SAND REPORT

SAND2004-6171
Unlimited Release
Printed Nov. 2004

PRAM C: A New Programming Environment for Fine-Grain and Coarse-Grain Parallelism

Jonathan Leighton Brown, Zhaofang Wen

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2004-6171
Unlimited Release
Printed Nov. 2004

PRAM C: A New Programming Environment for Fine-Grain and Coarse-Grain Parallelism

Jonathan Leighton Brown, Zhaofang Wen*

Abstract

In our search for "good" parallel programming environments for Sandia's current and future parallel architectures, we revisit a long-standing open question. Can the PRAM parallel algorithms designed by theoretical computer scientists over the last two decades be implemented efficiently? This open question has co-existed with ongoing efforts in the HPC community to develop practical parallel programming models that can simultaneously provide ease of use, expressiveness, performance, and scalability. Unfortunately, no single model has met all these competing requirements. Here we propose a parallel programming environment, PRAM C, to bridge the gap between theory and practice. This is an attempt to provide an affirmative answer to the PRAM question, and to satisfy these competing practical requirements. This environment consists of a new thin runtime layer and an ANSI C extension. The C extension has two control constructs and one additional data type concept, "shared". This C extension should enable easy translation from PRAM algorithms to real parallel programs, much like the translation from sequential algorithms to C programs. The thin runtime layer bundles fine-grained communication requests into coarse-grained communication to be served by message-passing. Although the PRAM represents SIMD-style fine-grained parallelism, a stand-alone PRAM C environment can support both fine-grained and coarse-grained parallel programming in either a MIMD or SPMD style, interoperate with existing MPI libraries, and use existing hardware. The PRAM C model can also be integrated easily with existing models. Unlike related efforts proposing innovative hardware with the goal to realize the PRAM, ours can be a pure software solution with the purpose to provide a practical programming environment for existing parallel machines; it also has the potential to perform well on future parallel architectures.

*email: zwen@sandia.gov

Acknowledgment

This work would not have started if Bill Camp had not asked us how to handle pointer jumping in Unified Parallel C. It was during the design of a library to handle pointer jumping on discrete data structures (linked-list, tree, and graph) on top of UPC that we came to the idea of developing a programming environment for general PRAM style fine-grained parallel algorithm implementation. We would like to express our appreciation to Bill Camp for his many insightful comments and suggestions on our past programming model research reports and technical presentations. We would like to thank Neil Pundit for his constant encouragement, support, and coaching. Numerous constructive comments were made by many others including Rolf Riesen, Ron Brightwell, Cindy Phillips, Bruce Hendrickson, Jonathan Berry, Sue Goudy, Rob Leland.

Contents

1 Introduction 7

1.1 The PRAM Model 7

1.2 Fine-grained vs Coarse-grained Parallelism 8

1.3 Our Proposed Programming Environment 9

1.4 Related Research 9

2 Design of PRAM C 12

2.1 Our Implementation of the PRAM Model: A High-level View 12

2.2 PRAM C Features 12

2.3 Design Considerations for PRAM C 16

3 PRAM C Strengths and Potential Benefits 17

4 Program Execution Model 19

4.1 Thread Creation and Physical Processor Allocation 19

4.2 Thread Execution 19

4.3 Physical Processor Execution 20

4.4 Virtual PRAM Processor Execution 20

4.5 Shared Memory Access By Virtual PRAM Processors 21

5 Runtime System Components 21

5.1 Communication Manager and Thread Scheduler Activities 21

5.2 Communication Manager Interface 23

5.3 Light-weight Thread Library Interface 23

5.4 Thread Scheduler Interface 24

6 Translator for the PRAM C 25

7 Possible Implementations 26

8 Performance Discussion and Comparison of Models 26

8.1 The Essence of Our Model 26

8.2 Many Threads and Communication Bundling 27

8.3 Comparison with Other Models 27

9 Program Style Guide 28

9.1 Memory Access Style Guide 29

10 Code Examples 29

10.1 Parallel Prefix 29

10.2 Parallel QuickSort 31

11 Conclusion 32

Figures

1 **The Parallel Random Access Machine (PRAM), an abstract model for SIMD parallel algorithm design.** Each algorithmic step is executed in parallel on each of the processors. Each processor has access to the shared memory. There are various PRAM models (CRCW, CREW, EREW) that differ in whether not concurrent-read (exclusive-read) and concurrent-write (exclusive-write) are allowed. 8

2 **Our Implementation of the PRAM Programming Model: (A Conceptual View)** We extend the basic PRAM model to provide an implementable model compatible with modern parallel hardware. A PRAM C program is written based on virtual PRAM processors, which interact with a virtual shared memory interface. Each physical processor supports multiple ($\frac{P}{K}$ on average) virtual PRAM processors via multiplexing. The execution of virtual PRAM processors on each physical processor is scheduled by a Thread Scheduler. The virtual shared memory accesses is registered with a Communication Manager that maps the global, shared accesses to distributed memory and uses the message passing layer (or other communication subsystem) for remote reads and writes. (1) Fine- and coarse-grained parallel accesses are allowed between virtual PRAM processors and the virtual shared memory. (2) The virtual shared memory registers the PRAM memory accesses with the Communication Manager. (3) Requests for physically non-local memory are bundled by the Communication Manager and served in bulk (by message-passing). (4) The virtual PRAM processors are implemented as light-weight threads. (5) PRAM processors are multiplexed to a much smaller number of physical processors by the thread library. **Note:** There is one Thread Scheduler and a Communication Mangager running on each physical processor. . . 13

3 **The Runtime System.** (a) Buffer of pending threads. (b) Queue of ready threads. (c) Shared data request buffer. 22

PRAM C: A New Programming Environment for Fine-Grain and Coarse-Grain Parallelism

1 Introduction

Sandia has a substantial investment in its parallel computing assets and in research to develop new parallel architectures and technologies to maintain its leadership position in high-performance computing. Similar effort is needed to provide an appropriate programming model and environment to exploit these existing and future supercomputers. Our work focuses on finding parallel programming models that will provide the performance Sandia has come to rely on from MPI while fully exploiting the potential of emerging supercomputing architectures as well as being expressive and easy-to-use for the programmer. The PRAM is an expressive, natural parallel model that has a firm theoretical foundation and a large body of algorithms designed for it, but is widely considered impractical. We claim that it is indeed practical and present an adaptation of the PRAM for existing and future parallel architectures, as well as design details for its implementation.

1.1 The PRAM Model

The Parallel Random Access Machine (PRAM) model has been widely used by academic algorithm designers for two decades. This work has led to a vast collection of parallel algorithms ([6]). The model is known for its simplicity (Figure 1). In the PRAM model, each instruction, or, rather, each algorithm step, is executed concurrently by all processors, providing synchronized, SIMD-style parallelism. The PRAM assumes shared memory available to all processors, and communication costs to access such shared memory are not counted when analyzing algorithms for the PRAM. The PRAM is a tool for the algorithm designer to discover, express, and exploit the natural parallelism inherent in a problem without being handicapped by the constraints of a specific hardware architecture.

