$ represents the dependence of task v on task u: task v cannot be executed until task u is.

For any arc $(u \to v) \in A_{\mathcal{G}}$, u is a *parent* of v, and v is a *child* of u in $\mathcal{G}$. A parentless node of $\mathcal{G}$ is a *source*, and a childless node is a *sink*.

Fig. 1 presents two sample DAGs that represent two algorithms for summing eight numbers, $a_1, a_2, \ldots, a_8$. One sees easily that the lefthand DAG, which organizes (read: parenthesizes) the summation as

$$(((a_1 + a_2) + (a_3 + a_4)) + ((a_5 + a_6) + (a_7 + a_8)))$$

exposes more opportunities for parallelization than does the righthand DAG, which organizes (read: parenthesizes) the summation as

$$(((((((a_1 + a_2) + a_3) + a_4) + a_5) + a_6) + a_7) + a_8).$$

The utility of modeling computations in terms of their underlying DAGs is amply documented in sources that focus on how to schedule computations in both sequential computing environments [14], [18], [19] and parallel ones

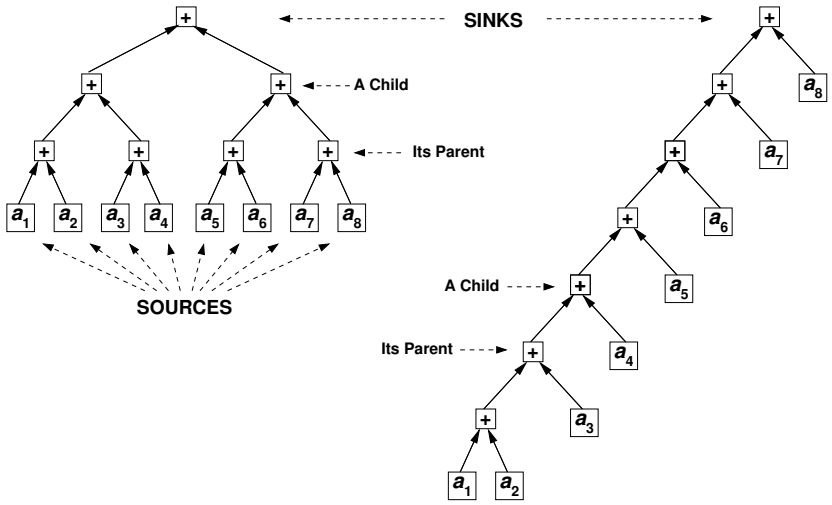

Fig. 1. Two DAGs that represent two algorithms/groupings for summing eight numbers: $a_1 + a_2 + a_3 + a_4 + a_5 + a_7 + a_8$.

[12], [13]. Additionally, various genres of transformations of DAGs can simplify computations, say by eliminating redundant data transfers [15], or can render them more amenable to certain computing paradigms; e.g., the "series-parallelization" algorithms of [11] transform a given computation into a functionally equivalent one that is more amenable to multithreaded scheduling systems such as *Cilk* [5].

III. Studying Algorithms and Paradigms via DAGs

A. Expansion-Reduction Computations

1) The structure of the DAG*s:* The computations we exemplify in this section are built via iterated composition from the two basic building blocks depicted in Fig. 2: the *Vee* DAG $\mathcal{V}$ on the left and the *Lambda* DAG Λ on the right. (Both

Fig. 2. The Vee DAG $\mathcal{V}$ (left) and the Lambda DAG Λ (right).

are named for the shapes of their drawings..) Via iterated composition: $\mathcal{V}$ is a typical building block for an "expansive" computation, in which one generates an out-tree to generate subcomputations—as, e.g., in the "divide" phase of a divide-and-conquer computation; Λ is a typical building block for a "reductive" computation, in which one generates an in-tree that accumulates previously computed results—as, e.g., in the recombination phase of a divide-and-conquer computation.

Our primary interest here is in two-phase computations, which compose an *expansive* computation—represented by an out-tree—and a *reductive* computation—represented by a complete in-tree.[1] (One can easily generalize the class of computations to more ambitious multiphase ones that iteratively compose alternating expansive computations and reductive ones.) The righthand subfigure of Fig. 3 presents an expansion-reduction DAG; the lefthand subfigure explicitly demonstrates the composition that produces the expansion-reduction DAG. In a typical computation on this DAG, the "bottom" expansion tree generates values which are then accumulated (in some way) by the "top" reduction tree. When the expansion tree is composed with (i.e., "feeds into") into the reduction tree—which DAG-theoretically amounts to merging the sinks of the expansion tree with the sources of the reduction tree—one obtains the complete expansion-reduction DAG.

2) Two sample computations: Expansion-reduction DAGs arise in myriad important divide-and-conquer computations; we describe two significant examples that represent two extremes in the "uniformity" of the expansion phase of the computation. In the first example, Mergesort, the expansion phase proceeds uniformly at all nodes—which means, technically, that all leaves of the expansion tree are at the same level of the tree. In the second example, numerical integration, the depth of expansion depends on the details of the local "smoothness" of the function being integrated, so different leaves of the expansion tree can be at very different levels of the tree.

(i) Mergesort [9]. Sorting is often the first algorithmic problem covered in an Algorithms course, and Mergesort is often the first comparison-optimal sorting algorithm covered[2], because of its simple expansive-reductive structure. Mergesort takes as input a sequence of *keys* to be sorted.

> *Keys,* the names of the items to be sorted, come from an ordered set, such as numbers or words. In a comparison-based algorithm such as Mergesort, the process of sorting involves comparing and rearranging the keys of the input sequence.

During the recursive *expansive* phase of Mergesort, if the sequence of keys in a DAG-node contains more than one key,

[1] Our restriction to *binary* trees in illustrations is irrelevant to the theory: any fixed degree works.

[2] "Comparison-optimal" means that Mergesort requires only $O(n \log n)$ comparisons to sort n items.

