2-1991

A Practical Hierarchial Model of Parallel Computation: The Model

Todd Heywood

Sanjay Ranka
Syracuse University

SU-CIS-91-06

A Practical Hierarchial Model of Parallel Computation: The Model

Todd Heywood and Sanjay Ranka

February 1991

School of Computer and Information Science
Suite 4-116
Center for Science and Technology
Syracuse, New York 13244-4100

(315) 443-2368

A Practical Hierarchical Model of Parallel Computation: The Model

Todd Heywood and Sanjay Ranka
School of Computer and Information Science
Syracuse University
Syracuse, NY 13244
Email: heywood@top.cis.syr.edu and ranka@top.cis.syr.edu

February 26, 1991

Abstract

We introduce a model of parallel computation that retains the ideal properties of the PRAM by using it as a sub-model, while simultaneously being more reflective of realistic parallel architectures by accounting for and providing abstract control over communication and synchronization costs. The Hierarchical PRAM (H-PRAM) model controls conceptual complexity in the face of asynchrony in two ways. First, by providing the simplifying assumption of synchronization to the design of algorithms, but allowing the algorithms to work asynchronously with each other; and organizing this "control asynchrony" via an implicit hierarchy relation. Second, by allowing the restriction of "communication asynchrony" in order to obtain determinate algorithms (thus greatly simplifying proofs of correctness). It is shown that the model is reflective of a variety of existing and proposed parallel architectures, particularly ones that can support massive parallelism. Relationships to programming languages are discussed. Since the PRAM is a sub-model, we can use PRAM algorithms as sub-algorithms in algorithms for the H-PRAM; thus results that have been established with respect to the PRAM are potentially transferable to this new model. The H-PRAM can be used as a flexible tool to investigate general degrees of locality ("neighborhoods" of activity) in problems, considering communication and synchronization simultaneously. This gives the potential of obtaining algorithms that map more efficiently to architectures, and of increasing the number of processors that can efficiently be used on a problem (in comparison to a PRAM that charges for communication and synchronization). The model presents a framework in which to study the extent that general locality can be exploited in parallel computing.

A companion paper demonstrates the usage of the H-PRAM via the design and analysis of various algorithms for computing the complete binary tree and the FFT/butterfly graph.

1 Introduction

In few areas of computer science is theoretical research as important to practice than in parallel computer systems. "Systems" is a very general term, broadly standing for the integrated hardware and software that comprises the computer that a *user* sees. One reason theoretical work is important is is that it is necessary to have sound conclusions about the operation of a system before undertaking the cost of building it. This is particularly important in parallel computing, since there are so many different ways of building and using parallel systems.

Another reason for the importance of theoretical work is that the youth and conceptual complexity of parallel computing necessitate simplified and clean descriptions of systems, in order to be able to discover and apply general principles and methods without excessive complicating "implementational" baggage. This is done by defining *models* at certain levels of abstraction, and designing and analyzing algorithms with respect to them. Although simplicity is of prime importance in a model, in order to be useful in the real world a model must allow the efficient usage of (some) realistic parallel computer system. This effectively means that a "good" (fast, efficient) algorithm for the model should translate to a"good" algorithm on a system; if this holds generally we could say that the model is reflective of the system. The simultaneous achievement of simplicity and reflectivity can be difficult, and the relative balance between these two properties in a model is the subject of much debate.

In this paper a model is presented that we believe achieves a good balance between simplicity of usage and reflectivity of realistic systems. Simplicity is attained through implicit hierarchical organizations, synchronous communication, and the use of the PRAM model as a sub-model. Reflectivity is achieved by accounting for communication and synchronization costs, and these can be controlled through the representation of a generalized form of locality. The model is defined in Section 2. Since parallel computing is a rather unorganized field, with conflicting and sometimes vague terms, it is necessary to first lay some groundwork, which we do in the remainder of this section.

1.1 Types of models

A "computing model", in the general sense, is a simplified and clean description of a computer. Models can differ not only in the specifics of their definitions, but also by the levels of abstraction away from the hardware that they reside in, which is what we are concerned with in this section. In sequential computing, with its standardized von Neumann architecture,

existing models differ *only* by their level of abstraction; this combined with the relative simplicity of sequential computing (to parallel computing) has made it unnecessary to explicitly classify types of models. In parallel computing, though, the inherent complexity of the field along with the existence of a wide range of architectures, and also models which are not (yet) buildable as architectures, makes it seem important to at least loosely classify models by level of abstraction. This would negate any possible confusion about purposes of models, and make explicit the translations a high level user program would need in order to execute on the low level hardware of some computer. Ideally, multiple compilations could be done, with each compilation mapping a program for a model in a higher class to a model in the next lowest class.

In this paper we use four basic types of models: *machine, architectural, computational,* and *programming.* The *machine model* is at the lowest level of abstraction. The description of the computer by this type of model consists of the hardware and operating system. Assembly language programming could be considered as using the machine model. Computer architectures are abstractions of the hardware and operating system, and are thus models in themselves. Thus the *architectural model* is at the next higher level of abstraction over the machine model. The architectural model of a parallel computer describes the interconnection network (thus how to perform communication, but not its implementation details), whether the computer is synchronous or asynchronous, SIMD or MIMD, and whether the *physical* memory of the computer consists of one large bank (shared memory) or the union of the local memories of the processors (distributed memory).

The *computational model* (or *model of computation*) is at the next higher level of abstraction over the architectural model. Ideally, a model of computation provides an abstract view of a class of computers while accurately reflecting costs and resources of those computers. It should be simple and general enough to make the design and analysis of algorithms, and the proving of lower bounds for problems, relatively easy. At the same time, complexity results of algorithms on the model should sufficiently predict their performance on computers. The RAM (Random Access Machine) [AHU] is a widely accepted model of sequential computation which satisfies these requirements with respect to the von Neumann architectural model.

The computational model is usually used in theoretical work, and is sometimes called an "abstract" or "formal" model corresponding to an architectural model. As this suggests, there has traditionally been (in sequential computation) a very close relationship between the computational and architectural models, to the point of considering them the same thing

except that the computational model has defined cost functions for the purpose of algorithm analysis. Although they may be considered equivalent, they need not and should not be. For example, the PRAM model of parallel computation is not physically implementable as an architectural model, and there has been much interest in simulating (mapping) the PRAM on realistic architectures.

The *programming model* is at the next higher level of abstraction over the computational model. We define this type of model to describe the computer in terms of the semantics of a particular implementation of a programming language, i.e. a formalization of a language. One key difference between a programming model and a computational model is that a computational model typically describes memory as a sequence of memory locations, while a programming model will describe it in terms of (possibly elaborate) data structures.

The current state of parallel computing consists of programming models mapped to (implemented on) architectural models, which results in programs being architecture dependent, and the existence of a nice model of computation (the PRAM) which is not reflective of realistic architectures. Additionally, programming directly on architectural models is quite complicated. What is needed is computational models possessing the properties of simplicity and reflectivity of realistic architectural models, that programming models can be mapped to.

1.2 Models of computation

In parallel computing, the PRAM (Parallel RAM) ([FW] [S2], among others) has become quite established as an ideal model of parallel computation; the evidence is the large body of work which now exists on parallel algorithm design and analysis with respect to it. The reason for this is that the PRAM provides an abstraction that removes many of the complications which make the programming of parallel computers so much more difficult than the programming of sequential ones. This allows algorithm designers to concentrate on the logical structure of a parallel computation. The PRAM provides a very simple basis upon which to design, analyze, and compare algorithms, and has proven to be a very useful vehicle for studying the power and limitations of parallel computing (see [KR] and [EG] for surveys of results). Many organizational strategies in algorithm design have been discovered because of the PRAM's simplicity.

The PRAM consists of a collection of sequential processors (RAMs), each with their own bounded (usually small) private memory, and each of which can access any location of a global shared memory in unit time. Processors operate synchronously (but may run

different programs); in any one step a processor may read from shared memory into its private memory, write into shared memory from its private memory, or perform a computation on operands in private memory, storing the result in its private memory. Each step (consisting of a read, a write or a compute) is assumed to take unit time. Different variants of the PRAM can be defined based upon how concurrent writes and/or reads to the same shared memory location are handled (EREW, CREW, CRCW with various methods of resolving write conflicts), and by adding read-modify-write (Fetch-&-Op) capabilities (strong PRAM).

Although it provides an ideal and simple view to the programmer, the PRAM has been criticized as being too far away from realistic architectures, in that it does not accurately reflect costs and resources. Thus there is no close relationship between the PRAM and realistic parallel architectural models like there exists between the RAM and von Neumann architectures in the sequential domain. In fact, one of the reasons that the PRAM has become established despite its distance from "reality" is that no standard parallel counterpart of the von Neumann model has emerged (to serve as a basis for comparison of algorithms); there is a great variety of existing and proposed architectures. An architectural model is of paramount importance as a "meeting place" for hardware and software designers, and has thus been called a "bridging model" by Valiant. Recently in [V2] he claims that what is required before general purpose parallel computation can succeed, as sequential computation has done, is the adoption of a standard bridging model; he then goes on to introduce one based on preservation of efficiency when mapping PRAM-like programs to hardware. Snyder [S3] has also discussed these issues in the context of "type architectures".

Regardless of the details of architectural models, a computational model still needs to accurately reflect costs and resources, so we can't necessarily blame the PRAM's distance from reality on lack of adequate architectural models. If there were a sufficiently close relationship between the PRAM and an architectural model, then it would be possible to address these issues in the context of the architectural model (i.e. in the accurate reflection of the underlying machine model). Still, a PRAM model more reflective of reality would only be beneficial. First, it would not require the architectural model to be so powerful, and thus possibly expensive. Second, it can only increase the retainment of the "general performance" of a PRAM program when it is mapped onto the underlying architecture. Third, there is no escaping the variety of existing and proposed architectures.

Because of the attractiveness of the PRAM's conceptual simplicity, considerable research has been performed to bring the model closer to realistic parallel computers. Three primary issues arise here: communication, synchronization, and memory granularity (or pipelining).

Communication is a cost in parallel computers that the PRAM does not account for. The weakest P processor PRAM (the EREW variant) allows any P distinct memory locations to be accessed simultaneously. This implies an unbounded fan-in of communication links to individual memory locations, and is clearly impractical. In reality, system memory is partitioned into memory modules which can only service a constant number of access requests per memory cycle. The memory modules may exist as physical banks (shared memory) or as local memories of processors (distributed memory). There has been extensive work on simulating the shared memory abstraction of PRAMs on architectures with bounded fan-in to memory modules, both probabilistically [DM] [KRS] [LPP1] [R1] [V2] [KU] and deterministically [LPP2] [H] [HP] [HB] [HGC] [AHMP] [UW]. However, there is another aspect of communication that no PRAM simulation can overcome: the *communication delay*, or *latency*, of an architecture. The latency of an architecture is typically the diameter of its interconnection network; many global memory accesses will need to traverse the full diameter of the network. Thus the latency of an architecture serves as a lower bound for any PRAM simulation on it. The significance of latency has resulted in recent work on PRAM variants which take it into account [ACS1] [PY] [PU2]. The thrust of this work is to use cheaper local computation as much as possible in order to limit more expensive communication.

Synchronization is a cost in parallel computers that the PRAM does not account for. The PRAM is synchronous, i.e. after every step on it there is a "free" synchronization step over all of the processors. While the assumption of free synchronization may be valid when the reflected underlying architecture is SIMD, in general synchronization has a significant cost (it is at least of the order of the latency, and may be up to a logarithmic factor greater). Additionally, the real world, and thus many general applications by definition are (naturally) asynchronous; to force synchrony where it is not needed or wanted is wasteful. There has been much recent work on incorporating asynchrony into PRAMs [KRS] [MSP] [G1] [G2] [CZ1] [CZ2] [N].

