LOCK-FREE PARALLEL ACCESS COLLECTIONS

Bruce P. Lester

Department of Computer Science, Maharishi University of Management, Fairfield, Iowa, USA

ABSTRACT

All new computers have multicore processors. To exploit this hardware parallelism for improved performance, the predominant approach today is multithreading using shared variables and locks. This approach has potential data races that can create a nondeterministic program. This paper presents a promising new approach to parallel programming that is both lock-free and deterministic. The standard forall primitive for parallel execution of for-loop iterations is extended into a more highly structured primitive called a Parallel Operation (POP). Each parallel process created by a POP may read shared variables (or shared collections) freely. Shared collections modified by a POP must be selected from a special set of predefined Parallel Access Collections (PAC). Each PAC has several Write Modes that govern parallel updates in a deterministic way. This paper presents an overview of a Prototype Library that implements this POP-PAC approach for the C++ language, including performance results for two benchmark parallel programs.

KEYWORDS

Parallel Programming, Lock-free Programming, Parallel Data Structures, Multithreading.

1. INTRODUCTION

The National Research Council of the USA National Academies of Sciences recently issued a report entitled *The Future of Computing Performance* [1]. Following is a brief excerpt from the Preface of this report (p. vii):

"Fast, inexpensive computers are now essential for nearly all human endeavors and have been a critical factor in increasing economic productivity, enabling new defense systems, and advancing the frontiers of science. ... For the last half-century, computers have been doubling in performance and capacity every couple of years. This remarkable, continuous, exponential growth in computing performance has resulted in an increase by a factor of over 100 per decade and more than a million in the last 40 years. ... The essential engine that made that exponential growth possible is now in considerable danger."

The response of the computer processor manufacturers (such as Intel) to this challenge has been the introduction of chip multiprocessors with multiple processing cores per chip, usually called multicore processors. Each core is essentially a miniature processor, capable of executing its own independent sequence of instructions in parallel with the other cores. Processors for new laptop computers now typically have 2 to 4 cores, processors for desktops have 4 to 6 cores, and high-performance processors have 8 to 12 cores. The number of cores per processor is expected to double every two to three years. This gradually increasing hardware parallelism offers the potential for greatly improved computer performance. Therefore, the NRC Report concludes the following (p. 105):

DOI : 10.5121/ijdps.2014.5301

"Future growth in computing performance will have to come from software parallelism that can exploit hardware parallelism. Programs will need to be expressed by dividing work into multiple computations that execute on separate processors that communicate infrequently or, better yet, not at all."

To achieve this goal, the NRC Report gives the following recommendation:
"Invest in research in and development of programming methods that will enable efficient use of parallel systems not only by parallel systems experts but also by typical programmers." (p. 138)
The purpose of this paper is to address this critical need in computing today, as expressed in the NRC Report. We explore a new high-level parallel programming abstraction that is easy to use and achieves good performance, without the need for locking and without the possibility of data races that can cause nondeterministic program execution.

2. BACKGROUND

The predominant approach to parallel programming in current computer technology is *Multithreading*: the programming language has special primitives for creating parallel threads of activity, which then interact through shared variables and shared data structures. To prevent conflicts in the use of the shared data, locking is used to provide atomic access. Examples of popular multithreading systems are OpenMP [2] and Java Threads [3]. One of the main problems with this multithreading approach is *data races* that result when parallel threads update the same memory location (or one thread reads and the other updates). Depending on which thread wins the race, the final result may be different. This essentially creates a **nondeterministic** program: a program that may produce different outputs for the same input data during different executions. This nondeterminism complicates the software development process, and makes it more difficult to develop reliable software [4]. This introduction of data races and nondeterminism into mainstream computer programs is hardly a step of progress in computer programming, and is considered by many to be unacceptable (see S. Adve, "Data Races are Evil with No Exceptions" [5]).

