Changing Challenges for Collaborative Algorithmics

Arnold L. Rosenberg
Department of Computer Science
University of Massachusetts at Amherst
Amherst, MA 01003, USA
`rsnbrg@cs.umass.edu`

October 6, 2004

Abstract

Technological advances and economic considerations have led to a wide variety of modalities of *collaborative computing:* the use of multiple computing agents to solve individual computational problems. Each new modality creates new challenges for the algorithm designer. Older "parallel" algorithmic devices no longer work on the newer computing platforms (at least in their original forms) and/or do not address critical problems engendered by the new platforms' characteristics. In this chapter, the field of "collaborative algorithmics" is divided into four epochs, representing (one view of) the major evolutionary eras of collaborative computing platforms. The changing challenges encountered in devising algorithms for each epoch are discussed, and some notable sophisticated responses to the challenges are described.

1 Introduction

Collaborative computing is a regime of computation in which multiple agents are enlisted in the solution of a single computational problem. Until roughly one decade ago, it was fair to refer to collaborative computing as *parallel computing.* Developments engendered by both economic considerations and technological advances make the older rubric both inaccurate and misleading, as the *multiprocessors* of the past have been joined by *clusters*—independent computers interconnected by a local-area network (LAN)—and by various modalities of *Internet computing*—loose confederations of computing agents of differing levels of commitment to the common computing enterprise. The agents in the newer collaborative computing milieux often do their computing at their own times and in their own locales—definitely not "in parallel."

Every major technological advance in all areas of computing creates significant new scheduling challenges even while enabling new levels of computational efficiency (measured in time and/or space and/or cost). This chapter presents one algorithmicist's view of the paradigm-challenges milestones in the evolution of collaborative computing platforms and of the algorithmic challenges each change in paradigm has engendered. The chapter is organized around a somewhat eccentric view of the evolution of collaborative computing technology through four "epochs," each distinguished by the challenges one faced when devising algorithms for the associated computing platforms.

1. In the epoch of *shared-memory multiprocessors*:

    - One had to cope with partitioning one's computational job into disjoint subjobs that could proceed in parallel on an assemblage of identical processors. One had to try to keep all processors fruitfully busy as much of the time as possible. (The qualifier "fruitfully" indicates that the processors are actually working on the problem to be solved, rather than on, say, bookkeeping that could be avoided with a bit more cleverness.)
    - Communication between processors was effected through shared variables, so one had to coordinate access to these variables. In particular, one had to avoid the potential races when two (or more) processors simultaneously vied for access to a single memory module, especially when some access was for the purpose of writing to the same shared variable.
    - Since all processors were identical, one had, in many situations, to craft protocols that gave processors separate identities—the process of so-called *symmetry breaking* or *leader election*. (This was typically necessary when one processor had to take a coordinating role in an algorithm.)

2. The epoch of *message-passing multiprocessors* added to the technology of the preceding epoch a user-accessible interconnection network—of known structure—across which the identical processors of one's parallel computer communicated. On the one hand, one could now build much larger aggregations of processors than one could before. On the other hand:

    - One now had to worry about coordinating the routing and transmission of messages across the network, in order to select short paths for messages, while avoiding congestion in the network.
    - One had to organize one's computation to tolerate the often-considerable delays caused by the point-to-point latency of the network and the effects of network bandwidth and congestion.

- Since many of the popular interconnection networks were highly symmetric, the problem of symmetry breaking persisted in this epoch. Since communication was now over a network, new algorithmic avenues were needed to achieve symmetry breaking.
- Since the structure of the interconnection network underlying one's multiprocessor was known, one could—and was well advised to—allocate substantial attention to network-specific optimizations when designing algorithms that strove for (near) optimality. (Typically, for instance, one would strive to exploit *locality:* the fact that a processor was closer to some processors than to others.) A corollary of this fact is that one often needed quite disparate algorithmic strategies for different classes of interconnection networks.

3. The epoch of *clusters*—also known as *networks of workstations* (*NOW*s, for short)—introduced two new variables into the mix, even while rendering many sophisticated multiprocessor-based algorithmic tools obsolete. In Section 3, we outline some algorithmic approaches to the following new challenges.

   - The computing agents in a cluster—be they pc's, or multiprocessors, or the eponymous workstations—are now independent computers that communicate with each other over a local-area network (LAN). This means that communication times are larger and that communication protocols are more ponderous, often requiring tasks such as: breaking long messages into packets, encoding, computing checksums, explicitly setting up communications (say, via a handshake). Consequently, tasks must now be coarser-grained than with multiprocessors, in order to amortize the costs of communication. Moreover, the respective computations of the various computing agents can no longer be tightly coupled, as they could be in a multiprocessor. Further, in general, network latency can no longer be "hidden" via the sophisticated techniques developed for multiprocessors. Finally, one can usually no longer translate knowledge of network topology into network-specific optimizations.
   - The computing agents in the cluster, either by design or chance (such as being purchased at different times), are now often *heterogeneous*, differing in speeds of processors and/or memory systems. This means that a whole range of algorithmic techniques developed for the earlier epochs of collaborative computing no longer work—at least in their original forms [127]. On the positive side, heterogeneity obviates symmetry breaking, as processors are now often distinguishable by their unique combinations of computational resources and speeds.

4. The epoch of *Internet computing*, in its several guises, has taken the algorithmics of collaborative computing precious near to—but never quite reaching—that of distributed computing. While Internet computing is still evolving in often-unpredictable

directions, we detail two of its circa-2003 guises in Section 4. Certain characteristics of present-day Internet computing seem certain to persist.

- One now loses several types of *predictability* that played a significant background role in the algorithmics of prior epochs.
  - Interprocessor communication now takes place over the Internet. In this environment:
    - a message shares the "airwaves" with an unpredictable number and assemblage of other messages; it may be dropped and resent; it may be routed over any of myriad paths. All of these factors make it impossible to predict a message's transit time.
    - a message may be accessible to unknown (and untrusted) sites, enhancing the need for security-enhancing measures.
  - The predictability of interactions among collaborating computing agents that anchored algorithm development in all prior epochs no longer obtains, due to the fact that remote agents are typically not dedicated to the collaborative task. Even in modalities of Internet computing in which remote computing agents promise to complete computational tasks that are assigned to them, they typically do not guarantee *when*. Moreover, even the guarantee of eventual computation is not present in all modalities of Internet computing: in some modalities remote agents cannot be relied upon *ever* to complete assigned tasks.
- In several modalities of Internet computing, computation is now *unreliable* in two senses.
  - The computing agent assigned a task may, without announcement, "resign from" the aggregation, abandoning the task. (This is the extreme form of temporal unpredictability just alluded to.)
  - Since remote agents are unknown and anonymous in some modalities, the computing agent assigned a task may maliciously return fallacious results. This latter threat introduces the need for computation-related security measures (e.g., result-checking and agent monitoring) for the first time to collaborative computing. This problem is discussed in a news article at

    ⟨`http://www.wired.com/news/technology/0,1282,41838,00.html`⟩.

In succeeding sections, we expand on the preceding discussion, defining the collaborative computing platforms more carefully and discussing the resulting challenges in more detail. Due to a number of excellent widely accessible sources that discuss and analyze the epochs of multiprocessors, both shared-memory and message-passing, our discussion of the first two of our epochs, in Section 2, will be rather brief. Our discussion of the epochs of cluster computing (in Section 3) and Internet computing (in Section 4) will be both broader and deeper. In each case, we describe the subject computing platforms in some detail and describe a variety of sophisticated responses to the algorithmic challenges of that epoch. Our goal is to highlight studies that attempt to develop algorithmic strategies that respond in novel ways to the challenges of an epoch. Even with this goal in mind, the reader should be forewarned that:

- her guide has an eccentric view of the field, which may differ from the views of many other collaborative algorithmicists;
- some of the still-evolving collaborative computing platforms we describe will soon disappear, or at least morph into possibly unrecognizable forms;
- some of the "sophisticated responses" we discuss will never find application beyond the specific studies they occur in.

This said, I hope that this survey, with all of its limitations, will convince the reader of the wonderful research opportunities that await her "just on the other side" of the systems and applications literature devoted to emerging collaborative computing technologies.

2 The Epochs of Multiprocessors

The quick tour of the world of multiprocessors in this section is intended to convey a sense of what stimulated much of the algorithmic work on collaborative computing on this computing platform. The following books and surveys provide an excellent detailed treatment of many subjects that we only touch upon and even more topics that are beyond the scope of this chapter: [5, 45, 50, 80, 93, 97, 134].

2.1 Multiprocessor Platforms

As technology allowed circuits to shrink, starting in the 1970's, it became feasible to design and fabricate computers that had many processors. Indeed, a few theorists had anticipated these advances in the 1960's [79]. The first attempts at designing such *multiprocessors* envisioned them as straightforward extensions of the familiar von Neumann architecture, in which a processor box—now populated with many processors—interacted with a single memory box; processors would coordinate and communicate with each other via shared variables. The resulting *shared-memory multiprocessors* were easy to think about, both for computer architects and computer theorists [61]. Yet, using such multiprocessors effectively turned out to present numerous challenges, exemplified by the following.

- Where/how does one identify the parallelism in one's computational problem? This question persists to this day, feasible answers changing with evolving technology. Since there are approaches to this question that often to not appear in the standard references, we shall discuss the problem briefly in Section 2.2.

- How does one keep all available processors fruitfully occupied—the problem of *load balancing*? One finds sophisticated multiprocessor-based approaches to this problem in primary sources such as [58, 111, 123, 138].

- How does one coordinate access to shared data by the several processors of (especially, a shared-memory) multiprocessor? The difficulty of this problem increases with the number of processors. One significant approach to sharing data requires establishing order among a multiprocessor's indistinguishable processors, by selecting "leaders" and "subleaders," etc. How does one efficiently pick a "leader" among indistinguishable processors—the problem of *symmetry breaking*? One finds sophisticated solutions to this problem in primary sources such as [8, 46, 107, 108].

A variety of technological factors suggest that shared memory is likely a better idea as an abstraction than as a physical actuality. This fact led to the development of *distributed shared memory* multiprocessors, in which each processor had its own memory module, and accesses to remote data was through an interconnection network. Once one had processors communicating over an an interconnection network, it was a small step from the distributed shared memory abstraction to explicit *message-passing,* i.e., to having processors communicate with each other directly rather than through shared variables. In one sense, the introduction of interconnection networks to parallel architectures was liberating: one could now (at least in principle) envision multiprocessors with many thousands of processors. On the other hand, the explicit algorithmic use of networks gave rise to a new set of challenges.

- How can one route large numbers of messages within a network without engendering congestion ("hot spots") that renders communication insufferably slow? This is one of the few algorithmic challenges in parallel computing that has an acknowledged champion. The two-phase randomized routing strategy developed in [150, 154] provably works well in a large range of interconnection networks (including the popular butterfly and hypercube networks) and empirically works well in many others.

- Can one exploit the new phenomenon—*locality*—that allows certain pairs of processors to intercommunicate faster than others? The fact that locality can be exploited to algorithmic advantage is illustrated in [1, 101]. The phenomenon of locality in parallel algorithmics is discussed in [124, 156].

- How can one cope with the situation in which the structure of one's computational problem—as exposed by the graph of data dependencies—is incompatible with the structure of the interconnection network underlying the multiprocessor that one has access to? This is another topic that is not treated fully in the references, so we discuss it briefly in Section 2.2.

- How can one organize one's computation so that one accomplishes valuable work while awaiting responses from messages, either from the memory subsystem (memory accesses) or from other processors. A number of innovative and effective responses to variants of this problem appear in the literature; see , e.g., [10, 36, 66].

In addition to the preceding challenges, one now also faced the largely unanticipated insuperable problem that one's interconnection network may not "scale." Beginning in 1986, a series of papers demonstrated that the physical realizations of large instances of the most popular interconnection networks could not afford one performance consistent with idealized analyses of those networks [31, 155, 156, 157]. A word about this problem is in order, since the phenomenon it represents influences so much of the development of parallel architectures. We live in a three-dimensional world: areas and volumes in space grow polynomially fast when distances are measured in units of length. This physical polynomial growth notwithstanding, for many of the algorithmically attractive interconnection networks—*hypercubes*, *butterfly networks*, and *de Bruijn networks*, to name just three—the number of nodes (read: "processors") grows *exponentially* when distances are measured in number of interprocessor links. This means, in short, that the interprocessor links of these networks must grow in length as the networks grow in number of processors. *Analyses that predict performance in number of traversed links do not reflect the effect of link-length on actual performance.* Indeed, the anaysis in [31] suggests—on the preceding grounds—that only the polynomially growing *mesh-like networks* can supply in practice efficiency commensurate with idealized theoretical analyses.[1]

We now discuss briefly a few of the challenges that confronted algorithmicists during the epochs of multiprocessors. We concentrate on topics that are not treated extensively in books and surveys as well as on topics that retain their relevance beyond these epochs.

2.2 Algorithmic Challenges and Responses

Finding Parallelism. The seminal study [37] was the first to systematically distinguish between the inherently sequential portion of a computation and the parallelizable portion. The analysis in that source led to "Brent's Scheduling Principle," which states, in simplest form, that the time for a computation on a p-processor computer need be no greater than $t + n/p$ where t is the time for the inherently sequential portion of the computation, and n is the total number of operations that must be performed. While the study illustrates

[1]Fig. 1 depicts the four mentioned networks. See [93, 134] for definitions and discussions of these and related networks. Additional sources such as [4, 21, 90] illustrate the algorithmic use of such networks.

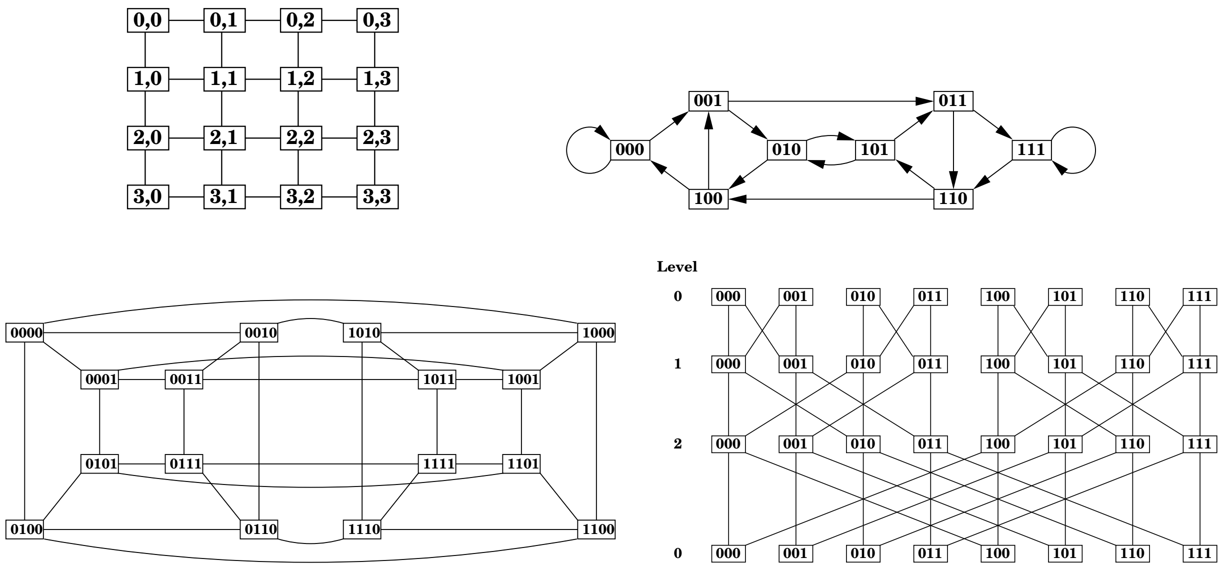

Figure 1: *Four interconnection networks. Row* 1*: the* 4 × 4 *mesh, the* 3*-dimensional de Bruijn network; row* 2*: the* 4*-dimensional boolean hypercube, the* 3*-level butterfly network (note the two copies of level* 0*)*

how to achieve the bound of the Principle for a class of arithmetic computations, it leaves open the challenge of discovering the parallelism in general computations. Two major approaches to this challenge appear in the literature and are discussed here.