Memory granularity is a possible resource in parallel computers that the PRAM does not represent. In some architectures there is a significant cost, typically the latency, for a processor to access a word in memory, but after that subsequent words can be accessed in unit time. Let ℓ be the latency of an architecture, and c be a constant representing the ratio between the time taken by a "communication operation" and the time taken by a "computation operation" in an architecture. Then the architecture has memory granularity g if $\leq g$ *contiguous* words in memory can be accessed in time $\ell + c \cdot g$. Typically, granularity is correlated with latency in architecture design, so that g contiguous words can be accessed

in time $O(\ell)$. Conceptually, granularity g means that information can be transferred in the architecture in blocks of size g or less. In contrast, if g words are accessed by a processor in a PRAM algorithm individually (as it must), then on the underlying architecture it will take time ℓg. In [ACS2] a PRAM variant (BPRAM) is introduced that can employ "block transfers" and thus provides a representation of the resource of granularity. Memory granularity may be defined without the restriction that words be in contiguous memory locations. Some architectures may be capable of performing g *arbitrary* memory references so that they complete in time $\ell + c \cdot g$; Gibbons' asynchronous PRAM model [G1] [G2] is an example of a model that represents this resource. Memory granularity has been referred to as the ability to *pipeline* memory references (e.g. *block pipelining* or *arbitrary pipelining*, according to whether there is a restriction that words accessed be in contiguous memory locations or not).

Valiant [V2] [V3] has addressed the above issues in a unified framework, only for an alternate purpose of designing architectural models that can "support" the PRAM abstraction.

The danger of adding constraints to a model of computation in order to make it more reflective of real architectures is that it may result in the sacrifice of the abstract properties of the model that have established it as a good model in the first place. The PRAM provides a conceptually simple and general view, but does not accurately reflect costs and resources. A model that accurately reflects costs and resources but is not conceptually simple and general would not be much better. For example, one of the two properties of the PRAM which lend it its "ideal" abstract view is its assumption of synchronous execution (the other property is unit time communication to any memory location). Thus it could be claimed that by incorporating asynchrony into the PRAM we are sacrificing half of its advantages. It is well known how difficult asynchronous programming is. This argument is stated in the general sense; it does not hold when the model is meant for a specific class of architectures which must be programmed asynchronously. This is the case for the above-mentioned work on asynchronous PRAMs; it is targeted towards existing shared memory MIMD architectures.

Another negative consequence of modifying the PRAM model too much is that the large body of work that has been done in respect to it may no longer be applicable to the new model. It is likely that new algorithms have to be designed for problems for which (efficient) PRAM algorithms already exist, in order to efficiently make use of the new model; this situation occurs for example in the asynchronous PRAM work referenced above. Ideally, we would like to build on past results.

It can be argued that, at least at the present time, simplicity of usage is a more impor-

tant property in a computational model than (perfect) reflectivity of realistic architectures. This follows from the fact that the largest "consumers" of parallel computers will not be computer scientists, but those from other science and engineering fields. Parallel computing will not become widespread unless models are sufficiently simple for non-computer scientists to use productively. This suggests the path of defining a computational model that retains the PRAM abstraction and finding a means, other than adding (complicating) constraints, to make it more reflective of realistic architectural models, but not insisting on perfect reflectivity. Perfect reflectivity of "reality" is not such a good idea anyway, one reason is that reality changes as technology advances.

1.3 Overview of the paper

In this paper we introduce a model which accounts for and provides control over communication and synchronization costs an abstract sense (as a computational model should) and which, instead of changing the properties of the PRAM, takes the opposite direction and makes the PRAM a sub-model. It also simultaneously controls conceptual complexity via the use of hierarchy. Hierarchy is well-established in computing to control conceptual complexity [M1]; here we employ it as a method of organizing asynchrony. Although the model is designed in an ideal sense, i.e. without considering its relation to any *specific* architectures, it turns out that it is reflective of a variety of existing and proposed architectural models. We also discuss possible relationships between the model and parallel programming languages (programming models).

The Hierarchical PRAM (H-PRAM) model is defined and discussed in Section 2; comparisons with other relevant computational models is also done there. In Section 3 we discuss the model's relationships to existing and proposed architectures. Relationships to overlaying programming models are discussed in Section 4, and Section 5 concludes the paper.

The companion paper [HR] provides an introduction to the use of the H-PRAM through algorithms for computing a complete binary tree, and gives a case study of the model through various algorithms for computing the FFT/butterfly graph.

2 The Hierarchical PRAM model

2.1 Definition

The Hierarchical PRAM (H-PRAM) model consists of a dynamically configurable hierarchy of synchronous PRAMs, or equivalently, a collection of individual synchronous PRAMs that operate asynchronously from each other. A *hierarchy relation* defines the organization of synchronization between independent PRAMs. We treat the hierarchy in a top-down way. Suppose we have a P-processor H-PRAM. At level 1 of the hierarchy, which is the entry point of an algorithm, or the view of the model before an algorithm executes its first step, we have a single P-processor synchronous PRAM. Allowable instructions are any PRAM instruction and one additional instruction: the partition instruction, which adds a controlled form of asynchrony to the model. A partition step in an algorithm partitions the P processors into disjoint subsets and assigns a synchronous PRAM algorithm to execute on each of them. Each subset of processors is a synchronous PRAM operating separately (asynchronously) from the others (the extent of the shared memory "belonging" to each "sub-PRAM" is defined below). This is seen as level 2 of the hierarchy. The hierarchy relation defines how the individual sub-PRAMs, or more specifically the PRAM algorithms running on them, relate to each other, i.e. synchronize with each other.

In this paper we define the hierarchy relation to be that of recursive decomposition, or a tree. Other relations are possible; see [M1] for a hierarchy relation based on condensation of cyclic subgraphs.

With a tree hierarchy relation, each individual sub-PRAM in level 2 runs its algorithm totally asynchronously from the others. When the algorithm terminates, the sub-PRAM waits until all of the algorithms on the other sub-PRAMs terminate, i.e. the individual sub-PRAMs synchronize with each other. We consider this to be implemented by a synchronization step on the sub-PRAMs which follows every partition step. When the synchronization step on the sub-PRAMs finishes, the next step of the P-processor PRAM algorithm at level 1 begins. Thus the time complexity of a partition step is the maximum of the times of the PRAM algorithms running on the individual sub-PRAMs. The non-partition steps of the level 1 PRAM algorithm (computation and communication steps on P processors) are synchronous by definition; we consider this to be implemented by a synchronization step on P processors which follows every non-partition step.

Each of the algorithms in level 2 is designed with a parameterized number of processors, so from its vantage point it is at level 1 of a hierarchy. Partition steps can be used here to

create level 3 of the hierarchy. This process may continue recursively until partitioning is no longer possible, i.e. when we are running a PRAM algorithm on a 1-processor PRAM. Thus, an H-PRAM algorithm consists of a hierarchy relation and a collection of synchronous PRAM sub-algorithms, each augmented with the partition instruction (except possibly those that operate at the bottom level of a hierarchy). Since each sub-algorithm applies to a two-level hierarchy (or one-level, if there are no partition instructions in it), the H-PRAM can be structurally defined as a recursive two-level hierarchy. In view of this, we often speak of a level λ synchronous PRAM augmented with the partition instruction (and/or the associated algorithm), and its level $\lambda + 1$ sub-PRAMs (and/or their associated algorithms). The H-PRAM, and the H-PRAM algorithm, is what we have when $\lambda = 1$.

In order to define the form of the partition instruction, consider a P'-processor (sub)-PRAM at some arbitrary level in a hierarchy. A partition instruction in its associated algorithm has the form

partition { P_1: *Algorithm-1(parameter-list)*;
P_2: *Algorithm-2(parameter-list)*;
...
P_Q: *Algorithm-Q(parameter-list)* }

where Q is the number of sub-PRAMs created, P_i is the number of processors in the ith sub-PRAM, *Algorithm-i* is the algorithm assigned to the ith sub-PRAM, $1 \leq i \leq Q$, and

$$\sum_{i=1}^{Q} P_i = P'$$

Intuitively, the partition instruction acts like a call of Q procedures in parallel.

Very often in H-PRAM algorithm design, the partition instruction is used to create a number of sub-PRAMs, each of which has the same number of processors and the same algorithm (with the same parameter list) assigned to it. In this case we use an abbreviated form of the partition instruction:

partition$\{Q, P_i,$ *Algorithm(parameter-list)*$\}$

where Q is the number of sub-PRAMs created, P_i is the number of processors in each of them, and $Q \cdot P_i = P'$. *Algorithm* is the name of the algorithm assigned to each sub-PRAM.

Any sub-PRAM sub-algorithm in the H-PRAM can be seen as a PRAM algorithm by simply viewing any partition instruction as a procedure/function call.

At this point an example is called for. Assume that P is the number of H-PRAM processors and that $P \geq 8$. The H-PRAM algorithm Alg1(P) consists of five sub-algorithms, and does nothing except create a hierarchy for demonstration purposes. The sub-algorithms Alg2(P'), Alg3(P'), Alg4(P'), and Alg5(P') do absolutely nothing (except exit the instance they are invoked). The *sub*-algorithm Alg1(P') is given below.

```
Alg1(P')
    if P' ≥ 16 then
        partition{2, P'/2, Alg1(P'/2)}
    else
        < Comment: P' = 8 >
        partition{
            P'/2: Alg2(P'/2);
            P'/4: Alg3(P'/4);
            P'/8: Alg4(P'/8);
            P'/8: Alg5(P'/8)}
    end-if-then-else
end-Alg1
```

We often give an H-PRAM algorithm the name of the sub-algorithm that runs at level 1 of the hierarchy. The algorithm is invoked by Alg1(P). Figure 1 shows the the hierarchy created by this algorithm when $P = 16$; each box in each level represents a sub-PRAM in that level, and the sub-algorithms running on the sub-PRAMs are noted in the boxes.

We define two variants of the H-PRAM:

- **Non-uniform H-PRAM**: The partition instruction partitions the shared memory along with the processors, such that each sub-PRAM has its own "private" block of shared memory *disjoint from* the shared memories of the other sub-PRAMs. If a level $\lambda + 1$ sub-PRAM has $1/x$ of the processors of its level λ PRAM, then the size of its shared memory is $1/x$ of the size of the level λ PRAM's shared memory.

- **Uniform H-PRAM**: The partition instruction does *not* partition the shared memory along with the processors. Each sub-PRAM in a hierarchy has the H-PRAM's entire

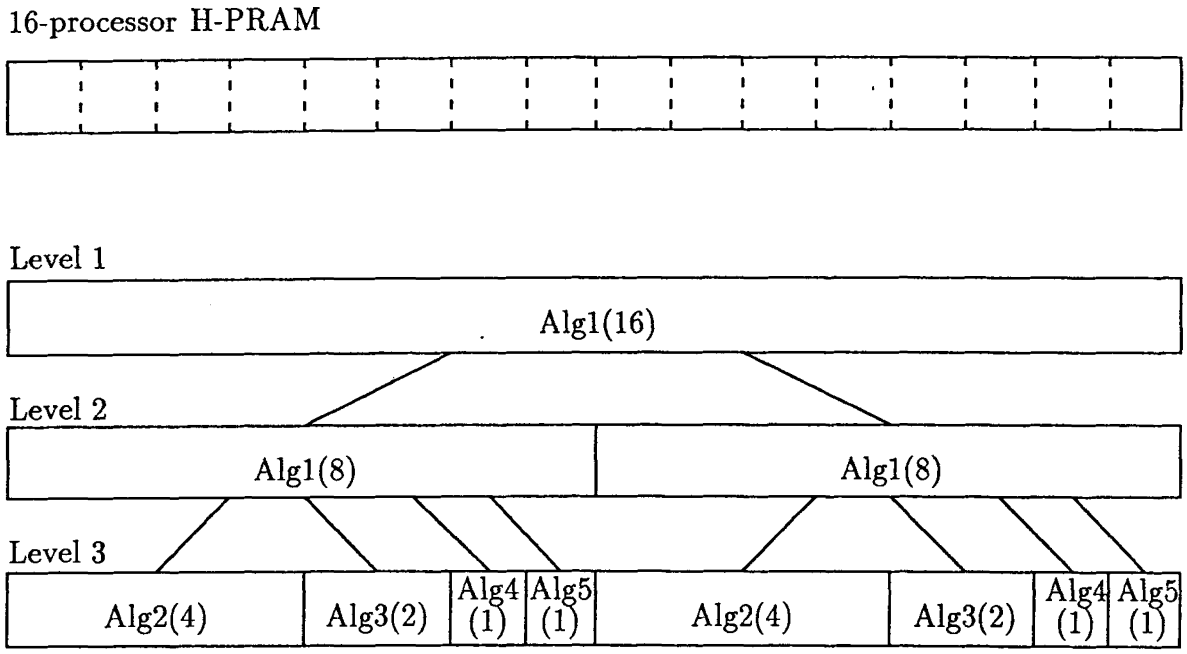

Figure 1: Hierarchy arising from Alg1(16)

shared memory (the shared memory seen at level 1 of a hierarchy) as "its" shared memory, i.e. the sub-PRAMs share the global shared memory.