In multithreaded parallel programming, *locking* is generally used to control access to shared data by parallel threads. If correctly used, locking helps to maintain the consistency of shared data structures. However, locking introduces other problems that complicate parallel programming. Even if used correctly, multiple locks used to protect different data structures can interact in unfortunate ways to create a *program deadlock*: a circular wait among a group of parallel threads which ends program execution prematurely. Furthermore, these deadlocks may occur in a nondeterministic manner, appearing and then disappearing during different executions of the same program with the same input data. Of course, there is a wide variety of proposed solutions to this deadlock problem, but none is completely effective or has achieved wide acceptance in practice [6].

Another problem with locking is performance degradation. The locking operation itself takes considerable time and is an additional execution overhead for the parallel program. Also, the lock by its very nature restricts parallelism by making threads wait, and thus reduces parallel program performance and limits the scalability of the program. (A parallel program is said to be *scalable* if the execution time goes down proportionately as more processors, or cores, are added.) There has been some research on lock-free parallel access data structures [7], but with limited success so far.

We have developed a new high-level parallel programming abstraction that does not require locking and does not allow data races. With this new abstraction, parallel programs are

guaranteed to be deterministic and deadlock-free even in the presence of program bugs. In the following sections, we will describe the details of this new approach to parallel programming.

3. OUR APPROACH

Anyone who has been a serious parallel programmer or parallel algorithm designer knows that the best source of scalable parallelism is program loops. Computationally intensive programs, which are candidates for parallel execution, typically have program loops (or nested loops) in which the loop body is executed many times sequentially. If the loop iterations are fairly independent, they can be executed in parallel by different processor cores to significantly speedup program execution. This property of programs was recognized early in the history of parallel computing. Early parallelizing compilers, such as Parafrase [8], had some limited success with automatic parallelization of sequential program loops. Numerical programs that operate on large multi-dimensional data arrays typically have lots of such loops that in principle can be parallelized. Unfortunately, the automatic parallelization by compilers has shown itself to be too limited. More often than not, a programmer is needed to restructure the underlying algorithm to expose a more complete level of parallelism [9].

This has led to explicit parallel loop constructs in many programming languages (or APIs for parallel programming). Recent versions the Fortran language have a *DOALL* directive for parallel *DO-loop* execution [10]. Many parallel versions of the C programming language have some kind of *forall* instruction, which is a parallel form of the ordinary sequential *for-loop* [11]. The popular OpenMP standard for parallel programming in the C language [2] has a *parallel-for* directive for assigning sequential *for-loop* iterations to different threads for parallel execution. Unfortunately, these parallel loop instructions alone do not solve the problem of data races for access to shared data structures. Therefore, additional primitives such as *locks* are needed to synchronize the parallel processes. This introduces the problems of deadlock and performance degradation as described earlier. Also, if the process synchronization is not done completely correctly by the programmer, data races can remain in the parallel program causing nondeterministic execution.
We feel the parallel *for-loop* is a good step in the right direction moving from sequential to parallel programming, but does not go far enough. A more highly structured (or more abstract) form of the *parallel-for* is needed to prevent data races and guarantee deterministic execution without explicit locking by the programmer. We have developed a new high-level parallel programming abstraction for the C/C++ language that has an *operational* component and a *data* component. The operational component is called a *Parallel Operation* (abbreviated POP). The data component is called a *Parallel Access Collection* (abbreviated PAC).

An ordinary program loop has three basic categories of data:

- Local Data: variables (or data structures) declared local to each loop iteration and only accessible by one loop iteration.
- Read-only Shared Data: variables (or data structures) read by multiple loop iterations, but not updated by any loop iterations.
- Write Shared Data: variables (or data structures) accessed by multiple loop iterations and updated by at least one loop iteration.

Local data and Read-only shared data do not pose a problem for parallel loop execution, and thus can be accessed freely within a POP. Data races and consequent nondeterminacy arise from Write Shared Data. Therefore, access to Write Shared Data must be carefully controlled. In our new parallel programming abstraction, all Write Shared Data of any POP must be selected from a special category of data structures called Parallel Access Collections (PACs). The POPs and PACs work together to allow deterministic access to the Write Shared Data of the parallel loop iterations.