The simplifying assumptions of the PRAM model, in particular the cost of parallel communication, has allowed for the "abuse" of fine-grained accesses to shared memory in many PRAM algorithms in order to reduce complexity and thus achieve optimal asymptotic (big-O) performance. Because existing parallel programming environments on real machines do not provide support for efficient fine-grained communication, few (if any) "fast" PRAM algorithms have provided efficient solutions on real parallel machines as compared to efficient sequential algorithms. Consequently, PRAM algorithms are widely considered unimplementable, and PRAM algorithm designers have almost all moved on to other research areas.

The PRAM story recently made a 180-degree turn to the positive. The Workshop on

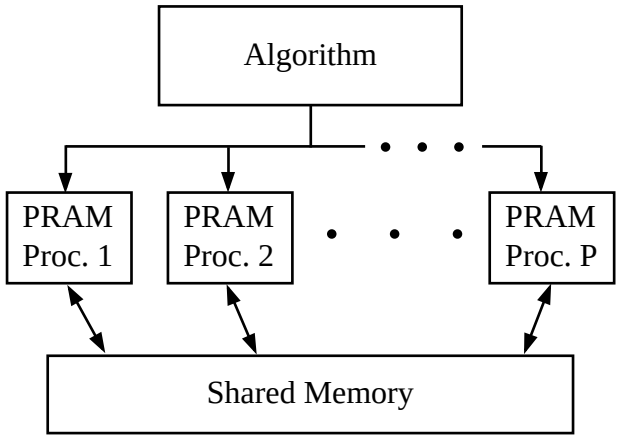

Figure 1. The Parallel Random Access Machine (PRAM), an abstract model for SIMD parallel algorithm design. Each algorithmic step is executed in parallel on each of the processors. Each processor has access to the shared memory. There are various PRAM models (CRCW, CREW, EREW) that differ in whether not concurrent-read (exclusive-read) and concurrent-write (exclusive-write) are allowed.

The Roadmap for the Revitalization of High-End Computing, Washington, D.C., June 2003 was commissioned by the White House Office of Science and Technology. According to the Summary of Workshop, edited by Daniel A. Reed, Computing Research Association, in Table 2.1, entitled Enabled Technology Opportunities, on page 19, the PRAM is noted as an enabling software and algorithms technology for Fiscal Years 2010-2014 (quoted from [10]).

1.2 Fine-grained vs Coarse-grained Parallelism

Probably related to the past PRAM experiences, it has become a common belief that coarse-grained parallelism is good, and fine-grained parallelism is bad for application performance. Parallel application developers have learned to avoid fine-grained parallelism and thus have lost the opportunities to exploit the (arguably) more important parallelism inherent in the problem. Consequently, most existing parallel programming environments provide little capabilities for the program to express fine-grained parallelism.

The lack of expressiveness for fine-grained parallelism can have a negative impact on the ease of use of the programming environments and thus application programmers' productivity. For many fundamental problems (such as pointer-jumping in linked list, graph manipulation, and sparse-matrix handling) that are algorithmic in nature, "efficient" coarse-grained parallel algorithms (if possible at all) are often harder to design, while fine-grained

parallel algorithms are much more natural. In such cases, the capability to do fine-grained parallel programming can be important to productivity and even application performance.

1.3 Our Proposed Programming Environment

We propose a PRAM-style parallel programming environment, called PRAM C. It consists of a thin runtime layer and a simple extension to ANSI C. The C extension is designed to enable fine-grained parallelism, allowing programmers to convert PRAM algorithms to a program as a matter of syntax translation, similar to the way sequential algorithms are implemented as C programs. We aim to satisfy the often-competing requirements on any production programming environment, including ease-of-use, expressiveness, application performance, and scalability. We also seek to provide an environment compatible with existing programming models.

This is an innovative programming environment that allows for both coarse- and fine-grained parallelism to be implemented efficiently on existing and future parallel architectures. Fine-grained parallelism is expressed with a new language construct (**PRAM_do**), which creates virtual PRAM processors as light-weight threads. Fine-grained communication is bundled by a thin runtime layer and exchanged in a coarse-grained fashion, thus leading to high performance. Multiplexing a large number of virtual PRAM processors to each physical processor maximizes the parallelism that can be exploited as well as the benefit from bundling communication requests. It is this separation of virtual PRAM processors and physical processors that lies at the heart of our innovation: the virtual PRAM processors are provided as an interface to programmers that allow for maximal parallel expression, whereas the mapping to physical processors via communication and thread libraries in the runtime allow this to run on existing hardware and interoperate with existing programming environments such as MPI.

1.4 Related Research

There are projects that are somewhat related to our PRAM C research. These projects typically develop their own special (proprietary) hardware architectures. Two of them, the XMT project [10] and the SB-PRAM project [9], have specific goals to realize the PRAM model and offer PRAM-like programming environments; while others, such as CRAY's MTA [1], provide programming language constructs or compiler directives to allow for creation of parallel threads, which can be helpful in implementing PRAM-style parallel algorithms.

1.4.1 The SB-PRAM Project

SB-PRAM is a physical realization of PRAM. Its proprietary hardware is a massively parallel, uniform memory access (UMA) shared memory computer [9]. The main ideas of the design

are multithreading on instruction level, hashing of the address space, and combining in the butterfly network. Its software includes a programming language FORK (as an extension to C and C++ with shared-variables and a few parallel programming constructs), compiler for FORK (*fcc*), a UNIX-like OS, and the P4 library (Communication primitives for C.) The FORK language allows for PRAM-like programming with key constructs as follows.

- **group concept**: all processors belonging to the same processor group are operating strictly synchronously, i.e., they follow the same path of control flow and execute the same instruction at the same time. Also, all processors with the same group have access to the common shared address subspace. Thus, newly allocated "shared" objects exist once for each group allocating them.
- **farm <statement>**
  within the farm body, any synchronization is suspended.
- **fork(e1; e2; e3) <statement>**
  First, the shared expression e1 is evaluated to the number of subgroups to be created. Then the current leaf group is split into that many subgroups. Evaluating e2, every processor determines the number of the newly created leaf group it will be member of. Finally, by evaluating e3, the processor c an readjust its current processor number within the new leaf group.

1.4.2 The XMT Project

The Explicit-multithreading (XMT) is a parallel programming approach for exploiting on-chip parallelism [10]. Its hardware is a single special proprietary chip (PRAM-on-a-chip) including special circuits for parallel prefix operations. Its software includes a parallel language (as an extension of C); it requires a very sophisticated parallel compiler to generate and schedule instructions for the special chip. Unlike many MPP machines, XMT aims to achieve speed-ups for smaller input computations, such as those which might be encountered on desktop hardware.

The XMT programming language allows for PRAM-like programming (but not exactly PRAM programming) [10] with key parallel constructs as follows.

- **spawn(n,0)** {
  statements
  } **join()**
  A parallel region is delineated by spawn and join statements. Synchronization is achieved through the prefix-sum and join commands. Every thread executing the parallel code is assigned a unique thread ID, designated TID. The spawn statement takes as arguments the number of threads to spawn and the ID of the first thread. (Note: The spawn region does not allow nesting of another spawn region due to the difficulty to realize with hardware support.)

- **xfork( );**
  The xfork statement is used within an **fspawn** block to indicate that an additional thread should be created.