The names "non-uniform" and "uniform" refer to the cost (time) of accessing a shared memory location. In the uniform H-PRAM any memory access by any sub-PRAM takes the same time (proportional to the "size" of the entire H-PRAM), while in the non-uniform H-PRAM a memory access by a sub-PRAM takes time proportional to the "size" of that sub-PRAM. Sections 2.2 and 2.3 discuss the concepts behind and implications of these definitions.

The PRAM sub-models of the H-PRAM may be any of the common variants *except* asynchronous PRAMs, e.g. EREW, CREW, CRCW, strong, LPRAM [ACS1], BPRAM [ACS2]; or PRAMs augmented with scan [B1] or multiprefix [R2] unit-time operations. We do not restrict all of the sub-models to be the same variant.

The use of the BPRAM as a sub-model implies block pipelining capability. Note that in block pipelining, only one *instruction* to access memory is outstanding at a time, but in arbitrary pipelining, as many instructions as there are memory references being pipelined are outstanding at a time [G2]. Essentially, the H-PRAM represents the resource of memory granularity by passing along responsibility for it to the sub-models. (However, it may be

that this resource in a parallel computer will be "consumed" by an architectural model for the purpose of supporting the efficient mapping of the overlaying H-PRAM computational model to it; if so, there will be no pipelining capability in the H-PRAM. We return to this in Section 3.)

Because the model is meant to be reflective of reality, it is parameterized by latency, synchronization cost, and memory granularity. The first two are functions of the number of processors being communicated amongst and synchronized, respectively; the functions are defined by the specific underlying architecture. The last is a function of latency and a quantity g defined by an architecture. These parameters allow H-PRAM algorithm design and analysis to be reflective of the costs and resources of realistic parallel architectures, as shall be seen.

Partition steps can be viewed as compound instructions on "super-processors", thus giving the appearance of lock-step execution. In other words, each level $\lambda + 1$ sub-PRAM involved in a partition step runs an algorithm that performs an operation; from the level λ vantage point it is as if (possibly different) operations are done in parallel on disjoint sets of processors, or disjoint super-processors. The operations do not take generally take unit time like a PRAM step, so they are seen as compound steps. Because there are two types of steps, unit-time PRAM steps on processors and compound steps on super-processors, there are two types of synchronization, *α-synchronization* on the processors of a PRAM step and *β-synchronization* on the super-processors of a compound step (this will be explained more fully below).

2.2 Discussion

The purpose in designing the H-PRAM was to obtain a flexible model that is more reflective of realistic parallel architectures than the PRAM, while retaining its simplicity. This can be reformulated as accounting for and providing control over four types of complexity simultaneously: *computation*, *communication*, *synchronization*, and *conceptual*. The first three are the fractions of the total time taken by an algorithm which are devoted to performing local computation, communication, and synchronization, respectively. These are measurable costs in parallel computation; we discuss how the H-PRAM represents and gives control of them towards the end of this section and in Section 2.3. The majority of this section discusses how the H-PRAM controls conceptual complexity (difficulty of programming) arising from asynchrony and communication.

As mentioned in the introduction, synchronization is costly and potentially wasteful (in

that real-world problems, and thus algorithms that solve them, have naturally asynchronous functional definitions), thus the H-PRAM admits asynchrony. There are two types of asynchrony in parallel computation, which we term *control asynchrony* (or asynchronous control) and *communication asynchrony* (or asynchronous communication).

Control asynchrony exists where different processes (sub-PRAMs in our case) execute separately from each other, but wait for each other at logical points in their individual algorithms. These logical points are usually called synchronization points for the participating processes. It can be quite difficult to explicitly organize control asynchrony, keeping track of when and where synchronization points occur, and among which sets of processes. The H-PRAM allows asynchronous control of multiple synchronous PRAM algorithms, organizing the control structure as a hierarchy. The model constrains conceptual complexity in the face of control asynchrony since synchronization between individual sub-PRAMs is *implicit*; the structure of an H-PRAM algorithm implies the organization of control via the hierarchy relation. It allows explicit management of control granularity; for example, a coarse grained computation can be obtained by using many sub-PRAMs running non-constant time algorithms, and a fine grained computation by very few sub-PRAMs running constant time algorithms (or no sub-PRAMs, i.e. no partition step, in which case we have one synchronous PRAM).

The other type of asynchrony that complicates asynchronous programming is communication asynchrony. Communication between asynchronously executing processes causes even more problems than control asynchrony. Programs become *indeterminate*, where different runs on the same set of data may result in different execution paths. We say that two processes (or processors) are *communicating asynchronously* if they are operating asynchronously from each other and each accesses the same shared memory location, where

1. one access is a write and the other is a read, or

2. each access is a write and the two values being written are different.

Since the timing of asynchronously operating processes is independent of each other and they may have different (and varying) relative speeds, there is no way to predict the ordering of the communication operations, i.e. values read or written in a particular instruction sequence may be different from run to run. Programmers must explicitly coordinate asynchronous communication operations in order to prevent this from happening, a very difficult task. This requires the existence of additional synchronization primitives (e.g. busy-waiting, arbitration) in the computer system, along with their corresponding (significant) costs. Since

programs are correct only if they work regardless of process speeds, it is almost impossible to prove correctness in the face of this indeterminacy. Program testing cannot guarantee that programs are error-free; debuggers must be left on at all times since a particular "bad" sequence of communication operations may not occur for thousands or more runs [G2]. This is infeasible since debuggers slow down the execution of programs.

As stated above, the H-PRAM uses hierarchy to control the conceptual complexity of control asynchrony. It *may or may not* use hierarchy to eliminate asynchronous communication, and thus the indeterminacy arising from it, depending on the variant: the *non-uniform* H-PRAM employs a "communication hierarchy" (or hierarchical memory), while the *uniform* H-PRAM does not. Recall that in the non-uniform variant the shared memory is partitioned along with the processors, such that each sub-PRAM created by a partition step consists of a disjoint set of processors and a corresponding disjoint block of shared memory that is private to it. No sub-PRAM algorithm may access a memory location belonging to the private memory of another sub-PRAM. Thus there is no asynchronous communication since each sub-PRAM synchronously operates on its private block of shared memory. This means that, assuming all synchronous sub-PRAM algorithms are determinate, any non-uniform H-PRAM algorithm is determinate; a single successful run of the algorithm on a particular input serves as a proof of correctness for that input. If there is an error in a program run, the error can be reproduced in future runs on the same input. A debugger need only be turned on when required. (Note that a sub-PRAM in the H-PRAM model corresponds to a process in the discussion of the previous paragraph.)

Indeterminacy can only arise in a synchronous PRAM algorithm if a read from and a write to the same memory location is done in the same step, or if a concurrent write of different values to the same memory location is done in the same step where the resultant written value may be different from run to run. Since a read and a write on the same location in the same step is disallowed in all PRAM variants, the only place where indeterminacy can occur is in the ARBITRARY CRCW PRAM variant, where the resultant written value is chosen randomly. This means that an ARBITRARY CRCW PRAM algorithm must be proven correct regardless of resultant written values, or that the underlying architecture must provide a primitive (black box routine) to "choose pseudo-randomly" the same value be written in different runs of the algorithm.

Another way of seeing how the non-uniform H-PRAM disallows asynchronous communication is to note that it enforces the paradigm that any two processors which are operating asynchronously from each other, and want to communicate with each other, must synchro-

nize with each other between the time one accesses the memory location being communicated through, and the time the other one does. Consider a partition step in a level λ PRAM algorithm which creates a number of level $\lambda + 1$ sub-PRAMs. Assume we have two processors in different sub-PRAMs (thus the processors are operating asynchronously from another) that want to communicate through a shared memory location which "belongs" to one of the sub-PRAMs. The "communication operation" consists of two successive accesses to this memory location. The only way for it to occur is for the processor in the sub-PRAM "owning" the memory location to make its access, then for all sub-PRAM algorithms to terminate, thus finishing the partition step in the level λ PRAM algorithm; and finally for the other processor to perform its access to the memory location in the level λ algorithm. Sub-PRAMs synchronize with each other upon the termination of their algorithms, thus a partition step can be seen as immediately followed by a synchronization step on the sub-PRAMs. We call this a β-synchronization step (where an α-synchronization step follows every non-partition step of a synchronous sub-PRAM algorithm, and operates on all processors in that sub-PRAM). The above discussion shows that the non-uniform H-PRAM disallows communication asynchrony by enforcing the existence of a β-synchronization step among any two asynchronously communicating processors (i.e. processors in different sub-PRAMs) between their memory accesses.

In the uniform H-PRAM, shared memory is not partitioned along with the processors; all sub-PRAMs in a hierarchy share the same (global) shared memory. Asynchronous communication, thus indeterminacy, is possible in this variant of the model. However, if an H-PRAM algorithm is written such that there are *always* β-synchronization steps among communicating processors belonging to different sub-PRAMs between the time one accesses the memory location being communicated through, and the time the other one does, then the algorithm will not use asynchronous communication and will thus be determinate. We note that in the uniform variant, a memory location being communicated through does not have to "belong" to either of the sub-PRAMs containing the communicating processors. Thus, instead of enforcing determinate behavior at the computational model level, as the non-uniform H-PRAM does, the responsibility for this is passed on to an overlaying *programming model* at the next highest level of abstraction. What this means is that the programmer is responsible for explicitly organizing communication such that it is not asynchronous (by inserting β-synchronization steps, i.e. by partitioning appropriately), and that the formal programming model being used to do this must provide enforcement (i.e. can find and flag any attempt at asynchronous communication as a program error, and abort the program). We consider

this a requirement for any programming model mapped to the uniform H-PRAM.

Generally, the H-PRAM controls conceptual complexity in the face of asynchrony by providing an implicit hierarchy for organizing control asynchrony, and by disallowing communication asynchrony, either explicitly (the non-uniform variant) or by depending on an overlaying programming model to do so (the uniform variant). The concept of allowing control asynchrony but not communication asynchrony can be seen as a compromise between the flexibility of completely asynchronous models and the simplicity of completely synchronous models. The resulting benefits of determinate computation and lack of need for expensive synchronization primitives should not be underestimated. Gibbons refers to models of this type as "semi-synchronous"; in [G2] he provides an extensive discussion which includes, in part, a new model called the Asynchronous PRAM, "repeatable" (determinate) algorithms for it, and relationships to possible overlaying programming models (as the model does not enforce determinacy itself). Steele [S4] has introduced a programming model based on a hierarchy of processes operating on a shared memory, including dynamic enforcement mechanisms for maintaining determinacy (error checking for asynchronous communication).

The H-PRAM retains the shared memory abstraction of the PRAM, thus retaining its property of simplicity the face of parallel communication (the practicality of this will be addressed below). The non-uniform variant can be seen as embodying a multi-level memory hierarchy, where the access time in a sub-PRAM to its private block of shared memory is a function of the size of the sub-PRAM. While "uniform" in the name uniform H-PRAM suggests that there is a one-level shared memory where all memory accesses by all sub-PRAMs take the same amount of time (which is a function of the size of the entire H-PRAM, regardless of the size of a sub-PRAM), we consider the uniform variant to embody a two-level memory consisting of the global shared memory and memory private to each H-PRAM processor. Clearly an access to private memory by a processor will be much faster than an access to shared memory. The LPRAM [ACS1] and BPRAM [ACS2] each have a two-level memory, where private memory is unbounded in size (although the processors in the PRAM are defined to have private memories they are seen as being small, e.g. consisting of a set of registers).

The other types of complexities mentioned at the beginning of this section were computation, communication, and synchronization; in contrast to conceptual complexity, these are measurable by adding up their total *costs* in an algorithm. The cost of a computation operation is usually defined to take unit time (operands are in a processor's registers). The H-PRAM provides control over communication (in the non-uniform variant) and synchro-

nization costs via the partitioning into smaller subsets of processors, with communication (in the non-uniform variant) and synchronization only occurring between processors of the same subset. The latency and the synchronization cost of an architecture are functions of the number of processors being communicated amongst and synchronized. Thus, the H-PRAM reflects and allows control over costs in an underlying architecture in an abstract sense, as a model of computation should. The property that the H-PRAM assumes of an underlying architecture is that it be "partitionable into contiguous segments", where the word "contiguous" implies that the processors, and memory (in the non-uniform variant), in a segment be close or in the physical neighborhood of each other (a stronger assumption would be that a segment have the same properties as the architecture, only a smaller size). We return to this in Section 3; the point is that the H-PRAM is representative of the situation where costs are functions of the number of processors participating.