A POP has the following basic syntax:

POP (integer n, FunctionPointer, PAC1, PAC2, ... , PACm)

where

n is the total number of parallel processes (parallel loop iterations)
FunctionPointer is a pointer to the executable function (code) forming the body of the loop
PAC1, ... , **PACm** is a list of all write shared data collections used by the loop body

In the POP abstraction, each loop iteration is viewed as a separate parallel process. However, since the number of physical processors (cores) is limited in practice, the POP may be implemented by assigning groups of parallel processes for execution on each physical processor. Problems with parallel access to shared data collections arise when parallel processes access the same memory location, and at least one is a writer. For the POP abstraction, this problem can be divided into two general categories:

Two parallel processes access the same memory location and
1. one process is a reader, and the other a writer
2. both processes are writers

We resolve the first case (reader and writer) by using deferred update of PACs inside a POP operation. Writes to a PAC by any parallel process (parallel loop iteration) are not seen by other processes of the same POP. During the POP execution, each process is reading the old values of all PACs. This solves the data race problem for parallel reader and writer processes. The second case where both parallel processes are writers is more difficult to resolve. We require that each writeable PAC of a POP has an assigned Write Mode that prevents conflicting writes by parallel processes of the POP. So far in our research we have identified two fundamental Write Modes for PACs:

Private Mode: Each location in the PAC can be written by at most one process during each POP. A single process may write to the same PAC location many times, but two different processes are not permitted to write to the same location. Locations in the PAC are not assigned in advance to the processes. Any process is allowed to write into any PAC location, provided no other process writes to the same location during the POP. This prevents the possibility of write-write data races by the parallel processes of each POP. If a process attempts to write a location already written by another process, this is a runtime error and generates a program exception.

Reduce Mode: Any number of processes is allowed to write the same location in the PAC. All the writes to a given location are combined using an associative and commutative reduction function, such as sum, product, minimum, maximum, logical AND, logical OR. The reduction guarantees that the final result of each PAC location is independent of the relative speeds of the writer processes, and thus removes the possibility of data races. User-defined reduction functions are also allowed, although this introduces the possibility of nondeterministic execution if the user-defined function is not completely associative and commutative.

For all of the input PACs of any POP, the program must assign a *Write Mode* to each PAC before the POP is executed. The implementation of the two *Write Modes* within each PAC is done automatically by the runtime system. The details of the implementation are not seen by the User Program.

4. POP-PAC LIBRARY

We have implemented a prototype library for this POP abstraction in the C++ programming language with three types of PACs. This library was then used for performance testing of two

```
initialize mindist array to infinity;
initialize queue to contain source vertex 0;
mindist[0] = 0;
while queue is not empty do {
  x = head of queue;
  foreach neighboring vertex w of x {
    newdist = mindist[x]+ weight[x][w];
    if (newdist < mindist[w]) {
      mindist[w] = newdist;
      if w not in queue then append w to queue;
    }
  }
}
```

Figure 1 Sequential Shortest Path Algorithm

parallel benchmark programs. In the following sections, we will describe the details of this library and the benchmarks.

The three types of PACs implemented in the C++ library are as follows:

- **pArray**: A simple one-dimensional array where each element has the same base type
- **pArray2**: A standard two-dimensional version of the *pArray*
- **pList**: An extension of *pArray* where elements can be appended to the end of the list

Each of these types of PACs is capable of functioning in either of the two Write Modes (*Private* or *Reduce*). Each PAC is represented in the C++ library as a template class. The prototype library also contains several template classes to implement the POP operation. We used this prototype library to implement two parallel algorithms:

- **Shortest Path in a Graph**: This parallel algorithm uses three *pArray* PACs all in *Reduce Mode*.
- **Jacobi Relaxation to solve a Partial Differential Equation**: This parallel algorithm uses a total of three *pArray2* PACs. Two of the PACs are in *Private* Mode, and one in *Reduce* Mode.