1.4.3 The CRAY MTA

The Cray MTA hardware is a multithreaded architecture that offers scalable uniform shared memory [7]. Each MTA processor has special hardware to switch quickly and efficiently between multiple threads of execution. Each processor holds, in hardware, the execution state of as many as 128 threads. Switching between threads on a processor is done in hardware with no software overhead. On each clock cycle, each processor switches to a different resident thread and issues one instruction from that thread. Thus, if a particular thread is waiting on some resource (e.g., memory, I/O, or synchronization), the processor remains busy executing instructions on behalf of other threads.

The MTA processors switch between threads to hide latencies, rather than running multiple threads concurrently.

MTA has the following key parallel language constructs and compiler directives for programming.

- **Explicit parallelism:**
  FUTURE (<future variable>) <future statement clauses>
  A future statement creates a new thread to execute the body of the future. The MTA "future" is best used to implement task-level parallelism (asynchronous) and the parallelism in recursive computations. When used in expressing fine-grain parallelism, the MTA "future" serves to provide tips for the parallelizing compiler to do software-pipelining in code generation.
- **Implicit parallelism:** The MTA programming environment allows automatic parallelization of the loops as indicated by the programmers using compiler directives. This requires the support of a heroic compiler.

1.4.4 Our Solution: PRAM C

In contrast, PRAM C is motivated by the need to provide a good programming model for current and future (general purpose) parallel architectures. The parallel programming constructs of PRAM C are designed with the intension to allow PRAM programming (not just PRAM-like). The PRAM became our model of choice after consideration of existing parallel programming models and a clear understanding of the desired features of any programming model. PRAM C will be compatible with existing parallel hardware architectures, and interoperate with existing software libraries. As such, new special hardware is not required and PRAM C is a software solution. The two central ideas of PRAM C are the use of a large

number of light-weight threads to simulate PRAM threads, the multiplexing of these threads to the physical processors, and the bundling of fine-grain communication to virtual shared memory. One distinction of PRAM C from other related research is that PRAM C can be a pure software solution to support PRAM programming; that is, it does not require special purpose hardware.

2 Design of PRAM C

In this section, we present the design of PRAM C, including an ANSI C language extension, a thin runtime layer, and discussion of design issues.

2.1 Our Implementation of the PRAM Model: A High-level View

Conceptually, our support for the PRAM model is visualized in in Figure 2.

We have added "private" memory to each of the PRAM processors; this is not usually assumed in the abstract PRAM model, but is not incompatible with the abstract PRAM model. The programmer can make good use of this private memory to reduce the amount of fine-grained communication, and thus any communication delay introduced, for performance improvement. On hardware systems with physically-distributed memory, the shared data structures are physically distributed across the memory of all real processors. The Communication Manager bundles the shared memory access requests, and periodically sends the bundles to the appropriate physical processors by message passing for resolution. Resolved shared accesses are then fed back to the virtual PRAM processors.

PRAM processors (as threads in the runtime) are virtual in the sense they are emulated by physical processors through multiplexing; and this work is handled by the Thread Scheduler. More details of the Communication Manager and the Thread Scheduler are discussed in the Section 5.

2.2 PRAM C Features

PRAM C is a simple extension to ANSI C that adds constructs and data types to support the PRAM model.

2.2.1 Parallel Control Constructs

- **PRAM_do**: This construct takes as argument an integer K. This construct indicates that function "f(...)" will be executed by K virtual PRAM processors. K is a way to

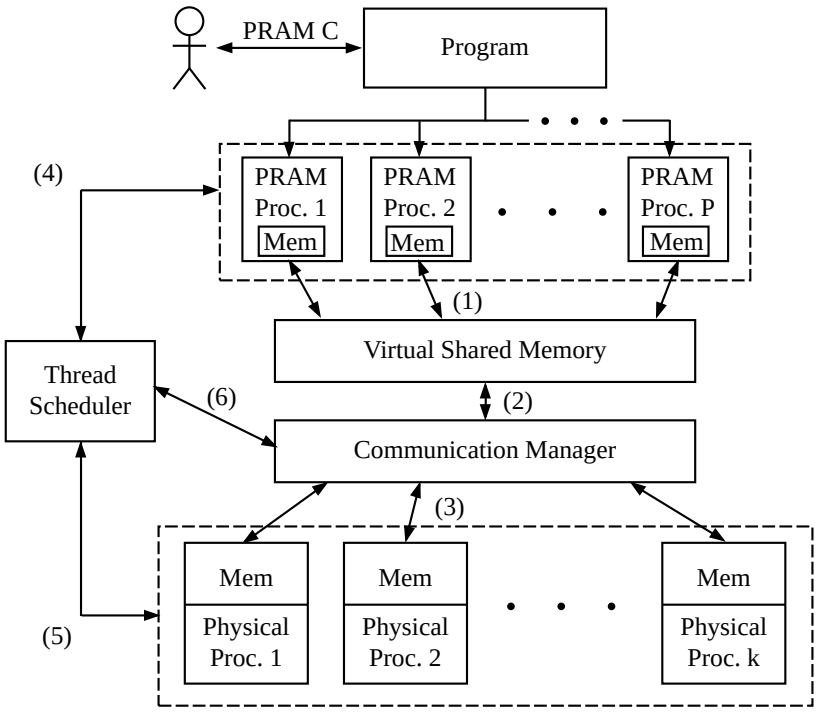

Figure 2. Our Implementation of the PRAM Programming Model: (A Conceptual View) We extend the basic PRAM model to provide an implementable model compatible with modern parallel hardware. A PRAM C program is written based on virtual PRAM processors, which interact with a virtual shared memory interface. Each physical processor supports multiple ($\frac{P}{K}$ on average) virtual PRAM processors via multiplexing. The execution of virtual PRAM processors on each physical processor is scheduled by a Thread Scheduler. The virtual shared memory accesses is registered with a Communication Manager that maps the global, shared accesses to distributed memory and uses the message passing layer (or other communication subsystem) for remote reads and writes. (1) Fine- and coarse-grained parallel accesses are allowed between virtual PRAM processors and the virtual shared memory. (2) The virtual shared memory registers the PRAM memory accesses with the Communication Manager. (3) Requests for physically non-local memory are bundled by the Communication Manager and served in bulk (by message-passing). (4) The virtual PRAM processors are implemented as light-weight threads. (5) PRAM processors are multiplexed to a much smaller number of physical processors by the thread library. **Note:** There is one Thread Scheduler and a Communication Mangager running on each physical processor.

express the degree of fine-grained parallelism in the algorithm that the programmer wants the computer system to exploit.

```
PRAM_do(K): f(...);
```

The theoretical PRAM model requires a SIMD style of execution of its instructions; that is, execution of every instruction is synchronized. Here inside a **PRAM_do** construct, execution of the statements follows a SPMD model; and synchronization is required only at the end of **PRAM_do** construct and also at (implicit) barriers as described next. This relaxation should help performance.

No explicit barrier should be necessary inside **PRAM_do**. However, there are implicit barriers within such a construct. These implicit barriers are honored by the group of virtual PRAM processors within PRAM_do construct:

- The programmer can use the ANSI C block statement syntax (statements enclosed within curly braces {...}) to indicate the execution of the block of code needs to be synchronized, implying a group barrier at the end of the C block.
- Any simple or compound statement implies a group barrier at the end whenever a shared array access appears in the statement. This should be sufficient to support the SIMD style of synchronization, because program correctness can only be affected by out-of-order shared memory accesses.