The term *communication locality* (or just *locality*) has been used in the literature primarily to describe certain algorithms where processors perform substantially more local computation steps than global communication steps. Since local computation steps take unit time and global communication steps take time Ω(latency), this results in more efficient algorithms, and normally is exploitable only when the input size is significantly larger than the number of processors. It is normally obtained by having processors run sequential algorithms on data in their local memories independently of each other, at certain stages of the algorithm. Since processors running independent sequential algorithms need not synchronize with each other (until the end of the stage), the term locality can be used to refer to synchronization in addition to communication. We call this type of locality *strict* locality. The non-uniform H-PRAM can represent strict communication and synchronization locality, and the uniform H-PRAM strict synchronization locality, by partitioning into 1-processor sub-PRAMs and assigning sequential algorithms to them (in the non-uniform variant, a 1-processor sub-PRAM's private block of shared memory is seen as the processor's local memory; the following subsection shows the validity of this statement).

The H-PRAM can also represent *neighborhood* locality, in which activity (communication, in the non-uniform variant, and synchronization) can be organized into independent neighborhoods (i.e sub-PRAMs). Members (processors, memory locations) of a neighborhood are in close proximity to each other in comparison with proximity to members of other neighborhoods. The costs of neighborhood activity are functions of the size of the neighborhood. Strict locality refers to the special case where neighborhoods have "size 1". We use the term *general locality* to encompass both strict and neighborhood locality. Many problems seem

to submit to natural solutions possessing the property of general locality. General locality has not seemed to be previously researched (although there has been some work on it in the field of distributed computing).

Clearly, the non-uniform variant of the H-PRAM is the most ideal since it provides the tightest control over all four types of complexity simultaneously. There are two basic reasons for the uniform variant. First, it may be more difficult for an architecture to support a partitionable, non-uniform shared memory than a uniform shared memory. Second, there may be difficulties in designing algorithms such that data required by a processor is always present in the shared memory block private to the sub-PRAM the processor belongs to. We will return to these issues later on in the paper.

Because the H-PRAM accounts for and provides control over communication and synchronization costs, it presents a more practical abstract reflection of realities in parallel computing, with a minimal amount of added conceptual complexity, than the PRAM. It shall be seen to be applicable to variety of architectures. It allows general degrees of locality to be employed by algorithm designers. This gives the potential of obtaining algorithms that map more efficiently to architectures than could be done otherwise. Since the PRAM is a sub-model, we can retain direct relevance of established theory of parallel computing, and possibly build on it in a manner of "construction from existing components". It is attractive from a practitioner's point of view because of its support of determinate (thus testable) parallel programming, and because of the methodology of modular construction from sub-algorithms (the traditional way of programming in the sequential domain).

2.3 Costs and complexity analysis

This section defines the types and costs of steps in an H-PRAM algorithm, and the method of analyzing the complexity of an algorithm. Section 2.3.1 provides complete and detailed definitions. Section 2.3.2 defines a streamlined complexity analysis methodology which simplifies H-PRAM usage without loss of accuracy or generality.

2.3.1 Definitions

Consider any "arbitrary" model of computation which can perform three types of steps: computation (on operands in processors' private memories), communication, and synchronization. A computation step is (always) defined to take unit time. In a PRAM, a communication step takes unit time and synchronization is free, i.e. it takes zero time. If an

algorithm analyzed on our "arbitrary" model is to accurately predict performance on an architecture, then its costs must be parameters of the model. The latency and the synchronization cost are functions of the number of processors P being communicated amongst and synchronized; we denote these parameters by $\ell(P)$ and $s(P)$ respectively. The values of these parameters are defined by the architectural model, where the communication and synchronization mechanisms are implemented. Possible values that the latency $\ell(P)$ could take on are the diameter of the network interconnecting P processors in the architecture, e.g. $\log P$ for the hypercube and $\sqrt{P}$ for the mesh, or the cost of implementing (simulating) a communication step of the "arbitrary" model on the architectural model. The latency function $\ell(x)$ is an non-decreasing function of x such that $\ell(x) = O(1)$ *iff* $x = O(1)$. Since $\ell(P)$ is not necessarily the diameter of a P-processor network, we use $\text{diam}(P)$ when diameter is meant. The synchronization cost is typically $\text{diam}(P) \leq s(P) \leq \text{diam}(P) \log P$. We often use the shorter forms of ℓ and s when their arguments are clear from context.

Let T, C, and S denote the number of computation, communication, and synchronization steps in an algorithm on this arbitrary model, respectively. We need to discern between these types of steps in order to properly account for costs, and predict performance on the architecture underlying the model. The *computation complexity* is T (since computation steps are defined to take unit time), the *communication complexity* is $C \cdot \ell$, and the *synchronization complexity* is $S \cdot s$. and the *total complexity* of the algorithm is $T + C \cdot \ell + S \cdot s$. Setting $\ell = 1$, $s = 0$ and $S = T + C$ turns the arbitrary model into a PRAM, and results in a communication complexity of C, a synchronization complexity of zero, thus a total complexity of $T + C$.

Recall that the H-PRAM has an additional type of step, the partition step. We extend the definitions to account for this. The partition step creates a two level (λ and $\lambda + 1$) hierarchy; T, C, and S refer to steps in the level λ algorithm. Because a partition step can be seen as a compound computation step, we denote the number of partition steps in the level λ algorithm as T_p. Let the number of level $\lambda + 1$ sub-PRAMs (or super-processors) created in the ith partition step be Q_i, $1 \leq i \leq T_p$, and index these sub-PRAMs by q, $1 \leq q \leq Q_i$. Different partition steps may have different costs, unlike the other types of steps: recall that a partition step ends when all of the sub-algorithms running on the sub-PRAMs have terminated. Therefore, letting $\tau(q, \lambda + 1)$ be the complexity of the algorithm running on the qth level $\lambda + 1$ sub-PRAM, $1 \leq q \leq Q_i$, of the ith partition step, the *cost* of the ith partition step is $\max\{\tau(q, \lambda+1) | 1 \leq q \leq Q_i\}$. The *partition complexity* of the level λ algorithm is $\sum_{i=1}^{T_p} \max\{\tau(q, \lambda + 1) | 1 \leq q \leq Q_i\}$.

When the sub-algorithms running on the sub-PRAMs of a partition step terminate, they must synchronize before the following level λ step can start. Only one representative processor from each sub-PRAM need participate in this synchronization step (since all processors within a sub-PRAM have synchronized after the final step of the sub-algorithm running on it). Thus there is a different type of synchronization step and a (possibly) different type of synchronization cost. Above we defined S to be the number of synchronization steps in an algorithm, and s to be the cost of one of these steps. In terms of a level λ PRAM in the H-PRAM, we call these *α-synchronization steps* and redenote S as S_α and s as s_α. Note that $S_\alpha = T + C$. We term the synchronization steps following partition steps as *β-synchronization steps* and denote the number of them as S_β (note $S_\beta = T_p$).

The cost of a β-synchronization step is usually a function of the number of sub-PRAMs Q in the partition step being synchronized, but may *also* be a function of the level λ latency, and thus the number of processors, say P', in the PRAM executing the partition step. Therefore we denote the β-synchronization cost as $s_\beta(Q, P')$, or just s_β when usage is clear from context. From the synchronization cost bounds for the "arbitrary" model above, β-synchronization cost in the case that s_β is solely a function of the number of sub-PRAMs being synchronized is typically $\text{diam}(Q) \leq s_\beta(Q, P') \leq \text{diam}(Q) \log Q$. For the case where β-synchronization cost is also a function of P', its bounds are typically $\text{diam}(P') \leq s_\beta(Q, P') \leq \text{diam}(P') \log Q$. We return to β-synchronization in Section 3, where these cost definitions are justified in the context of underlying architectures.

In the non-uniform H-PRAM, the latency within a sub-PRAM is a function of the number of processors in that sub-PRAM, i.e. $\ell = \ell(P')$ where P' is the number of processors in the sub-PRAM, and in the uniform variant, ℓ is fixed and a function of the number of processors in the entire *H-PRAM*, regardless of the size of a sub-PRAM.

Complexities on the H-PRAM are formally defined via recursive functions, because of the hierarchy relation of recursive decomposition (however, in practice this formalism is generally not needed, as shall be seen).

Let P be the number of processors in the H-PRAM. For any P'-processor sub-PRAM q_0 in level λ of the hierarchy, the total complexity of its associated algorithm on an architectural model with latency ℓ, α-synchronization cost s_α, and β-synchronization cost s_β is

$$\begin{aligned}
\tau(q_0, \lambda) &= T + C \cdot \ell + S_\alpha \cdot s_\alpha + \sum_{i=1}^{T_p} \max\{\tau(q, \lambda+1) | 1 \leq q \leq Q_i\} + \sum_{i=1}^{S_\beta} s_\beta(Q_i, P') \\
&= T \cdot (s_\alpha + 1) + C \cdot (\ell + s_\alpha) + \sum_{i=1}^{T_p} (\max\{\tau(q, \lambda+1) | 1 \leq q \leq Q_i\} + s_\beta(Q_i, P'))
\end{aligned}$$

(recall that $S_\alpha = T + C$ and $T_p = S_\beta$), where

- in the *non-uniform* variant: ℓ and s_α are functions of P'.
- in the *uniform* variant: ℓ is a function of P, and s_α is a function of P'.

Let H denote the sub-PRAM at level 1 of the hierarchy, i.e. the PRAM that exists at the entry point of an H-PRAM algorithm, or simply the P-processor H-PRAM. Then the *total complexity* of its associated H-PRAM algorithm is $\tau(H, 1)$, where ℓ and s_α are functions of P.

The recursion in this definition bottoms out when $T_p = 0$, i.e. when there are no partition steps in a level λ sub-algorithm. Intuitively, the definitions state that when analyzing a level λ algorithm, the complexities of the level $\lambda + 1$ sub-algorithms running in the sub-PRAMs of partition steps are already known. These complexities have been determined in the same way, meaning that analysis starts at the deepest level of the hierarchy and proceeds upward.

Note that from a level λ algorithm's vantage point, the complexities of the level $\lambda + 1$ sub-algorithms are *total* complexities, i.e. all communication and synchronization costs have been accounted for by the sub-algorithm analysis. Thus a partition step can be seen as a compound (non-unit time) step consisting of local computation on super-processors, as there is no level λ communication or synchronization "within" such a step (β-synchronization is performed after the step). The larger the super-processors, the faster they are (potentially, through more parallelism); at the same time, the greater the costs of communication and synchronization overhead in the sub-algorithms running on them. This is another way of looking at the concept of general locality.

In the literature, latency $\ell(P)$ has normally been correlated with the diameter of a network interconnecting P processors. Alternatively, it could take the value of the cost of simulating a communication step of a PRAM algorithm on a distributed memory architecture, since this is the true "cost of communication" when PRAMs are employed as computational models in this case. Network diameter is a function of P, and lower bounds any PRAM simulation. The cost of simulating a PRAM communication step is also a function of P, so ℓ remains a function of P regardless of whether it represents diameter or simulation overhead (however, if a simulating architecture and a simulated PRAM have different numbers of processors, then ℓ will be a function of both quantities). The non-uniform H-PRAM model allows for the control of communication cost via the simulation of multiple "smaller" sub-PRAMs on sub-networks of the underlying architecture. In fact, the non-uniform variant seems to relate best to physically distributed memory architectures that "implement" shared

memory as the union of its processor's local memories, since an architecture's memory partitions naturally along with its processors. The uniform variant is also suitable; memory accesses are just allowed to go outside of the sub-network that a sub-PRAM is simulated on. The details of PRAM simulation, if used, are part of the architectural model; we return to this in Section 3. The point here is that the latency is a *parameter* representing the cost of an H-PRAM communication step; it is most accurate when a mapping of the H-PRAM to an architecture defines it as the full cost of "implementing" the communication step on the architecture.

2.3.2 Simplifications

It is well known that too many parameters in a model of computation can make it hard to use. The difficulty of having to consider multiple concepts (parameters) simultaneously, and their interactions, tends to cloud up otherwise discernible properties and patterns in algorithm design.

The previous subsection defined three parameters $(\ell, s_\alpha, s_\beta)$ and their usage for complexity analysis on the H-PRAM (we also have the additional obvious parameters of number of H-PRAM processors P and input size N). These definitions were necessary for the sake of rigor. In this subsection we define a streamlined complexity analysis procedure which simplifies H-PRAM usage by reducing the number of parameters that need be considered at once.