We will begin our discussion of the feasibility study by describing the details of the parallel program for determining the Shortest Path in a weighted, directed graph. To simplify the algorithm, only the shortest distance from the source vertex 0 to every other vertex will be computed, and stored in a one-dimensional array *mindist*. To represent a graph with *n* vertices, use an *n* by *n* two-dimensional array called *weight*. The value of *weight*[*x*][*w*] will give the weight of the edge from vertex *x* to vertex *w* in the graph. The algorithm for finding the shortest distances is based on gradually finding new shorter distances to each vertex. Whenever a new short distance is found to a vertex *x*, then each neighboring vertex *w* of vertex *x* is examined to determine if *mindist*[*x*] + *weight*[*x*][*w*] is less than *mindist*[*w*]. If so, a new shorter distance has been found to vertex *w* via *x*. Vertex *w* is then put into a queue of vertices that have to be further explored. When the neighbors of *x* have all been considered in this way, the algorithm selects a new vertex *x* from the queue and examines all of its neighbors. When the queue is empty, the algorithm terminates and the *mindist* array contains the final shortest distance from the source vertex to every other vertex. A sequential version of this shortest path algorithm is shown in Figure 1 (ref. [11], p. 412).

For a large graph, the queue will grow in size very quickly and then gradually diminish as the algorithm progresses until it is eventually empty and the algorithm terminates. The sequential

```
void main(  ) {
  initialize mindist array to infinity;
  initialize inflag array to False;
  inflag[0] = True;  mindist[0] = 0;  // set vertex 0 as source
  Set Write Mode of mindist, inflag, outflag, more to Reduce;
  Set Reduction Function of mindist to minimum( );
  Set Reduction Function of inflag, outflag, more to logicalOR( );
  do {
    more = False;
    POP ( n, ShortPathFunction, &mindist, &outflag, &more );
    Exchange outflag and inflag arrays;
  } while (!more)
}


// this function forms the body of each process
void ShortPathFunction( int x, mindist, outflag, more ){
  if (inflag[x]) { // if vertex x is in the queue
    foreach neighboring vertex w of x {
      newdist = mindist[x]+ weight[x][w];
      if (newdist < mindist[w]) {
        // found a new shorter distance to w via x
        mindist[w] = newdist;
        outflag[w] = True;
        more = True;
      }
    }
    inflag[x] = False; // vertex x is removed from queue
}
```

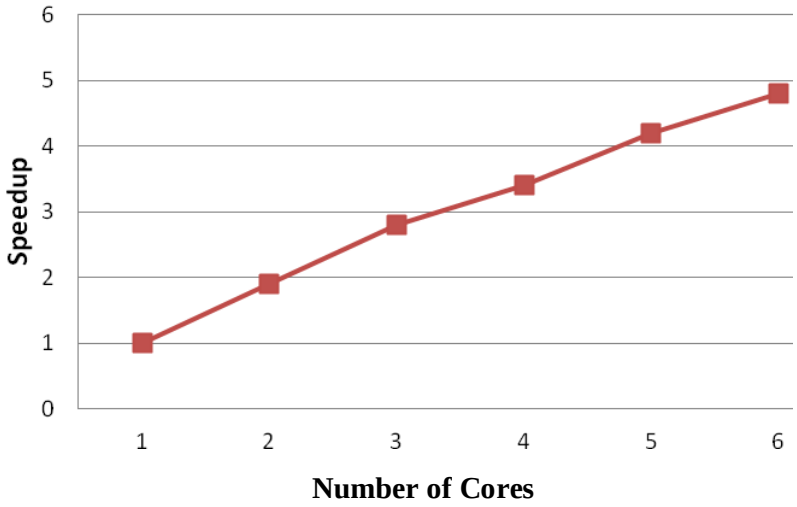
Figure 2 Parallel Shortest Path Algorithm

algorithm is naturally parallelized by considering all the vertices in the queue in parallel. Let us use an *inflag*[*n*] array of *True* and *False* values, which indicates whether each vertex of the graph is currently in the queue. Then a POP can be created that assigns one process to each element of the *inflag* array. If any element *inflag*[*x*] is *True*, then the process examines the neighboring vertices *w* of vertex *x*, as described above for the sequential algorithm. If a new shorter path is found to vertex *w*, then *mindist*[*w*] is updated, vertex *w* is added to the queue, and *outflag*[*w*] is set to *True*. When the current POP finishes, then the *outflag* array replaces the *inflag* array and becomes the starting point for the next POP. These POPs are iteratively repeated until an *outflag* array of all *False* results.

