Determinacy and Repeatability of Parallel Program Schemata

Jack B. Dennis
Computer Science and Artificial Intelligence Laboratory
MIT, Cambridge, Massachusetts
Email: dennis@csail.mit.edu

Guang R. Gao
University of Delaware
Newark, Delaware
Email: ggao@capsl.udel.edu

Vivek Sarkar
Rice University
Houston, Texas
Email: vsarkar@rice.edu

***Abstract*—The concept of "determinism" of parallel programs and parallel systems has received a lot of attention since the dawn of computing, with multiple proposals for formal and informal definitions of deterministic execution. In this paper, we present precise definitions of two related properties of program schemata — *determinacy* and *repeatability*. A key advantage of providing definitions for schemata rather than concrete programs is that it simplifies the task for programmers and tools to check these properties. The definitions of these properties are provided for schemata arising from data flow programs and task-parallel programs, thereby also establishing new relationships between the two models. Our hope is that these sound definitions will help provide a more precise framework for discussions of these properties.**

I. Introduction

The challenge of running applications on parallel systems has received increased attention in recent years due to the foreseeable growth of multicore and many-core processors. There is widespread agreement that a major obstacle to developing parallel software with comparable productivity to that of sequential software arises from the potential for programs written in current parallel programming models to exhibit non-repeatable behavior for the same input. The term, "nondeterministic", is often used to refer to this behavior, but without a sound definition of exactly what it means. In Wikipedia we find these uses of "nondeterministic":

> "Nondeterministic algorithms are often used when the problem solved by the algorithm inherently allows multiple outcomes (or when there is a single outcome with multiple paths by which the outcome may be discovered, each equally preferable). Crucially, every outcome the nondeterministic algorithm produces is valid."
>
> "In computational complexity theory, nondeterministic algorithms are ones that, at every possible step, can allow for multiple continuations. . . . These algorithms do not arrive at a solution for every possible computational path; however, they are guaranteed to arrive at a correct solution for some path."

[1]This work was supported in part by the National Science Foundation through grant CCF-0937832, CCF-0833122, CCF-0937907, and OCI-0904534. This work was also partially supported by European FP7 project TERAFLUX, id. 249013.

The terms "deterministic" and "nondeterministic" were used in the classical switching theory of Finite State Machines (FSMs). An FSM is defined by a state transition function that specifies a next state for every combination of current state and input symbol. If the transition function specifies a unique next state for every combination, the FSM is said to be *deterministic*. However, if the transition function specifies several possible next states (being a relation rather than a function), the meaning is that the actual next state may be any one of those that are possible according to the transition relation. Such an FSM is said to be a *nondeterministic* FSM or an NDFSM. A well known result of switching theory is that for any NDSM, one can construct an equivalent deterministic FSM, possibly with a larger set of states.

In this paper, we review the original concept of *determinacy* as applied by Richard Karp and Raymond Miller to their model of Parallel Program Schemata (Section II). This motivates our interest in the property we call *repeatability*, which captures the informal notion of a parallel program having consistent behavior in every run. We provide definitions of related properties of determinacy and repeatability in the context of parallel program schemata arising from data flow programs (Section III) and task-parallel programs (Section IV). Section IV also introduces two analytic tools, computation graphs and function graphs, that help understanding the semantics and behavior of task parallel programs. Section V discusses means of ensuring repeatability of programs, illustrating ideas using the Habanero Java programming language. Our hope is that this knowledge will be of use in developing tools for easing the programming burden for parallel computing and lead to improved parallel programming languages.

II. Determinacy

Interest in repeatability of asynchronous parallel computations dates back to at least the 1960s. The concept of determinacy of parallel program schemata was published by Karp and Miller in a series of papers and reports from 1964 through 1969 [1], [2], [3][1]. Karp and Miller studied a model of

[1]Karp and Miller published a version of their work in the IEEE Eighth Annual Symposium on Switching and Automata Theory[4], where uses of the terms "deterministic" and "nondeterministic" were well-established. We believe they chose "determinate" to distinguish their new concept from the established usage.

parallel computation they called *parallel program schemata*. A parallel program schema is a set of memory locations, a set of operators and a *control*. Each operator A is defined by two functions:

1) A mapping (total function) F_A from values in a *domain* of memory locations to a set of values put into a *range* of memory locations. The *domain* and *range* need not be disjoint. Function F_A is applied when the operator is selected for execution by the control.
2) A total function G_A that determines which of a finite number of possible "outcomes" occurs when operator execution terminates.