Note that these implicit group barriers can be handled by the thread scheduling scheme, and thus do not require explicit synchronization.

- **PRAM_fork_join**: This indicates that C functions "f1(...)", "f2(...)", ... will be called in parallel.

```
PRAM_fork
   C_function:  f1(...);
   C_function:  f2(...);
     ...
PRAM_join;
```

Different branches of execution of a PRAM_fork proceed independently; that is, there is no synchronization between any two separate branches. This will be useful in writing parallel recursive functions. This construct also provides a way to express MIMD style coarse-grained parallelism; and group synchronization will be provided by runtime services when needed.

The new constructs (**PRAM_fork** and **PRAM_do**) can be nested. They can also be nested with other compound statements of ANSI C.

2.2.2 Data Types and Their Physical Layout

We augment the ANSI C data types with the concept of **shared**. A shared variable is accessible by all the virtual PRAM processors as well as threads executing other parts of the program. This is introduced to support the shared memory precessing needed by the PRAM. For example,

```
shared double A[N];
```

A shared array is physically laid out across all the physical processors with equal partitions. In this example, assuming P physical processors, elements $A[0]$ ... $A[\frac{N}{P}-1]$ will be located in the the first processor; elements $A[\frac{N}{P}]$ ... $A[\frac{2N}{P}-1]$ will be located in the second processor; and so on. Based on such a formulation, it is easy to determine in which physical processor any element of a shared array is located. This is important in the processes of bundling fine-grained shared memory (array) accesses. Other layouts could be used to handle shared items, so long as it is possible to determine the location of memory from information available at runtime.

Variables that are not shared are local to the thread and thus local to the processor executing the thread, therefore, requests to local variables do not require remote communication and can all be resolved within the host processors.

Variables declared in a PRAM_do are also local variables. Conceptually, it helps to think that there are duplicated sets of such variables with each set belonging to the **private** memory space of one of the virtual PRAM processors. A virtual PRAM processor's accesses to its private memory certainly do not require remote communications between physical processors. This is in contrast to a virtual PRAM processor's shared memory accesses; some of these may be resolvable within the host physical processor while others will have to be served via remote requests to other physical processors.

2.2.3 A Thin Runtime Layer

The thin runtime layer provides a **group_barrier(...)** library functions to manage synchronization of different threads for MIMD style coarse-grained parallelism. Typically, this should only be needed to synchronize threads created for a **PRAM_fork**; for fine-grained parallelism at the virtual PRAM processors, their SIMD style synchronization is done by implicit barriers (as discussed in the definition of the **PRAM_do** construct).

The runtime also provides an interface to a light-weight thread library for thread creation. New threads are created when executing **PRAM_fork** and **PRAM_do** statements, to do the functions in the fork branches and to represent the virtual PRAM processors, respectively. The runtime provides some library functions for the jobs, **create_threads_for_function(...)** and **create_threads_for_PRAM(...)**.

Shared memory accesses are all registered with the runtime, which will periodically decide which access requests can be resolved locally within the host physical processor and which remotely. Remote requests will be bundled together according to destination, and then the bundles will be sent out by message passing. Therefore we will need runtime library function to register shared memory read and shared memory write requests, **PRAM_read(...)**, and **PRAM_write(...)**. The runtime also provides services for copying of sections of the elements between shared and local arrays. This is useful to support coarse-grained communication. **PRAM_copy_private_to_shared(...)** and **PRAM_copy_shared_to_private(...)**. More details are provided in Section 5.

Another useful function is **rank_in_PRAM_do(...)**, which returns the relative rank in the group of virtual PRAM processors introduced in a **PRAM_do**.

2.3 Design Considerations for PRAM C

2.3.1 Separation Between Physical Processors and Virtual PRAM Processors

In reality, the actual number of physical processors available is usually several order of magnitude smaller than the number of processors imagined by the academic parallel algorithm designers in order for their algorithms to be interesting. We choose to separate the concept of virtual PRAM processors from the physical processors for the following reasons.

- PRAM algorithms are usually designed assuming a large number of processors. The processors count in the algorithm is usually a function of the size of input data (say N). One of the purposes for this is to maximize the inherent parallelism of the problem that can be exploited.
- Such a practice can actually be beneficial to performance here. This is because in our program execution model, multiple virtual PRAM processors are served by a physical processor; and fine-grained shared memory accesses by the virtual PRAM processors are bundled together for batch processing. The more virtual PRAM processor as compared to the physical processors, the more likely bundling can reduce average fine-grained communication cost, and thus improve performance.

2.3.2 Brent's Lemma in Action: Making It Easier for Algorithm Designers

Our idea discussed in this subsection is motivated by the following lemma.

Lemma 2.1 (Brent's Lemma [2]) *If a computation can be performed in t steps with q operations on a parallel computer (formally, a PRAM) with an unbounded number of processors, then the computation can be performed in $t + (q - t)/p$ steps with p processors.*

The ability to use a much larger number of PRAM processors compared to the number of physical processors has potential benefits.

Academic (PRAM) parallel algorithm designers typically try to achieve two goals: (1) to achieve maximum parallelism by using as many processors as possible to bring down the parallel time complexity, and (2) to achieve the so-called cost-optimality. A parallel algorithm is cost-optimal if

$$O(\text{ parallel time } \star \text{ processor count }) = O(\text{optimal sequential time complexity})$$

Very often in practice, a simpler sub-optimal-cost parallel algorithm is first designed using a large number of processors to achieve the best the parallel time complexity, say $T(N)$. Then a smaller number of processors are used to simulate the work of the original larger number of processors with complicated and ad-hoc processor load-balancing techniques. It is very often possible to use a much smaller number of processors to do the work of more processors in the sub-optimal-cost algorithm and yet maintain the time complexity $T(N)$; this is because in the sub-optimal-cost algorithm some processors are either idle or finished long before other processors. The "simulation" using less processors allows for those idle processors to be ignored. However, the resulted cost-optimal algorithm is usually much harder to implement. For example, a trivial parallel algorithm to merge two sorted lists of sizes N each takes $O(\log N)$ time using N processors. It takes a much more involved algorithm to achieve $O(\log N)$ using $\frac{N}{\log N}$ processors to achieve the $O(N)$ optimal cost complexity.

In our model, this kind of simulation will done by the runtime with a much smaller number of physical processors executing the threads (representing the virtual PRAM processors.) Naturally, only ready threads will be served. This way, the user can choose to implement an easier but sub-optimal parallel algorithm and yet hope to achieve comparable or even better performance than the much more sophisticated asymptotically cost-optimal algorithm for the same problem.

3 PRAM C Strengths and Potential Benefits

Let us look at PRAM C in terms of the requirements for practical parallel programming environments: expressiveness, ease of use, application performance, and scalability. We shall also consider other potential benefits.