In an ordinary parallel implementation of this algorithm using multithreading, locking would be needed to prevent conflicts during updates of the arrays. However, by using PACs with the proper Write Modes, locking is not needed in our parallel formulation. This parallel shortest path algorithm requires three PACs: *inflag*, *outflag*, and *mindist* are each represented as an integer *pArray* with *n* elements. To simplify the termination detection, an additional integer *pArray* called *more* is used with one element. All of the *pArrays* are set to *Reduce* Mode. Since the *inflag* array indicates which vertices are currently in the queue, a queue data structure is not actually needed in the parallel algorithm. A centralized queue would pose a performance bottleneck and limit scalability. A high-level pseudo-code description of the parallel algorithm is shown in Figure 2.

The *main*( ) function contains the initializations and the high-level *do-while* loop that drives the algorithm. Each loop iteration contains one POP that creates *n* parallel processes: one for each

Figure 3 Self-Speedup of Parallel Shortest Path



vertex of the graph. Each process is given its own unique index from 0 to *n*-1. The *ShortPathFunction*( ) forms the body of each process. Notice that the *ShortPathFunction*( ) is almost identical to the sequential version of the Shortest Path Algorithm shown in Figure 1. The *inflag* array is a PAC, but it is read-only in *ShortPathFunction*( ), and thus does not have to be included in the list of input PACs. It is possible that two parallel processes may update the same element in the *mindist* array. This potential for a data race is resolved by the *minimum*( ) function, which is the Reduction Function assigned to the *mindist* array. Since *minimum*( ) is associative and commutative, the final result in *mindist* will be independent of the relative timing of the process updates to *mindist*. The same is true of the PACs *outflag* and *more*, which have *logicalOR*( ) as the Reduction Function. Thus, no locking or other process synchronization is needed in the parallel algorithm. All parallel writes to the PACs are resolved in a deterministic way by the assigned Write Modes.

As part of our feasibility study, we implemented a prototype C++ library for the POP and the three types of PACs described above (*pArray*, *pArray2*, *pList*). The POP and PACs are all implemented as template classes in the library. The library also contains some additional "helper" template classes to implement the Write Modes. This POP and PAC library was used to execute the Parallel Shortest Path Algorithm shown in Figure 2 on a Dell Studio XPS workstation with a six-core AMD Opteron processor. Figure 3 shows the performance results for a graph with 4000 vertices, as the number of cores applied to the parallel computation is varied from one to six. The graph in Figure 3 shows that the POP program for Shortest Path does appear to scale well: the program runs 4.8 times faster using six cores than using one core.

5. ADDITIONAL EXAMPLE: JACOBI RELAXATION

The next example program considered in the Feasibility study is Solving a Partial Differential Equation using Jacobi Relaxation. Consider a simple application to determine the voltage level across the surface of a two-dimensional rectangular metal plate, assuming that the voltage along the four boundaries is held constant. Using a two-dimensional coordinate system with x and y axes, the voltage function on the metal plate $v(x, y)$ can be computed by solving Laplace's Equation in two dimensions:

$$\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} = 0$$

This equation can be solved numerically using a two-dimensional array of discrete points across the surface of the metal sheet. Initially, the points along the boundaries are assigned the appropriate constant voltage. The internal points are all set to 0 initially. Then *Jacobi Relaxation* is used to iteratively recompute the voltage at each internal point as the average of the four

```
void main(  ) {
  initialize pArray2 A;
  Set Write Mode of A, B to Private;
  Set Write Mode of pArray maxchange to Reduce;
  Set Reduction Function of maxchange to maximum( );
  tolerance = .00001;
  do {
    // perform 10 Jacobi iterations
    for ( i = 0; i < 10; i++) {
      POP ( n, JacobiFunction, &B );
      Exchange A and B arrays;
    }
    // convergence test every ten iterations
    POP ( n, ConvergenceTestFunction, &maxchange);
  } while (maxchange > tolerance)
}

// this function forms the body of each process
void JacobiFunction( int myNum, B ){
  // compute row and column position of this process myNum
  row = myNum/n;  col = myNum % n;
  // compute new value of this point as average of four neighbors
  B[row][col] = A[row-1][col] + A[row+1][col]
              + A[row][col-1] + A[row][col+1] / 4.0;
}

// this function performs the convergence test
void ConvergenceTestFunction( int myNum, maxchange ) {
  // compute row and column position of this process myNum
  row = myNum/n;  col = myNum % n;
  // compute change in value of this point
  maxchange = fabs( B[row][col] - A[row][col] );
}
```