The *control* of a schema is a nondeterministic state machine and concerns a set of states and an alphabet of output symbols containing an initiation symbol for each operator and a termination symbol for each possible outcome of each operator. The control is specified by a transition function that assigns a "next state" for each combination of state and symbol. It is required that the transition function assign a next state for every combination of state and termination symbol, but is partial for the initiation symbols. One state q_0 is specified as the initial state.

A run of a "Parallel Program Schema", starting from some initial memory contents and the initial control state, yields a sequence of operator initiations and terminations that Karp and Miller call a *computation*, and a corresponding sequence of memory states. In any state the control may allow many choices of action, generating runs in which different interleavings of operator execution occur. Karp and Miller defined a schema as *determinate* if it satisfies the following property:

> Given any two runs of the schema with the same initial contents of memory and same initial control state, the sequences of values for each memory location in the two runs are identical, *regardless of the specific functions F and G chosen for mapping and selection by each operator.*

Thus, determinacy is a property of "uninterpreted program schemata", a kind of computation model studied by the Russian scientist Ianov [5] and extended by Luckham, Park and Patterson [6], [7].

The authors of computer programs are typically interested in reproducibility of the results produced by their programs, and are generally not concerned whether intermediate results are the same in every run. This is what authors appear to mean when they write that a program is deterministic. (For example, see Blelloch [8]). This is the property we will call *final value repeatability*:

> A parallel program is final value repeatable if, for a *chosen specific interpretation* of its operators this condition is satisfied: For any given input set, and for every execution of the program, the final value of each variable is unique.

To contrast this property of a program with the notion of determinacy, we introduce a weaker property that we call *final value determinacy*:

```
Program:                          Schema:

cilk int foo (int x)              cilk int foo (int x)
{                                 {
   spawn { x = x + 1; }              spawn { A; }
   spawn { x = x + 2; }              spawn { B; }

   sync;                             sync;

   return (x);                       return (x);
}                                 }
```

Fig. 1. A simple parallel program in Cilk and its schema.

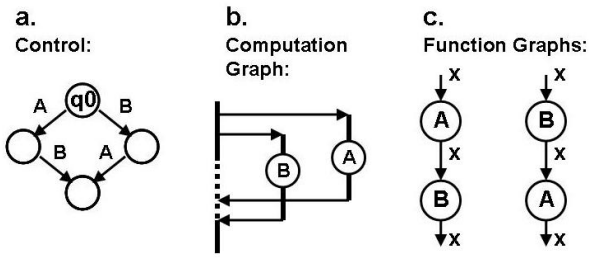

Fig. 2. Karp-Miller schema for a Cilk program. a. The control. b. Execution sequences. c. Function composition graphs.

> A parallel program schema is final value determinate if, for *any interpretation* of its operators this condition is satisfied: For any given input set, and for every execution of the program, the final value of each variable is unique.

Note that, to apply the ideas of Karp and Miller to programs, we have substituted "variable" for "memory location". Figure 1 provides an illustration using a simple parallel program written in the Cilk language [9]. The abstraction made in the schema is to replace the two statements with operators `A` and `B` that have the following interpretations in the Cilk program:

```
Statement:          Operator:
x = x + 1;        A (x + 1 -> x);
x = x + 2;        B (x + 2 -> x);
```

The control component of the schema is the simple nondeterministic state machine in Figure 2a[2]. The *computation graph* for this program schema is shown in Figure 2b. In it, each vertical bar represents one task – the master task and the two worker tasks spawned by spawn statements. Along each bar is shown the sequence of operations performed by the task. The horizontal arrows show each event that either initiates or terminates a task. The master task, which has no operations to perform, waits until the two worker tasks have completed. Clearly, the computation graph is a directed acyclic graph (DAG). We define an *execution sequence* of the schema to be a total ordering of the operations of the computation graph. For this simple schema there are just two execution sequences

[2]In fact, the control in the Karp-Miller schema can represent the intermediate states in which an operator has been initiated but not terminated. We omit those intermediate states because the operators in this example are atomic.