Parallelizing computations via clustering/partitioning. Two related major approaches have been developed for scheduling computations on parallel computing platforms, when the computation's intertask dependencies are represented by a *computation-dag*—a directed acyclic graph, each of whose arcs $(x \rightarrow y)$ betokens the dependence of task y on task x; sources never appear on the righthand side of an arc; sinks never appear on the lefthand side.

The first such approach is to *cluster* a computation-dag's tasks into "blocks" whose tasks are so tightly coupled that one would want to allocate each block to a single processor to obviate any communication when executing these tasks. A number of efficient heuristics have been developed to effect such clustering for general computation-dags [67, 83, 103, 139]. Such heuristics typically base their clustering on some easily computed characteristic of the dag, such as its *critical path*—the most resource-consuming source to-sink path, including both computation time and volume of intertask data—or its *dominant sequence*—a source-to-sink path, possibly augmented with dummy arcs, that accounts for the entire makespan of the computation. Several experimental studies compare these heuristics in a variety of settings [54, 68], and systems have been developed to exploit such clustering in devising schedules [43, 140, 162]. Numerous algorithmic studies have demonstrated analytically the *provable* effectiveness of this approach for special scheduling classes of computation-dags [65, 117].

Dual to the preceding clustering heuristics is the process of clustering by *graph separation*. Here one seeks to partition a computation-dag into subdags by "cutting" arcs that interconnect loosely coupled blocks of tasks. When the tasks in each block are mapped to a single processor, the small numbers of arcs interconnecting pairs of blocks lead to relatively small—hence, inexpensive—interprocessor communications. This approach has been studied extensively in the parallel-algorithms literature, with regard to myriad applications, ranging from circuit layout to numerical computations to nonserial dynamic programming. A small sampler of the literature on specific applications appears in [28, 55, 64, 99, 106]; heuristics for accomplishing efficient graph partitioning (especially into roughly equal-size subdags) appear in [40, 60, 82]; further sample applications, together with a survey of the literature on algorithms for finding graph separators appears in [134].

Parallelizing using dataflow techniques. A quite different approach to finding parallelism in computations builds on the *flow of data* in the computation. This approach originated with the VLSI revolution fomented by Mead and Conway [105], which encouraged computer scientists to apply their tools and insights to the problem of designing computers. Notable among the novel ideas emerging from this influx was the notion of *systolic array*—a dataflow-driven special-purpose parallel (co)processor [86, 87]. A major impetus for the development of this area was the discovery, in [109, 120], that for certain classes of computations—including, e.g., those specifiable via nested for-loops—such machines could be designed "automatically." This area soon developed a life of its own as a technique for finding parallelism in computations, as well as designing special-purpose parallel machines. There is now an extensive literature on the use of systolic design principles for a broad range of specific computations [38, 39, 89, 91, 122], as well as for large general classes of computations that are delimited by the structure of their flow of data [49, 75, 109, 112, 120, 121].

Mismatches between network and job structure. Parallel efficiency in multiprocessors often demands using algorithms that accommodate the structure of one's computation to that of the host multiprocessor's network. This was noticed by systems builders [71] as well as algorithms designers [93, 149]. The reader can appreciate the importance of so tuning one's algorithm by perusing the following studies of the operation of sorting: [30, 52, 52, 74, 77, 92, 125, 141, 148]. The overall groundrules in these studies are constant: one is striving to minimize the worst-case number of comparisons when sorting n numbers; only the underlying interconnection network changes. We now briefly describe two broadly applicable approaches to addressing potential mismatches with the host network.

Network emulations. The theory of network emulations focuses on the problem of making one computation-graph—the *host*—"act like" or "look like" another—the *guest*.

In both of the scenarios that motivate this endeavor, the host $\mathcal{H}$ represents an existing interconnection network. In one scenario, the guest $\mathcal{G}$ is a directed graph that represents the intertask dependencies of a computation. In the other scenario, the guest $\mathcal{G}$ is an undirected graph that represents an ideal interconnection network that would be a congenial host for one's computation. In both scenarios, computational efficiency would clearly be enhanced if $\mathcal{H}$'s interconnection structure matched $\mathcal{G}$'s—or could be made to appear to.

Almost all approaches to network emulation build on the theory of graph embeddings, which was first proposed as a general computational tool in [126]. An *embedding* $\langle \alpha, \rho \rangle$ of the graph $\mathcal{G} = (V_\mathcal{G}, E_\mathcal{G})$ into the graph $\mathcal{H} = (V_\mathcal{H}, E_\mathcal{H})$ consists of a one-to-one map $\alpha : V_\mathcal{G} \rightarrow V_\mathcal{H}$, together with a mapping of $E_\mathcal{G}$ into *paths* in $\mathcal{H}$ such that: for each edge $(u, v) \in E_\mathcal{G}$, the path $\rho(u, v)$ connects nodes $\alpha(u)$ and $\alpha(v)$ in $\mathcal{H}$. The two main measures of the quality of the embedding $\langle \alpha, \rho \rangle$ are: the *dilation*, which is the length of the longest path of $\mathcal{H}$ that is the image, under ρ, of some edge of $\mathcal{G}$; the *congestion*, which is the maximum, over all edges e of $\mathcal{H}$, of the number of ρ-paths that edge e occurs in; in other words, it is the maximum number of edges of $\mathcal{G}$ that are routed across e by the embedding.

It is easy to use an embedding of a network $\mathcal{G}$ into a network $\mathcal{H}$ to translate an algorithm designed for $\mathcal{G}$ into a computationally equivalent algorithm for $\mathcal{H}$. Basically: the mapping α identifies which node of $\mathcal{H}$ is to emulate which node of $\mathcal{G}$; the mapping ρ identifies the routes in $\mathcal{H}$ that are used to simulate internode message-passing in $\mathcal{G}$. This sketch suggests why the quantitative side of network-emulations-via-embeddings focuses on dilation and congestion as the main measures of the quality of an embedding. A moment's reflection suggests that, when one uses an embedding $\langle \alpha, \rho \rangle$ of a graph $\mathcal{G}$ into a graph $\mathcal{H}$ as the basis for an emulation of $\mathcal{G}$ by $\mathcal{H}$, any algorithm that is designed for $\mathcal{G}$ is slowed down by a factor $O(\text{congestion} \times \text{dilation})$ when run on $\mathcal{H}$. One can *sometimes* easily orchestrate communications to improve this factor to $O(\text{congestion} + \text{dilation})$; cf. [13]. Remarkably, one can *always* improve the slowdown to $O(\text{congestion} + \text{dilation})$; a nonconstructive proof of this fact appears in [94]; and, even more remarkably, a constructive proof and efficient algorithm appear in [95].

There are myriad studies of embedding-based emulations with specific guest and host graphs. An extensive literature follows up one of the earliest studies, [6], which embeds rectangular meshes into square ones, a problem having nonobvious algorithmic consequences [18]. The algorithmic attractiveness of the boolean hypercube mentioned in Section 2.1 is attested to by countless specific algorithms [93], but also by several studies that show the hypercube to be a congenial host for a wide variety of graph families that are themselves algorithmically attractive. Citing just two examples: (1) One finds in [24, 161] two quite distinct efficient embeddings of complete trees—hence, of the ramified computations they represent—into hypercubes. Surprisingly, such embeddings exist also for trees that are not complete [98, 158] and/or that grow *dynamically* [27, 96]. (2) One finds in [70] efficient embeddings of butterfly-like networks—hence, of the convolutional computations they represent—into hypercubes. A number of related algorithm-motivated embeddings into hypercubes appear in [72]. [57] embeds the mesh-of-trees network, which is shown in [93] to be an efficient host for many parallel computations, into hypercubes; [142] embeds this network into the de Bruijn network. The emulations in [11, 12] attempt to exploit the algorithmic attractiveness of the hypercube, despite its earlier-mentioned physical intractability. The study in [13], unusual for its algebraic underpinnings, was motivated by the (then-)unexplained fact—observed, e.g., in [149]—that algorithms designed for the butterfly network run equally fast on the de Bruijn network. An intimate algebraic connection discovered in [13] between these networks—the de Bruijn network is a *quotient* of the butterfly—led to an embedding of the de Bruijn network into the hypercube that had *exponentially* smaller dilation than any competitors known at that time.

The embeddings discussed thus far exploit structural properties that are peculiar to the target guest and host graphs. When such enabling properties are hard to find, a strategy pioneered in [25] can sometimes produce efficient embeddings. This source crafts efficient embeddings based on the ease of recursively decomposing a guest graph $\mathcal{G}$ into subgraphs. The insight underlying this embedding-via-decomposition strategy is that recursive bisection—the repeated decomposition of a graph into like-sized subgraphs by "cutting" edges—affords one a representation of $\mathcal{G}$ as a binary-tree-like structure.[2] The root of this structure is the graph $\mathcal{G}$; the root's two children are the two subgraphs of $\mathcal{G}$—call them $\mathcal{G}_0$ and $\mathcal{G}_1$—that the first bisection partitions $\mathcal{G}$ into. Recursively, the two children of node $\mathcal{G}_x$ of the tree-like structure (where x is a binary string) are the two subgraphs of $\mathcal{G}_x$—call them $\mathcal{G}_{x0}$ and $\mathcal{G}_{x1}$—that the bisection partitions $\mathcal{G}_x$ into. The technique of [25] transforms an (efficient) embedding of this "decomposition tree" into a host graph $\mathcal{H}$ into an (efficient) embedding of $\mathcal{G}$ into $\mathcal{H}$, whose dilation (and, often, congestion) can be bounded using a standard measure of the ease of recursively bisecting $\mathcal{G}$. A very few studies extend and/or improve the technique of [25]; see, e.g., [78, 114].

When networks $\mathcal{G}$ and $\mathcal{H}$ are incompatible—i.e., there is no efficient embedding of $\mathcal{G}$ into $\mathcal{H}$—graph embeddings cannot lead directly to efficient emulations. A technique developed in [84] can sometimes overcome this shortcoming and produce efficient network emulations. The technique has $\mathcal{H}$ emulate $\mathcal{G}$ by alternating the following two phases:

Computation phase. Use an embedding-based approach to emulate $\mathcal{G}$ piecewise for short periods of time (whose durations are determined via analysis).

Coordination phase. Periodically (frequency is determined via analysis) coordinate the piecewise embedding-based emulations to ensure that all pieces have fresh information about the state of the emulated computation.

This strategy will produce efficient emulations if one makes enough progress during the computation phase to amortize the cost of the coordination phase. Several examples in

[2]See [134] for a comprehensive treatment of the theory of graph decomposition, as well as of this embedding technique.

[84] demonstrate the value of this strategy; each presents a phased emulation of a network $\mathcal{G}$ by a network $\mathcal{H}$, that incurs only constant-factor slowdown, while any embedding-based emulation of $\mathcal{G}$ by $\mathcal{H}$ incurs slowdown that depends on the sizes of $\mathcal{G}$ and $\mathcal{H}$.

We mention one final, unique use of embedding-based emulations. In [115], a suite of embedding-based algorithms is developed, to endow a multiprocessor with a capability that would be prohibitively expensive to supply in hardware. The *gauge* of a multiprocessor is the common width of its CPU and memory bus. A multiprocessor can be *multigauged* if, under program control, it can dynamically change its (apparent) gauge. (Prior studies had determined the algorithmic value of multigauging, as well as its prohibitive expense [53, 143].) Using an embedding-based approach that is detailed in [114], the algorithms of [115] efficiently endow a multiprocessor architecture with a multigauging capability.

The use of parameterized models. A truly revolutionary approach to the problem of matching computation structure to network structure was proposed in [153], the birthplace of the *bulk-synchronous* parallel (*BSP*) programming paradigm. The central thesis in [153] is that, by appropriately reorganizing one's computation, one can obtain almost all of the benefits of message-passing parallel computation while ignoring all aspects of the underlying interconnection network's structure, save its end-to-end latency. The needed reorganization is a form of task-clustering: one organizes one's computation into a sequence of computational "supersteps"—during which processors compute locally, with no intercommunication—punctuated by communication "supersteps"—during which processors synchronize with one another (whence the term "bulk-synchronous") and perform a stylized intercommunication in which each processor sends h messages to h others. (The choice of h depends on the network's latency.) It is shown that a combination of artful message routing—say, using the congestion-avoiding technique of [154]—and latency-hiding techniques—notably, the method of *parallel slack* that has the host parallel computer emulate a computer with more processors—allows this algorithmic paradigm to achieve within a constant factor of the parallel speedup available via network-sensitive algorithm design. A number of studies, such as [69, 104], have demonstrated the viability of this approach for a variety of classes of computations.

The focus on network latency and number of processors as the sole architectural parameters that are relevant to efficient parallel computation limits the range of architectural platforms that can enjoy the full benefits of the BSP model. In response, the authors of [51] have crafted a model that carries on the spirit of BSP but that incorporates two further parameters related to interprocessor communication. The resulting *LogP* model accounts for *latency* (the "L" in "LogP), *overhead* (the "o") [the cost of setting up a communication], *gap* (the "g") [the minimum interval between successive communications by a processor], and *processor number* (the "P"). Experiments described in [51] validate the predictive value of the LogP model in multiprocessors, at least for computations involving only short interprocessor messages. The model is extended in [7], to allow long, but equal-length, messages. One finds in [29] an interesting study of the efficiency of parallel algorithms developed under the BSP and LogP models.

3 Clusters/Networks of Workstations

3.1 The Platform

Many sources eloquently argue the technological and economic inevitability of an increasingly common modality of collaborative computing—the use of a *cluster* (or, equally commonly, a *network*) of computers to cooperate in the solution of a computational problem; see [9, 119]. Note that while one typically talks about a network of *workstations* (a *NOW*, for short), the constituent computers in a NOW may well be pc's or multiprocessors; the algorithmic challenges change quantitatively but not qualitatively depending on the architectural sophistication of the "workstations." The computers in a NOW intercommunicate via a LAN—local area network—whose detailed structure is typically neither known to nor accessible by the programmer.

3.2 Some Challenges

Some of the challenges encountered when devising algorithms for (H)NOWs differ only quantitatively from those encountered with multiprocessors. For instance:

- The typically high latencies of LANs (compared to interconnection networks), coupled with the relatively heavyweight protocols needed for robust communication, demand coarse-grained tasks, in order to amortize the costs of communication.

Some new challenges arise from the ineffectiveness in NOWs of certain multiprocessor-based algorithmic strategies. For instance:

- The algorithm designer typically cannot exploit the structure of the LAN underlying a NOW.
- The higher costs of communication, coupled with the loose coordination of a NOW's workstations, render the (relatively) simple latency-hiding techniques of multiprocessors ineffective in clusters.

Finally, some algorithmic challenges arise in the world of collaborative computing for the first time in clusters. For instance:

- The constituent workstations of a NOW may differ in processor and/or memory speeds; i.e., the NOW may be *heterogeneous* (be an *HNOW*).

All of the issues raised here make parameterized models such as those discussed at the end of Section 2.2 an indispensable tool to the designers of algorithms for (H)NOWs. The challenge is to craft models that are at once faithful enough to ensure algorithmic efficiency on real NOWs and simple enough to be analytically tractable. The latter goal is particularly elusive in the presence of heterogeneity. Consequently, much of the focus in this section is on models that have been used successfully to study several approaches to computing in (H)NOWs.

3.3 Some Sophisticated Responses

Since the constituent workstations of a NOW are at best loosely coupled, and since interworkstation communication is typically rather costly in a NOW, the major strategies for using NOWs in collaborative computations center around three loosely coordinated scheduling mechanisms—workstealing, cycle-stealing, and worksharing—that, respectively, form the foci of the following three subsections.

3.3.1 Cluster computing via workstealing

Workstealing is a modality of cluster computing wherein an idle workstation seeks work from a busy one. This allocation of responsibility for finding work has the benefit that idle workstations, not busy ones, do the unproductive chore of searching for work. The most comprehensive study of workstealing is the series of papers [32]–[35], which schedule computations in a multiprocessor or in a (homogeneous) NOW. These sources develop their approach to workstealing from the level of programming abstraction through algorithm design and analysis through implementation as a working system (called Cilk [32]). As will be detailed imminently, these sources use a strict form of multithreading as a mechanism for subdividing a computation into chunks (specifically, threads of unit-time tasks) that are suitable for sharing among collaborating workstations. The strength and elegance of the results in these sources has led to a number of other noteworthy studies of multithreaded computations, including [1, 14, 59]. A very abstract study of workstealing, which allows one to assess the impact of changes in algorithmic strategy easily, appears in [110], which we describe a bit later.