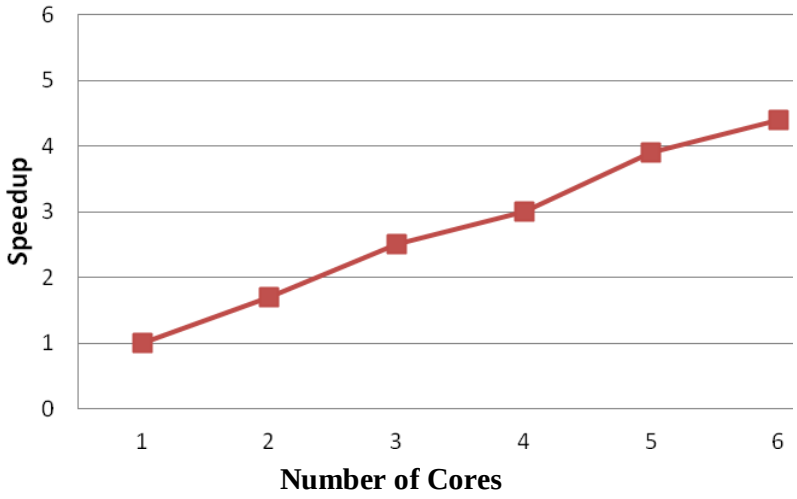
Figure 4 Parallel Jacobi Relaxation Program

immediate neighboring points (above, below, left, right). Convergence is tested by comparing a desired tolerance value to the maximum change in voltage across the entire grid during each iteration.

The parallel version of this program requires two different POP operations: one for performing the Jacobi iterations, and another for the Convergence test. Three different PACs are required:

- Array *A*[*n*][*n*] contains the current value at each point at the start of the iteration
- Array *B*[*n*][*n*] holds the newly computed value at each point (average of four neighboring points)
- Scalar *maxchange* holds the maximum change across the whole array during a given iteration

Figure 5 Self-Speedup of Parallel Jacobi Relaxation



Arrays *A* and *B* are represented in the parallel program as *pArray2* PACs in *Private* Mode. Variable *maxchange* can be represented as a *pArray* (with one element) in *Reduce* Mode. The parallel program is shown in Figure 4. The *main*( ) function drives the whole program with two nested loops. The first loop performs ten Jacobi iterations using a POP that calls the *JacobiFunction*( ). The ten iterations are sequential, but within each iteration, a separate parallel process is assigned to compute each point of the two-dimensional grid. Since only one parallel process writes to each point of *pArray2 B*, *Private* mode is used for the *Write* Mode. After ten Jacobi iterations, the second POP is invoked to perform the convergence test by subtracting the values in the corresponding locations of *A* and *B*. By having each process write its change directly into *maxchange*, the overall maximum value of the change is computed because *maxchange* is set to *Reduce* Mode with reduction function *maximum*( ).

The Jacobi Relaxation program of Figure 4 was tested using an array of 40,000 points on a Dell Workstation with a six-core processor. The graph of Figure 5 shows the performance results as the number of cores assigned to the program is varied from one to six. The results are similar to the Shortest Path program. The parallel Jacobi program appears to exhibit good scalability: the program runs 4.4 times faster using six cores than one core.

6. RESEARCH COMPARISON

There has been a considerable amount of research over the years in lock-free parallel programming and deterministic parallel programming. We will briefly focus on some of the major trends in this research that are most relevant to our proposed research direction. One important approach to lock-free parallel programming is *Software Transactional Memory* (STM) [ref. 12, 13, 14]. With STM, the programmer can specify certain portions of code to be transactions. The STM implementation will automatically guarantee that each transaction is executed in an *atomic* way, without interference from other transactions. This is usually implemented with some form of *optimistic scheduling*: parallel transactions may proceed freely until some possibility of conflict is detected. Then one of the conflicting transactions is rolled back and rescheduled. This guarantees a *serializable* execution of parallel transactions – equivalent to some serial (sequential) execution of the transactions. The main problem with STM has been the implementation overhead that slows parallel program execution.