Consider an arbitrary P'-processor (sub)-PRAM in a hierarchy, and let R denote its corresponding algorithm's partition and β-synchronization complexity:

$$R = \sum_{i=1}^{T_p} \max\{\tau(q, \lambda + 1) | 1 \leq q \leq Q_i\} + \sum_{i=1}^{S_\beta} s_\beta(Q_i, P')$$

Then the algorithm's complexity as defined in the previous subsection is

$$T + T \cdot s_\alpha + C \cdot (\ell + s_\alpha) + R$$

where s_α is a function of P', and ℓ is a function of P' in the non-uniform variant and of P in the uniform variant. This definition charges the α-synchronization cost s_α to every one of the the T computation steps (i.e. there is an α-synchronization step following every computation step).

The first simplification can be arrived at by requiring that, in any one step, all (active) processors must execute a computation step or all (active) processors must execute a communication step. This is very reasonable since most, if not all, PRAM algorithms are SIMD

anyway (in any one step all active processors execute the same instruction, not just the same type of instruction). If an algorithm is MIMD, we can break a step into two sub-steps, the first in which all (active) processors with computation instructions execute these instructions, and the second in which all (active) processors with communication instructions execute them.

Define a *computation-phase step* to be a "longest sequence" of *consecutive* computation steps. Thus a computation-phase step is immediately preceded by a communication step or the entry point of the algorithm (i.e. it is the first step in the algorithm), and is immediately followed by a communication step or the exit point of the algorithm (i.e. it is the last step in the algorithm). Note that when all processors are running a sequence of local computations (a computation-phase step), it is unnecessary for them to α-synchronize with each other with each other until the end of the computation-phase step. Furthermore, it is a reasonable assumption that an architecture would not force them to do so. If the information to turn off synchronization within a computation-phase step is not available to a compiler, an architecture can be directed to only synchronize between any two steps which are not both computation steps.

Therefore, letting T' be the number of computation-phase steps in a sub-PRAM algorithm, we need only charge α-synchronization to T' computation steps rather than T. This changes the algorithm complexity to

$$T + T' \cdot s_\alpha + C \cdot (\ell + s_\alpha) + R$$

Clearly $T' \leq C + 1 = O(C)$ and most likely we will have $T' \leq C$ (since most algorithms consist of repeated computation-then-communication or communication-then-computation stages), so we make the assumption that $T' \leq C$ with the confidence that little, if any, accuracy will be lost. This simplifies the complexity to be

$$\leq T + C \cdot (\ell + 2s_\alpha) + R$$

We need to consider the parameters on the cost functions now. Recall that $s_\alpha = s_\alpha(P')$ on both the non-uniform and uniform H-PRAM variants, $\ell = \ell(P')$ on the uniform variant, and $\ell = \ell(P)$ on the uniform variant. Also recall that $s_\alpha(P')$ is lower bounded by diam(P'), and is at most a logarithmic (in P') factor greater than diam(P'). However, it is difficult to believe that, because of the importance of synchronization, any architecture will be built without special support for implementing barrier synchronization of P' processors at the (order of the) diameter diam(P') of the network interconnecting P' processors in the architecture

(Section 3 addresses this further). Remember that $\ell(P') \geq \text{diam}(P')$. Therefore, on the uniform variant we have

$$\begin{aligned} \ell + 2s_\alpha &= \ell(P) + 2s_\alpha(P') \\ &= \ell(P) + O(\text{diam}(P')) \\ &\leq \ell(P) + O(\ell(P')) \\ &= O(\ell(P)) \end{aligned}$$

and on the non-uniform variant we have

$$\begin{aligned} \ell + 2s_\alpha &= \ell(P') + 2s_\alpha(P') \\ &= \ell(P') + O(\text{diam}(P')) \\ &= O(\ell(P')) \end{aligned}$$

For either variant, the complexity becomes

$$O(T + C \cdot \ell + R)$$

which is natural and intuitive (charging latency cost alone to communication steps alone). If in fact $s_\alpha > \ell$ in an architecture, we could simply use s_α rather than ℓ. Either way, little accuracy is lost due to the small implied constant in the "big Oh" notation.

As it stands now, complexity analysis employs two parameters, ℓ and s_β (which is part of the term denoted by R; see above). We can simplify further by defining a two-stage complexity analysis procedure so that only one parameter is under consideration at one time:

1. determine the H-PRAM algorithm complexity, ignoring β-synchronization costs.

2. determine the total cost of β-synchronization in the H-PRAM algorithm.

The total complexity of the the H-PRAM algorithm is the sum of the quantities obtained in each of these two stages. The first stage consists of counting the numbers of computation and communication steps of the sub-PRAM algorithms in the hierarchy, charging a cost of $\ell(P')$ (non-uniform variant) or $\ell(P)$ (uniform variant) for each communication step in a P'-processor sub-PRAM, and adding on the partition complexities of the sub-PRAM algorithms (which have already been determined as analysis proceeds from bottom-up in the hierarchy). The second stage usually just consists of a check to make sure that the total cost of β-synchronization does not dominate the complexity of the first stage analysis.

The separation of β-synchronization is probably the proper thing to do anyway, as partitionable architectures would likely have special-purpose mechanisms to do β-synchronization.

In practice, analysis is even simpler since the recursion in the complexity definitions is not needed. Algorithms will have regular structures, such that each level of a hierarchy consists of sub-PRAMs of the same size running the same algorithm. In this case we can analyze the complexity of each level (one sub-PRAM algorithm per level) and sum the complexities over all levels in the hierarchy. For example, consider an H-PRAM algorithm that employs a hierarchy of L levels, where all sub-PRAM algorithms have one partition instruction (except at the bottom level L). Let T_λ, C_λ, ℓ_λ, and $(s_\beta)_\lambda$ be the number of computation steps, number of communication steps, latency, and β-synchronization cost, respectively, in the (identical) sub-PRAM algorithms in level λ of the hierarchy. Then the first stage complexity is

$$\sum_{\lambda=1}^{L}(T_\lambda + C_\lambda \cdot \ell_\lambda)$$

the second stage complexity is

$$\sum_{\lambda=1}^{L-1}(s_\beta)_\lambda$$

and the total complexity is the sum of these.

The companion paper [HR] demonstrates the design and analysis (using the techniques of this subsection) of some H-PRAM algorithms.

2.4 Comparisons to other models

Clearly, if an H-PRAM algorithm uses no partition steps, then it is a valid PRAM algorithm. Thus the H-PRAM subsumes the PRAM in that the PRAM is an instance, or configuration, of the H-PRAM (where $\ell = 1$ and $s_\alpha = 0$).

In the introduction, we briefly discussed various approaches to making the PRAM more realistic as a computational model. These approaches generally involve three main issues: communication, synchronization, and memory granularity (or pipelining). In this section we compare the H-PRAM to related models under these issues.

Previous work on asynchronous PRAMs has been targeted at more accurately reflecting shared memory MIMD architectures by relaxing the assumption of synchronous execution. Maintaining synchrony on these architectures is very expensive since processes may run at different relative speeds, so that a single slow process can slow down the whole computation. Process speeds may differ from instructions taking different times to complete and

from delays that may occur between instruction executions; delays may be due to process swapping, contention, interrupts, etc. On all proposed asynchronous PRAMs, algorithms are correct only if they work regardless of any delays that may occur among processes. Differences between them are based on assumptions made about the characteristics of delays; in general algorithms designed and analyzed on a model assuming smaller delays are inefficient on models allowing for larger delays.

Gibbons [G1] [G2] has introduced the Asynchronous PRAM model, where processors operate asynchronously on a global shared memory but requires that two communicating processors (in the sense described in Section 2.2) explicitly synchronize with each other between the time one accesses the memory location being communicated through and the time the other one does. Thus the model is semi-synchronous in that it disallows asynchronous communication (although it doesn't enforce it; that is left for an overlaying programming model to do), unlike other work on asynchronous PRAMs (next paragraph). Algorithms are designed and analyzed on Gibbons' model assuming that delays are small and spread evenly among the processors, such that processors progress through their programs at roughly the same rate. This is a reasonable assumption for certain system environments (see Section 3).

Kruskal, Rudolph, and Snir [KRS], and later Cole and Zajicek [CZ1], have introduced asynchronous PRAM variants that assume bounded deterministic delays. The work of Cole and Zajicek [CZ2], and Nishimura [N], assume probabilistic delays. Martel et al. [MSP] [MS] allow for completely arbitrary delays, including fail-stop faults, giving "expected work" complexity results. The thrust of the Kruskal et al. model is that it is more realistic than a synchronous PRAM, but can efficiently simulate a PRAM. The thrust of the other work is towards using known (assumed) properties of delays in the algorithm design process to obtain algorithms that are efficient in the face of those delays.

The problem with retrofitting the PRAM to reflect shared memory MIMD architectures is that it takes on the latter's property of relative difficulty of programming. The resulting algorithms are much more conceptually complex than their PRAM counterparts. The difference between the H-PRAM and the above-mentioned work is that, instead of relaxing the synchronization assumption, we accept (insist) on the synchronization assumption as critical to controlling conceptual complexity but give means of controlling its *extent.* For this reason, the H-PRAM seems not especially well suited for the existing shared memory MIMD computers that the above-mentioned work, except Gibbons', is geared toward; thus we consider it orthogonal to this work (in Section 3 we discuss architectures that it is geared toward). Gibbons' target architecture consists of a (nearly) regular network of asynchronously oper-

ating processors where shared memory is implemented as a union of the memory modules local to each processor (i.e. on top of physically distributed memory). The H-PRAM is most similar with Gibbons' model as it restricts asynchrony by requiring that two communicating processors have either an α-synchronization or β-synchronization step between them. In fact, Gibbons' model can be seen as an instance, or configuration, of the *uniform* H-PRAM. His "all processor synchronization" variant corresponds to a two-level hierarchy with partition steps in the level 1 algorithm, where all level 2 sub-PRAMs consist of a single processor. His "subset synchronization" variant corresponds to a three-level hierarchy with partition steps in the level 1 and 2 algorithms, where all level 3 sub-PRAMs consist of a single processor. In addition to synchronization costs, Gibbons also accounts for latency, unlike the other work on asynchronous PRAMs.

Previous work on communication complexity in PRAM computations has been aimed at reducing global memory references, with their associated costs (latency), by employing computation on data in processors' private memory as much as possible. Aggarwal, Chandra, and Snir [ACS1] propose the LPRAM model, where processors have unbounded private memories, and use it to study communication requirements in problems and algorithms. This is a uniform, two-level memory, as discussed in Section 2.2. The basic idea here is to read a collection of variables from shared memory into private memory, perform a non-trivial amount of local computation on them (communication to private memory is unit-time), and write the result(s) back to global memory. The model implied by Papadimitriou and Yannakakis [PY], and Papadimitriou and Ullman [PU2], is similar to the LPRAM except that it pipelines global memory references. The LPRAM can be seen as an instance of the H-PRAM corresponding to a two-level *non-uniform* H-PRAM where all global communication is done in level 1 and where level 2 sub-PRAMs consist of single processors performing local computation; the block of global shared memory "private" to a 1-processor sub-PRAM is seen as the processor's private memory (as communication to it is unit-time since $\ell(P) = 1$ when $P = 1$). The LPRAM is also trivially an instance of the H-PRAM corresponding to a one-level *uniform* H-PRAM with an LPRAM level 1 sub-model.

As discussed in Section 2.2, the ratio of computation steps to communication steps in a computation has been known as the *communication locality* (or just the *locality*) of it; the higher it is the better the performance of a PRAM algorithm will be on realistic architecture. The difference between the *non-uniform* H-PRAM and the models of [ACS1] [PY] [PU2] is that these models represent *strict* locality, which distinguishes between comparatively "free" communication (unit-time communication to private memory) and communication charged

at the full latency of the architecture, while the non-uniform H-PRAM represents *neighborhood* locality in addition to strict locality. Many problems seem to have the characteristic of neighborhoods of communication, where communication need not be charged at the full latency of the architecture but at the latency of the neighborhood. We give examples in [HR]. The point is that there is a limit to replacing communication with local computation; communication must exist in parallel computing and often it need not traverse the full "width" of the architecture. An analog of communication locality is *synchronization locality*; Gibbons' algorithm work [G1] [G2] is essentially towards those that exhibit good *strict* synchronization locality. We defined *general* locality to subsume both strict and neighborhood locality, in terms of both communication and synchronization (simultaneously).