- Language expressiveness:
  - Prior (fine-grain) parallel programming languages do not provide the users with direct control of the processors. For example, in data-parallel programming languages typically provide a mask array of boolean values, which is used to control the operations of other arrays (elements) depending on the true or false values

of the mask array. UPC provides the **UPC_forall** loop to operate on shared arrays, while an "affinity" expression is used to hint the compiler to schedule the operations on the appropriate physical processors according to locality of data layout.

- However, fine-grain parallel algorithm designers would like to have direct and exact control over the operations of each individual processor, because it is a natural way to think in terms of which processor does what. For example, they often like to express something like *Let processor P_i work on all the elements of a shared array whose indices are divisible by i (or other conditions based on the value of i.*

  Our new **PRAM_do** is introduced to provide exactly this kind of expressiveness. Converting parallel PRAM algorithms to a program in our language should be straightforward, similar to the way sequential algorithms are converted to C program.

- Our programming environment supports the CRCW PRAM model, where CRCW means concurrent-reads and concurrent-writes to a shared memory location are allowed. In the case of concurrent-writes, one of the write attempts will succeed; but which one will succeed is not deterministic. This is the most relaxed and the most powerful version of the PRAM model.

- It would be the easiest (to use) parallel programming model to date because it requires minimal efforts to translate from a parallel algorithm to a parallel program. Therefore, this would revitalize the interests in parallel algorithm research and also attract a larger number of parallel application programmers.

- As a stand-alone programming environment, it supports both fine-grain and coarse-grain (both SPMD and MIMD styles) parallel programming. It has the potential to provide better application performance than current message-passing systems. (Detailed discussion is in the Performance section.)

- This programming environment should be scalable to a hardware architecture of any number of processors.

- PRAM C can be easily merged into existing programming environments, such as MPI, OpenMP, and even UPC ([5, 8, 4])

- The idea in our proposed environment has the potential to enable the most efficient parallel programming environment for parallel applications over all existing models (MPI, OpenMP, GAS), because it bundles communications across the whole application and it exploits both fine-grain and coarse-grain parallelism together in one program.

- The ability to implement parallel algorithms designed for a PRAM provides a bridge into this lost "treasure island" (two decades of intensive algorithm research).

4 Program Execution Model

A program will be executed by threads. Initially, one thread is created and assigned to one of the available processors to be executed. Additional threads are created as the flow of control enters various PRAM constructs (**PRAM_do** and **PRAM_fork**). Threads are partitioned among the executing processors for load balancing of work.

4.1 Thread Creation and Physical Processor Allocation

New threads are created in the following instances:

- When the flow of control enters the **PRAM_do(N)** construct, N threads will be created to emulate the behavior of the virtual PRAM processors. (These newly created threads are evenly distributed across all physical processors.)
- When the flow of control enters a **PRAM_fork()** construct, multiple threads will be created, one for each branch (function call) within the fork statement.

When possible, newly created threads are evenly distributed across all physical processors. More sophisticated processor-allocation or load-balancing schemes could be implemented.

4.2 Thread Execution

A thread is executed by one physical processor. The processor executes the threads until one of the following events:

- **Access to shared memory:** In this case, execution will be suspended; the thread will be placed in a queue of pending threads; and the shared memory access request will be queued by the *Communication Manager*.
- **An (implicit) PRAM barrier statement:** In this case, execution will be suspended; the thread will be placed in a queue of pending threads related to the particular barrier. The programmer can use curly braces {...} (a C notation) to group statements inside a **PRAM_do** to indicate that executions of this group by all the virtual PRAM processors need to be synchronized. Therefore, there is an implicit barrier statement at the end of the group.
- **End of the thread:** In this case, the thread is "removed" from the processor.
- **A PRAM construct:** New threads will be created as discussed above.

- **End of an ANSI C compound statement if shared memory access is involved:** Because the abstract PRAM model is silent on the details of SIMD operation around branch and loop statements, we have adopted the convention that all PRAM processors synchronize at the end of such compound statements if any PRAM processor accessed shared memory within the compound statement.

4.3 Physical Processor Execution

Each physical processor maintains several queues of threads according to their state of execution. At a high level, these are:

- a pending queue due to communication
- a pending queue due to PRAM barrier
- a ready queue

A physical processor serves each thread in the ready queue one by one until the queue is emptied. The processor then turns to serve the pending queue due to communication. Those shared memory access requests that touch remote physical memory are bundled, along with thread migration requests if needed for load balancing. Communication managers on each processor exchange their payloads of shared memory access requests and thread migration packages. If the processor receives a thread migration package, it creates the threads as requested and places them in its ready queue. It then serves the shared memory access requests. The requests resolvable locally within the physical processor's own memory will be processed and the corresponding pending threads will be released to the ready queue. The requests received from remote processors are served and then bundled for return. The processor then transmits these to their respective remote processors, and waits to receive its serviced remote requests. Once these remotely-serviced requests are received and processed, the processor can then resume its execution of threads out of the ready queue.

4.4 Virtual PRAM Processor Execution

The number of virtual PRAM processors used in the algorithm can be much larger than the real physical processors available for the program. During program execution, the PRAM processors are partitioned and assigned to the physical processors for real execution. Therefore, it is typical that one physical processor will be serving multiple virtual PRAM processors.

4.5 Shared Memory Access By Virtual PRAM Processors

Shared memory accesses by virtual PRAM processors are queued and those PRAM processors are placed in a pending queue. These shared memory accesses are then processed all together. Those requiring remote physical accesses will be bundled together and sent over to other processors for service using a message-passing library; and those resolvable locally within the physical processors will then be processed. The pending shared memory access requests are considered resolved when both the locally resolvable requests are served and the remote requests are returned and processed. At this point, all the pending virtual PRAM processors (threads) will become ready.

5 Runtime System Components

The runtime system has the following components.

- **A Message-passing library (MPI)** for sending and receiving bundles of remote memory accesses. Thread migration packages are also transported using this library.
- **A Communication Manager** for managing the shared memory accesses for the the virtual PRAM processors.
- **A Thread Scheduler** to run on each physical processor for scheduling various queues of threads running on the processor.
- **A Light-weight Thread Library** for creating, running and retiring threads.

5.1 Communication Manager and Thread Scheduler Activities

To help understand the runtime system, we shall describe the behavior of the Communication Manager and the Thread Scheduler according to one of the numerous possible scheduling strategies.

5.1.1 Shared Data Access Request Handling

On a non-shared memory hardware architecture, virtual shared data is implemented by distributing the data across the local memory of the physical processors using a simple layout pattern. Each physical processor runs a copy of the Communication Manager.