In our opinion, Software Transactional Memory has some usefulness, but does not go far enough. A *serializable* schedule of transactions is not necessarily *deterministic*. For example, consider two parallel transactions to update a variable *x* with initial value 50: transaction *A* reads the current value of *x*, squares it, and writes the result back to *x*; transaction *B* reads the current value of *x*, adds 10, and writes the result back to *x*. If Transaction *A* executes first then *B*, the result is $x = 2510$. If Transaction *B* executes first then *A*, the result is $x = 3600$. Both schedules are serializable, and thus permitted by STM. Since both transactions read and write variable *x*, there is a possibility of interference between the transactions. This kind of interference is prevented by STM, but it still does not guarantee determinacy. STM eliminates some of the data races in a parallel program, but not all. Thus, the parallel programmer will still be faced with debugging nondeterministic programs that may produce different outputs using the same input data.

Our proposed use of POPs and PACs for parallel programming is really quite different from Software Transactional Memory. The POP is a very specific way of creating a team of parallel process. The PACs are special data structures with *Write Modes*. Whereas, with STM, the programmer just inserts some *begin* and *end* transaction primitives into the program. These can be inserted anywhere and apply to any type of data. Our PACs have *Reduce* and *Private* Mode – there is nothing like this in Software Transactional Memory.

Another notable trend of research in parallel programming that has some relevance to our research on POPs and PACs is *data parallel* programming [15, 16, 17], sometimes called "Collection-Oriented" (CO) parallel programming [18]. With CO programming, certain commonly used highly parallel operations on collections (data structures) are built into a collections library. This saves the parallel programmer from having to "reinvent the wheel" in every program. Some simple examples of such highly parallel operations are Broadcast, Reduce, Permute, Map, Scatter, Gather [19]. These standard operations are implemented internally in the collections library in a highly parallel way. As is the case with ordinary sequential program libraries, this approach is very useful and improves programmer productivity. However, practical experience has shown that the standard library operations alone are not sufficient to structure a whole parallel program. Often the parallel program requires some "custom" operations that are not easily created from a combination of the available collections library operations. Thus, our proposed POPs and PACs are not a replacement for collection-oriented parallel programming, but a supplement to it. The two approaches are complementary and can be used together in the same program.

Another research direction for deterministic parallel programming is *deterministic scheduling* of multithreaded programs with shared variables [20, 21, 22, 23]. This approach allows the data races to remain in the parallel program, but forces the data races to be resolved in a repeatable (deterministic) manner. One major drawback of this approach is the resultant parallel programs are *fragile*: even an insignificant modification in the program code can change the outcome of a data race in some unrelated portion of the program, thus changing the final outcome of the program. As long as the program is not modified at all, the deterministic scheduling will guarantee a repeatable outcome for all the data races, but a small change in the code may expose a data race bug in a completely different portion of the program. Our POP-PAC approach is quite different than this *deterministic scheduling* approach. Our approach is more highly structured because we have a specific parallel control structure (POP) and specific parallel access data structures (PACs) with many Write Modes.

7. FUTURE RESEARCH

As defined in Section 3, each POP has an associated function, which is the executable code supplied by the User. *Local variables* are (non-static) variables declared inside the function body. Variables defined outside the function body are considered as *shared variables*. For the POP to be deterministic, the User's POP function must satisfy the following simple rules:

- Local variables may be read or written freely.
- Shared variables must be read-only, except for those explicitly included as arguments in the POP invocation.

These are fairly simple rules for a programmer to follow in order to guarantee determinacy of each POP. However, the prototype library we describe in this paper does not enforce these rules. Therefore, it is possible for bugs in the User function to violate these rules and cause nondeterministic execution of the POP. The next phase of our research on the POP-PAC approach will be to modify the C++ compiler (and language runtime system) to enforce these rules. Then all program POPs will be guaranteed to be deterministic even in User programs with bugs.