In [ACS2], Aggarwal, Chandra, and Snir augment the LPRAM with block pipelining capability and call the resulting model a BPRAM. The BPRAM can transfer a contiguous block of g words from global memory into local memory in time $\ell + g$, and thus represents the resource of memory granularity in parallel machines. If we allow block pipelining in the H-PRAM, the BPRAM is an instance of the H-PRAM in the same way that the LPRAM is.

Valiant's *architectural* (in our opinion) model (BSP [V2]; XPRAM [V3]) considers communication, synchronization, and memory granularity in a unified framework, so in this sense is related to our work. In it processors run asynchronously but synchronize with each other at fixed periods of time; the amount of time between synchronizations is called the *periodicity*. Processors employ arbitrary pipelining and simulate a number of virtual processors proportional to the periodicity in order to efficiently simulate a PRAM. It does not employ hierarchy or represent general locality.

Neighborhood locality has received some attention in terms of networks in distributed computing [AGLP] [AP1].

3 Relations to architectures

The original purpose in designing the H-PRAM model was to obtain an *idealized* model retaining as many properties of the PRAM as possible, but accounting for and providing abstract control over communication and synchronization costs. It turns out that the H-PRAM reflects (is fairly close to) a variety of existing and proposed architectures, in that they are able to support its computing style.

In the previous section we noted that some other models of computation can be seen as instances of the (uniform and/or non-uniform) H-PRAM. It follows that these models are

reflective of any architecture the H-PRAM is reflective of (when the H-PRAM cost functions are charged in the analysis of their algorithms). Also, these restricted instances of the H-PRAM are reflective of any architectures that those models are.

In Subsection 3.1 we discuss relations, or mappings, of the H-PRAM to architectural models with respect to the issues of structure (partitioning) and operation. Subsection 3.2 considers some specific, practical issues arising from the mapping of the H-PRAM's shared memory to an architecture's physical memory.

3.1 Structural and operational mapping

An architecture where processes may proceed at different and varying speeds is said to present a non-uniform *environment* [CZ2]. Non-uniformity arises in "process oriented" (or implicit scheduling) architectures, where a user program may create an arbitrary number of processes and an architecture's operating system is responsible for scheduling and managing their execution on a (usually) small number of powerful processors (which is a very involved procedure). Without extremely efficient dynamic scheduling and load balancing mechanisms in process oriented architectures, a synchronization assumption results in algorithms which perform poorly in the resultant non-uniform environments, thus the significant amount of work on asynchronous PRAMs which relax this assumption. Since the H-PRAM retains the assumption, even though controlling the extent of it, it seems not especially well suited for architectures presenting a non-uniform environment, such as most existing shared memory MIMD architectures.

On the other hand, an architecture where processes proceed at predictable and relatively similar speeds presents a uniform environment. "Virtual processor oriented" (or explicit scheduling) architectures generally result in uniform environments. In these types of architectures, a user program is given a fixed set of processors to work with explicitly, where each processor may simulate a number of virtual processors. Since physical processors are identical, each generally simulates the same number of virtual processors, and their interconnection networks are have a regular structure, there is little variation between the speeds of the processors as seen from the user program. Distributed memory architectures are usually virtual processor oriented, and thus present uniform environments; they also have more potential for general purpose massively parallel computing because of their relative scalability.

In relating the H-PRAM to architectures we generally consider architectures that present uniform environments, i.e. virtual processor oriented, distributed memory architectures.

However, it may be that certain hierarchically structured, process oriented architectures (with efficient partitioning, scheduling, and load balancing mechanisms) would be able to support the H-PRAM. A hierarchically structured system may be able to reduce the degree of non-uniformity via partitioning, thus "controlling its extent" analogously to the way we control the extent of synchronization and communication costs.

In order to fully support the H-PRAM, an architecture must basically be capable of recursively partitioning itself into independent "subsystems". If it is able to do this is such a way that the subsystems are "contiguous", in that they may be seen as smaller parallel computers that sub-models of the H-PRAM can be logically mapped onto, then the H-PRAM provides an accurate and abstract reflection of it, as a computational model is supposed to. For the *uniform* H-PRAM variant, a subsystem in the architecture consists of processors, since α-synchronization cost in an overlaying sub-PRAM is a function of number of processors participating. For the *non-uniform* variant, a subsystem consists of processors *and* memory, since α-synchronization *and* latency are functions of number of processors participating in this case. The property of partitionability into "contiguous" subsystems is a very reasonable requirement for a general purpose parallel computer; one argument for this is that any multi-user computer will necessarily be partitionable in some way. By recursively partitionable, we mean that subsystems obtained from a partitioning may be further partitioned themselves, and this may continue until a certain smallest subsystem size is reached (such as a single processor); this gives rise to a hierarchy of subsystems.

There are two issues to address in partitioning into subsystems: the partitioning of synchronization (or control communication), resulting in asynchronous control, and the partitioning of the physical resources of the architecture (e.g. processors, memory). There are two methods of partitioning synchronization. The first is by computing in Multiple SIMD (MSIMD) style, where subsystems operate in SIMD mode. In this sort of system the sub-PRAMs of the H-PRAM would run in SIMD mode rather than just synchronously (this is no drawback; as noted in Section 2.3.2 synchronous MIMD execution has questionable value, and we are unaware of any MIMD PRAM algorithms). PASM [SSKD] [SSKMSS] is a system that has been built that can support this type of computing; it consists of a two-level hierarchy where subsystems can run in either SIMD or MIMD mode. The GPA Machine [B2] is a proposed two-level MSIMD architecture which has more partitioning flexibility than PASM. The other method of partitioning control is by having a completely asynchronous system, and using synchronization primitives to implement (α) synchronization steps within subsystems. Barrier synchronization (see [KRS] and [G2]) and (maybe) network synchronizers adapted

from distributed computing [A] [AP2] [PU1] [ER] are possible primitives. Barrier synchronization of P' sub-PRAM processors should be implementable in time $O(\ell(P'))$ on most architectures. This follows from the fact that prefix/scan operations (which can implement a barrier synchronization step) can be implemented in time $O(\ell(P'))$ on most architectures (actually $O(\text{diam}(P'))$; recall that $O(\ell(P'))$ is not necessarily the diameter of a P'-processor network but $\text{diam}(P') \leq \ell(P')$). Busy-waiting would have to be incorporated into a prefix/scan barrier synchronization operation, but the overhead of it would be negligible since a sub-PRAM algorithm is tightly synchronized, with processors executing the same types of instructions at the same rate. The maximum difference between times needed by processors participating in one parallel step to complete their instructions is $O(\ell(P'))$ (the shortest time required by a single instruction in a single processor is unit time, while the longest is $O(\ell(P')))$, which does not dominate the time needed by a prefix/scan operation.

Note that in massively parallel computing, MSIMD architectures may be preferable to both SIMD and MIMD from a synchronization cost point of view. Synchronization under MIMD has a high cost, but is not done after every algorithm step. Under SIMD, synchronization has lower cost but is done after every step, and becomes more costly as the number of processors to synchronize grows.

Feitelson and Rudolph [FR] have addressed the issue of partitioning control in the context of operating systems; they propose a hierarchical system for general purpose, interactive, multi-user parallel computing. Partitioning of physical resources is also addressed. In the context of user programs, the H-PRAM might be reflective of a system such as this. However, we note that this system is a process oriented one.

The partitioning of the physical resources of an architecture depends on its interconnection network. There have been conflicting definitions of "partitionable" and "hierarchical" in the literature in the context of interconnection networks. We use the following definitions. A network is *partitionable* if it can be divided (once) into independent sub-networks that have the same interconnection properties (topology) as the larger, whole network. The network is *recursively partitionable* if each sub-network resulting from a partitioning can be further partitioned in the same way, and this recursive decomposition of the network can continue until some "smallest" possible sub-network is obtained (usually a single processor). A network is *hierarchical* if it can be recursively decomposed, as a recursively partitionable network can, except that the topology of a sub-network need not be the same as the network that was partitioned to obtain it. Thus "hierarchical" subsumes "recursively partitionable", which in turn subsumes "partitionable". Siegel [S1] has discussed recursive partitionability

of general interconnection networks.

In the following, we are solely concerned with distributed memory architectures, where the global memory consists of the union of the local memories of the processors. Thus the partitioning of processors naturally partitions the memory proportionately, and these architectures can support the non-uniform H-PRAM. They can also support the uniform H-PRAM by simply ignoring the effect that partitioning of processors has on the global memory. (Subsection 3.2 considers issues of supporting the H-PRAM's shared memory abstraction on architectures' physically distributed memory.)

The H-PRAM is reflective of partitionable (although restricted to being a two-level H-PRAM), recursively partitionable, and hierarchical networks. The PASM and GPA are both partitionable architectures. Examples of recursively partitionable networks are the hypercube, mesh, and star graph [AK] (the hypercube and star graph are Cayley graphs [AK], a class of graphs that are highly symmetric and recursively partitionable; thus, the H-PRAM would be reflective of any Cayley graph). Kruskal, Rudolph, and Snir [KRS] have stated that "most models" satisfy the property that Q processors can simulate $\Omega(Q/P)$ independent P processor models with constant overhead, for $Q > P$. This could be seen as an alternative definition of recursive partitionability. PRAM models, hypercubes, and meshes have this property, but butterfly and related networks do not (in particular, constant degree networks that permute in logarithmic time do not possess this property [KRS]).

There have been numerous proposed hierarchical architectures where interconnection properties are identical *within* the same level of hierarchy but may differ *between* different levels. This work is based on the recognition that many problems exhibit natural (neighborhood) locality of communication and it is thus possible to reduce the number of communication links between sub-networks and as a result efficiently accommodate a larger number of processors. These architectures are typically more sparsely connected at the higher levels of hierarchy and more densely at the lower levels. Cm* [SFS] and CEDAR [GKLS] have two-level cluster-based hierarchies. Cluster-based hierarchical systems have been proposed based on crossbars [AM] and shared buses [WL]. Carlson [C1] has proposed mesh based hierarchies, and Hwang and Ghosh [HG] have designed hypercube and tree based hierarchical networks. Dandamudi and Eager [DE] have investigated hierarchical networks composed of combinations of a number of different topologies. Youssef et al. [YN1] [YN2] [BY] have recently defined "banyan-hypercube" networks. There is undoubtably many more such proposed architectures which we have not referenced here.

Since hierarchical architectures have been considered one of the most realistic ways to

build massively parallel computers, due to their cost-effective scalability (via minimization of the number of communication links), the H-PRAM naturally suggests itself an ideal model of massively parallel computation. Additionally, the property of recursive decomposability in a network greatly helps manufacture, since it can be constructed from a large number of relatively few kinds of components [V3]. The conclusion is that the H-PRAM seems to have a nice "fit" with large scale parallel architectures that are likely to be built.

An H-PRAM variant with different cost parameters for each level of hierarchy would be most reflective of hierarchical architectures, due to differences in interconnection properties between different levels. Potentially, design and analysis of H-PRAM algorithms could use parameters ℓ_λ, $(s_\alpha)_\lambda$, and $(s_\beta)_\lambda$ according to which level of hierarchy λ a sub-PRAM algorithm is meant for, depending on how much accuracy in cost reflection is desired. However, single latency, α-synchronization, and β-synchronization parameters may very well provide adequate approximation while keeping things simpler (again, we do not want too many parameters).

The cost of β-synchronization has been defined to "usually" be a function of the number of sub-PRAMs created by the partition step being synchronized. This can be justified as follows. Since all processors within a level $\lambda+1$ sub-PRAM have been synchronized following the last step of its algorithm, only one designated processor in each sub-PRAM needs to participate in a level λ β-synchronization step among the sub-PRAMs. Sub-PRAMs are mapped to sub-networks in the architecture; let Q again be the number of sub-PRAMs/sub-networks. The logical network over which β-synchronization communication is done is obtained by considering each sub-network as a node and linking these nodes according to the interconnection links between the designated processors of each sub-network (the designated processors of the sub-networks should be chosen such that they are collectively "close"). Thus, for the purpose of β-synchronization, we get a logical network of Q nodes. Usually, in hierarchical architectures the diameter of this logical network is a function of Q, however there may be exceptions where it is also a function of P', where P' is the number of processors in the level λ PRAM. Hence, in most cases, the cost of β-synchronization is a function of Q. When this is so, it may be quite common to have $s_\beta(Q, P') = s_\alpha(Q)$ for networks possessing a regular structure. This is the case for the hypercube; it partitions in such a way that both the sub-networks and the logical network that β-synchronization is performed on are also hypercubes.