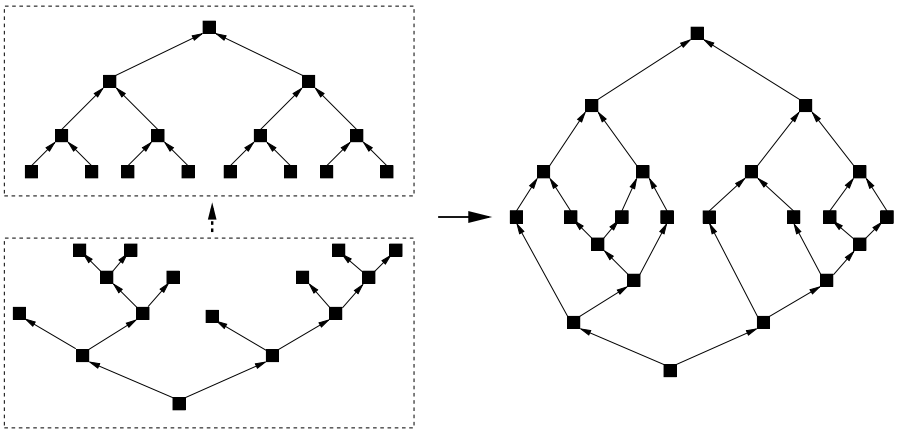

Fig. 3. A sample expansion-reduction computation-DAG. The expansive computation on the lower left is composed with ("feeds into") the reductive computation on the upper left to yield the expansive-reductive DAG on the right. Note that the expansive computation is a tree whose sinks *need not be on the same level*.

then the node bisects the sequence, sending the first half to one of its children and the second half to the other; if the sequence contains only one key, then the node just retains the key. Fig. 4(left), which is the Mergesort-specialized version of Fig. 2; Fig. 4(left) is the Mergesort-specialized version of Fig. 2(left)'s generic expansive building block. During the

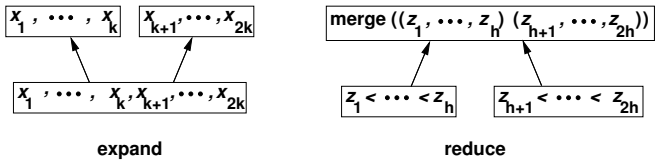

Fig. 4. The expansive (left) and reductive (right) base-DAGs for Mergesort.

recursive *reductive* phase of Mergesort, every pair of DAG-nodes that are *siblings*, in that they share a child, merge the sequences in the two nodes and pass the resulting sequence to their child. Fig. 4(right) is the Mergesort-specialized version of Fig. 2(right)'s generic reductive building block. Fig. 5 provides

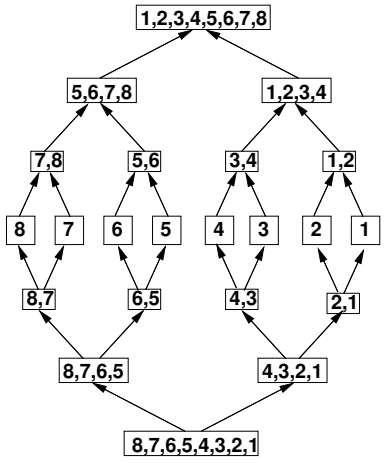

Fig. 5. A DAG-based depiction of Mergesort in action.

an example of Mergesort's complete expansion-reduction tree for the case when the input sequence has 8 keys. *Full disclosure:* When the number of input keys is not a power of 2, the algorithm is a trifle more complicated than our description suggests, because of the rounding needed during the expansion phase.

(*ii*) Numerical Integration [6]. A number of numerical integration algorithms proceed in the following way. (One can imagine the following as specifying the task that is computed in each node of the out-tree that represents the expansive portion of the computation.) Say that one wants to integrate a function F over an interval $[a_0, b_0]$.[3] One chooses a computationally simple functional form that provides a numerically adequate approximation to the area under F, at least over a very small interval. The Trapezoid method, e.g., uses a *linear* approximation to F to generate an approximation to the area under F within a small interval; Simpson's Rule uses a *quadratic* approximation; for simplicity, we describe the Trapezoid method, which is based on the approximation

$$A(X,Y) \stackrel{\text{def}}{=} \frac{1}{2}(F(X)+F(Y))(Y-X).$$

One then computes the following two quantities

$$\begin{aligned} A_0 &= A(a_0, b_0) \\ A_1 &= A\left(a_0,\ \frac{1}{2}(a_0+b_0)\right) + A\left(\frac{1}{2}(a_0+b_0),\ b_0\right). \end{aligned}$$

A_0 is a linear approximation to the area under F over the interval $[a_0,\ b_0]$, while A_1 is the approximation obtained by bisecting the interval $[a_0,\ b_0]$, thereby making some accommodation for F's curvature within the interval. Fig. 6 schematically illustrates the described scenario regarding an interval $[a_k, b_k]$: Case 0 in the figure depicts the trapezoid that forms the linear approximation; Cases 1 and 2 depict "reality" as exposed by the Trapezoid method, *as discerned from a single bisection of the interval*. In Case 1, the trapezoid is seen to underestimate the desired approximation; in Case 2, it

[3] "$[X, Y]$" denotes the closed real interval $\{Z \mid X \leq Z \leq Y\}$.

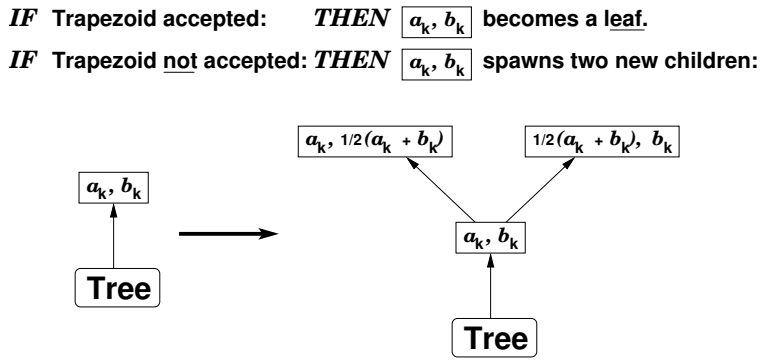

Fig. 7. A geometric view of the decision moment under the Trapezoid method.