A. Case study: [34] From an algorithmic perpsective, the main paper in the series about Cilk and its algorithmic underpinnings is [34], which presents and analyzes a (randomized) mechanism for scheduling "well-structured" multithreaded computations, achieving both time and space complexity that are within constant factors of optimal.

Within the model of [34], a *thread* is a collection of unit-time tasks, linearly ordered by dependencies; graph-theoretically, a thread is, thus, a linear computation-dag. A *multithreaded computation* is a set of threads that are interconnected in a stylized way. There is a *root thread.* Recursively, any task of any thread T may have $k \geq 0$ *spawn-arcs* to the initial tasks of k threads that are *children* of T. If thread T' is a child of thread T via a spawn-arc from task t of T, then the last task of T' has a *continue-arc* to some task t' of T that is a successor of task t. Both the spawn-arcs and continue-arcs individually thus give the computation the structure of a tree-dag. See Fig. 2. All of the arcs of a multithreaded

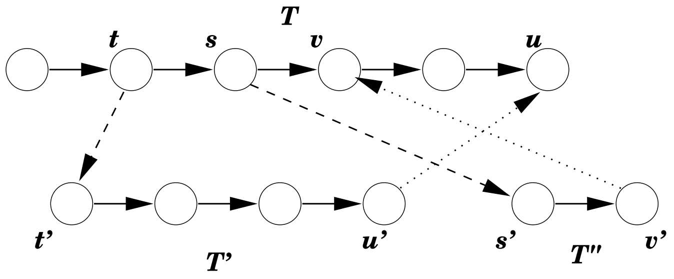

Figure 2: *An exemplary multithreaded computation. Thread T' (resp., T'') is a child of thread T, via the spawn-arc from task t to task t' (resp., from task s to task s') and the continue-arc from task u' to task u (resp., from task v' to task v).*

computation represent data dependencies that must be honored when executing the computation. A multithreaded computation is *strict* if all data-dependencies for the tasks of a thread T go to an ancestor of thread T in the thread-tree; the computation is *fully strict* if all dependencies in fact go to T's parent in the tree. Easily, any multithreaded computation can be made fully strict by altering the dependency structure; this restructuring may affect the available parallelism in the computation but will not compromise its correctness. The study in [34] focuses on scheduling fully strict multithreaded computations.

In the computing platform envisioned in [34], a multithreaded computation is stored in shared memory. Each individual thread T has a block of memory (called an *activation frame*) within the local memory of the workstation that "owns" T, that is dedicated to the computation of T's tasks. Space is measured in terms of activation frames.

Time is measured in [34] as a function of the number of workstations that are collaborating in the target computation. T_p is the minimum computation time when there are p collaborating workstations; therefore, T_1 is the total amount of work in the computation. T_∞ is *dag-depth* of the computation, i.e., the length of the longest source-to-sink path in the associated computation-dag; this is the "inherently sequential" part of the computation. Analogously, S_p is the minimum space requirement for the target computation, S_1 being the "activation depth" of the computation.

Within the preceding model, the main contribution of [34] is a provably efficient randomized workstealing algorithm, **Procedure** Worksteal (see Fig. 3), which executes the fully strict multithreaded computation rooted at thread T. In the Procedure, each work-

Normal execution. A workstation P seeking work removes (pops) the thread at the bottom of its ready deque—call it thread T—and begins executing T's tasks seriatim.

A stalled thread is enabled. If executing one of T's tasks enables a stalled thread T', then the now-ready thread T' is pushed onto the bottom of P's ready deque. (A thread *stalls* when the next task to be executed must await data from a task that belongs to another thread.)

/*Because of full strictness: thread T' must be thread T's parent; thread T's deque must be empty when T' is inserted.*/

A new thread is spawned. If the task of thread T that is currently being executed spawns a child thread T', then thread T is pushed onto the bottom of P's ready deque, and P begins to work on thread T'.

A thread completes or stalls. If thread T completes or stalls, then P checks its ready deque.

Nonempty ready deque. If its deque is not empty, then P pops the bottommost thread and starts working on it.

Empty ready deque. If its deque is empty, then P initiates workstealing. It chooses a workstation P' uniformly at random, "steals" the topmost thread in P''s ready deque, and starts working on that thread. If P''s ready deque is empty, then P chooses another random "victim."

Figure 3: **Procedure** Worksteal*(T) executes the multithreaded computation rooted at thread T*

station maintains a *ready deque* of threads that are eligible for execution; these deques are accessible by all workstations. Each deque has a *bottom* and a *top*; threads can be inserted at the bottom and removed from either end. A workstation uses its ready deque as a procedure *stack*, pushing and popping from the bottom. Threads that are "stolen" by other workstations are removed from the top of the deque. It is shown in [34] that **Procedure** Worksteal is close to optimal in both time and space complexity.

- *For any fully strict multithreaded computation,* **Procedure** Worksteal*, when run on a p-workstation NOW, uses space* $\leq S_1 p$.

- *Let* **Procedure** Worksteal *execute a multithreaded computation on a p-workstation NOW. If the computation has dag-depth* T_∞ *and work* T_1*, then the expected running time, including scheduling overhead, is* $O(T_1/p + T_\infty)$*. This is clearly within a constant factor of optimal.*

B. Case study: [110] The study in [34] follows the traditional algorithmic paradigm. An algorithm is described in complete detail, down to the design of its underlying data structures. The performance/behavior of the algorithm is then analyzed in a setting appropriate to the genre of the algorithm. For instance, since **Procedure** Worksteal is a randomized algorithm, its performance is analyzed in [34] under the assumption that its input multithreaded computation is selected uniformly at random from the ensemble of such computations. In contrast to the preceding approach, the study in [110] describes an algorithm abstractly, via its state space and state-transition function. The performance/behavior of the algorithm is then analyzed by positing a process for generating the inputs that trigger state changes. We illustrate this change of worldview by describing **Procedure** Worksteal and its analysis in the framework of [110] in some detail. We then briefly summarize some of the other notable results in that source.

In the setting of [110], when a computer (such as a homogeneous NOW) is used as a *workstealing system*, its workstations execute tasks that are generated dynamically via a Poisson process of rate $\lambda < 1$. Tasks require computation time that is distributed exponentially with mean 1; these times are not known to workstations. Tasks are scheduled in a First-Come-First-Served fashion, with tasks awaiting execution residing in a FIFO queue. The *load* of a workstation P at time t is the number of tasks in P's queue at that time. At certain times (characterized by the algorithm being analyzed), a workstation P' can steal a task from another workstation P. When that happens, a task at the output end of P's queue (if there is one) *instantaneously* migrates to the input end of P''s queue. Formally, a workstealing system is represented by a sequence of variables that yield snapshots of the state of the system as a function of the time t. Say that the NOW being analyzed has n constituent workstations.

- $n_l(t)$ is the number of workstations that have load l.
- $m_l(t) \overset{\text{def}}{=} \sum_{i=0}^{l} n_i(t)$ is the number of workstations that have load $\geq l$.
- $p_l(t) \overset{\text{def}}{=} n_l(t)/n$ is the fraction of workstations of load l.
- $s_l(t) \overset{\text{def}}{=} \sum_{i=l}^{\infty} p_i(t) = m_l(t)/n$ is the fraction of workstations of load $\geq l$.

The *state* of a workstealing system at time t is the infinite-dimensional vector $\vec{s}(t) \overset{\text{def}}{=} \langle s_0(t), s_1(t), s_2(t), \ldots \rangle$.

The goal in [110] is to analyze the limiting behavior, as $n \to \infty$, of n-workstation workstealing systems under a variety of randomized workstealing algorithms. The mathematical tools that characterize the study are enabled by two features of the model we have described thus far. (1) Under the assumption of Poisson arrivals and exponential service times, the entire workstealing system is *Markovian:* its next state, $\vec{s}(t+1)$, depends only on its present state, $\vec{s}(t)$, not on any earlier history. (2) The fact that a workstealing system changes state instantaneously allows one to view time as a *continuous* variable, thereby enabling the use of differentials rather than differences when analyzing changes in the variables that characterize a system's state.

We enhance legibility henceforth by omitting the time variable t when it is clear from context. Note that $s_0 \equiv 1$ and that the s_l are nonincreasing, since $s_{l-1} - s_l = p_l$. The systems analyzed in [110] also have $\lim_{l\to\infty} s_l = 0$.

We introduce the general process of characterizing a system's (limiting) performance by focusing momentarily on a system in which no workstealing takes place. Let us represent by $\mathrm{d}t$ a small interval of time, in which only one event (a task arrival or departure) takes place at a workstation. The model of task arrivals (via a Poisson process with rate λ) means that the expected change in the variable m_l due to task arrivals is $\lambda(m_{l-1} - m_l)\mathrm{d}t$. By similar reasoning, the expected change in m_l due to task departures—recall that there is no stealing going on—is just $(m_l - m_{l+1})\mathrm{d}t$. It follows that the expected net behavior of the system over short intervals is:

$$\frac{\mathrm{d}m_l}{\mathrm{d}t} = \lambda(m_{l-1} - m_l) - (m_l - m_{l+1}),$$

or, equivalently, (by eliminating the ubiquitous factor of n, the size of the NOW),

$$\frac{\mathrm{d}s_l}{\mathrm{d}t} = \lambda(s_{l-1} - s_l) - (s_l - s_{l+1}). \tag{3.1}$$

This last characterization of state changes illustrates the changes' independence from the aggregate number of workstations, depending instead only on the densities of workstations with various loads. The technical implications of this fact is discussed in some details in [110], with appropriate pointers to the underlying mathematical texts.

In order to analyze the performance of **Procedure** Worksteal within the current model, one must consider how the Procedure's various actions are perceived by the workstations of the subject workstealing system. First, under the Procedure, a workstation P that completes its last task seeks to steal a task from a randomly chosen fellow workstation, P', succeeding with probability s_2 (the probability that P' has at least two tasks). Hence, P now perceives completion of its final task as emptying its queue only with probability $1 - s_2$. Mathematically, we thus have the following modified first equation of system (3.1):

$$\frac{\mathrm{d}s_1}{\mathrm{d}t} = \lambda(s_0 - s_1) - (s_1 - s_2)(1 - s_2). \tag{3.2}$$

For $l > 1$, s_l now decreases whenever a workstation with load l *either* completes a task *or* has a task stolen from it. The rate at which workstations steal tasks is just $s_1 - s_2$, i.e., the rate at which workstation complete their final tasks. We thus complete our modification of system (3.1) as follows.

$$\text{For } \ l > 1, \qquad \frac{\mathrm{d}s_l}{\mathrm{d}t} = \lambda(s_{l-1} - s_l) - (s_l - s_{l+1})(1 + s_1 - s_2). \tag{3.3}$$

The limiting behavior of the workstealing system is characterized by seeking the *fixed point* of system (3.2, 3.3), i.e., the state $\vec{s}$ for which every $\mathrm{d}s_l/\mathrm{d}t = 0$.

Denoting the sought fixed point by $\vec{\pi} = \langle \pi_0, \pi_1, \pi_2, \ldots \rangle$, we have:

- $\pi_0 = 1$, because $s_0 = 1$ for all t;
- $\pi_1 = \lambda$, because:
  - tasks complete at rate $s_1 n$, the number of busy workstations;
  - tasks arrive at rate λn;
  - at the fixed point, tasks arrive and complete at the same rate;
- from (3.2) and the fact that $\mathrm{d}s_1/\mathrm{d}t = 0$ at the fixed point, we have
$$\pi_2 \ = \ \frac{1 + \lambda - \sqrt{1 + 2\lambda - 3\lambda^2}}{2};$$
- from (3.3) and the fact that $\mathrm{d}s_l/\mathrm{d}t = 0$ at the fixed point, we have, by induction,
$$\text{For } \ l > 2, \qquad \pi_l \ = \ \left(\frac{\lambda}{1 + \lambda - \pi_2}\right)^{l-2} \pi_2.$$

The message of the preceding analysis becomes clear only when one performs the same exercise with the system (3.1), which characterizes a "workstealing system" in which there is no workstealing. For that system, one finds that $\pi_l = \lambda^l$, indicating that, in the limiting state, tasks are being completed at rate λ. Under the workstealing regimen of **Procedure** Worksteal, we still have the π_l, for $l > 2$, decreasing geometrically, but now the damping ratio is $\dfrac{\lambda}{1 + \lambda - \pi_2} < \lambda$. In other words, workstealing under the Procedure has the same effect as increasing the service rate of tasks in the workstealing system!

Simulation experiments in [110] help one evaluate the paper's abstract treatment. The experiments indicate that, even with $n = 128$ workstations, the model's predictions are quite accurate, at least for smaller arrival rates. Moreover, the quality of these predictions improve with larger n and smaller arrival rates.

The study in [110] goes on to consider several variations on the basic theme of workstealing, including precluding: • stealing work from workstations whose queues are almost empty; • stealing work when load gets below a (positive) threshold. Additionally, one finds in [110] refined analyses and more complex models for workstealing systems.

3.3.2 Cluster computing via cycle-stealing

Cycle-stealing, the use by one workstation of idle computing cycles of another, views the world through the other end of the computing telescope from workstealing. The basic observation that motivates cycle-stealing is that the workstations in clusters tend to be idle much of the time—due, say, to a user's pausing for deliberation or for a telephone call, etc.—and that the resulting idle cycles can fruitfully be "stolen" by busy workstations [100, 145]. Although cycle-stealing ostensibly puts the burden of finding available computing cycles on the busy workstations (the criticisms leveled against cycle-stealing by advocates of workstealing), the just-cited sources indicate that this burden can often be offloaded onto a central resource, or at least onto a workstation's operating system (rather than its application program).

The literature contains relatively few rigorously analyzed scheduling algorithms for cycle-stealing in (H)NOWs. Among the few such studies, [16] and the series [26, 128, 129, 131] view cycle-stealing as an *adversarial* enterprise, in which the cycle-stealer attempts to accomplish as much work as possible on the "borrowed" workstation before its owner returns—which event results in the cycle-stealer's job being killed!

A. Case study: [16] One finds in this source a randomized cycle-stealing strategy which, with high probability, accomplishes within a logarithmic factor of optimal work production. The underlying formal setting is as follows.

- All of the n workstations that are candidates as cycle donors are equally powerful computationally; i.e., the subject NOW is homogeneous.
- The cycle-stealer has a job that requires d steps of computation an any of these candidate donors.
- At least one of the candidate donors will be idle for a period of $D \geq 3d \log n$ time units (= steps).

Within this setting, the following simple randomized strategy provably steals cycles successfully, with high probability.

Phase 1. At each step, the cycle-stealer checks the availability of all n workstations in turn: first P_1, then P_2, and so on.

Phase 2. If, when checking workstation P_i, the cycle-stealer finds that it was idle at the last time unit, s/he flips a coin and assigns the job to P_i with probability $(1/d)n^{3x/D-2}$, where x is the number of time units for which P_i has been idle.

The provable success of this strategy is expressed as follows.

- *With probability* $\geq 1 - O((d \log n)/D + 1/n)$*, the preceding randomized strategy will allow the cycle-stealer to get his/her job done.*

It is claimed in [16] that same basic strategy will actually allow the cycle-stealer to get $\log n$ d-step jobs done with the same probability.

B. Case study: [131] In [26, 128, 129, 131], cycle-stealing is viewed as a game against a malicious adversary who seeks to interrupt the borrowed workstation in order to kill all work in progress and thereby minimize the work amount of produced during a cycle-stealing opportunity. (In these studies, cycles are stolen from one workstation at a time, so the enterprise is unaffected by the presence or absence of heterogeneity.) Clearly, cycle-stealing within the described adversarial model can accomplish productive work only if the metaphorical "malicious adversary" is somehow restrained from just interrupting every period when the cycle-donor is doing work for the cycle-stealer, thereby killing all work done by the donor. The restraint studied in the *Known-Risk* model of [26, 128, 131] resides in two assumptions: (1) we know the instantaneous probability that the cycle-donor has *not* been reclaimed by its owner; (2) the *life function* $\mathcal{P}$ that exposes this probabilistic information—$\mathcal{P}(t)$ is the probability that the donor has not been reclaimed by its owner by time t—is "smooth." The formal setting is as follows.