The two-dimensional mesh is an architecture where β-synchronization is also a function of the number of processors P' in the PRAM performing a partition step on Q sub-PRAMs.

It is generally not possible to choose designated processors in each sub-network such that they form a logical network whose diameter is a function of Q.

As some of the referenced papers mention, patterns of communication locality can be used to optimize topologies at various levels of hierarchical networks. These patterns can be isolated by considering certain application domains, such as computer vision. We suggest that an abstract model such as the non-uniform H-PRAM would be an excellent tool for studying general properties of locality in problems, and thus drive topological optimization in hierarchical networks for general purpose parallel computing.

The point may be raised that the H-PRAM allows arbitrary partitioning but realistic architectures naturally restrict the sizes and numbers of sub-networks. This can be countered by a weak argument and a stronger argument. The weak argument is that the H-PRAM is by definition an abstract model, and will adequately reflect an architecture even if its sub-PRAMs do not perfectly map onto an architecture's sub-network; in other words costs will be proportional to the numbers and sizes of sub-models, and the depth of the hierarchy, in both the H-PRAM and the architectural model. Remember from the introduction that we are not (necessarily) looking for *perfect* reflectivity of architectures. The strong argument is that we can map a sub-PRAM onto the nearest-sized sub-network; most networks can simulate differently-sized versions of themselves with only a constant loss of efficiency [KRS].

If hierarchy were introduced into Valiant's architectural model [V2] [V3], there would be a close relationship between it and the H-PRAM. This seems like a reasonable prospect since hardware designs have exhibited hierarchy (and Valiant specifically discusses the hypercube as an implementing network), and ideal since it is proposed as a parallel counterpart of the von Neumann architectural model, i.e. as a standard. A hierarchy capability does not *have* to be used.

3.2 Memory mapping

Since latency is a parameter of the H-PRAM, the results of this paper are independent of (the costs of) any particular technique for "implementing" the H-PRAM's shared memory on a distributed memory architecture. Still, for practical reasons, it is important to consider the issues behind it.

The PRAM effectively automates all memory management; an algorithm deals with a collection of memory locations, each of which can be accessed in unit time. As stated, this results in simplicity of usage, but its realization (simulation) has a cost. On the other hand, distributed memory architectures require manual memory management; an algorithm needs

to explicitly map data to the local memories of processors and change mappings (using explicit communication links) as necessary. As stated, this is significantly more difficult but leads to more efficient use of architectures.

Automated memory management essentially means PRAM simulation, which consists of the mapping of a shared memory space onto the physical memory of the simulating architecture. Since the uniform H-PRAM employs one large, fixed shared memory, like the PRAM, it embodies automated memory management, and known techniques using hashing can be used to simulate it (this is one reason for its existence). The nature of memory management in the non-uniform H-PRAM lies between the extremes of totally automated and totally manual. Therefore, in terms of memory management it provides a good balance (compromise) between simplicity and efficiency (or reflectivity of an architecture). In the non-uniform variant, H-PRAM algorithms need to map data to blocks of memory that will become sub-PRAMs' private blocks of memory when a partition step is executed. To be more specific, consider a partition step in an algorithm running on a (sub)-PRAM in level λ of a hierarchy, and suppose it creates Q level $\lambda+1$ sub-PRAMs. Then prior to the partition step the level λ algorithm maps groups of data (not necessarily contiguous in the level λ PRAM's memory) into Q blocks of contiguous memory according to the purpose of the algorithm. In other words, the level λ PRAM's memory is permuted prior to a partition step so that data are grouped into blocks of memory; each block will become the private memory of a level $\lambda+1$ sub-PRAM. In the remainder of this section, we concentrate on the non-uniform variant of the H-PRAM.

Since, in a distributed memory architecture, the physical memory is partitioned into blocks along with the processors, there needs to be a method (a memory mapping scheme) that maps contiguous blocks of the non-uniform H-PRAM's shared memory to contiguous blocks of physical memory (i.e. to the local memories of processors forming a sub-network that a sub-PRAM is mapped to). Given this mapping scheme, the algorithm designer has abstract control over the mapping of data to the physical memory of an architecture. This would be analogous to the data is explicitly mapped to processors in distributed memory algorithms, except that data need only be kept in the right "groups", and "groups" mapped to contiguous sets of processors (sub-networks). This is what we mean by "abstract control" over memory mapping.

The details of a mapping scheme depend on the details of the automated memory management, or simulation of the level λ PRAM, with respect to the underlying architecture. Known PRAM simulation techniques use hash functions to map shared memory locations to

physical memory locations so that any combination of memory accesses completes quickly with high probability. Given that this technique is used to simulate sub-PRAMs, a possible mapping scheme could just unhash the memory, assuming that the given hash function is a permutation (i.e. one-to-one, which linear-polynomial pseudo-random hash functions are [V2]). In this case, unhashing entails performing a permutation on the memory defined by the inverse of the hash function. This operation could be part of an implementation of a partition step. An implementation of a partition step would also have to hash the memories of the Q level $\lambda + 1$ sub-PRAMs onto the physical memories of Q sub-networks of an architecture, so as to simulate them. Since both the mapping scheme (unhashing) and the hashing operation entail permutations, their composition is a single permutation and can implement the memory management part of a partition step. When level $\lambda + 1$ sub-PRAM algorithms terminate, i.e. the level λ partition step terminates, the inverse permutation is executed, i.e. the memories of the level $\lambda + 1$ sub-PRAMs are unhashed and the memory of the level λ PRAM is (re)hashed.

The reader is referred to [V2] and references therein for discussion on properties of hash functions.

The permutation of N' elements by P' processors can be done in time $O((N'/P')\ell(P'))$ on a P'-processor sub-PRAM (any variant) of the H-PRAM. Thus the overhead of this possible memory mapping scheme would not normally effect the complexity of a sub-PRAM algorithm by more than a constant factor. Recall that sub-PRAM algorithms which undertake memory management execute a permutation prior to (and after) a partition step anyway.

PRAM simulation and mapping schemes are research topics in their own right, and depend on underlying architectural models.

One drawback to current PRAM simulation techniques is that they obliterate (through hashing) any natural communication locality that may be present in an algorithm. The non-uniform H-PRAM has the potential to reduce the scope of this problem, via simulation of multiple smaller sub-PRAMs (this constrains the extent of obliteration to be within the sub-network simulating the sub-PRAM). Another possible advantage of the non-uniform variant is that it may allow the efficient usage of larger parallel architectures than could be done otherwise. To see this, note that as an architecture scales up, its communication bandwidth must grow faster than its computation bandwidth, in order to maintain the same level of efficiency [V1] [V3]. By increasing the number of sub-PRAMs in a non-uniform H-PRAM algorithm as an architecture scales up, thus reducing the sizes of the sub-PRAMs relative to an architecture's size, the need for higher communication bandwidth is reduced.

If a parallel computer has the resource of memory granularity, or pipelining, thought needs to be given as to whether it is best represented in the computational model level of abstraction or the architectural model level of abstraction. An arbitrary pipelining capability can be combined with a sufficient reduction in parallelism in an architecture to achieve PRAM simulation with only a constant loss of efficiency [V2] [KRS] (reduced parallelism means that the simulating architecture has less processors than the simulated PRAM). This suggests that the correct placement of of arbitrary pipelining representation is in the architectural model. The choice for block pipelining is less clear and reduces to whether we consider the BPRAM [ACS2] an architectural or computational model. Although the decision is subjective, we do note that first, the BPRAM does not need simulation on an architecture as (or to the extent) the PRAM does (since blocks of memory accessed in parallel must be disjoint), and second, block pipelining may in some way be useful in an architectural model to support a memory mapping scheme in the overlaying H-PRAM computational model.

There is one situation where employing the BPRAM as a sub-model of the H-PRAM (therefore representing block pipelining in the H-PRAM) may be beneficial. If a programming model overlaying the H-PRAM has data structures that use sets, thus operating on sets of data in parallel, these sets could be mapped to blocks of memory in the BPRAM sub-models of the H-PRAM. This could potentially allow a programmer to use an architecture very efficiently.

4 Relations to programming models

In this section we briefly discuss relationships between the H-PRAM computational model, and existing and potential programming models which could overlay it. Although we have defined a programming model to be a formalized definition of a programming language, we may use the terms "programming model" and "programming language" interchangeably without ambiguity. Generally, a programming model is the view of a computer as seen by an applications programmer.

Most existing parallel programming languages are either strictly SIMD or strictly MIMD, so could not directly take advantage of the full capabilities of the H-PRAM. However, SIMD languages could be augmented with partitioning operations to obtain MSIMD languages that can effectively use the H-PRAM (where sub-PRAMs operate in SIMD fashion, rather than just synchronously).

Steele [S4] has recently discussed the design of a programming model that organizes

asynchronous control via a hierarchy, and where asynchronous communication is disallowed (enforcement mechanisms are part of the model). Although based on sequential processes, rather than collections of processors (i.e. sub-PRAMs), it is a promising development. Something along these lines may provide an ideal programming model overlaying the uniform H-PRAM, as it assumes a single (i.e. non-partitionable) global shared memory. Gibbons [G2] has discussed general issues of determinate ("semi-synchronous") programming models with respect to computational models that partition synchrony but not communication (memory), as the uniform H-PRAM does. We stress that any programming model overlaying the uniform H-PRAM must enforce determinacy (synchronous communication; see Section 2.2).

Programming models for the non-uniform variant would be easier to design, since this variant of the H-PRAM takes on the responsibility of enforcing synchronous communication itself. We have noted that the non-uniform H-PRAM has a natural fit with the paradigm of divide-and-conquer. Mou [M2] has recently designed a programming model for scientific computing based on divide-and-conquer called Divacon, which could likely make very effective use of the non-uniform H-PRAM.

A programming model that has set data structures, operating on and communicating sets of data in parallel, could efficiently use an architecture capable of block pipelining. This was discussed in Section 3.2; the sub-models of the H-PRAM underlying a programming model of this type would be BPRAMs.

Finally, the ability of the H-PRAM to control the conceptual complexity of large parallel applications requires that overlaying programming models have facilities for supporting the modular construction of software.

5 Conclusions

We have introduced the Hierarchical PRAM (H-PRAM), a model of parallel computation that retains the ideal properties of the PRAM by using it as a sub-model, while simultaneously being more reflective of realistic parallel architectures by accounting for and providing abstract control over communication and synchronization costs. The use of hierarchy controls conceptual complexity in the face of asynchrony in two ways. First, by organizing control asynchrony such that synchronization points are implicit in an H-PRAM algorithm's structure. Second, by restricting communication asynchrony such that algorithms are determinate and thus easy to prove correct, unlike algorithms for most other asynchronous environments. Additionally, this means that there is no need for costly synchronization

primitives in underlying architectures. Since the PRAM is a sub-model of the H-PRAM, we can retain direct relevance of established theory of parallel computing, and build on it in a manner of construction from components. From a purely aesthetic viewpoint, the H-PRAM is logical model in that it provides the simplifying assumption of synchronous execution to the design of algorithms, but allows the algorithms to work asynchronously with each other, which corresponds to the way real-world applications are: collections of individual algorithms which, as systems, have asynchronous functional definitions. The process of constructing an H-PRAM algorithm is analogous to the software engineering process in the sequential domain, i.e. modular construction of larger and larger components.

The *uniform* variant of the H-PRAM passes the responsibility for enforcing determinate computation to an overlaying programming model. As noted, there has been encouraging work on programming models that do this. Programming models for the *non-uniform* variant (which provides more control over costs) would be easier to design, since this variant of the H-PRAM takes on the responsibility of enforcing determinacy itself. In the other direction, we have shown that the H-PRAM is reflective of a variety of existing and proposed architectural models. Thus, the H-PRAM could possibly serve as a bridging model at a higher level of abstraction than that proposed by Valiant [V2] (and higher than the von Neumann model is in sequential computing).