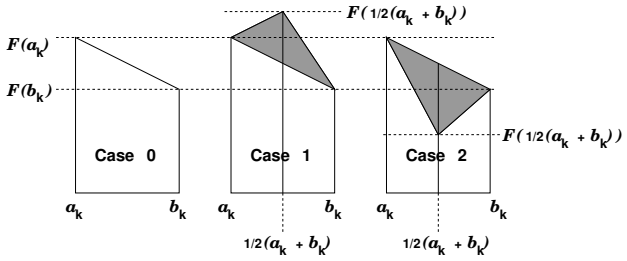

Fig. 6. A geometric view of the decision moment under the Trapezoid method. The leftmost figure is the trapezoid that yields the linear approximation to F in the interval. The other two figures show how the linear approximation can be too large (left) or too small (right), *as discerned from a bisection of the interval.*

overestimates the approximation. (Of course, a single bisection affords one just one additional snapshot into the true share of F. One must await subsequent bisections to refine one's picture of F.) The size of the grey shaded triangles represent an estimate of the error incurred by the linear approximation of F in this interval. In detail, if the difference $|A_0 - A_1|$ is sufficiently small (relative to a predetermined tolerance), then the approximation A_0 is accepted, and the current task-node becomes a leaf of the out-tree. If the difference is too large—i.e., exceeds the tolerance—then the current task spawns two new tasks, representing the two summands of A_1, which become its children in the out-tree: the left child-task seeks to integrate F over the interval $[a_0,\ \frac{1}{2}(a_0 + b_0)]$, the right child over the interval $[\frac{1}{2}(a_0 + b_0),\ b_0]$. Fig. 7 depicts the local refinement of the expansive out-tree engendered by this step in the integration algorithm. Linking up with the DAG $\mathcal{V}$ in Fig. 2, the variables w, x_0, x_1 represent the intervals over which the task-node must integrate the function F:

$$\begin{array}{lrcl} \textbf{if} & w & = & [a_0,\ b_0] \\ \textbf{then} & x_0 & = & [a_0,\ \frac{1}{2}(a_0 + b_0)] \\ \textbf{and} & x_1 & = & [\frac{1}{2}(a_0 + b_0),\ b_0] \end{array}$$

The initial task—the root of the expansive out-tree—represents the entire interval $[a_0,\ b_0]$.

Note. The preceding **if-then-else** prescription specifies intertask dependencies within the out-tree portion of the expansion-reduction DAG. It does *not* specify a computation that *we* are doing.

The integration procedure ends by composing the final out-tree $\mathcal{T}$, whose leaves contain (approximations to) the area of F over the subintervals wherein a linear approximation to F suffices, with its follow-on reductive in-tree $\widetilde{\mathcal{T}}$, which accumulates these approximations; hence, the sink of $\widetilde{\mathcal{T}}$ provides the sought approximation to the area under F over the entire interval. Linking up with the DAG Λ in Fig. 2, the variables y_0, y_1, z represent the areas under F over the various subintervals:

$$\begin{array}{lrcl} \textbf{if} & y_0 & = & A(a_0,\ \frac{1}{2}(a_0 + b_0)) \\ \textbf{and} & y_1 & = & A(\frac{1}{2}(a_0 + b_0),\ b_0) \\ \textbf{then} & z & = & y_0 + y_1 \end{array}$$

The described computation thus generates a (*possibly quite irregular*) expansive out-tree whose leaves contain (approximations to) the areas under the curve in regions that are small enough for a linear (Trapezoid method) or quadratic (Simpson's Rule) approximation to F to provide an adequate approximation to the true area. It then uses a reductive in-tree to accumulate these approximations over subintervals into the (approximate) area of F over the entire interval $[a_0,\ b_0]$.

We remark that by appropriately coarsening the expansion-reduction DAG that represents this computation, one can decrease the volume of internode communication, as well as render the computation's tasks more coarse-grain.

B. Parallel-Prefix: A Complex Expansion Paradigm

We now describe the *parallel-prefix* (or, *scan*) operator, a computational paradigm that provides myriad examples of important computations that share a readily parallelizable structure. It has been shown in many sources, including [1], [3], that the ability to compute the parallel-prefix operator efficiently automatically enables one to compute a large variety of computations efficiently, ranging from fine-grained ones such as carry-lookahead addition, to medium-grained ones such the Discrete Laplace Transform, to coarse-grained ones such as path-computation in graphs; cf. Section III-B2.

1) The operator and its DAG*:* The parallel-prefix operator is defined for an arbitrary *associative* binary operation that

we denote $*$ (think, e.g., of $+$, $\times$, $\min$, $\max$,'concatenate").[4] The $*$-parallel prefix of the input vector $\langle x_1, x_2, \ldots, x_n\rangle$ is the output vector $\langle y_1, y_2, \ldots, y_n\rangle$, where

$$\begin{array}{lcll} y_1 & = & x_1 & \\ y_2 & = & y_1 * x_2 & = \quad x_1 * x_2 \\ \vdots & & \vdots & \quad\quad \vdots \\ y_n & = & y_{n-1} * x_n & = \quad x_1 * x_2 * \cdots * x_{n-1} * x_n \end{array}$$

There are many ways of implementing the $*$-parallel prefix computation. The following way is attractive because it performs the computation in $O(\log n)$ parallel steps (in the presence of n-fold parallel computation).

for $j = 0$ **to** $\lfloor \log_2(n-1) \rfloor$ **do**
  for $i = 2^j$ **to** $n-1$ **do in parallel**
    $x_i \leftarrow x_{i-2^j} * x_i$
for $i = 0$ **to** $n-1$ **do in parallel** $y_i \leftarrow x_i$

The 8-input parallel-prefix DAG $\mathcal{P}_8$ that represents this algorithm is depicted in Fig. 8.

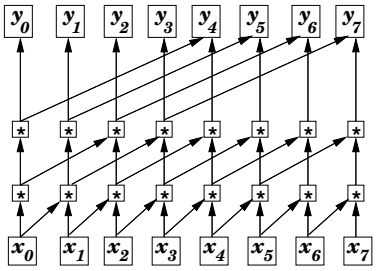

Fig. 8. The 8-input parallel-prefix DAG $\mathcal{P}_8$. Each $y_k = x_0 * x_1 * \cdots * x_k$.