- The cycle-stealer, A, has a large bag of mutually independent tasks of equal *sizes* (which measures the cost of describing each task) and *complexities* (which measures the cost of computing each task).
- Each pair of communications—in which A sends work to the donor, B, and B returns the results of that work to A—incurs a fixed cost c. This cost is kept independent of the marginal per-task cost of communicating between A and B by incorporating the latter cost into the time for computing a task.
- B is dedicated to A's work during the cycle-stealing opportunity, so its computation time is known exactly.
- Time is measured in work-units (rather than wall-clock time); one *unit of work* is the time it takes for:
  - workstation A to transmit a single task to workstation B. (This is the marginal transmission time for the task: the (fixed) setup time for each communication—during which many tasks will typically be transmitted—is accounted for by the parameter c.)

– workstation B to execute that task;

– workstation B to return its results for that task to workstation A.

Within this setting, a cycle-stealing opportunity is a sequence of *episodes* during which workstation A has access to workstation B, punctuated by *interrupts* caused by the return of B's owner. When scheduling an opportunity, the vulnerability of A to interrupts, with their attendant loss of work in progress on B, is decreased by partitioning each episode into *periods*, each beginning with A sending work to B and ending either with an interrupt or with B returning the results of that work. A's discretionary power thus resides solely in deciding how much work to send in each period, so an *(episode-)schedule* is simply a sequence of positive period-lengths: $\mathcal{S} = t_0, t_1, \ldots$. A length-$t$ period in an episode accomplishes $t \ominus c \stackrel{\text{def}}{=} \max(0,\ t - c)$ units of work if it is not interrupted and 0 units of work if it is interrupted. Thus, the episode scheduled by $\mathcal{S}$ accomplishes $\sum_{i=1}^{k-1}(t_i \ominus c)$ units of work when it is interrupted during period k.

Focus on a cycle-stealing episode whose lifespan ($\stackrel{\text{def}}{=}$ its maximum possible duration) is L time units. As noted earlier, we are assuming that we know the risk of B's being reclaimed, via a decreasing *life function*,

$$\mathcal{P}(t) \ \stackrel{\text{def}}{=}\ Pr(B \text{ has not been interrupted by time } t),$$

which satisfies: • $\mathcal{P}(0) = 1$ (to indicate B's availability at the start of the episode); • $\mathcal{P}(L) = 0$ (to indicate that the interrupt will have occurred by time L). The earlier assertion that life functions must be "smooth" is embodied in the formal requirement that $\mathcal{P}$ be *differentiable* in the interval $(0, L)$. The goal is to maximize the *expected work production* from an episode governed by the life function $\mathcal{P}$, i.e., to find a schedule $\mathcal{S}$ whose expected work production,

$$\text{Exp-Work}(\mathcal{S}; \mathcal{P}) \ \stackrel{\text{def}}{=}\ \sum_{i=0}^{L}(t_i \ominus c)\mathcal{P}(T_i), \tag{3.4}$$

is maximum, over all schedules for $\mathcal{P}$. In summation (3.4): each T_i is the partial sum

$$T_i \ \stackrel{\text{def}}{=}\ t_0 + t_2 + \cdots + t_i.$$

The presence of positive subtraction, $\ominus$, in (3.4) makes analyses of life functions difficult technically. Fortunately, one can avoid this difficulty for all but the last term of the summation. Say that a schedule is *productive* if each period—save possibly the last—has length $> c$. The following is proved in [26] and, in the following strict form, in [128].

- *One can effectively*[3] *replace any schedule $\mathcal{S}$ for life function $\mathcal{P}$ by a productive schedule $\widehat{\mathcal{S}}$ such that* $\text{EXP-WORK}(\widehat{\mathcal{S}}; \mathcal{P}) \geq \text{EXP-WORK}(\mathcal{S}; \mathcal{P})$.

One finds in [131] a proof that the following characterization of optimal schedules allows one to compute such schedules effectively.

- *The productive schedule $\mathcal{S} = t_0, t_1, \ldots, t_{m-1}$ is optimal for the differentiable life function $\mathcal{P}$ if, and only if, for each period-index $k \geq 0$, save the last, period-length t_k is given by*[4]

$$\mathcal{P}(T_k) \ = \ \max\left(0, \ \mathcal{P}(T_{k-1}) + (t_{k-1} - c)\mathcal{P}'(T_{k-1})\right). \tag{3.5}$$

Since the explicit computation of schedules from system (3.5) can be computationally inefficient, relying on general function optimization techniques, the following simplifying initial conditions are presented in [131] for certain simple life functions.

- *When $\mathcal{P}$ is convex (resp., concave),*[5] *the initial period-length t_0 is bounded above and below as follows, with the parameter $\psi = 1$ (resp., $\psi = 1/2$).*

$$\sqrt{\frac{c^2}{4} - \frac{c\mathcal{P}(t_0)}{\mathcal{P}'(t_0)}} + \frac{c}{2} \ \leq \ t_0 \ \leq \ 2\sqrt{\frac{c^2}{4} - \frac{c\mathcal{P}(t_0)}{\mathcal{P}'(\psi t_0)}} + c.$$

3.3.3 Cluster computing via worksharing

Whereas workstealing and cycle-stealing involve a transaction between two workstations in an (H)NOW, *worksharing* typically involves many workstations working cooperatively. The qualifier "cooperatively" distinguishes the enterprise of worksharing from the passive cooperation of the work donor in workstealing and the grudging cooperation of the cycle donor in cycle-stealing.

In this section, we describe three studies of worksharing: the study in [2], one of four problems studied in [20], and the most general HNOW model of [17]. (We deal with these sources in the indicated order to emphasize relevant similarities and differences.) These sources differ markedly in their models of the HNOW in which worksharing occurs, the characteristics of the work that is being shared, and the way in which worksharing is orchestrated. Indeed, part of our motivation in highlighting these three studies is to illustrate how apparently minor changes in model—of the computing platform or the workload—can lead to major changes in the algorithmics required to solve the worksharing problem

[3]The qualifier "effectively" means that the proof is constructive.

[4]As usual, f' denotes the first derivative of the univariate function f.

[5]The life function $\mathcal{P}$ is *concave* (resp., *convex*) if its derivative $\mathcal{P}'$: • never vanishes at a point x where $\mathcal{P}(x) > 0$; • is everywhere nonincreasing (resp., everywhere nondecreasing).

(nearly) optimally. (Since the model of [20] is described at a high level in that paper, we have speculatively interpreted the architectural antecedents of the model's features for the purposes of enabling the comparison in this section.)

All three of these studies focus on some variant of the following scenario. A master workstation P_0 has a large bag of mutually independent tasks of equal sizes and complexities. P_0 has the opportunity to employ the computing power of an HNOW $\mathcal{N}$ comprising workstations $P_1, P_2, \ldots, P_n$. P_0 transmits work to each of $\mathcal{N}$'s workstations, and each workstation (eventually) sends results back to P_0. Throughout the worksharing process, $\mathcal{N}$'s workstations are dedicated to P_0's workload. Some of the major differences among the models of the three sources are highlighted in Table 1. The "N/A" ("Not Applicable") entries in the table reflect the fact that only short messages (single tasks) are transmitted in [17]. The goal of all three sources is to allocate and schedule work optimally, within the

Model Feature	[2]	[20]	[17]
Does each communication incur a substantial "setup" overhead?	Yes	No	No
Is complex message (un)packaging allowed/accounted for?	Yes	No	N/A
Can a workstation send and receive messages simultaneously?	No	No	Yes
Is the HNOW $\mathcal{N}$'s network pipelineable? (A "Yes" allows savings by transmitting several tasks or results at a time, with only one "setup.")	Yes	Yes	N/A
Does P_0 allocate multiple tasks at a time?	Yes	Yes	No
Are $\mathcal{N}$'s workstations allowed to redistribute tasks?	No	No	Yes
Are tasks "partitionable?" (A "Yes" allows the allocation of fractional tasks.)	Yes	No	No

Table 1: *Comparing the models of [2], [20], and [17].*

context of the following problems.

The HNOW-Utilization Problem. *P_0 seeks to reach a "steady-state", in which the average amount of work accomplished per time unit is maximized.*

The HNOW-Exploitation Problem. *P_0 has access to $\mathcal{N}$ for a prespecified fixed period of time (the* lifespan*) and seeks to accomplish as much work as possible during this period.*

The HNOW-Rental Problem. *P_0 seeks to complete a prespecified fixed amount of work on $\mathcal{N}$ during as short a period as possible.*

The study in [17] concentrates on the HNOW-Utilization Problem. Those of [2, 20] concentrate on the HNOW-Exploitation Problem; this concentration is just for expository convenience, since the HNOW-Exploitation and -Rental Problems are computationally equivalent within the models of [2, 20]; i.e., an optimal solution to either can be converted to an optimal solution to the other.

A. Case study: [2] This study employs a rather detailed architectural model for the HNOW $\mathcal{N}$, the HiHCoHP model of [41], which characterizes each workstation P_i of $\mathcal{N}$ via the parameters in Table 2. A word about message packaging and unpackaging is in order.

Computation-related parameters for $\mathcal{N}$'s workstations	
Computation	Each P_i needs ρ_i <u>*work units*</u> to compute a task. By convention: $\rho_1 \leq \rho_2 \leq \cdots \leq \rho_n \equiv 1$.
Message-(un)packaging	Each P_i needs: $\pi_i \overset{\text{def}}{=} \rho_i \pi_n$ time units per packet to package a message for transmission (e.g., break into packets, compute checksums, encode); $\overline{\pi}_i \overset{\text{def}}{=} \rho_i \overline{\pi}_n$ time units per packet to unpackage a received message.
Communication-related parameters for $\mathcal{N}$'s network	
Communication setup	Two workstations require σ time units to set up a communication (say, via a handshake).
Network latency	The first packet of a message traverses $\mathcal{N}$'s network in λ time units.
Network transit time	Subsequent packets traverse $\mathcal{N}$'s network in τ time units.

Table 2: *A summary of the HiHCoHP model.*

- In many actual HNOW architectures, the packaging (π) and unpackaging ($\overline{\pi}$) rates are (roughly) equal. One would lose little accuracy, then, by equating them.
- Since (un)packaging a message requires a fixed, known computation, the (common) ratio ρ_i/π_i is a measure of the granularity of the tasks in the workload.
- When message encoding/decoding is not needed (e.g., in an HNOW of trusted workstations), message (un)packaging is likely a lightweight operation; when encoding/decoding is needed, the time for message (un)packaging can be significant.

In summary, within the HiHCoHP model, a p-packet message from workstation P_i to workstation P_j takes an aggregate of $(\sigma + \lambda - \tau) + (\pi_i + \overline{\pi}_j + \tau)p$ time units.

The computational protocols considered in [2] for solving the HNOW-Exploitation Problem build on single paired interactions between P_0 and each workstation P_i of $\mathcal{N}$: P_0 sends work to P_i; P_i does the work; P_i sends results to P_0. The total interaction between P_0 the single workstation P_i is orchestrated as shown in Fig. 4. This interaction is extrapo-

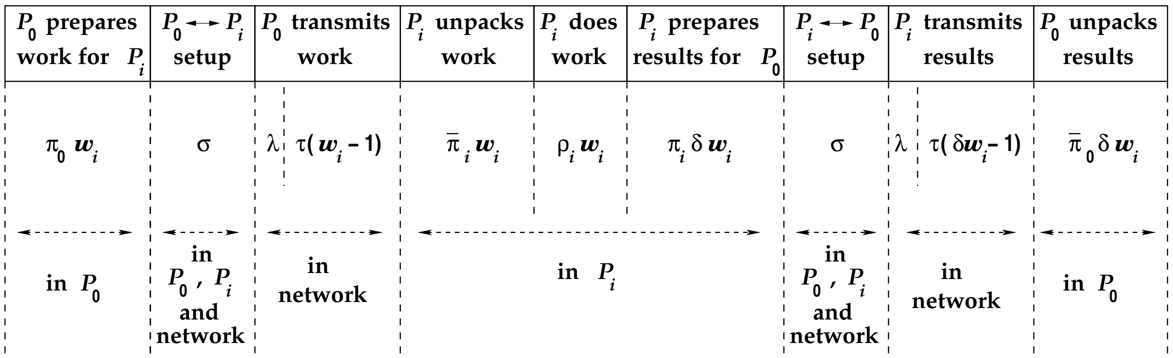

Figure 4: *The timeline for P_0's use of a single "rented" workstation P_i (not to scale)*

lated into a full-blown worksharing protocol via a pair of ordinal-indexing schemes for $\mathcal{N}$'s workstations, to supplement the model's power-related indexing described in the "Computation" entry of Table 2. The *startup indexing* specifies the order in which P_0 transmits work to $\mathcal{N}$'s workstations; for this purpose, we label the workstations $P_{s_1}, P_{s_2}, \ldots, P_{s_n}$, to indicate that P_{s_i} receives work—hence, begins working—before $P_{s_{i+1}}$ does. The *finishing indexing* specifies the order in which $\mathcal{N}$'s workstations return their work-results to P_0; for this purpose, we label the workstations $P_{f_1}, P_{f_2}, \ldots, P_{f_n}$, to indicate that P_{f_i} ceases working—hence, transmits its results—before $P_{f_{i+1}}$ does. Fig. 5 depicts a multi-workstation protocol. If we let w_i denote the amount of work allocated to workstation P_i, for $i = 1, 2, \ldots, n$, then the goal is to find a protocol (of the type described) that maximizes the overall work production, $W = w_1 + w_2 + \cdots + w_n$.

Abbrev.	Quantity	Meaning
$\widetilde{\tau}$	$\tau(1+\delta)$	Two-way transmission rate
$\widetilde{\pi}_i$	$\overline{\pi}_i + \pi_i\delta$	Two-way message-packaging rate for P_i
FC	$(\sigma + \lambda - \tau)$	*Fixed* overhead for an interworkstation communication
VC_i	$\pi_0 + \widetilde{\tau} + \widetilde{\pi}_i$	*Variable* communication overhead *rate* for P_i

Table 3: *Some useful abbreviations*

Importantly, *when one allows work-allocations to be fractional*, the work production of a protocol of the form we have been discussing can be specified in a computationally tractable, perspicuous way. Enhancing legibility via the abbreviations of Table 3, the

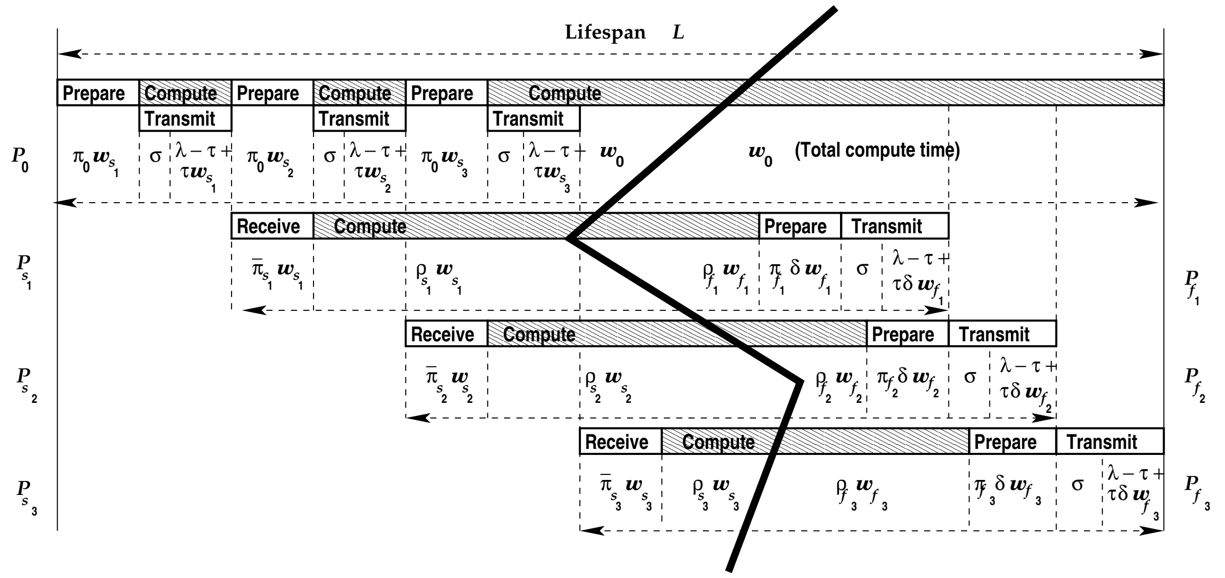

Figure 5: *The timeline (*not to scale*) for* 3 *"rented" workstations, indicating each workstation's lifespan. Note that each P_i's lifespan is partitioned in the figure between its incarnations as some P_{s_a} and some P_{f_b}.*

work production of the protocol $\mathcal{P}(\Sigma, \Phi)$ that is specified by the startup indexing $\Sigma = \langle s_1, s_2, \ldots, s_n\rangle$ and finishing indexing $\Phi = \langle f_1, f_2, \ldots, f_n\rangle$ over a lifespan of duration L is given by the following system of linear equations.