Since hierarchical architectures have been proposed as the most realistic way of building cost-efficient massively parallel computers, the H-PRAM suggests itself as an ideal model of massively parallel computing. Inter-level topologies of hierarchical architectures can be optimized to match certain common patterns of general locality; it thus follows that the H-PRAM could potentially be used to discover these patterns and thus drive topological optimization. Additionally, general locality (using the non-uniform H-PRAM) could allow the efficient usage of larger parallel architectures than could be done otherwise since, as an architecture scales up, its communication bandwidth must grow faster than its computation bandwidth, in order to maintain the same level of efficiency [V1] [V3]. By increasing the number of sub-PRAMs performing "localized" communication in a non-uniform H-PRAM algorithm as an architecture scales up, thus reducing the sizes of the sub-PRAMs relative to the architecture's size, the need for higher communication bandwidth is reduced.

Strict communication locality, as defined by the ratio of computation to communication steps, seems to have its limits in that certain problems do not submit to it, and those that do have sometimes severe restrictions on the number of processors efficiently usable. The H-PRAM provides a flexible tool to investigate general degrees of locality ("neighborhoods" of

activity) in problems, considering communication *and* synchronization simultaneously, which gives the potential of obtaining algorithms that map more efficiently to architectures, and of allowing the efficient usage of more processors (in comparison to a PRAM that charges for communication and synchronization). We demonstrate this in the companion paper [HR] via various algorithms for computing complete binary trees and FFT graphs. The H-PRAM presents a framework in which to study the extent that general locality can be exploited in parallel computing.

Finally, it would be interesting to investigate alternative hierarchy relations for organizing asynchrony, such as in [M1].

References

[A] B. Awerbuch, Complexity of network synchronization, *Journal of the ACM*, Oct. 1985, pp.804–823.

[ACS1] A. Aggarwal, A.K. Chandra and M. Snir, Communication complexity of PRAMs, *Theoretical Computer Science*, Vol. 71, 1990, pp. 3–28.

[ACS2] A. Aggarwal, A.K. Chandra and M. Snir, On communication latency in PRAM computations, Tech. Rep. RC 14973 (#66882) 9/27/89, IBM T.J. Watson Research Center, Yorktown Heights, NY.

[AGLP] B. Awerbuch, A.V. Goldberg, M.Luby and S.A. Plotkin, Network decomposition and locality in distributed computation, *Proc. 30th Annual IEEE Symposium on Foundations of Computer Science*, 1989, pp.364–369.

[AHMP] H. Alt, T. Hagerup, K. Mehlhorn and F. Preparata, Deterministic simulation of idealized parallel computers on more realistic ones, *SIAM Journal on Computing*, Oct. 1987, pp. 808–835.

[AHU] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[AK] S.B. Akers and B. Krishnamurthy, A group-theoretic model for symmetric interconnection networks, *IEEE Trans. on Computers*, April 1989, pp. 555–566.

[AM] D.P. Agrawal and I.E.O. Mahgoub, Performance analysis of cluster-based supersystems, *Proc. IEEE Intl. Conference on Supercomp. Systems*, 1985, pp. 593–602.

[AP1] B. Awerbuch and D. Peleg, Sparse partitions, *Proc. 31st Annual IEEE Symposium on Foundations of Computer Science*, 1990.

[AP2] B. Awerbuch and D. Peleg, Network synchronization with polylogarithmic overhead, *Proc. 31st Annual IEEE Symposium on Foundations of Computer Science*, 1990.

[B1] G. Blelloch, Scans as primitive parallel operations, *Proc. Intl. Conference on Parallel Processing*, 1987, pp. 355–362.

[B2] T. Bridges, The GPA Machine: A generally partitionable MSIMD architecture, *Proc. 3rd Symp. on Frontiers of Massively Parallel Computation*, 1990, pp. 196–203.

[BY] A. Bellaachia and A. Youssef, Partitioning on banyan-hypercube networks, *Proc. 3rd Symp. on Frontiers of Massively Parallel Computation*, 1990, pp. 343–351.

[C1] D. Carlson, The mesh within a global mesh: a flexible, high-speed organization for parallel computation, *Proc. IEEE Intl. Conference on Supercomp. Systems*, 1985, pp. 618–627.

[CZ1] R. Cole and O. Zajicek, The APRAM: incorporating asynchrony into the PRAM model, *Proc. 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, 1989, pp. 169–178.

[CZ2] R. Cole and O. Zajicek, The expected advantage of asynchrony, *Proc. 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, 1990, pp.85–94.

[DE] S.P. Dandamudi and D.L. Eager, Hierarchical interconnection networks for multicomputer systems, *IEEE Trans. on Computers*, June 1990, pp. 786–797.

[DM] M. Dietzfelbinger and F. Meyer auf der Heide, How to distribute a dictionary in a complete network, *Proc. 21st Annual ACM Symposium on Theory of Computing*, 1990, pp. 117–127.

[EG] D. Eppstein and Z. Galil, Parallel algorithmic techniques for combinatorial computation, *Annual Review of Computer Science, Vol. 3*, 1988, pp. 233–283.

[ER] S. Even and S. Rajsbaum, The use of a synchronizer yields maximum computation rate in distributed networks, *Proc. 21st Annual ACM Symposium on Theory of Computing*, 1990, pp. 95–105.

[FW] S. Fortune and J. Wyllie, Parallelism in random access machines, *Proc. 10th Annual ACM Symposium on Theory of Computing*, 1978, pp. 114–118.

[FR] D.G. Feitelson and L. Rudolph, Distributed hierarchical control for parallel processing, *Computer*, May 1990, pp. 65–77.

[G1] P.B. Gibbons, A more practical PRAM model, *Proc. 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, 1989, pp. 158–168.

[G2] P.B. Gibbons, The asynchronous PRAM: a semi-synchronous model for shared memory MIMD machines, Ph.D. thesis, Computer Science Division, University of California, Berkeley, California, Dec. 1989.

[GKLS] D. Gajski, D. Kuck, D. Lawrie and A. Sameh, CEDAR, *Tutorial on Supercomputers: Design and Applications*, IEEE Computer Society Press, 1983, pp. 251–275.

[H] K.T. Herley, Efficient simulations of small shared memories on bounded degree networks, *Proc. 30th Annual IEEE Symposium on Foundations of Computer Science*, 1989, pp. 390–395.

[HB] K.T. Herley and G. Bilardi, Deterministic simulations of PRAMs on bounded degree networks, *Proc. 26th Annual Allerton Conference on Communication, Control, and Computation*, 1988.

[HG] K. Hwang and J. Ghosh, Hypernet: A communication-efficient architecture for constructing massively parallel computers, *IEEE Trans. on Computers*, Dec. 1987, pp. 1450–1466.

[HGC] T. Heywood, A. Ghafoor and J.K. Chan, Deterministic simulation of PRAMs on hypercube networks without look-up tables, *Proc. 2nd Annual IEEE Symposium on Parallel and Distributed Processing*, 1990, pp. 195–199.

[HP] S.W. Hornick and F.P. Preparata, Deterministic PRAM simulation with constant redundancy, *Proc. 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, 1989, pp. 103–109.

[HR] T. Heywood and S. Ranka, A practical hierarchical model of parallel computation: binary tree and FFT graph algorithms, Technical Report, School of Computer and Information Science, Syracuse University, Feb. 1991.

[K] C.P. Kruskal, Performance bounds on parallel processors: an optimistic view, *Control Flow and Data Flow: Concepts of Distributed Programming*, NATO ASI Series, Series F: Computer and System Sciences, Vol. 14, Springer-Verlag, 1985, pp. 331-344.

[KU] A. Karlin and E. Upfal, Parallel hashing: an efficient implementation of shared memory, *Journal of the ACM*, Oct. 1988, pp. 876–892.

[KR] R.M. Karp and V. Ramachandran, A survey of parallel algorithms for shared-memory machines, Tech. Rep. UCB/CSD 88/408, Computer Science Division, Univ. of Calif., Berkeley, 1988. Also in: *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed., North-Holland, Amsterdam, 1990.

[KRS] C.P. Kruskal, L. Rudolph and M. Snir, A complexity theory of efficient parallel algorithms, *Theoretical Computer Science*, Vol. 71, 1990, pp. 95–132.

[LPP1] F. Luccio, A. Pietracaprina and G. Pucci, A probabilistic simulation of PRAMs on a bounded-degree network, *Information Processing Letters*, July 1988, pp. 141–147.

[LPP2] F. Luccio, A. Pietracaprina and G. Pucci, A new scheme for the deterministic simulation of PRAMs in VLSI, to appear in *SIAM Journal on Computing.*

[M1] M.J. Manthy, Hierarchy in sequential and concurrent systems, *The Characteristics of Parallel Algorithms*, MIT Press, 1988, pp. 139–164.

[M2] Z.G. Mou, Divacon: A parallel language for scientific computing based on divide-and-conquer, *Proc. 3rd Symp. on Frontiers of Massively Parallel Computation*, 1990, pp. 451–461.

[MS] C. Martel and R. Subramonian, Asynchronous PRAM algorithms for list ranking and transitive closure, *Proc. Intern. Conference on Parallel Processing*, 1990.

[MSP] C. Martel, R. Subramonian and A. Park, Asynchronous PRAMs are (almost) as good as synchronous PRAMs, to appear in: *Proc. 31st IEEE Symposium on Foundations of Computer Science*, 1990.

[N] N. Nishimura, Asynchronous shared memory parallel computation, *Proc. 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, 1990, pp. 76–84.

[PU1] D. Peleg and J.D. Ullman, An optimal synchronizer for the hypercube, *SIAM Journal on Computing*, Aug. 1989, pp.740–747.

[PU2] C.H. Papadimitriou and J.D. Ullman, A communication-time tradeoff, *SIAM Journal on Computing*, Aug. 1987, pp.639–646.

[PY] C.H. Papadimitriou and M. Yannakakis, Towards an architecture independent analysis of parallel algorithms, *SIAM Journal on Computing*, April 1990, pp.322–328.

[R1] A. Ranade, How to emulate shared memory, *Proc. 28th Annual IEEE Symposium on Foundations of Computer Science*, 1987, pp. 185–194.

[R2] A.Ranade, *Fluent Parallel Computation*, PhD Thesis, Yale University, May 1989.

[S1] H.J. Siegel, The theory underlying the partitioning of permutation networks, *IEEE Trans. on Computers*, Sept. 1980, pp. 791–801.

[S2] M. Snir, On parallel searching, *SIAM Journal on Computing*, Vol. 14, 1985, pp. 688-708.

[S3] L. Snyder, Type architectures, shared memory, and the corollary of modest potential, *Annual Review of Computer Science, Vol.1*, 1986, pp. 289–317.

[S4] G.L. Steele, Making asynchronous parallelism safe for the world, *Proc. ACM Symposium on Principles of Programming Languages*, 1990, pp. 218–227.

[SFS] R.J. Swan, S.H. Fuller and D. Siewiorek, Cm*: a modular multiprocessor, *Tutorial on Parallel Processingv*, IEEE Computer Society Press, 1981, pp. 39–46.

[SSKD] H.J. Siegel, T. Schwederski, J.T Kuehn and N.J. Davis, An overview of the PASM parallel processing system, in *Tutorial: Computer Archictecture*, IEEE Computer Society Press, 1987, pp. 387–407.

[SSKMSS] H.J. Siegel, L.J. Siegel, F.C. Kemmerer, P.T. Mueller, H.E. Smalley and S.D. Smith, PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition, *IEEE Trans. on Computers*, Dec. 1981, pp. 934–947.

[UW] E. Upfal and A. Wigderson, How to share memory in a distributed system, *Journal of the ACM*, Jan. 1987, pp. 116–127.

[V1] L.G. Valiant, Optimally universal parallel computers, *Phil. Trans. R. Soc. Lond.*, A326, 1988, pp. 373–376.

[V2] L.G. Valiant, A bridging model for parallel computation, *Commun. of the ACM*, Aug. 1990, pp. 103–111.

[V3] L.G. Valaint, General purpose parallel architectures, *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed., MIT Press, 1990.

[VS] J.S. Vitter and R.A. Simons, New classes for parallel complexity: a study of unification and other complete problems for $\mathcal{P}$, *IEEE Trans. on Computers*, May 1986, pp. 403–418.

[WL] S.B. Wu and M.T. Liu, A cluster structure as an interconnection network for large multimicrocomputer systems, *IEEE Trans. on Computers*, April 1981, pp. 254–264.

[YN1] A. Youssef and B. Narahari, Banyan-hypercube networks, *IEEE Trans. on Parallel and Distributed Systems*, April 1990, pp. 160–169.

[YN2] A. Youssef and B. Narahari, Topological properties of banyan-hypercube networks, *Proc. 3rd Symp. on Frontiers of Massively Parallel Computation*, 1990, pp. 324–332.