2) Three sample computations: (i) Carry-Lookahead Addition. The fundamental operation of addition takes two operands expressed as sequences of bits (i.e., 0-1 digits)

$$A_{n-1}A_{n-2}\cdots A_1A_0 \quad \text{and} \quad B_{n-1}B_{n-2}\cdots B_1B_0$$

and produces a third sequence

$$(S_n)S_{n-1}S_{n-2}\cdots S_1S_0$$

that is the sum of A and B, in the classical sense. (We parenthesize S_n because explicit mention of S_n is traditionally suppressed (a) in common parlance when $S_n = 0$, and (b) in computers with "small" words that express only the low-order n bits of a number.) The simplest way to calculate the sum S from the summands A and B uses the *carry-ripple algorithm* that we all learned in elementary school. This algorithm looks at A_0 and B_0 and produces S_0 as their (modulo 2) sum, in addition to a *carry-out bit* $C_0 \in \{0, 1\}$ that it carries forward to the next pair of bits. From that point on, inductively at each step i, the algorithm looks at A_i and B_i and the *carry-in bit* C_{i-1} (which is the carry-out bit from the preceding pair of bits) and produces from them S_i and the next *carry-out bit* C_i. As is well known, the carry-ripple algorithm takes $\Theta(n)$ steps to generate the sum S of A and B.

It turns out that there is an alternative algorithm for producing S from A and B which is exponentially faster than the carry-ripple algorithm. Specifically, the following *carry-lookahead algorithm* produces S in $\Theta(\log n)$ steps. We describe the essence of this faster algorithm by adapting the treatment from [16]. First, for the sake of a uniform presentation, we create a carry-in bit for position 0 also, and we call this carry-bit C_{-1}. (Typically, one would set $C_{-1} = 0$, but, say, in a multiple-precision addition algorithm, we may want to set $C_{-1} = 1$.) We then define the following $2n$ functions that specify when carry-bits of value 1 *are generated* (the functions G_i) and/or *propagated* (the functions P_i): for $k = 0, \ldots n-1$,[5]

$$\begin{array}{lcll} G_k & = & A_k \cdot B_k & \text{(Arithmetic multiplication of bits} \\ & & & \quad \text{is equivalent to logical AND)} \\ P_k & = & A_k \oplus B_k & \text{(“}\oplus\text{” denotes } \textit{exclusive or} \text{ (}\textit{XOR}\text{):} \\ & & & \quad X \oplus Y = X\overline{Y} \vee \overline{X}Y) \end{array}$$

Using these functions, we can compute the sequence of carry bits that occur when adding A and B via the following system: for $k = 0, \ldots n-1$,

$$C_k \;=\; G_k \oplus C_{k-1}P_k$$

Expanding the preceding recursion, we arrive at the following system of equations for the sequence $\langle C_0, C_1, \ldots, C_{n-1}\rangle$.

$$\begin{array}{lcl} C_0 & = & G_0 \oplus C_{-1}P_0 \\ C_1 & = & G_1 \oplus G_0P_1 \oplus C_{-1}(P_0P_1) \\ C_2 & = & G_2 \oplus G_1P_2 \oplus G_0(P_1P_2) \oplus C_{-1}(P_0P_1P_2) \\ \vdots & & \vdots \\ C_k & = & G_k \oplus G_{k-1}P_k \oplus G_{k-2}(P_{k-1}P_k) \oplus \cdots \\ & & \quad \oplus\, G_0(P_1P_2\cdots P_k) \oplus C_{-1}(P_0P_1\cdots P_k) \\ \vdots & & \vdots \\ C_{n-1} & = & G_{n-1} \oplus G_{n-2}P_{n-1} \oplus \cdots \\ & & \quad \oplus\, C_{-1}(P_0P_1\cdots P_{n-1}) \end{array}$$

The groupings of P_i-products indicated by the parenthesizations in these expressions highlight places where the parallel-prefix operator can simplify the computation of the C_i's. Fig. 9 uses a DAG-theoretic specification of the same computation, again highlighting where the parallel-prefix operator would be useful. Both specifications of the C_i-computations illustrate that the ability to compute parallel-prefix-products (of products of the P_i's) in logarithmic time enables us to compute the sequence of carry bits, i.e., the C_i's, in logarithmic time.

(ii) The Discrete Laplace Transform. The n-dimensional *Discrete Laplace Transform* (*DLT*, for short)—aka the *Z-Transform*—transforms an n-dimensional vector $\langle x_0, x_1, \ldots, x_{n-1}\rangle$ of real numbers to an m-dimensional vector of complex functions $\langle y_0(\zeta), y_1(\zeta), \ldots, y_{m-1}(\zeta)\rangle$,

[4] Associativity guarantees that $x*(y*z) = (x*y)*z$ for all triples x, y, z, so that the unparenthesized expression $x * y * z$ is unambiguous.

[5] "$\oplus$" denotes the *exclusive or* operation: $A_k \oplus B_k = A_k\overline{B}_k \vee \overline{A}_kB_k$.