$$\begin{pmatrix} \mathsf{VC}_1+\rho_1 & B_{1,2} & \cdots & B_{1,n} \\ B_{2,1} & \mathsf{VC}_2+\rho_2 & \cdots & B_{2,n} \\ \vdots & \vdots & \cdots & \vdots \\ B_{n-1,1} & B_{n-1,2} & \cdots & B_{n-1,n} \\ B_{n,1} & B_{n,2} & \cdots & \mathsf{VC}_n+\rho_n \end{pmatrix} \cdot \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_{n-1} \\ w_n \end{pmatrix} = \begin{pmatrix} L-(c_1+2)\mathsf{FC} \\ L-(c_2+2)\mathsf{FC} \\ \vdots \\ L-(c_{n-1}+2)\mathsf{FC} \\ L-(c_n+2)\mathsf{FC} \end{pmatrix}, \quad (3.6)$$

where

- SB_i is the set of startup indices of workstations that *start before* P_i;
- FA_i is the set of finishing indices of workstations that *finish after* P_i;
- $c_i \stackrel{\text{def}}{=} |\mathsf{SB}_i| + |\mathsf{FA}_i|$;
- $B_{i,j} = \begin{cases} \pi_0+\tau+\tau\delta & \text{if } j \in \mathsf{SB}_i \text{ and } j \in \mathsf{FA}_i \\ \pi_0+\tau & \text{if } j \in \mathsf{SB}_i \text{ and } j \notin \mathsf{FA}_i \\ \tau\delta & \text{if } j \notin \mathsf{SB}_i \text{ and } j \in \mathsf{FA}_i \\ 0 & \text{otherwise.} \end{cases}$

The nonsingularity of the coefficient matrix in (3.6) indicates that the work production of protocol $\mathcal{P}(\Sigma, \Phi)$ is, indeed, specified completely by the indexings Σ and Φ.

Of particular significance in [2] are the *FIFO* worksharing protocols, which are defined by the relation $\Sigma = \Phi$. For such protocols, system (3.6) simplifies to:

$$\begin{pmatrix} \mathsf{VC}_{s_1} + \rho_{s_1} & \tau\delta & \cdots & \tau\delta \\ \pi_0 + \tau & \mathsf{VC}_{s_2} + \rho_{s_2} & \cdots & \tau\delta \\ \vdots & \vdots & \cdots & \vdots \\ \pi_0 + \tau & \pi_0 + \tau & \cdots & \tau\delta \\ \pi_0 + \tau & \pi_0 + \tau & \cdots & \mathsf{VC}_{s_n} + \rho_{s_n} \end{pmatrix} \cdot \begin{pmatrix} w_{s_1} \\ w_{s_2} \\ \vdots \\ w_{s_{n-1}} \\ w_{s_n} \end{pmatrix} = \begin{pmatrix} L - (n+1)\mathsf{FC} \\ L - (n+1)\mathsf{FC} \\ \vdots \\ L - (n+1)\mathsf{FC} \\ L - (n+1)\mathsf{FC} \end{pmatrix} \quad (3.7)$$

It is proved in [2] that, surprisingly:

- *All FIFO protocols produce the same amount of work in L time units, no matter what their startup indexing. This work production is obtained by solving system (3.7).*

FIFO protocols solve the HNOW-Exploitation Problem asymptotically optimally [2]:

- *For all sufficiently long lifespans L, a FIFO protocol produces at least as much work in L time units as any protocol $\mathcal{P}(\Sigma, \Phi)$.*

It is worth noting that having to schedule the transmission of results, in addition to inputs, is the source of much of the complication encountered in proving the preceding result.

B. Case study: [20] As noted earlier, the communication model in [20] is specified at a high level of abstraction. In an effort to compare that model with the HiHCoHP model, we have cast the former model within the framework of the latter, in a way that is consistent with the algorithmic setting and results of [20]. One largely cosmetic difference between the two models is that all speeds are measured in absolute (wall-clock) units in [20], in contrast to the relative work units in [2]. More substantively, the communication model of [20] can be obtained from the HiHCoHP model via the following simplifications.

- There is no cost assessed for setting up a communication (the HiHCoHP cost σ). Importantly, the absence of this cost removes any disincentive to replacing a single long message by a sequence of shorter ones.
- Certain costs in the HiHCoHP model are deemed negligible, hence, ignorable:
  - the per-packet transit rate (τ) in a pipelined network,
  - the per-packet packaging (the π_i) and unpackaging (the $\overline{\pi}_i$) costs.

These assumptions implicitly assert that the tasks in one's bag are very coarse, especially if message-(un)packaging includes en/decoding.

These simplifications imply that, within the model of [20]:

- The heterogeneity of the HNOW $\mathcal{N}$ is manifest only in the differing computation rates of $\mathcal{N}$'s workstations.
- In a pipelined network, the distribution of work to and the collection of results from each of $\mathcal{N}$'s workstation take fixed constant time. Specifically, P_0 sends work at a cost of $t_{\text{com}}^{(\text{work})}$ time units *per transmission* and receives results at a cost of $t_{\text{com}}^{(\text{results})}$ time units *per transmission*.

Within this model, [20] derives efficient optimal or near-optimal schedules for the four variants of the HNOW-Exploitation Problem that correspond to the four paired answers to the questions: "Do tasks produce nontrivial-size results?" "Is $\mathcal{N}$'s network pipelined?" For those variants that are NP-Hard, *near*-optimality is the most that one can expect to achieve efficiently—and this is what [20] achieves.

The *Pipelined* HNOW-Exploitation Problem—which is the only version we discuss—is formulated in [20] as an *integer* optimization problem. (Tasks are atomic, in contrast to [2].) One allocates an integral number—call it a_i—of tasks to each workstation P_i via a protocol that has the essential structure depicted in Fig. 5, altered to accommodate the simplified communication model. One then solves the following optimization problem.

Find:	A startup indexing:	$\Sigma = \langle s_1,\ s_2, \ldots,\ s_n \rangle$
	A finishing indexing:	$\Phi = \langle f_1,\ f_2, \ldots,\ f_n \rangle$
	An allocation of tasks:	Each P_i gets a_i tasks

That maximizes: $\displaystyle\sum_{i=1}^{n} a_i$ (the number of tasks computed)

Subject to the constraint: All work gets done within the lifespan; formally:

$$(\forall 1 \le i \le n)\ [s_i \cdot t_{\text{com}}^{(\text{work})} \ + \ a_i \cdot t_i \ + \ f_i \cdot t_{\text{com}}^{(\text{results})} \ \le \ L] \tag{3.8}$$

Not surprisingly, the (decision version of the) preceding optimization problem is NP-Complete, hence, likely computationally intractable. This fact is proved in [20] via reduction from a variant of the Numerical 3-D Matching Problem. Stated formally:

- *Finding an optimal solution to the HNOW-Exploitation Problem within the model of [20] is NP-complete in the strong sense*[6].

[6]The strong form of NP-completeness measures the sizes of integers by their magnitudes rather than the lengths of their numerals.

Those familiar with discrete optimization problems would tend to expect a Hardness result here because this formulation of the HNOW-Exploitation Problem requires finding a maximum "paired-matching" in an edge-weighted version of the tripartite graph depicted in Fig. 6: A "paired-matching" is one that uses both of the permutations Σ and Φ in

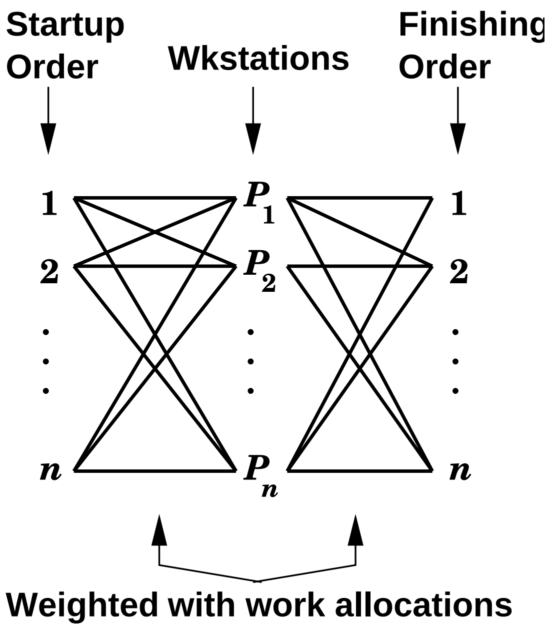

Figure 6: *An abstraction of the HNOW-Exploitation Problem within the model of [20].*

a coordinated fashion in order to determine the a_i. The matching gives us the startup and finishing orders of $\mathcal{N}$'s workstations. Specifically, the edge connecting the lefthand instance of node i with node P_j (resp., the edge connecting the righthand instance of node k with node P_j) is in the matching when $s_j = i$ (resp., $f_j = k$). In order to ensure that an optimal solution to the HNOW-Exploitation Problem is within our search space, we have to accommodate the possibility that $s_j = i$ and $f_j = k$, for every distinct triple of integers, $i, j, k \in \{1, 2, \ldots, n\}$. In order to ensure that a maximum matching in the graph of Fig. 6 yields this optimal solution, we weight the edges of the graph in accordance with constraint (3.8), which contains both s_i and f_i. If we let $\omega(u, v)$ denote the weight on the edge from node u to node v in the graph, then, for each $1 \le i \le n$, the optimal weighting must end up with

$$\omega(s_i, P_i) + \omega(P_i, f_i) \ = \ \left\lfloor \frac{L - s_i \cdot t_{\text{com}}^{(\text{work})} - f_i \cdot t_{\text{com}}^{(\text{results})}}{t_i} \right\rfloor .$$

While the desired weighting would lead to an optimal solution, it also leads to NP-Hardness. We avoid this complexity by relinquishing our demand for an *optimal* solution. A simple approach to ensuring reasonable complexity is to decouple the matchings derived for the lefthand and righthand sides of the graph of Fig. 6, which is tantamount to ignoring the interactions between Σ and Φ when seeking work-allocations. We achieve the desired decoupling via the following edge-weighting

$$\omega(i, P_j) \ = \ \left\lfloor \frac{L/2 - i \cdot t_{\text{com}}^{(\text{work})}}{t_j} \right\rfloor \qquad \text{and} \qquad \omega(P_j, k) \ = \ \left\lfloor \frac{L/2 - k \cdot t_{\text{com}}^{(\text{results})}}{t_j} \right\rfloor .$$

We then find independent lefthand and righthand maximum matchings, each within time $O(n^{5/2})$. It is shown in [20] that the solution produced by this decoupled matching problem deviates from the true optimal solution by only an additive discrepancy of $\leq n$.

- *There is an $O(n^{5/2})$-time work-allocation algorithm whose solution (within the model of [20]) to the HNOW-Exploitation Problem in an n-workstation HNOW is (additively) within n of optimal.*

C. Case study: [17] The framework of this study is quite different from that of [2, 20], since it focuses on the HNOW-Utilization Problem rather than the HNOW-Exploitation Problem. In common with the latter sources, a master workstation enlists the computational resources of an HNOW $\mathcal{N}$ in computing a bag of tasks that are equal in both size and complexity. Here, however, the master workstation is a member—call it P_m—of the HNOW $\mathcal{N}$. Moreover, here the bag of tasks is massive, and there is no *a priori* limit to the duration of the worksharing enterprise. Additionally, the form of worksharing considered is different from and, in some ways, more ambitious than in [2, 20]. Now, P_m allocates one task at a time, *and* workstations may redistribute these work allocations (one task at a time) at will, along direct communication links between selected pairs of workstations. Finally, in contrast to the HNOW-Exploitation Problem, one wants here to have the worksharing regimen reach an optimal "steady state," in which the average aggregate number of tasks computed per time-step is maximized. We describe here only the most general of the scheduling results in [17], which places no *a priori* restriction on which pairs of workstations can communicate directly with each other.

As in the HiHCoHP model, each workstation P_i of [17] has a computation rate ρ_i (cf. Table 2) which indicates the amount of time P_i takes to compute one task—but the indices here do not reflect relative speeds. Every pair of workstations, P_i and P_j, has an associated cost c_{ij} of transmitting a single task (with all material necessary for its computation) between P_i and P_j, in either direction. To simplify the development, the cost associated with a task is "double-ended," in the sense that it includes the cost of transmitting both that task and (at a later time) the results from that task. If P_i and P_j can communicate directly with one another—for short, are *neighbors*—then c_{ij} is finite; if they cannot, then, by convention, $c_{ij} = \infty$. The communication model in [17] is thus closer to that of [131] than to that of [2], for in the latter, the possible differences between packaging and unpackaging times may render communication costs asymmetric. Several regimens are considered in [17] concerning what processes may occur in parallel. We focus here only on their "base model," in which a workstation can simultaneously receive a task (or a result) from one neighbor, send a task (or a result) to one (possibly different) neighbor, and process some task (that it already has). In summation, if workstation P_i sends a task to workstation P_j at time-step t, then, until time $t + c_{ij}$:

- P_j cannot start executing this task nor initiate another receive operation;
- P_i cannot initiate another send operation.

Within the preceding model, the goal of the study—optimal steady-state performance—is formalized as follows. For each $1 \le i \le n$, let $n(i)$ be the set of indices of workstation P_i's neighbors. During a snapshot depicting one unit of activity by the HNOW $\mathcal{N}$:

- κ_i is the fraction of time during which P_i is *computing;*
- s_{ij} is the fraction of time during which P_i is *sending* to neighbor P_j;
- r_{ij} is the fraction of time during which P_i is *receiving* from neighbor P_j.

The quantity κ_i/ρ_i is the *throughput* of workstation P_i during the isolated time unit. To wit, P_i is capable of computing $1/\rho_i$ tasks in one time unit; in the snapshot, only the fraction κ_i of that time unit is spent computing. The goal is to maximize the quantity

$$\mathsf{Throughput\text{-}rate} \stackrel{\text{def}}{=} \sum_{i=1}^{n} \frac{\kappa_i}{\rho_i}. \tag{3.9}$$

subject to the following seven sets of constraints imposed by the model.

1. for all i: $\quad 0 \le \kappa_i \le 1$
   for all i, $j \in n(i)$: $\quad 0 \le s_{ij} \le 1$
   for all i, $j \in n(i)$: $\quad 0 \le r_{ij} \le 1$
   These reflect the fact that κ_i, s_{ij}, and r_{ij} are proper fractions.

2. for all i, $j \in n(i)$: $\quad s_{ij} = r_{ji}$
   Each P_j receives whatever each neighbor P_i sends it.

3. for all i: $\quad \sum_{j \in n(i)} s_{ij} \le 1$
   for all i: $\quad \sum_{j \in n(i)} r_{ij} \le 1$
   These reflect the single-port communication regimen.

4. for all i, $j \in n(i)$: $\quad s_{ij} + r_{ji} \le 1$
   Even though a link is bidirectional, its bandwidth can never be exceeded.
   (Multiply the inequality by the bandwidth $1/c_{ij}$ to clarify the constraint.)

5. for all $i \neq m$: $\displaystyle\sum_{j \in n(i)} \frac{r_{ij}}{c_{ij}} = \frac{\kappa_i}{\rho_i} + \sum_{j \in n(i)} \frac{s_{ij}}{c_{ij}}$
   A conservation law: For every P_i except the master P_m—which starts out with "infinitely many" tasks—the number of tasks that P_i receives should equal the number that it computes, plus the number that it relays to other P_j.

6. for all $j \in n(m)$: $r_{mj} = 0$
   Since P_m is saturated with tasks *ab initio*, there can be no advantage to sending it additional tasks.