REFERENCES

[1] S. H. Fuller and L. I. Millett, Editors, (2011) *The Future of Computing Performance: Game Over or Next Level?*, National Research Council, National Academies Press.

[2] B. Chapman, et al., (2007) *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press.

[3] C. Horstmann and G. Cornell, (2007) *Core Java, Volume I--Fundamentals* (8th Edition). Prentice Hall, pp. 715-808.

[4] E. Lee, "The problem with threads,"(2006) *Computer*, Vol. 39, No. 5, pp. 33–42.

[5] S. Adve, (2010) "Data Races are Evil with No Exceptions," *Communications of the ACM*, vol. 53, no. 11, p. 84.

[6] S. Bensalem, et al., (2006) "Dynamic Deadlock Analysis of Multi-threaded Programs," In *Proceedings First Haifa International Conference on Hardware and Software Verification and Testing*, pp. 208-223.

[7] N.Shavit, (2011) "Data Structures in the Multicore Age," *Communications of the ACM*, vol. 54, no. 3, p. 76-84.

[8] Polychronopoulos, C., et al., (1989) "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing and Scheduling Programs on Multiprocessors," In Proceedings 1989 International Conference on Parallel Processing, Vol. II, pp. 39-48.

[9] G. Tournavitis, et al., (2009) "Towards a Holistic Approach to Auto-Parallelization" presented at *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 2009), Toronto, Canada, pp. 177-187.

[10] *Fortran Programming Guide*, (2001) Sun Microsystems, Palo Alto, CA.

[11] B. Lester, (2006) *The Art of Parallel Programming* (Second Edition). First World Publishing.

[12] A. Dragojevic et al., (2011) "Why STM Can Be More Than a Research Toy," *Communications of the ACM*, vol. 54, no. 4, pp. 70-77.

[13] J. Larus and C. Kozyrakis, (2008) "Transactional Memory," *Communications of the ACM*, vol. 51, no. 7, pp. 80-88.

[14] C. Caşcaval, et al. , (2008) "Software Transactional Memory: Why is it Only a Research Toy?" *Communications of the ACM*, vol. 51, no. 1, pp. 40-46.

[15] G. E. Blelloch, (1990) *Vector Models for Data-Parallel Computing*. The MIT Press, Cambridge, Massachusetts.

[16] G. Tanase, et. al., (2011) "The STAPL Parallel Container Framework," In *Proceeding ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPOPP), pp. 235-246.

[17] *Array Building Blocks Application Programming Interface Reference Manual*. (2011) Intel Corporation.

[18] B. Lester, (2011) "Improving Performance of Collection-Oriented Operations through Parallel Fusion," In *Proceedings of The World Congress on Engineering 2011*, pp. 1519-1529.

[19] B. Lester, (1993) *The Art of Parallel Programming* (First Edition). Prentice Hall.

[20] T. Bergan et al. (2010) "CoreDet: A compiler and runtime system for deterministic multithreaded execution," In *Proceedings Architectural Support for Programming Languages and Operating Systems* (ASPLOS 2010), pp. 53-64.

[21] M. Olszewski, J. Ansel, and S. Amarasinghe, (2009) "Kendo: Efficient Deterministic Multithreading in Software," In *Proceedings Architectural Support for Programming Languages and Operating Systems* (ASPLOS 2009), pp. 97-108.

[22] E. D. Berger et al., (2009) "Grace: Safe multithreaded programming for C/C++," In *Proceedings ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (OOPSLA 2009), pp. 81-96.
[23] H. Cui, J. Wu, and J. Yang. (2010) "Stable deterministic multithreading through schedule memoization," In *Proceedings 9th USENIX Conference on Operating Systems Design and Implementation* (OSDI 10).

Author

Dr. Bruce Lester received his Ph.D. in Computer Science from M.I.T. in 1974. He was a Lecturer in the Department of Electrical Engineering and Computer Science at Princeton University for two years. Dr. Lester is one of the pioneering researchers in the field of parallel computing, and has published a textbook and numerous research papers in this area. He founded the Computer Science Department at Maharishi University of Management (MUM), where he has been a faculty member for sixteen years.