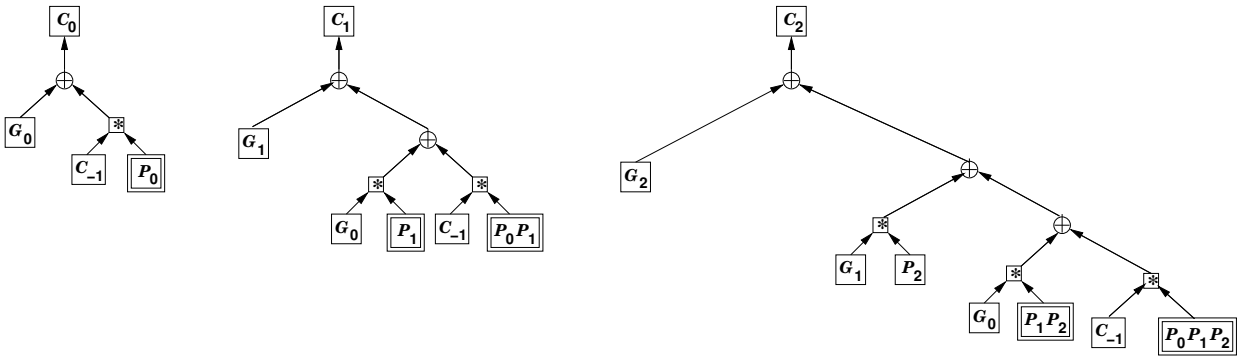

Fig. 9. Specifications via DAGs of the first three carry-out bits, C_0, C_1, C_2, for an adder. The binary associative operations are logical multiplication ($*$) and XOR ($\oplus$). The nodes that are highlighted by double "boxes" indicate products of P_i's that can be produced using the parallel-prefix operator.

where ζ is a complex number. Each function $y_k(\zeta)$ is given by

$$\begin{aligned} y_k(\zeta) &= x_0 + x_1\zeta^k + x_2\zeta^{2k} + \cdots + x_{n-1}\zeta^{(n-1)k} \\ &= \sum_{i=0}^{n-1} x_i\zeta^{ik}. \end{aligned} \tag{1}$$

We build on [4] to develop an algorithm for computing the DLT that uses: (a) the parallel-prefix operator to generate the required powers of the argument ζ; (b) a reductive "in-tree" to accumulate the terms of the sum (1). For stage (a) of the algorithm, we use an n-input parallel-prefix DAG to generate the vector $\langle 1, \zeta^k, \ldots, \zeta^{(n-1)k}\rangle$ of powers of ζ from the input vector $\langle \zeta^k, \ldots, \zeta^k\rangle$. For stage (b) of the algorithm, we use an in-tree having n sources to accumulate the required sum. Each source v_i begins by multiplying x_i times the power of ζ that v_i has received. In terms of the DAG Λ in Fig. 2, the variables y_0, y_1, z represent subsums of (1):

$$\begin{array}{lrcl} \textbf{if} & y_0 & = & x_i \cdot \zeta^{ik} \\ \textbf{and} & y_1 & = & x_j \cdot \zeta^{jk} \\ \textbf{then} & z & = & y_0 + y_1 \end{array}$$

Fig. 10 presents the resulting 8-input composite DLT DAG.

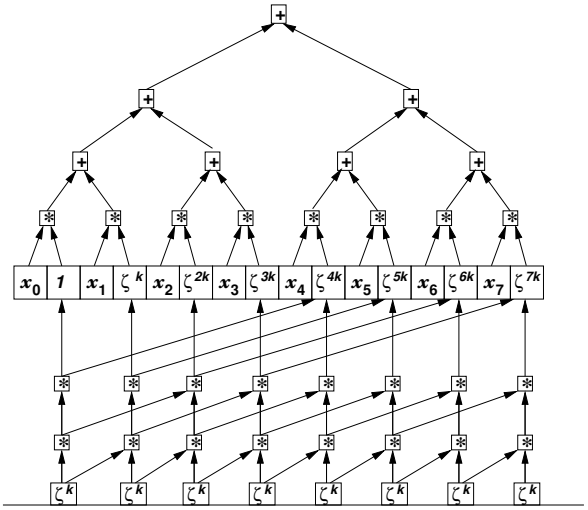

Fig. 10. The 8-input DLT DAG $\mathcal{L}_8$. The binary associative operation $*$ is complex multiplication.

(iii) Computing paths in a graph. We now consider a rather coarse-grain computation that falls within the framework of this section. Consider Fig. 11 as we describe the computation. Say that we have a 9-node graph $\mathcal{G}$, presented via its 9×9 boolean adjacency matrix A. (We choose the integer 9 to make Fig. 11 attractive.) We wish to compute a 9×9 matrix M of integers whose (i, j) entry is a vector $\vec{v}_{i,j} = \langle \beta_{i,j}^{(1)}, \ldots, \beta_{i,j}^{(8)}\rangle$, where

$$\beta_{i,j}^{(k)} = \begin{cases} 1 & \text{if there is a path of length } k \text{ in } \mathcal{G} \\ & \text{between nodes } i \text{ and } j \\ 0 & \text{if there is no such path} \end{cases}$$

We compute M via a computation whose intertask dependencies are depicted in Fig. 11.

1) We use an 8-input parallel-prefix to compute all logical powers of A. Within each power A^k, the (i, j) entry is 1 or 0, indicating whether or not there is a path of length k in $\mathcal{G}$ between nodes i and j.
2) We use an in-tree to accumulate the information from the eight power matrices A^k into the 64 vectors $\vec{v}_{i,j}$ of matrix M.

There are, of course, many variations on the indicated theme, all of which yield to computations having the structure depicted in Fig. 11.

C. Butterfly-Structured Computations

1) The structure of the DAG*s:* The computations we discuss in this section are built via iterated composition from the *butterfly building block* DAG $\mathcal{B}$ of Fig. 12, so named for its shape in the drawing. A large variety of transformations effected by butterfly building blocks—i.e., specifications of y_0 and y_1 as functions of x_0 and x_1 in Fig. 12—lead, via iterated composition, to useful computations. Notable among the DAGs constructed by composing butterfly building blocks are the DAGs in the *butterfly* family of DAGs, which are also known as *FFT* DAG*s*, in honor of the landmark *Fast Fourier Transform* algorithm [9]. Butterfly DAGs are sized by their *dimensionality*, which is the base-2 logarithm of their common number of *inputs* and *outputs*. The 1-*dimensional* butterfly

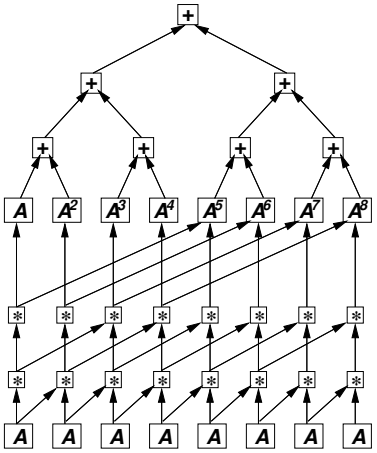

Fig. 11. Computing the paths in a 9-node graph that is specified via its adjacency matrix A. The binary associative operation $*$ is matrix multiplication, and the operation $+$ is matrix addtion.