7. $\kappa_m \equiv 1$
   The model allows P_m to compute without interruption.

The preceding formulation of the goal affords one an efficient alogorithm for optimally solving the HNOW-Utilization Problem on the HNOW $\mathcal{N}$ [17].

- *The optimization problem (3.9), augmented with the seven sets of constraints, comprises a linear program whose solution yields the optimal solution for the HNOW-Utilization Problem on the HNOW $\mathcal{N}$.*

- *This linear program finds this schedule in time polynomial in the size of $\mathcal{N}$, as measured by the number of workstations and the number of direct interworkstation links.*

Significant related studies. One finds in [3] a model that captures the same features as does HiHCoHP, but without allowing for workstation heterogeneity. Using this model, it is proved that the FIFO Protocol provides optimal solutions for the HNOW-Exploitation Problem in *homogeneous* NOWs.

We remarked earlier that one finds in [20] four variants of the HNOW-Exploitation Problem, not just the one variant we have described. In all four variants, the master workstation sends an allocation of equal-size, equal-complexity tasks to all workstations of the "exploited" HNOW $\mathcal{N}$ and receives the results of those tasks; all tasks are assumed to produce the same amount of data as results; all communication is single-ported. Two families of worksharing protocols are considered, one of which has work distributed and results collected in the staggered manner depicted in Fig. 5; the other of which has work distributed via a scatter operation and results collected via a gather operation.

The HNOW-Rental Problem is studied in [163], under a model in which tasks produce no output and communication can overlap with computation, even on the same workstation. Worksharing proceeds by having the master workstation transmit equal-size chunks of work to the rented HNOW's workstations at a frequency determined by an analysis of the workstations' powers. A near-optimal algorithm is derived within this setting.

One finds in [22, 23, 42] and sources cited therein a model that is simpler than those discussed thus far. These sources employ a very abstract model that suppresses many of the costs accounted for in the other cited studies.

Employing a rather different approach to worksharing, the study in [15] considers how to allocate a single compute-intensive task within an HNOW $\mathcal{N}$. The decision about which workstation(s) will receive the task is made based on an "auction." The master workstation determines which aggregation of $\mathcal{N}$'s workstations will—according to the source's cost model—yield the best performance on the auctioned task.

Finally, one finds in [56] a largely experimental study of worksharing in HNOWs whose workstations share resources in a nondedicated manner. As in a Computational Grid (see Section 4.1), the workstations of [56] *timeshare* their cycles with partners' work, rather than dedicating cycles to that work. As in [15], work is allocated among the HNOW's workstations based on anticipated performance on that work; in contrast to [15]: "anticipated performance" is explicitly determined empirically; all workstations simultaneously and continuously monitor the "anticipated performance" of their fellow HNOW members.

4 Internet Computing

Advancing technology has rendered the Internet a viable medium for collaborative computing, via mechanisms such as Grid computing (*GC*, for short) and Web-based computing (*WC*, for short). Our interest in these modalities of Internet computing resides in their (not-uncommon) use for computing a massive collection of (usually compute-intensive) tasks that reside at a "master" computing site. When so used, the "master" site views its "collaborators" as remotely situated "volunteers" who must be supplied with work in a manner that enhances the completion of the massive job.

4.1 The Platform(s)

Computational Grids. A *GC project* presupposes the formation of a *Computational Grid*—a consortium of computing sites that contract to share resources [62, 63]. From time to time, a Grid computing site will send a task to a companion Grid site that has agreed to share its computing cycles. When this companion site returns the result of its current task, it becomes eligible for further worksharing.

Web-based computing. In a *WC project*, a volunteer registers with the "master" site and receives a task to compute. When a volunteer completes its current task, it revisits the "master" site to return the results of that task and to receive a new task. Interesting WC projects include: [85, 159], which perform astronomical calculations; [137], which performs security-motivated number-theoretic calculations; [76, 116, 160], which perform medical and biological computations. Such sites benefit from Internet computing either because of the sheer volume of their workloads or because of the computational complexity of their individual tasks.

4.2 Some Challenges

The endeavor of using the Internet for collaborative computing gives rise to two algorithmic challenges that are not encountered in environments in which the computing agents are more tightly coupled. We term these challenges *temporal* and *factual unpredictability*.

Temporal unpredictability. Remote computing agents in an Internet computing project—be it a WC or GC project—typically tender no guarantee of when the results from an allocated task will be returned to the "master" site. Indeed, in a WC project, that site typically has no guarantee that a "volunteer" will *ever* return results. This lack of a time guarantee is an annoyance when the tasks comprising the collaborative workload are mutually independent—i.e., form a bag of tasks—but at least one never runs out of tasks that are eligible for allocation. (Of course, if all tasks must eventually be executed—which is *not* the case with several WC projects—then this annoyance must trigger some action, such as reallocation, by the "master" site.) However, when the tasks in the workload have interdependencies that constrain their order of execution, this temporal unpredictability can lead to a form of gridlock wherein no new tasks can be allocated for an indeterminate period, pending the execution of already allocated tasks. Although "safety devices" such as deadline-triggered reallocation of tasks address this danger, they do not eliminate it, since the "backup" remote participant assigned a given task may be as dilatory as the primary one. A major challenge is how to orchestrate the allocation of tasks in a way that minimizes the likelihood of this form of gridlock.

Factual unpredictability. The volunteers who participate in a WC project typically need not authenticate their alleged identities. In many such projects, the sheer number of participants would render the use of costly trusted authentication mechanisms impracticable. This fact renders all interchanges with—and information from—volunteers totally insecure. As noted in Section 1, this situation apparently creates an irresistible temptation for hackers, who plague many WC projects, greatly increasing the overhead for these projects. For this reason, one might suggest using WC only for security-insensitive applications (relating, say, to processing astronomical data [85, 159]) where erroneous or even mischievously or maliciously false results are not likely to have dire consequences. However, many of the most important applications of WC involve very sensitive applications, such as security-related [137] or health-related [76, 116] ones. Indeed, for many applications that generate truly massive numbers of identical tasks, Web-based computing is one of the only imaginable ways to assemble massive computing power at manageable cost. The challenge is to coordinate the volunteers in a WC project in a way that minimizes potential disruptions by hackers, while not excessively slowing down the progress of legitimate participants.

4.3 Some Sophisticated Responses

There have thus far been few rigorously analyzed algorithmic studies of computing on the Internet, via either WC or GC. One significant such study is [17], which studies worksharing in Grids. By rescaling model parameters, this study applies also to worksharing in HNOWs, which is the context in which we discuss it (Section 3.3.3.C). We have opted to reserve this section for studies that address problems unique to Internet computing.

4.3.1 Scheduling to cope with temporal unreliability

A. Case study: [133, 136] These sources craft and study a model that abstracts the process of scheduling computation-dags for either GC or WC. The goal of the model is to allow one to avoid the gridlock encountered when a computation stalls because all tasks that are eligible for execution have been allocated but not yet returned. The model is inspired by the many *pebble games* on dags that have been shown, over several decades, to yield elegant formal analogues of a variety of problems related to scheduling the task-nodes of computation-dags [47, 73, 118]. Such games use tokens called *pebbles* to model the progress of a computation on a dag: the placement or removal of the various available types of pebbles—which is constrained by the dependencies modeled by the dag's arcs—represents the changing (computational) status of the dag's task-nodes. The *Internet-Computing* (*IC*, for short) *Pebble Game* on a computation-dag $\mathcal{G}$ involves one player S, the *Server*, and an indeterminate number of players $C_1, C_2, \ldots$, the *Clients*. The Server has access to unlimited supplies of three types of pebbles: ELIGIBLE-BUT-UNALLOCATED (EBU, for short) pebbles, ELIGIBLE-AND-ALLOCATED (EAA, for short) pebbles, and EXECUTED (XEQ, for short) pebbles. The Game's moves reflect the successive stages in the "life-cycle" of a node in a computation-dag, from eligibility for execution through actual execution. Fig. 7 presents the rules of the IC Pebble Game. The reader should note how the moves of the Game expose the danger of a play's being stalled indefinitely by dilatory Clients.

There is little that one can do to forestall the chances of gridlock when playing the IC Pebble Game, absent some constraint on the actions of the Clients. Without some constraint, a malicious adversary (read: unfortunate behavior by Clients) could confute any attempt to guarantee the availability of a node containing an EBU pebble—by imposing a pessimal order on the execution of allocated tasks. The constraint imposed by the study in [133, 136] is the assumption that *tasks are executed in the same order as they are allocated.* (Since many GC and WC "master" sites monitor the state of remote participants, this

- At any step of the game, S may place an EBU pebble on any unpebbled source node of $\mathcal{G}$.
  /*Unexecuted source nodes are always eligible for execution, having no parents whose prior execution they depend on.*/
- Say that Client C_i approaches S requesting a task. If C_i has previously been allocated a task that it has not completed, then C_i's request is ignored; otherwise, the following occurs.
  - If at least one node of $\mathcal{G}$ contains an EBU pebble, then S gives C_i the task corresponding to one such node and replaces that node's pebble by an EAA pebble.
  - If no node of $\mathcal{G}$ contains an EBU pebble, then C_i is told to withdraw its request, and this move is a no-op.
- When a Client returns (the results from) a task-node, S replaces that task-node's EAA pebble by an XEQ pebble. S then places an EBU pebble on each unpebbled node of $\mathcal{G}$ all of whose parents contain XEQ pebbles.
- S's goal is to allocate nodes in such a way that every node v of $\mathcal{G}$ *eventually* contains an XEQ pebble.
  /*This modest goal is necessitated by the possibility that $\mathcal{G}$ is infinite.*/

Figure 7: *The rules of the IC Pebble Game*

assumption is not totally fanciful.) With this assumption in place, these studies attempt to optimize the quality of a play of the IC Pebble Game on a dag $\mathcal{G}$ by maximizing, at all steps t, *the aggregate number of* EBU *pebbles on* $\mathcal{G}$*'s nodes, as a function of the number of* EAA *and* XEQ *pebbles on* $\mathcal{G}$*'s nodes.*

The computation-dags studied in [133, 136] are the four depicted in Fig. 8: the (infinite) *evolving mesh-dag*, reduction-oriented versions of *mesh-dags* and *tree-dags*, and the *FFT-dag* [48]. It is shown in [133] (for evolving 2-dimensional mesh-dags) and in [136] (for the other dags in Fig. 8) that a schedule for the dags in Fig. 8 is optimal if, and only if, it allocates nodes in a *parent-oriented* fashion—i.e., it executes all parents of each node in consecutive steps. This general result translates to the following dag-specific instances.

- *The strategy of executing nodes of evolving mesh-dags along successive* levels *of the dag—level k comprises all nodes $\langle x, y\rangle$ such that $x + y = k$—is optimal for 2-dimensional mesh-dags. (It is shown in [133] that this strategy is within a constant factor of optimal for mesh-dags of higher (fixed) dimensionalities.)*

The proof for 2-dimensional mesh-dags is immediate from the following observation. No two eligible nodes can reside in the same row or the same column of the mesh-dag at any

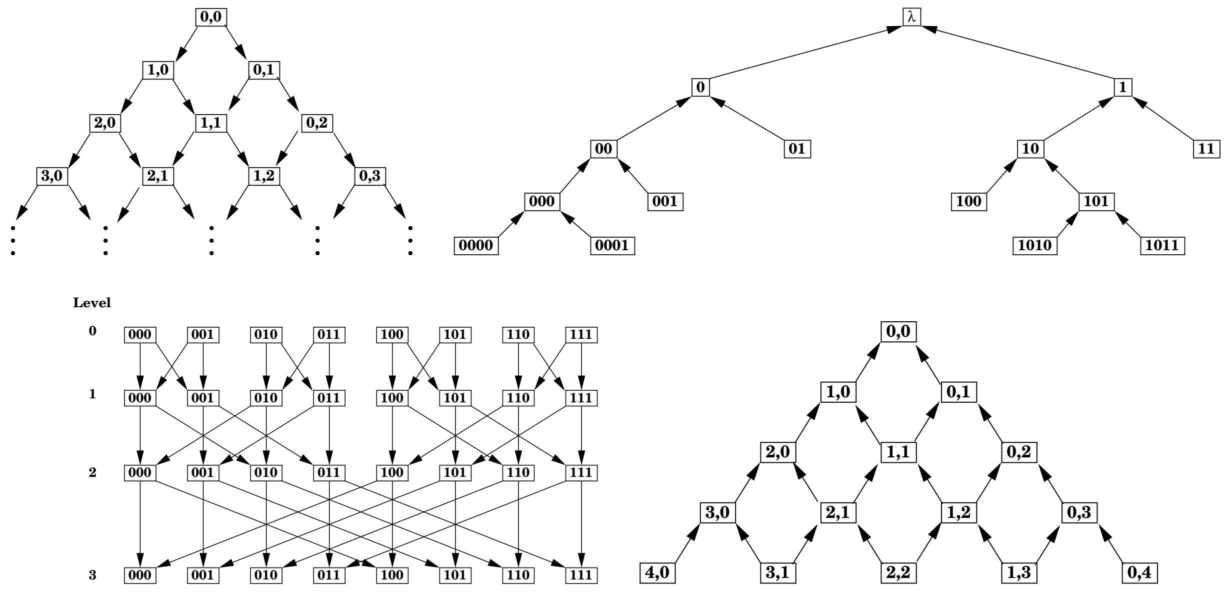

Figure 8: *Clockwise from upper left: the evolving (2-dimensional) mesh-dag, a (binary) reduction-tree dag, the 5-level (2-dimensional) reduction-mesh dag, the 4-level FFT-dag*

step of the IC Pebble Game; moreover, all "ancestors" of each EBU node must contain XEQ pebbles. Hence, when there are n EBU nodes on the dag, there must be at least $\binom{n}{2}$ XEQ nodes "supporting" them. The argument for higher dimensionalities is similar in strategy but significantly more complex.

- *For reduction-oriented mesh-dags, a schedule is optimal if it executes nodes along successive levels of the dag.*
- *For reduction-oriented tree-dags and for the FFT-dag, a schedule is optimal if "sibling" nodes*—nodes that share a parent—*are always executed in consecutive steps.*

For reduction-mesh dags, the optimality condition follows from the fact that the aggregate number of EBU nodes on the dag at any step of the IC Pebble Game is bounded above by (one plus) the smallest index of a level of the dag that contains a pebble at step t; one therefore wants this index to shrink as slowly as possible. For the other dags, the aggregate number of EBU nodes on the dag at a step of the IC Pebble Game is bounded above by a quantity depending on the structure of the dag and the number of XEQ nodes at that step, *minus the number of nodes that contain* XEQ *pebbles while their siblings don't.*

Significant progress is made in [102] toward developing techniques for crafting optimal schedules for a broad range of computation-dags, by abstracting and generalizing the scheduling principles underlying the case studies in [133, 136].

4.3.2 Scheduling to cope with factual unreliability

There is substantial work going on in the secure-computing community that is aimed at identifying attempts to compromise collaborative computing projects; see, e.g., [144] and sources cited therein. We know, however, of only one study aimed at possibly eliminating hackers from a WC project once they are identified.

A. Case study: [132] This source studies an unusual facet of the security problem in WC. It develops a computationally lightweight scheme for keeping track of which volunteers in a WC project computed which tasks. Much of the scheme employs familiar algorithmic techniques involving search trees for point- and range-queries. The unique aspect of the scheme is a strategy that assigns positive-integer indices to:

1. the set of all tasks at the master site,
2. all volunteers (who are allowed to arrive and depart dynamically),
3. the set of tasks reserved for each volunteer v

and that interrelates the resulting three sets of indices. The interrelation mechanism is a *task-allocation function* (*TAF*, for short), i.e., a *pairing function* φ that maps the set $\mathsf{N} \times \mathsf{N}$ of pairs of positive integers *one-to-one, onto* the set N of positive integers; symbolically, $\varphi : \mathsf{N} \times \mathsf{N} \leftrightarrow \mathsf{N}$. Each copy of the set N plays the role of one of the indicated sets of indices. The potential practicality of such a scheme demands that the functions φ, φ^{-1}, and $\varphi(v, t+1) - \varphi(v, t)$ all be easily computed; to wit, the "master" site must compute:

- $\varphi(v, t)$ to determine the index in the overall workload of the tth task in volunteer v's workload;
- $\varphi^{-1}(t)$ to determine which volunteer, v, was assigned task t, and what index task t has in v's workload;
- $\varphi(v, t+1) - \varphi(v, t)$ to determine which task to allocate to volunteer v when s/he returns the results of his/her task t.