As shown in Figure 3, the Communication Manager is tightly integrated with the Thread Scheduler. When a thread makes a request for shared data, the thread's execution is suspended and placed into a pending buffer. Meanwhile, the shared data request is registered

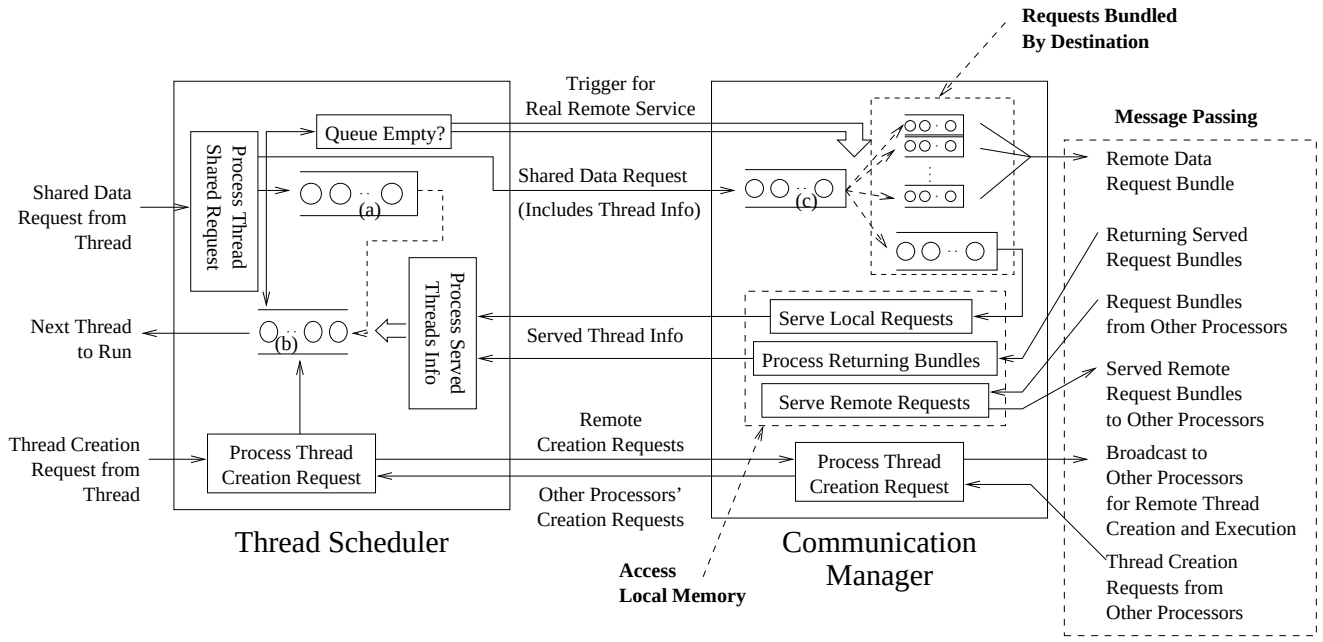

Figure 3. The Runtime System. (a) Buffer of pending threads. (b) Queue of ready threads. (c) Shared data request buffer.

with the Communication Manager, which puts the request into a buffer of all pending requests. The Thread Scheduler then selects another thread from the ready queue for execution. This goes on until the ready queue becomes empty; at this point the Thread Scheduler sends a trigger to the Communication Manager to start processing the buffered requests. The Communication Manager then bundles the requests according to the the destination processor, and sends them out via message passing; requests resolvable locally within the host processor will be handled separately. Next, the Communication Manager turns to receive and serve remote request bundles from other physical processors; it will send response packets back to other processors after their request bundles are served. The Communication Manager then receives the responses from the other processors that serve its request bundles. After all the data access requests are resolved, either locally or remotely, the Communication Manager passes information on the related threads to the Thread Scheduler; the Thread Scheduler can then release all those threads from the pending buffer into the ready queue.

5.1.2 Thread Creation Request Handling

When the Thread Scheduler receives a request for creation of threads, it will create some of the threads locally and then pass on the rest to the Communication Manager which will then evenly partition and broadcast them to the rest of the processors for thread creation and execution. Similarly, the Communication Manager and the Thread Scheduler also handle

thread creation requests from other processors.

5.2 Communication Manager Interface

The communication manager provides certain library functions to be called by the generated code from the PRAM C translator.

- **PRAM_read(...)**: This function returns the value of a (remote) memory address.

```
type PRAM_read(type * addr) {
    type temp;
    CM_read(&temp, addr);
          /* a request to the communication manager */
    return temp;
}
```

- **PRAM_write(...)**: This function writes a value into a (remote) memory address.

```
void PRAM_write(type * addr, type value) {
    type temp;
    temp = value;
    CM_write(temp, addr);
          /* a request to the communication manager */
}
```

- **PRAM_copy_private_to_shared(A, B, ...)**: This function copies a segment of elements from private array A to shared array B
- **PRAM_copy_shared_to_private(A, B, ...)**: This function copies a segment of elements from shared array A to private array B

5.3 Light-weight Thread Library Interface

We list some of the functions needed for the generated code produced by our translator.

- **create_threads_for_PRAM:** This function creates a group of N threads, distributed evenly across processors.

```
create_threads_for_PRAM(int N, void * funcName) {
     - P = physical processor count
     - for (i = 0; i < P; i++) {
            - Create a request for physical processor i to
              create threads for "funcName" with group ranks
              n the range of [(i-1)N/P, iN/P]
       }

     - Send all the thread creation requests to the rest of
       the physical processors

     - Locally create N/P number of threads for "funcName"
       with ranks in the range [0, N/P-1]
}
```

- **create_threads_for_C:** This function creates a thread locally for a C function.

```
create_threads_for_function(* func_ptr) {
     - create a thread for function "func_ptr";
}
```

5.4 Thread Scheduler Interface

- **PRAM_group_barrier(G, N)** provides a group synchronization (barrier). Here G is the name of an existing group; and the calling thread must be its member. N is the number of group members that must call PRAM_group_barrier before they are all released. The count argument is required because processes could be joining the given group after other processes have called the group barrier (the group may be dynamic). All participating group members must call PRAM_group_barrier with the same count value. Having been successfully passed, PRAM_group_barrier can be called again by the same group using the same group name. Our group idea is borrowed from PVM [**?**].

- **PRAM_join_group(G)** adds the executing thread to group G.

- **PRAM_leave_group(G)** removes the executing thread from group G.

- Inside the PRAM_do, sometimes it is necessary use the runtime service call **rank_in_PRAM_do()** to get the relative rank among the group virtual PRAM processors introduced by the PRAM_do.

6 Translator for the PRAM C

The translator inputs a PRAM C program and outputs a program written in C with embedded runtime library calls. Here we provide the general idea of how the new programming constructs will be translated.

- **PRAM_do** construct

```
{
   ...
   PRAM_do(K): f(...);
   ...
}
```

  will be translated into something like

```
{
    ...
    create_threads_for_PRAM(K, *f);
    ...
}
```

- The **PRAM_fork** construct will be translated similarly.
- Shared array access in an expression will be translated into library function calls **PRAM_read(...)**, and **PRAM_write(...)** which will submit shared data access requests to the Communication Manager; and when the requests are later served, it will return the data so that computation for the expression can proceed.
- An assignment statement into a shared array element such as the following

```
{
  A[i] = ...
}
```

  will be translated into something like the following.

```
{
  temp = ...
  PRAM_write(A[i], temp);
}
```

7 Possible Implementations

The PRAM C programming environment can be implemented as a stand-alone environment. It can also be merged into existing programming environments.

- The translator inputs the source program written in PRAM C; and outputs a program written in C with embedded calls to a runtime library. This translator should be straightforward to write, as it need not be a full-blown compiler, but can leverage existing compilers.
- The runtime system implementation needs the following components:
  - A message-passing library. We can use MPI.
  - A light-weight thread library for thread creation. Such a light-weight thread library can either be found from open-source or be written by ourselves.
  - A Communication Manager (library) to process the fine-grained shared-memory access (bundling, sending, receiving, and servicing). This communication manager should be relatively easy to implement.
  - A Thread Scheduler to maintain the various queues of threads.