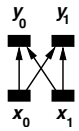

Fig. 12. The butterfly building-block DAG $\mathcal{B}$.

DAG is the butterfly building block with two inputs (x_0 and x_1 in Fig. 12) and two outputs (y_0 and y_1 in Fig. 12): $\mathcal{B}_1 = \mathcal{B}$. Inductively, the $(n+1)$-dimensional butterfly DAG, $\mathcal{B}_{n+1}$, is built by taking two copies of $\mathcal{B}_n$, and 2^{n+1} new output nodes, y_0, y_1, ..., $y_{2^{n+1}-1}$ and providing arcs from each output of each copy of $\mathcal{B}_n$, call it v_i, to two of the new output nodes, y_i and y_{i+2^n}; the inputs of $\mathcal{B}_{n+1}$ are the inputs of the two copies of $\mathcal{B}_n$; see Fig. 13.

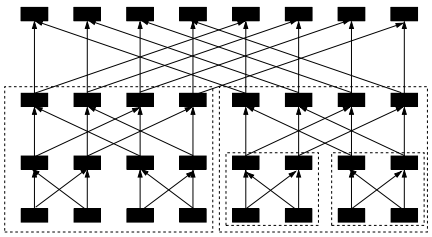

Fig. 13. The 3-dimensional butterfly DAG $\mathcal{B}_3$, with dashed boxes that illustrate the recursive structure of the family of butterfly DAGs.

2) Sample computations: (i) Odd-even mergesort. This is one of the oldest parallel sorting algorithms. It shares with "ordinary" mergesort (Section III-A2(i)) the strategy of mechanically breaking a sequence that is to be sorted into subsequences and then iteratively merging the subsequences. In contrast to "ordinary" mergesort, though, the breaking into subsequences and the merging of subsequences is accomplished in a nonobvious, albeit regular, way. We build on the description in [17].

The base case of Odd-even mergesort sorts a two-item sequence by using the butterfly building block of Fig. 2 as a *comparator;* see Fig. 14(a):

$$y_0 \;=\; \min(x_0,\; x_1) \quad \text{and} \quad y_1 \;=\; \max(x_0,\; x_1). \tag{2}$$

Building inductively on this base case, let us examine how Odd-even mergesort merges the final pair of sorted lists. Say that n is a power of 2, and consider how to merge the sorted lists

$$\begin{aligned} X &= (x_0 < x_1 < \cdots < x_{n/2-1}) \\ Y &= (y_0 < y_1 < \cdots < y_{n/2-1}). \end{aligned}$$

(We include the less-than signs to emphasize that the lists are sorted.)

1. In a manner suggestive of its name, Odd-even mergesort partitions these lists into four lists—each of which is still clearly sorted.

$$\begin{aligned} \text{Even}(X) &= (x_0 < x_2 < \cdots < x_{n/2-1}) \\ \text{Odd}(X) &= (x_1 < x_3 < \cdots < x_{n/2-2}) \\ \text{Even}(Y) &= (y_0 < y_2 < \cdots < y_{n/2-1}) \\ \text{Odd}(Y) &= (y_1 < y_3 < \cdots < y_{n/2-2}). \end{aligned}$$

2. We invoke Odd-even mergesort recursively, to merge pairs of these lists, to form two $n/2$-item sorted lists:

$$\begin{aligned} U &= \text{Merge}\big(\text{Even}(X), \text{Odd}(Y)\big) \\ &= (u_0 < u_1 < \cdots < u_{n/2-1}) \\ V &= \text{Merge}\big(\text{Odd}(X), \text{Even}(Y)\big) \\ &= (v_0 < v_1 < \cdots < v_{n/2-1}). \end{aligned}$$

3. Finally, we merge the (sorted) sequences U and V. It is shown in sources such as [17] that this last step is guaranteed to be easy—in contrast, possibly, to merging sequences X and Y. In particular, we can merge U and V by just

3(a) interleaving U and V to obtain the (not necessarily sorted) sequence

$$W \;=\; (u_0, v_0, u_1, v_1, \ldots, u_{n/2-1}, v_{n/2-1}).$$

3(b) using the comparator transformation of Eq. (2) $n/2$ times, with each adjacent pair (u_i, v_i).

Fig. 14(b) provides a schematic description of the algorithm.

(ii) Convolutions. The *Fast-Fourier Transform* (*FFT*) algorithm is the "personification" of convolutional algorithms, both because many other convolutions can be reformulated in terms of the FFT and because of its pedigree as the first of genre to be sped up to logarithmic time; see [7]. The n-dimensional *Discrete Fourier Transform* is the special case of the n-dimensional Discrete Laplace Transform (Section III-B2(ii)) where the argument ζ is a *primitive complex nth root of unity*, i.e., a solution to the equation $\zeta^n = 1$ such that $\zeta^k \neq 1$ for any integer $0 < k < n$. In this situation, by convention, we rename ζ as ω_n, and we note that every power $\upsilon = \omega_n^i$ also satisfies $\upsilon^n = 1$ (although υ is often not *primitive*).[6] We inherit a specification of the function $y_k(\omega)$ from Eq. (1). The

[6] $e^{2\pi i/n}$ is often viewed as the *standard* primitive nth root of unity.