In a quest for computational ease, the primary focus in [132] is on TAFs that are *additive* (are *ATAFs*, for short). An ATAF assigns each volunteer v a *base task-index* B_v and a *stride* S_v; it then uses the formula

$$\varphi(v, t) \ = \ B_v + (t-1)S_v$$

to determine the workload task-index of the tth task assigned to volunteer v. From a system perspective, ATAFs have the benefit that a volunteer's stride need be computed only when s/he first registers at the website and can be stored for subsequent appearances.

The main results of [132] determine how to assign base task-indices and strides to volunteers efficiently, both in terms of computing these indices and in terms of having the indices grow as slowly as possible, as functions of the volunteer-index v. The slow growth of B_v and S_v is argued in [132] to facilitate management of the memory in which the tasks are stored. Toward this end, a procedure for contructing ATAFs is presented, based on the following well-known property of the set $\emptyset$ of positive odd integers; see [113].

- *For any positive integer c, every odd integer can be written in precisely one of the 2^{c-1} forms: $2^c n + 1,\ 2^c n + 3,\ 2^c n + 5, \ldots,\ 2^c n + (2^c - 1)$, for some nonnegative integer n.*

Procedure ATAF-Constructor(φ) (see Fig. 9) builds on the preceding result to construct ATAFs efficiently.

Step 1. Partition the set of volunteer-task-indices into *groups* whose sizes are powers of 2 (with any desired mix of equal-size and distinct-size groups). Order the groups linearly in some (arbitrary) way.

/*We can now talk unambiguously about group 0 (whose members share *group-index* $g = 0$), group 1 (whose members share group-index $g = 1$), and so on.*/

Step 2. Assign each group a distinct copy of the set $\emptyset$, via a *copy-index* $\kappa(g)$ expressed as a function of the group-index g.

/*We can now talk unambiguously about group g's copy $\emptyset_{\kappa(g)}$ of the odd integers.*/

Step 3. Allocate group g's copy $\emptyset_{\kappa(g)}$ to its members via the $(c = \kappa(g))$ instance of the cited property of the odd integers, using the multiplier 2^g as a *signature* to distinguish group g's copy of the set $\emptyset$ from all other groups' copies.

Figure 9: **Procedure** ATAF-Constructor*(φ), which constructs an ATAF φ*

An explicit expression for the ATAFs of Procedure ATAF-Constructor**.** If we denote the $2^{\kappa(g)}$ rows of group g as $x_{g,1}, x_{g,2}, \ldots, x_{g,2^{\kappa(g)}}$, then for all $i \in \{1, 2, \ldots, \kappa(g)\}$,

$$\varphi(x_{g,i}, y) \ \stackrel{\text{def}}{=}\ 2^g \left[2^{1+\kappa(g)}(y-1) + (2x_{g,i} + 1 \bmod 2^{1+\kappa(g)})\right]$$

Fig. 10 illustrates the construction via a sampler of argument-result values from three sample ATAFs. The first two exemplified ATAFs, $\varphi^{<1>}$ and $\varphi^{<3>}$, stress ease of computation; the third, $\varphi^{\#}(x, y)$, stresses slowly growing strides.

$\varphi^{<1>}(x,y) \ \stackrel{\text{def}}{=}\ 2^{x-1}\,[2(y-1) + (2x-1 \bmod 2)]$

$\langle x,g\rangle$	$\varphi^{<1>}(x,y)$					
$\langle 14,13\rangle$	8192	24576	40960	57344	73728	$\cdots$
$\langle 15,14\rangle$	16384	49152	81920	114688	147456	$\cdots$

$\varphi^{<3>}(x,y) \ \stackrel{\text{def}}{=}\ 2^{\lfloor (x-1)/4\rfloor}\,[8(y-1) + (2x-1 \bmod 8)]$

$\langle x,g\rangle$	$\varphi^{<3>}(x,y)$					
$\langle 14,3\rangle$	24	88	152	216	280	$\cdots$
$\langle 15,3\rangle$	40	104	168	232	296	$\cdots$
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\langle 28,6\rangle$	448	960	1472	1984	2496	$\cdots$
$\langle 29,7\rangle$	128	1152	2176	3200	4224	$\cdots$

$\varphi^{\#}(x,y) \ \stackrel{\text{def}}{=}\ 2^{\lfloor \log x\rfloor}\left(2^{1+\lfloor \log x\rfloor}(y-1) + (2x+1 \bmod 2^{1+\lfloor \log x\rfloor})\right)$

$\langle x,g\rangle$	$\varphi^{\#}(x,y)$					
$\langle 28,4\rangle$	400	912	1424	1936	2448	$\cdots$
$\langle 29,4\rangle$	432	944	1456	1968	2480	$\cdots$

Figure 10: *Sample values by three ATAFs*

Acknowledgments. The research of the author was supported in part by NSF Grant CCR-00-73401. Thanks are due many colleagues who helped in varied ways: by commenting on the text, by pointing out references, and by giving permission to have their work summarized here. I hesitate to enumerate them for fear of inadvertently forgetting some. Let me, therefore, merely assure them of my sincere gratitude.

References

[1] U. Acar, G.E. Blelloch, R.D. Blumofe (2002): The data locality of work stealing. *Theory of Computing Systs. 35*, 321–347.

[2] M. Adler, Y. Gong, A.L. Rosenberg (2003): Optimal sharing of bags of tasks in heterogeneous clusters. *15th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA'03)*, 1–10.

[3] J. Agrawal and H.V. Jagadish (1988): Partitioning techniques for large-grained parallelism. *IEEE Trans. Computers 37*, 1627–1634.

[4] W. Aiello, S.N. Bhatt, F.R.K. Chung, A.L. Rosenberg, R.K. Sitaraman (2001): Augmented ring networks. *IEEE Trans. Parallel and Distr. Systs. 12*, 598–609.

[5] S. Akl (1989): *The Design and Analysis of Parallel Algorithms.* Prentice-Hall, Englewood Cliffs, NJ.

[6] R. Aleliunas and A.L. Rosenberg (1982): On embedding rectangular grids in square grids. *IEEE Trans. Comp., C-31*, 907–913.

[7] A. Alexandrov, M.I. Ionescu, K.E. Schauser, C. Scheiman (1997): LogGP: incorporating long messages into the LogP model for parallel computation. *J. Parallel Distr. Comput. 44*, 71–79.

[8] R.J. Anderson and G.L. Miller (1990): A simple randomized parallel algorithm for list-ranking. *Inform. Proc. Let. 10.*

[9] T.E. Anderson, D.E. Culler, D.A. Patterson, and the HNOW Team (1995): A case for NOW (networks of workstations). *IEEE Micro 15*, 54–64.

[10] M. Andrews, F.T. Leighton, P.T. Metaxas, L. Zhang (1996): Improved methods for hiding latency in high bandwidth networks. *8th ACM Symp. on Parallel Algorithms and Architectures*, 52–61.

[11] F.S. Annexstein (1991): SIMD-emulations of hypercubes and related networks by linear and ring-connected processor arrays. *3rd IEEE Symp. on Parallel and Distr. Processing*, 656–659.

[12] F.S. Annexstein (1994): Embedding hypercubes and related networks into mesh-connected processor arrays. *J. Parallel Distr. Comput. 23*, 72–79.

[13] F.S. Annexstein, M. Baumslag, A.L. Rosenberg (1990): Group action graphs and parallel architectures. *SIAM J. Comput. 19*, 544–569.

[14] N.S. Arora, R.D. Blumofe, C.G. Plaxton (2001): Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systs. 34*, 115–144.

[15] M.J. Atallah, C.L. Black, D.C. Marinescu, H.J. Siegel, T.L. Casavant (1992): Models and algorithms for coscheduling compute-intensive tasks on a network of workstations. *J. Parallel Distr. Comput. 16*, 319–327.

[16] B. Awerbuch, Y. Azar, A. Fiat, F.T. Leighton (1996): Making commitments in the face of uncertainty: how to pick a winner almost every time. *28th ACM Symp. on Theory of Computing*, 519–530.

[17] C. Banino, O. Beaumont, L. Carter, J. Ferrante, A. Legrand, Y. Robert (2003): Scheduling strategies for master-slave tasking on heterogeneous processor grids. Typescript, ENS-Lyon.

[18] A. Bar-Noy and D. Peleg (1991): Square meshes are not always optimal. *IEEE Trans. Comp. 40*, 196–204.

[19] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, Y. Robert (2002): Bandwidth-centric allocation of independent tasks on heterogeneous platforms. *Intl. Parallel and Distr. Processing Symp. (IPDPS'02).*

[20] O. Beaumont, A. Legrand, Y. Robert (2003): The master-slave paradigm with heterogeneous processors. *IEEE Trans. Parallel and Distr. Systs. 14*, 897–908.

[21] J.-C. Bermond and C. Peyrat (1989): The de Bruijn and Kautz networks: a competitor for the hypercube? In *Hypercube and Distributed Computers* (F. Andre and J.P. Verjus, eds.) North-Holland, Amsterdam, 279–293.

[22] V. Bharadwaj, D. Ghose, V. Mani (1994): Optimal sequencing and arrangement in distributed single-level tree networks. *IEEE Trans. Parallel and Distr. Systs. 5*, 968–976.

[23] V. Bharadwaj, D. Ghose, V. Mani (1995): Multi-installment load distribution in tree networks with delays. *IEEE Trans. Aerospace and Electron. Systs. 31*, 555–567.

[24] S.N. Bhatt, F.R.K. Chung, F.T. Leighton, A.L. Rosenberg (1992): Efficient embeddings of trees in hypercubes. *SIAM J. Comput. 21*, 151–162.

[25] S.N. Bhatt, F.R.K. Chung, J.-W. Hong, F.T. Leighton, B. Obrenić, A.L. Rosenberg, E.J. Schwabe (1996): Optimal emulations by butterfly-like networks. *J. ACM 43*, 293–330.

[26] S.N. Bhatt, F.R.K. Chung, F.T. Leighton, A.L. Rosenberg (1997): On optimal strategies for cycle-stealing in networks of workstations. *IEEE Trans. Comp. 46*, 545–557.

[27] S.N. Bhatt, D.S. Greenberg, F.T. Leighton, P. Liu (1999): Tight bounds for on-line tree embeddings. *SIAM J. Comput. 29*, 474–491.

[28] S.N. Bhatt and F.T. Leighton (1984): A framework for solving VLSI graph layout problems. *J. Comput. Syst. Scis. 28*, 300–343.

[29] G. Bilardi, K.T. Herley, A. Pietracaprina, G. Pucci, P. Spirakis (1998): BSP vs. LogP. *Algorithmica.*

[30] G. Bilardi and A. Nicolau (1989): Adaptive bitonic sorting: An optimal algorithm for shared memory machines. *SIAM J. Comput. 18*, 216–228.

[31] G. Bilardi and F.P. Preparata (1995): Horizons of parallel computation. *J. Parallel Distr. Comput. 27*, 172–182.

[32] 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).*

[33] R.D. Blumofe and C.E. Leiserson (1998): Space-efficient scheduling of multithreaded computations. *SIAM J. Comput. 27*, 202–229.

[34] R.D. Blumofe and C.E. Leiserson (1999): Scheduling multithreaded computations by work stealing. *J. ACM 46*, 720–748.

[35] R.D. Blumofe and D.S. Park (1994): Scheduling large-scale parallel computations on networks of workstations. *3rd Intl. Symp. on High-Performance Distr. Computing*, 96–105.

[36] B. Boothe and A.G. Ranade (1992): Improved multithreading techniques for hiding communication latency in multiprocessors. *19th Intl. Symp. on Computer Architecture.*

[37] R.P. Brent (1974): The parallel evaluation of general arithmetic expressions. *J. ACM 21*, 201–206.

[38] R.P. Brent and H.T. Kung (1984): Systolic VLSI arrays for polynomial gcd computation. *IEEE Trans. Comp., C-33*, 731–737.

[39] R.P. Brent, H.T. Kung, F.T. Luk (1983): Some linear-time algorithms for systolic arrays. In *Information Processing 83* (R.E.A. Mason, ed.), North-Holland, Amsterdam, 865–876.

[40] T.N. Bui, S. Chaudhuri, F.T. Leighton, M. Sipser (1987): Graph bisection algorithms with good average case behavior. *Combinatorica 7*, 171–191.

[41] F. Cappello, P. Fraigniaud, B. Mans, A.L. Rosenberg (2001): HiHCoHP—Toward a realistic communication model for hierarchical hyperclusters of heterogeneous processors. *Intl. Parallel and Distr. Processing Symp. (IPDPS'01).*

[42] Y.C. Cheng and T.G. Robertazzi (1990): Distributed computation for tree networks with communication delays. *IEEE Trans. Aerospace and Electron. Systs. 26*, 511–516.

[43] S. Chingchit, M. Kumar, L.N. Bhuyan (1999): A flexible clustering and scheduling scheme for efficient parallel computation. *13th IEEE Intl. Parallel Processing Symp.*, 500–505.

[44] W. Cirne and K. Marzullo (1999): The Computational Co-op: gathering clusters into a metacomputer. *13th Intl. Parallel Processing Symp.*, 160–166.

[45] M. Cole (1989): *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, Cambridge, Mass.

[46] R. Cole and U. Vishkin (1986): Deterministic coin tossing with applications to optimal parallel list ranking. *Inform. Contr. 70*, 32–53.

[47] S.A. Cook (1974): An observation on time-storage tradeoff. *J. Comp. Syst. Scis. 9*, 308–316.

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

[49] M. Cosnard and M. Tchuente (1988): Designing systolic algorithms by top-down analysis. *3rd Intl. Conf. on Supercomputing.*

[50] M. Cosnard and D. Trystram (1995): *Parallel Algorithms and Architectures.* International Thompson Computer Press.

[51] D.E. Culler, R.M. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, T. von Eicken (1996): LogP: towards a realistic model of parallel computation. *C. ACM 39*, 78–85.

[52] R. Cypher and C.G. Plaxton (1993): Deterministic sorting in nearly logarithmic time on the hypercube and related computers. *J. Comput. Syst. Scis. 47*, 501–548.

[53] T.D. deRose, L. Snyder, C. Yang (1987): Near-optimal speedup of graphics algorithms using multigauge parallel computers. *Intl. Conf. on Parallel Processing*, 289–294.

[54] M.D. Dikaiakos, K. Steiglitz, A. Rogers (1994): A comparison of techniques for mapping parallel algorithms to message-passing multiprocessors. *6th IEEE Symp. on Parallel and Distr. Processing*, 434–442.

[55] K. Diks, H.N. Djidjev, O. Sykora, I. Vrťo (1993): Edge separators of planar and outerplanar graphs with applications. *J. Algorithms 14*, 258–279.

[56] X. Du and X. Zhang (1997): Coordinating parallel processes on networks of workstations. *J. Parallel Distr. Comput. 46*, 125–135.

[57] K. Efe (1991): Embedding mesh of trees into the hypercube. *J. Parallel Distr. Comput. 11*, 222–230.

[58] R. Elsässer, B. Monien, R. Preis (2002): Diffusion schemes for load balancing on heterogeneous networks. *Theory of Computing Systs. 35*, 305–320.

[59] P. Fatourou and P. Spirakis (2000): Efficient scheduling of strict multithreaded computations. *Theory of Computing Systs. 33*, 173–232.

[60] C.M. Fiduccia and R.M. Mattheyses (1982): A linear-time heuristic for improving network partitions. *19th ACM-IEEE Design Automation Conf.*, 175–181.

[61] S. Fortune and J. Wyllie (1978): Parallelism in random access machines. *10th ACM Symp. on Theory of Computing*, 114–118.

[62] I. Foster and C. Kesselman [eds.] (1999): *The Grid: Blueprint for a New Computing Infrastructure*, Morgan-Kaufmann.

[63] I. Foster, C. Kesselman, S. Tuecke (2001): The anatomy of the Grid: enabling scalable virtual organizations. *Intl. J. Supercomputer Applications.*

[64] D. Gannon (1980): A note on pipelining a mesh-connected multiprocessor for finite element problems by nested dissection. *Intl. Conf. on Parallel Processing*, 197–204.