8 Performance Discussion and Comparison of Models

There are features of our model that may impact application performance. These features include a large number of threads, bundling of fine-grain parallelism, and thread assignment to physical processors.

8.1 The Essence of Our Model

Our proposed model introduces a (software) layer for thread and communication scheduling. This layer has performance overhead. The question is whether the potential saving in communication can outweigh the overhead. We believe it should for most cases.

- This new layer enables the introduction of our PRAM_do construct and the separation of virtual (PRAM) processors and physical processors in the PRAM_do construct, which makes our model the most natural, expressive, and easy to program over all existing models. The PRAM_do, on the other hand, creates opportunities for communcation scheduling (bundling).
- If this layer works out in practice, it can (and should) be broadly adopted to be merged in or sit on top other models (such as MPI).

8.2 Many Threads and Communication Bundling

Potentially a large number of threads can be created during the program execution. Managing the threads requires some system overhead. However, these threads tend to be very light-weight; and the thread maintenance is local to each individual processor.

On the other hand, the large number of threads are mostly created to emulate the virtual PRAM processors due to PRAM_do. As we discussed earlier in the design consideration section, this is to enable more bundling of fine-grain communications. We expect the saving in communication cost to far outweigh the overhead of thread management. Communication bundling is performed across the whole application, not just on those requests from a single group of virtual PRAM processors belonging to one PRAM_do.

Typically PRAM algorithms can be written such that the number of PRAM processors is a separate parameter independent of the size of the input data to be processed. In coding such algorithms, the programmer has the freedom to adjust the virtual PRAM processor count in the PRAM_do statements, if it is felt that the total number of threads in the whole application may be overwhelming the runtime system.

When threads are first created due to PRAM_do, they are evenly distributed and assigned to run across all the physical processors. Such a thread creation scheme is designed to keep the physical processor work load balanced. Since these threads tend to be very light-weight, and threads created for a PRAM_do are fairly synchronized, we expect the physical processor work load to be reasonably balanced throughout the program execution.

8.3 Comparison with Other Models

In our model, virtual PRAM processors are implemented as threads and multiplexed onto the physical processors, which should keep physical processors busy most of the time. **Fine-grain communications are bundled; and most importantly, this bundling applies all the pending remote data requests in the whole application, not just those incurred by the virtual PRAM processors in a single PRAM_do.** This way, the runtime system can be bundling a large volume of fine-grain communications not even anticipated by the programmers. Besides, this model allows exploitation of fine-grain parallelism inherent to the problems as well as coarse-grain parallelism.

Compared to a message-passing programming model (MPI), only coarse-grain parallelism is supported, and the programmers have to "bundle" their own remote data requests related to a relatively small region of the application. This bundling is limited since different components in an application are often based on different algorithms, and written by different programmers over an extended period of time. This difference in communication "bundling" along with the ability to exploit fine-grain parallelism provide good opportunities for PRAM C to outperform the current message passing systems in some applications.

Recently introduced programming languages such as Unified Parallel C (UPC) and Co-Array Fortran (CAF) support the Partitioned Global Address Space (GAS) model. UPC, for example, also supports fine-grain parallelism. But programmers are expected to manage the data locality by themselves. Communication bundling is not currently implemented in any UPC compilers. In fact, we are developing a whole application support layer for UPC just to manage data locality for common algorithms used in both numerical and discrete computation. It requires some real effort to convert a fine-grain parallel algorithm to a UPC program [3]. In comparison, our model allows both coarse-grain and fine-grain parallelism. Converting a fine-grain parallel algorithm to PRAM C is a matter of simple syntax translation exercise, very much like converting a sequential algorithm into C code.

Compared to the OpenMP model, our model does not require true shared-memory hardware and it is expected to be very scalable to large scale parallel machines. And our model should also be much easier to program, and is not as loop-oriented as OpenMP.

9 Program Style Guide

PRAM C supports both fine-grain and coarse-grain parallel programming. We list some of the possibilities as to what capabilities in the programming environment to use in order to support various style of parallelism.

- MIMD style coarse-grain parallel programming.
  - **PRAM_fork** can be used to create parallel threads.
  - Execution of these threads can be synchronized using **group_barrier** provided by the runtime services.
  - Coarse-grain communications can be done by copying sections of arrays between shared and local variables.
- SPMD style coarse-grain parallel programming:
  - **PRAM_do** can be used.
  - Copy sections of shared array data into the virtual PRAM processors private memory. Try to do computation using private data as much as possible.
  - There will be few implicit barriers for the PRAM processors when computation is done mostly on private memory, thus avoiding (SIMD style) frequent synchronizations
- SIMD style find-grain parallel programming:
  - **PRAM_do** can be used;
  - The implicit barriers **PRAM_do** for the PRAM processors enforce the SIMD semantics.

9.1 Memory Access Style Guide

For portability of the threads created by the PRAM constructs, each thread should only access shared data or data private to it; here shared data includes globally declared variables (such as C static variables) and function parameters of shared types; and data private to a function includes variables declared inside the functions and its formal parameters.

To ensure this PRAM code compliance, we have the following requirements.

- **The "shared" requirement:** all memory to be accessed by more than one virtual PRAM processor should be marked as "shared".
- **The "private" requirement:** all other memory accesses by a virtual PRAM processor must be to its function arguments or to the variables declared the function.
- **Any other memory accesses** by a virtual PRAM processor has undefined behavior.

10 Code Examples

The following are several small examples to provide a flavor of code written in PRAM C.

10.1 Parallel Prefix

The parallel prefix algorithm is guided by an implicit binary tree. We also assume that the length of the array, N, is a power of 2. The generalization is immediate to using an arbitrary binary function.

```
shared int A[N];  /* numbers to be summed for parallel prefix */
main {
    ...;  // initialization
    parallel_prefix(A, N);
}
void parallel_prefix(shared int A[], int N) {
    if (N > 1)
      PRAM_do(N): prefix_labor(A,N);     /* Use N virtual PRAM processors */
}
void prefix_labor(shared int A[], int N) {
       int i, rounds, left, PRAM_ID;
       rounds = log(2, N);
       PRAM_ID = rank_in_PRAM_do();
       /* Populate the tree upwards */
```

```
        for (i = 1; i < rounds; i++) {
            if (PRAM_ID % exp(2,i) == 0) {
                left = A[PRAM_ID - exp(2,i-1)];
                A[PRAM_ID] += left;
            }
        }
        /* Populate the tree downwards */
        for (i = rounds - 1; i > 0; i--) {
            if (PRAM_ID % exp(2,i) == 0) {
                A[PRAM_ID - exp(2,i-1)] =
                    A[PRAM_ID] - left;
            }
        }
}
```