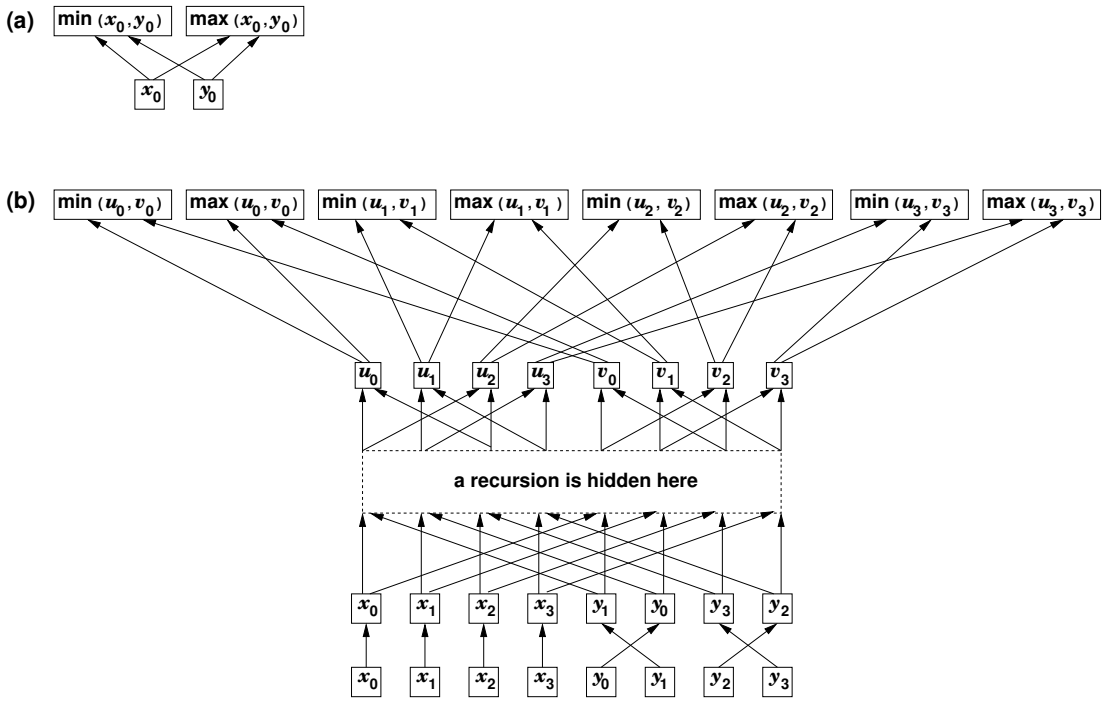

Fig. 14. Illustrating the Odd-even mergesort algorithm. (a) The base case of a 2-element list; (b) the "general" case of an 8-item list.

DFT can be implemented quite efficiently via an algorithm known as the *Fast Fourier Transform*, which has the structure of the FFT DAG—in fact, the DAG is named for the algorithm! While the FFT algorithm is not difficult conceptually, it is quite complex technically—hence would be relegated to an advanced spot in most curricula. This complexity, coupled with current space limits, leads us to offer just the following hint as to the butterfly stucture hidden within Eq. (1). If we define

$$\begin{aligned} & y_k^{(m)}(\omega; x_0, x_1, \ldots, x_{m-1}) \\ & \quad = \; x_0 + x_1\omega^k + x_2\omega^{2k} + \cdots + x_{m-1}\omega^{(m-1)k}. \end{aligned}$$

then we note that

$$\begin{aligned} & y_k^{(2m)}(\omega; x_0, x_1, \ldots, x_{2m-1}) \\ & \quad = \; x_0 + x_1\omega^k + x_2\omega^{2k} + \cdots + x_{2m-1}\omega^{(2m-1)k} \\ & \quad = \; y_k^{(m)}(\omega; x_0, x_1, \ldots, x_{m-1}) \; + \\ & \qquad \omega^{mk} \times y_k^{(m)}(\omega; x_m, x_{m+1}, \ldots, x_{2m-1}). \end{aligned}$$

This fact makes the algorithmic structure sketched in Fig. 15 plausible. An important technical note in understanding the algorithm is that the structure of the DAG mandates presenting the inputs x_i in *bit-reversal order* of their indices; i.e., write the length-$(\log_2 n)$ binary numerals for $1, 2, \ldots, n$, reverse each resulting bit-string, and interpret each string as a numeral. We refer the reader to Section 3.7 of [17] or Section 30.2 of [9] for a detailed derivation of the FFT algorithm and the structure of its DAG.

We close this section by noting the important fact that the problem of multiplying polynomials can be formulated as a convolution. To sketch why this is true, let us be given two univariate polynomials of degree n,

$$f(x) \;=\; a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$$

and

$$g(x) \;=\; b_0 + b_1x + b_2x^2 + \cdots + b_nx^n.$$

The product of $f(x)$ and $g(x)$ is the polynomial

$$[f \otimes g](x) \;=\; A_0 + A_1x + A_2x^2 + \cdots + A_{2n}x^{2n}.$$

Each coefficient A_k of this product polynomial is a *convolution*, i.e., a sum of the form

$$A_k = a_0b_k + a_1b_{k-1} + \cdots + a_{k-1}b_1 + a_kb_0 = \sum_{i=0}^{k} a_ib_{k-i}.$$

IV. Conclusions

This paper addresses the quesiton of how to teach PDC algorithmics—i.e., the algorithmic component of parallel and distributed computing—in a way that will "stay" with the student as s/he progresses from introductory courses to more advanced ones. The thesis of the paper is that incorporating the dependency structure of algorithms, as exposed by dependency DAGs, into the teaching of algorithms at all levels of the curriculum will: (a) add the desired staying power to the topic and (b) will benefit the student by emphasizing *why* certain algorithms parallelize more easily and more effectively than others.