[65] 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 Distr. Systs. 10*, 813–824.

[66] V. Garg and D.E. Schimmel (1998): Hiding communication latency in data parallel applications. *12th IEEE Intl. Parallel Processing Symp.*, 18–25.

[67] A. Gerasoulis, S. Venugopal, T. Yang (1990): Clustering task graphs for message passing architectures. *ACM Intl. Conf. on Supercomputing*, 447–456.

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

[69] M.W. Goudreau, K. Lang, S.B. Rao, T. Suel, T. Tsantilas (1999): Portable and efficient parallel computing using the BSP model. *IEEE Trans. Comp. 48*, 670–689.

[70] D.S. Greenberg, L.S. Heath and A.L. Rosenberg (1990): Optimal embeddings of butterfly-like graphs in the hypercube. *Math. Syst. Th. 23*, 61–77.

[71] V.C. Hamacher and H. Jiang (1994): Comparison of mesh and hierarchical networks for multiprocessors. *Intl. Conf. on Parallel Processing*, I:67–71.

[72] C.-T. Ho and S.L. Johnsson (1986): Graph embeddings for maximum bandwidth utilization in hypercubes. *Intl. Conf. Vector and Parallel Computing.*

[73] 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.

[74] Y. Hong and T. Payne (1989): Parallel sorting in a ring network of processors. *IEEE Trans. Comp. 38*, 458–464.

[75] O.H. Ibarra and S.T. Sohn (1990): On mapping systolic algorithms onto the hyercube. *IEEE Trans. Parallel Distr. Systs. 1*, 238–249.

[76] *The Intel Philanthropic Peer-to-Peer program.* ⟨`www.intel.com/cure`⟩.

[77] C. Kaklamanis and D. Krizanc (1992): Optimal sorting on mesh-connected processor arrays. *4th ACM Symp. on Parallel Algorithms and Architectures*, 50–59.

[78] C. Kaklamanis, D. Krizanc, S.B. Rao (1997): New graph decompositions with applications to emulations. *Theory of Computing Systs. 30*, 39–49.

[79] R.M. Karp and R.E. Miller (1966): Properties of a model for parallel computations: determinacy, termination, queueing. *SIAM J. Appl. Math. 14*, 1390–1411.

[80] R.M. Karp and V. Ramachandran (1990): A survey of parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science, vol. A* (J. van Leeuwen, ed.) Elsevier Science, Amsterdam, 869–941.

[81] R.M. Karp, A. Sahay, E. Santos, K.E. Schauser (1993): Optimal broadcast and summation in the logP model. *5th ACM Symp. on Parallel Algorithms and Architectures*, 142–153.

[82] B.W. Kernighan and S. Lin (1970): An efficient heuristic procedure for partitioning graphs. *Bell Syst. Tech. J. 49*, 291–307.

[83] S.J. Kim and J.C. Browne (1988): A general approach to mapping of parallel computations upon multiprocessor architectures. *Intl. Conf. on Parallel Processing*, III:1–8.

[84] R. Koch, F.T. Leighton, B.M. Maggs, S.B. Rao, A.L. Rosenberg, E.J. Schwabe (1997): Work-preserving emulations of fixed-connection networks. *J. ACM 44*, 104–147.

[85] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, M. Lebofsky (2000): SETI@home: massively distributed computing for SETI. In *Computing in Science and Engineering* (P.F. Dubois, Ed.) IEEE Computer Soc. Press, Los Alamitos, CA.

[86] H.T. Kung (1985): Systolic arrays. In *McGraw-Hill 1985 Yearbook of Science and Technology.*

[87] H.T. Kung and C.E. Leiserson (1980): Systolic arrays (for VLSI). In Chapter 8 of [105].

[88] H.T. Kung and W.T. Lin (1983): An algebra for VLSI algorithm design. *Conf. on Elliptic Problem Solvers*, Monterey, CA.

[89] H.T. Kung and R.L. Picard (1984): One-dimensional systolic arrays for multidimensional convolution and resampling. In *VLSI for Pattern Recognition and Image Processing*, Springer-Verlag, Berlin, 9–24.

[90] C. Lam, H. Jiang, V.C. Hamacher (1995): Design and analysis of hierarchical ring networks for shared-memory multiprocessors. *Intl. Conf. on Parallel Processing*, I:46–50.

[91] H.W. Lang, M. Schimmler, H. Schmeck, H. Schroeder (1985): Systolic sorting on a mesh-connected network. *IEEE Trans. Comp., C-34*, 652–658.

[92] F.T. Leighton (1985): Tight bounds on the complexity of parallel sorting. *IEEE Trans. Comp., C-34*, 344–354.

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

[94] F.T. Leighton, B.M. Maggs, S.B. Rao (1994): Packet routing and job-shop scheduling in O(congestion + dilation) steps. *Combinatorica 14*, 167–186.

[95] F.T. Leighton, B.M. Maggs, A.W. Richa (1998): Fast algorithms for finding O(congestion + dilation) packet routing schedules. *Combinatorica 18.*

[96] F.T. Leighton, M.J. Newman, A.G. Ranade, E.J. Schwabe (1992): Dynamic tree embeddings in butterflies and hypercubes. *SIAM J. Comput. 21*, 639–654.

[97] G. Lerman and L. Rudolph (1993): *Parallel Evolution of Parallel Processors.* Plenum Press, New York.

[98] K. Li and J. Dorband (1999): Asymptotically optimal probabilistic embedding algorithms for supporting tree structured computations in hypercubes. *7th Symp. on Frontiers of Massively Parallel Computation.*

[99] R.J. Lipton and R.E. Tarjan (1980): Applications of a planar separator theorem. *SIAM J. Comput. 9*, 615–627.

[100] M. Litzkow, M. Livny, M.W. Mutka (1988): Condor – A hunter of idle workstations. *8th Intl. Conf. on Distr. Computing Systs.*, 104–111.

[101] B.M. Maggs, F. Meyer auf der Heide, B. Vöcking, M. Westermann (1997): Exploiting locality for data management in systems of limited bandwidth. *38th IEEE Symp. on Foundations of Computer Science*, 284–293.

[102] G. Malewicz, A.L. Rosenberg M. Yurkewych (2004): On scheduling complex dags for Internet-based computing. Typescript, Univ. Massachusetts. Submitted for publication.

[103] D.W. Matula and L.L. Beck (1983): Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM 30*, 417–427.

[104] W.F. McColl and A. Tiskin (1998): Memory-efficient matrix computations in the BSP model. *Algorithmica.*

[105] C. Mead and L. Conway (1980): *Introduction to VLSI Systems.* Addison-Wesley, Reading, Mass.

[106] G.L. Miller, V. Ramachandran, E. Kaltofen (1988): Efficient parallel evaluation of straight-line code and arithmetic circuits. *SIAM J. Comput. 17*, 687–695.

[107] G.L. Miller and J.H. Reif (1989): Parallel tree contraction, Part 1: fundamentals. In *Randomness and Computation*, vol. 5 (S. Micali, ed.), JAI Press, Greenwich, Ct., 47–72.

[108] G.L. Miller and J.H. Reif (1991): Parallel tree contraction, Part 2: further applications. *SIAM J. Comput. 20*, 1128–1147.

[109] W.L. Miranker and A. Winkler (1984): Spacetime representations of computational structures. *Computing 32*, 93–114.

[110] M. Mitzenmacher (1998): Analyses of load stealing models based on differential equations. *10th ACM Symp. on Parallel Algorithms and Architectures*, 212–221.

[111] M. Mitzenmacher (1999): On the analysis of randomized load balancing schemes. *Theory of Computing Systs. 32*, 361–386.

[112] J. F. Myoupo (1992): Synthesizing linear systolic arrays for dynamic programming problems. *Parallel Proc. Let. 2*, 97–110.

[113] I. Niven and H.S. Zuckerman (1980): *An Introduction to the Theory of Numbers.* (4th Ed.) J. Wiley & Sons, New York.

[114] B. Obrenić (1994): An approach to emulating separable graphs. *Math. Syst. Th. 27*, 41–63.

[115] B. Obrenić, M.C. Herbordt, A.L. Rosenberg, C.C. Weems (1999): Using emulations to enhance the performance of parallel architectures. *IEEE Trans. Parallel and Distr. Systs. 10*, 1067–1081.

[116] *The Olson Laboratory Fight AIDS@Home project.* ⟨`www.fightaidsathome.org`⟩.

[117] C.H. Papadimitriou and M. Yannakakis (1990): Towards an architecture-independent analysis of parallel algorithms. *SIAM J. Comput. 19*, 322–328.

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

[119] G.F. Pfister (1995): *In Search of Clusters.* Prentice-Hall.

[120] P. Quinton (1984): Automatic synthesis of systolic arrays from uniform recurrence equations. *11th IEEE Intl. Symp. on Computer Architecture*, 208–214.

[121] P. Quinton (1988): Mapping recurrences on parallel architectures. *3rd Intl. Conf. on Supercomputing.*

[122] P. Quinton, B. Joinnault, P. Gachet (1986): A new matrix multiplication systolic array. *Parallel Algorithms and Architectures* (M. Cosnard et al., eds.) North-Holland, Amsterdam, 259–268.

[123] M.O. Rabin (1989): Efficient dispersal of information for security, load balancing, and fault tolerance. *J. ACM 36*, 335–348.

[124] A.G. Ranade (1993): A framework for analyzing locality and portability issues in parallel computing. In *Parallel Architectures and Their Efficient Use: The 1st Heinz-Nixdorf Symp.*, Paderborn, Germany (F. Meyer auf der Heide, B. Monien, A.L. Rosenberg, eds.) *Lecture Notes in Computer Science 678*, Springer-Verlag, Berlin, 185–194.

[125] J.H. Reif and L.G. Valiant (1987): A logarithmic time sort for linear networks. *J. ACM 34*, 60–76.

[126] A.L. Rosenberg (1981): Issues in the study of graph embeddings. In *Graph-Theoretic Concepts in Computer Science: Proceedings of the Intl. Wkshp. WG80* (H. Noltemeier, ed.) *Lecture Notes in Computer Science 100*, Springer-Verlag, Berlin, 150–176.

[127] A.L. Rosenberg (1994): Needed: a theoretical basis for heterogeneous parallel computing. In *Developing a Computer Science Agenda for High-Performance Computing* (U. Vishkin, ed.) ACM Press, New York, 137–142.

[128] A.L. Rosenberg (1999): Guidelines for data-parallel cycle-stealing in networks of workstations, I: on maximizing expected output. *J. Parallel Distr. Comput. 59*, 31–53.

[129] A.L. Rosenberg (2000): Guidelines for data-parallel cycle-stealing in networks of workstations, II: on maximizing guaranteed output. *Intl. J. Foundations of Computer Science 11*, 183–204.

[130] A.L. Rosenberg (2001): On sharing bags of tasks in heterogeneous networks of workstations: greedier is not better. *3rd IEEE Intl. Conf. on Cluster Computing (Cluster'01)*, 124–131.

[131] A.L. Rosenberg (2002): Optimal schedules for cycle-stealing in a network of workstations with a bag-of-tasks workload. *IEEE Trans. Parallel and Distr. Systs. 13*, 179–191.

[132] A.L. Rosenberg (2003): Accountable Web-computing. *IEEE Trans. Parallel and Distr. Systs. 14*, 97–106.

[133] A.L. Rosenberg (2004): On scheduling mesh-structured computations on the Internet. *IEEE Trans. Comput.*, 1176–1186.

[134] A.L. Rosenberg and L.S. Heath (2001): *Graph Separators, with Applications.* Kluwer Academic/Plenum Publishers, New York.

[135] A.L. Rosenberg and I.H. Sudborough (1983): Bandwidth and pebbling. *Computing 31*, 115–139.

[136] A.L. Rosenberg and M. Yurkewych (2004): Guidelines for scheduling some common computation-dags for Internet-based computing." *IEEE Trans. Comput.*, to appear.

[137] *The RSA Factoring by Web Project.* ⟨`http://www.npac.syr.edu/factoring`⟩ (with Foreword by A. Lenstra). Northeast Parallel Architecture Center.

[138] L. Rudolph, M. Slivkin, E. Upfal (1991): A simple load balancing scheme for task allocation in parallel machines. *3rd ACM Symp. on Parallel Algorithms and Architectures*, 237–244.

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

[140] V. Sarkar and J. Hennessy (1986): Compile-time partitioning and scheduling of parallel programs. *SIGPLAN Notices 21* (7) 17–26.

[141] C.P. Schnorr and A. Shamir (1986): An optimal sorting algorithm for mesh connected computers. *18th ACM Symp. on Theory of Computing*, 255–263.

[142] E.J. Schwabe (1992): Embedding meshes of trees into de Bruijn graphs. *Inform. Proc. Let. 43*, 237–240.

[143] L. Snyder (1985): An inquiry into the benefits of multigauge parallel computation. *Intl. Conf. on Parallel Processing*, 488–492.

[144] D. Szada, B. Lawson, J. Owen (2003): Hardening functions for large-scale distributed computing. *IEEE Security and Privacy Conf.*

[145] M.M. Theimer and K.A. Lantz (1989): Finding idle machines in a workstation-based distributed environment. *IEEE Trans. Software Eng'g. 15*, 1444–1458.

[146] C.D. Thompson (1979): Area-time complexity for VLSI. *11th ACM Symp. on Theory of Computing*, 81–88.

[147] C.D. Thompson (1980): *A Complexity Theory for VLSI.* Ph.D. Thesis, CMU.

[148] C.D. Thompson and H.T. Kung (1977): Sorting on a mesh-connected parallel computer. *C. ACM 20.*

[149] J.D. Ullman (1984): *Computational Aspects of VLSI.* Computer Science Press, Rockville, Md.

[150] L.G. Valiant (1983): Optimality of a two-phase strategy for routing in interconnection networks. *IEEE Trans. Comp., C-32*, 861–863.

[151] L.G. Valiant (1989): Bulk-synchronous parallel computers. In *Parallel Processing and Artificial Intelligence* (M. Reeve and S.E. Zenith, eds.) J. Wiley and Sons, New York, 15–22.

[152] L.G. Valiant (1990): General purpose parallel architectures. In *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.) Elsevier Science, Amsterdam, 943–972.

[153] L.G. Valiant (1990): A bridging model for parallel computation. *C. ACM 33*, 103–111.

[154] L.G. Valiant and G.J. Brebner (1981): Universal schemes for parallel computation. *13th ACM Symp. on Theory of Computing*, 263–277.

[155] P.M.B. Vitanyi (1986): Nonsequential computation and laws of nature. *VLSI Algorithms and Architectures* (Aegean Wkshp. on Computing) *Lecture Notes in Computer Science 227*, Springer-Verlag, Berlin, 108–120.

[156] P.M.B. Vitanyi (1988): Locality, communication and interconnect length in multicomputers. *SIAM J. Comput. 17*, 659–672.

[157] P.M.B. Vitanyi (1988): A modest proposal for communication costs in multicomputers. In *Concurrent Computations: Algorithms, Architecture, and Technology* (S.K. Tewksbury, B.W. Dickinson, S.C. Schwartz, eds.) Plenum Press, New York, 203–216.

[158] A.S. Wagner (1989): Embedding arbitrary binary trees in a hypercube. *J. Parallel Distr. Comput. 7*, 503–520.

[159] C. Weth, U. Kraus, J. Freuer, M. Ruder, R. Dannecker, P. Schneider, M. Konold, H. Ruder (2000): XPulsar@home — schools help scientists. Typescript, Univ. Tübingen.

[160] S.W. White and D.C. Torney (1993): Use of a workstation cluster for the physical mapping of chromosomes. *SIAM NEWS*, March, 1993, 14–17.

[161] A.Y. Wu (1985): Embedding of tree networks into hypercubes. *J. Parallel Distr. Comput. 2*, 238–249.

[162] T. Yang and A. Gerasoulis (1992): PYRROS: static task scheduling and code generation for message passing multiprocessors. *6th ACM Conf. on Supercomputing*, 428–437.

[163] Y. Yang and H. Casanova (2003): UMR: A multi-round algorithm for scheduling divisible workloads. *17th Intl. Parallel and Distributed Processing Symp. (IPDPS'03).*