10.2 Parallel QuickSort

QuickSort can be naturally expressed as a recursive algorithm: given an array of numbers, select the pivot, and then proceed recursively on each portion of the array. a pivot element, split the array into all numbers less than the pivot and those greater than

```
shared double A[N];      // to store the input
shared int temp[N];      // for temporary use
main{
    ...;  // input
    qSort(A, 1, N);
    ...;  // output
}
/*************************************************************************
 This function recursively sorts A[low.. high].
 *************************************************************************/
void qSort(shared double A[], int low, int high)
{       double v;  /* a local variable */
        if (low >= high)
             return;
        v = A[low];
        /* Use (high-low+1) virtual PRAM processors */
        PRAM_do(high-low+1): split_labor(A, v, low, high);

        k = temp[high]; /* a side-effect of the parallel_prefix()
                           call inside parallel_split() */
        /* Parallel recursive calls */
        PRAM_fork
                func: qSort(A, low, k-1);
                func: qSort(A, k+1, high);
        PRAM_join;
}
/*************************************************************************
 This function (executed by each virtual PRAM processor) splits A[low.. high]
 into two lists by "v"  such that (A[low..k] <= v) and (A[k+1..high] > v)
 *************************************************************************/
void split_labor(shared int A[], double v, int low, int high)
{       int PRAM_ID, j;     /* private variables to each PRAM processor */
        PRAM_ID = rank_in_PRAM_do();

        /* Every PRAM processor takes an item to compare against "v" */
        if (A[PRAM_ID] < v)
            temp[PRAM_ID] = 1;
        else
```

```
            temp[PRAM_ID] = 0;

        /* Only one of the PRAM processors calls prefix */
        /* function to calculate where each data item should go */
        if (PRAM_ID = low)
             parallel_prefix(&temp[low], high-low+1)
        /* An implicit group barrier right after the above if-then statement */

        /* The i-th PRAM processor moves A[PRAM_ID] to PRAM_ID destination A[j] */
        j = low + temp[PRAM_ID];
        A[j] = A[PRAM_ID];
}
```

11 Conclusion

We have presented the design of a new parallel programming environment, PRAM C. This new environment is based on two new ideas: a thin runtime layer and a simple extension to ANSI C. The thin runtime layer is responsible for thread scheduling and communication bundling throughout the whole application; and it provides the foundation for the introduction of a new parallel programming construct, **PRAM_do**, to allow natural expression of fine-grained PRAM style algorithms. In the **PRAM_do**, the concept of virtual PRAM parallel processors used in a parallel algorithm are separated from the physical parallel processors, which implements the the virtual processors via multiplexing. Such a separation not only allows for the programmer to express maximal parallelism in the algorithm, it also provides opportunities for fine-grained communication to be bundled by the thin runtime layer.

As a stand-alone environment, PRAM C supports both SIMD-style fine-grained as well as SPMD and MIMD style coarse-grained parallel programming. It provides the most expressive and easiest (to use) parallel language constructs for fine-grained parallelism over all existing parallel programming environments.

The proposed thin runtime layer bundles communication across the whole program, therefore PRAM C has to the potential to match and even outperform existing programming environments such as MPI. PRAM C is scalable to systems of any number of parallel processors.

PRAM C can also be integrated into existing parallel programming environments. Separately, the thin runtime layer can also adopted by existing parallel runtime systems for performance improvement.

Finally, ability to efficiently implement parallel fine-grained (PRAM) algorithms will provide a bridge into the lost "treasure island" of two decades of intensive algorithm research.

It will revitalize the interests in new parallel algorithm research; it will also attract a larger number of application programmers.

References

[1] (MTA at NPACI) (url). http://www.npaci.edu/MTA/tera-doc/pg/html/TOC.html.

[2] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of ACM*, pages 201–206, 1974.

[3] R. Brightwell, J. Brown, Q. Stout, and Z. Wen. Experiences implementing parallel sorting algorithms in UPC, 2004. The 5-th UPC Workshop, Washington D.C, September 2004.

[4] W. W. Carlson et al. Introduction the UPC and language specification. Technical Report CCS-TR-99-157, UPC Consortium, 1999.

[5] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 2nd edition, 1999.

[6] Y. Shiloach and U. Vishkin. Finding the maximum, merging and sorting in a parallel computation model. *Journal of Algorithms*, 2:88–102, 1981.

[7] Cray Multi-Thtreaded Architecture (url). http://www.cmf.nrl.navy.mil/CCS/help/mta/.

[8] OpenMP Architecture Review Board. (url). OpenMP fortran application interface version 1.1. http://www.openmp.org.

[9] SB PRAM Project (url). http://www.ida.liu.se/čhrke/fork95.html#Fork95.

[10] U. Vishkin. Explicit Multi-Threading (XMT): A PRAM-On-Chip Vision. http://www.umiacs.umd.edu/users/vishkin/XMT/.

DISTRIBUTION:

1 MS 0817
James Ang, 1422

1 MS 0807
Bob Ballance, 4328

1 MS 0817
Bob Benner, 1422

1 MS 0376
Ted Blacker, 1421

1 MS 1110
Ron Brightwell, 1423

1 MS 1110
Jonathan L. Brown, 1423

1 MS 0382
Kevin Brown, 1543

1 MS 0320
William J. Camp, 1400

1 MS 1110
S. Scott Collis, 1414

1 MS 0318
George Davidson, 1412

1 MS 1110
Erik DeBenedictis, 1423

1 MS 0817
Doug Doerfler, 1422

1 MS 0316
Sudip Dosanjh, 1420

1 MS 0817
Brice Fisher, 1422

1 MS 0382
Mike Glass, 1541

1 MS 0817
Sue Goudy, 1422

1 MS 8960
James Handrock, 9151

1 MS 1110
William Hart, 1415

1 MS 0822
Rena Haynes, 1424

1 MS 1110
Bruce Hendrickson, 1414

1 MS 1110
Michael Heroux, 1414

1 MS 0316
Scott Hutchinson, 1437

1 MS 0817
Sue Kelly, 1422

1 MS 0378
Marlin Kipp, 1431

1 MS 1111
Patrick Knupp, 1411

1 MS 0801
Rob Leland, 4300

1 MS 0370
Scott Mitchell, 1411

1 MS 1110
Steve Plimpton, 1412

1 MS 0807
Mahesh Rajan, 4328

1 MS 1110
Rolf Riesen, 1423

1 MS 0378
Allen Robinson, 1431

1 MS 0318
Elebeorba May, 1412

1 MS 0321
Jennifer Nelson, 1430

1 MS 1110
Cynthia Phillips, 1415

1 MS 1110
Neil Pundit, 1423

1 MS 1111
Mark D. Rintoul, 1412

1 MS 1110
Suzanne Rountree, 1415

1 MS 1111
Andrew Salinger, 1416

1 MS 0378
Stewart Silling, 1431

1 MS 0378
James Strickland, 1433

1 MS 0378
Randall Summers, 1431

1 MS 1110
Jim Tomkins, 1420

1 MS 0370
Tim Trucano, 1411

1 MS 0817
John VanDyke, 1423

1 MS 0817
Courtenay Vaughan, 1422

1 MS 1110
Zhaofang Wen, 1423

1 MS 0822
David White, 1424

1 MS 1110
David Womble, 1410

1 MS 0823
John Zepper, 4320

2 MS 9018
Central Technical Files, 8945-1

2 MS 0899
Technical Library, 9616