Of course, the intertask dependencies exposed by the DAGs by no means supply the whole picture regarding parallel efficiency. Heavy communication requirements can negate any computational benefits that the absence of intertask dependencies seems to offer. Recognizing well that DAGs do not tell

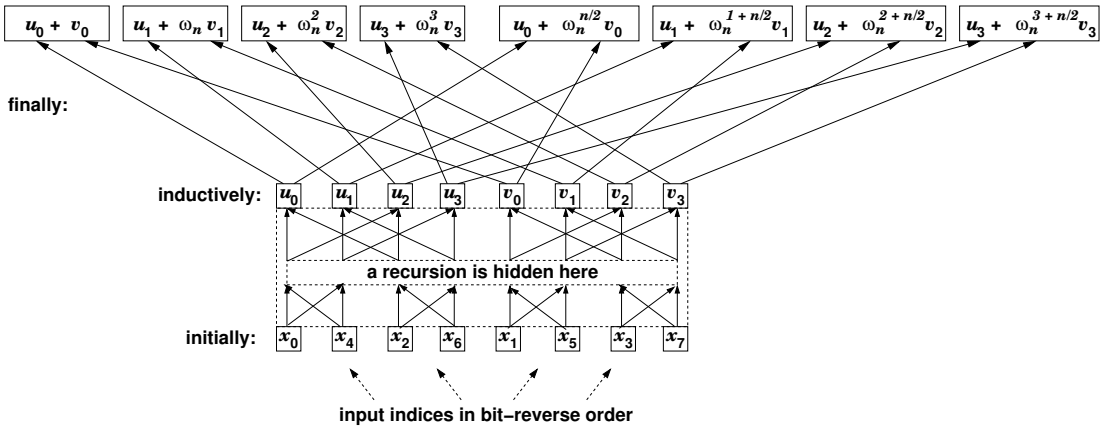

Fig. 15. Hinting at how the 3-dimensional DFT can be computed using the 3-dimensional butterfly DAG. This figure is adapted from [17].

the whole story, one can argue that the presence or absence of such dependencies is an important facet of understanding parallelism—facet that the student should be encouraged to always be aware of. One reason for this—perhaps the least important one—is that the dependency structure that is exposed by DAGs is *inherent:* it is a property of the computation. The importance of communication in selecting a good parallel algorithm: (a) depends on the characteristics of the hst platform, not just on the characteristics of the computation, (b) can require an analysis that is daunting even for computing professionals, let alone beginning students; a glance at [10] will amply support point (b).

Among the invaluable benefits of incorporating DAGs into one's treatment of algorithms are the facts that always keeping DAG-structure in mind:

- will help the student recognize nonobvious kinships among important problems
- will help build conceptual bridges between areas that are not obviously related. Examples such as those we have presented should illustrate this point.

Acknowledgment. A.L. Rosenberg is a co-PI of the CEDR Center for PDC curriculum development. His research was supported in part by NSF Grant CRI-1205650. S.K. Prasad offered several helpful comments and suggestions.

REFERENCES

[1] S. Aluru (2012): Teaching parallel computing through parallel prefix. *Supercomputing'12,* HPC Educator session: http://sc12.supercomputing.org/schedule/event_detail.php?evid=eps115.

[2] V.E. Beneš (1964): Optimal rearrangeable multistage connecting networks. *Bell Syst. Tech. J. 43*, 1641–1656.

[3] G.E. Blelloch (1989): Scans as primitive parallel operations. *IEEE Trans. Comput. 38*, 1526–1538.

[4] L.I. Bluestein (1970): A linear filtering approach to the computation of the Discrete Fourier Transform. *IEEE Trans. Audio Electroacoustics, AU-18*, 451–455.

[5] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, Y. Zhou (1995): Cilk: An efficient multithreaded runtime system. *5th ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming (PPoPP'95)*.

[6] R.L. Burden, J.D. Faires (2000): *Numerical Analysis* (7th ed.), Brooks/Cole, ISBN 0-534-38216-9.

[7] J.W. Cooley and J.W. Tukey (1965): An algorithm for the machine computation of complex Fourier series. *Mathematics of Computation 19(90)*, 297–301.

[8] G. Cordasco, G. Malewicz, A.L. Rosenberg (2007): Applying IC-scheduling theory to some familiar classes of computations. *Wkshp. on Large-Scale, Volatile Desktop Grids (PCGrid'07)* (2007).

[9] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein (1999): *Introduction to Algorithms* (2nd Edition). MIT Press, Cambridge, Mass.

[10] L.-X. Gao, A.L. Rosenberg, R.K. Sitaraman (1999): Optimal clustering of tree-sweep computations for high-latency parallel environments. *IEEE Trans. Parallel and Distributed Systems 10*, 813–824.

[11] A. González-Escribano, A. van Gemund, V. Cardeñoso-Payo (2002): Mapping unstructured applications into nested parallelism. *High Performance Computing for Computational Science (VECPAR '02)*.

[12] A. Gerasoulis and T. Yang (1992): A comparison of clustering heuristics for scheduling DAGs on multiprocessors. *J. Parallel Distr. Comput. 16*, 276–291.

[13] L. He, Z. Han, H. Jin, L. Pan, S. Li (2000): DAG-based parallel real time task scheduling algorithm on a cluster. *Intl. Conf. on Parallel and Distr. Processing Techniques and Applications (PDPTA'2000)*, 437–443.

[14] J.-W. Hong and H.T. Kung (1981): I/O complexity: the red-blue pebble game. *13th ACM Symp. on Theory of Computing*, 326–333.

[15] H.T. Hsu (1975): An algorithm for finding a minimal equivalent graph of a digraph. *J. ACM 22*, 11–16.

[16] K. Hwang (1979): *Computer Arithmetic: Principles, Architecture, and Design*. John Wiley and Sons, New York.

[17] F.T. Leighton (1992): *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, San Mateo, Cal.

[18] M.S. Paterson, C.E. Hewitt (1970): Comparative schematology. *Project MAC Conf. on Concurrent Systems and Parallel Computation*, ACM Press, 119–127.

[19] N.J. Pippenger (1980): Pebbling. In *5th IBM Symp. on Math. Foundations of Computer Science*.

[20] S.K. Prasad, A. Gupta, K. Kant, A. Lumsdaine, D. Padua, Y. Robert, A.L. Rosenberg, A. Sussman, C.C. Weems (2012): Toward a core undergraduate curriculum in parallel and distributed computing. *Computer Education,* No. 2012-6.

[21] S.K. Prasad, A. Gupta, K. Kant, A. Lumsdaine, D. Padua, Y. Robert, A.L. Rosenberg, A. Sussman, C.C. Weems (2012): Literacy for all in parallel and distributed computing: Guidelines for an undergraduate core curriculum. *CSI J. Computing 1(2)* (15 pages).

[22] V. Sarkar (1989): *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press, Cambridge, Mass.