– (A, B) and (B, A). For any execution sequence of a schema, we may construct a graph, called a *function composition graph* of the schema, that shows the composition of operator functions evaluated by the execution to determine the final values of each variable. Our simple schema has the two FCGs shown in Figure 2c. This shows that the final value of variable x may be either $B(A(x_0))$ or $A(B(x_0))$, where x_0 is the initial value of variable x. Because we can easily choose interpretations of A and B such that the final value of variable x is different, we conclude that this schema is not determinate.

On the other hand, the parallel program is repeatable because, for the specific interpretations of operators A and B the final value of x will always be $x_0 + 3$. Thus:

> Any parallel program with a determinate schema is repeatable. However, a schema of a repeatable program is not necessarily determinate.

III. Data Flow Program Schemata

Data flow models of parallel computation are often cited as originating with work at MIT [10], [11]. However, several versions of the data flow concept appeared in the late 1960s. Bert Sutherland completed a PhD thesis on the "On-Line Graphical Specification of Computer Procedures" in which he demonstrated construction of data flow diagrams using the display and light pen of the MIT TX-2 computer [12]. In 1968, Duane Adams completed a thesis at Stanford entitled "A Computation Model with Data Flow Sequencing" which was, to the best of our knowledge, the first use of the term "data flow" to describe a model of computation. At MIT, Jorge Rodriguez completed a PhD Thesis in 1967 (although not published until 1969 [13]), entitled "A Graph Model of Parallel Computations". The work of Rodriguez had the most direct influence on subsequent research on data flow in the MIT Computation Structures Group.

A *data flow schema* [11], [10] is an interconnection of components (actors) enabled by the presence of data items at input ports; the components act by sending data items to inputs of other components. The behavior of each component is described by a function that defines output values in terms of input values and determines on which output ports data items are sent. Figure 3 shows a basic set of data flow actors.

For discussing determinacy of data flow schemata, we need a broader definition than provided by Karp and Miller. There is no memory with domain and range locations for the operators in a data flow model. For these reasons, we discuss the determinacy of a *system* with m input ports and n output ports as shown in Figure 4. The system runs by accepting data items from inputs, performing internal activity, and delivering data items at outputs. The figure illustrates the system responding to a set of sequences of data items presented at each input port (the *presented input*). The *ultimate output* is the set of sequences of output data items delivered to the output ports, if the system is permitted to run as long as it chooses (perhaps forever). We adopt the following definition for determinacy of a system:

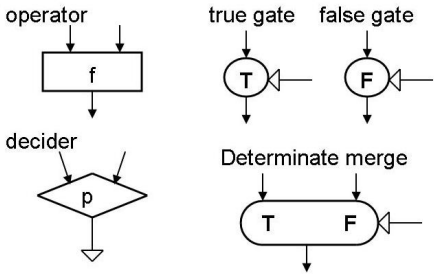

Fig. 3. A set of basic data flow actors. An operator applies a specified function to its inputs, a decider performs a test of its inputs using a specified predicate; a true or false gate actor passes a value if its control input is true or false, respectively, otherwise absorbing the input value with no response; the determinate merge uses its control input to pass one value from the indicated side – if a value is present at the other input it is left for future action.

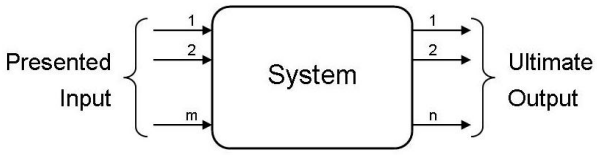

Fig. 4. A system having m inputs and n outputs. The inputs and outputs are sequences of values.

> A system is *determinate* if the same ultimate output is produced for every run of the system for a presented input, and this is true for all choices of input and for all interpretations of operators.

It can readily be seen that the components of data flow schemata are determinate when individually considered to be systems. As in the case of Karp-Miller determinacy, the property of determinacy applies to uninterpreted data flow schemata – the property holds regardless of the specific functions assigned as the behavior of operators, or predicates to deciders. Patil showed that any interconnection of determinate systems yields a system that also is determinate [14]. Consequently any data flow schema constructed of components with determinate (functional) behavior is itself determinate.

We may illustrate the distinction between determinate and repeatable systems using data flow schemata. This requires use of a nondeterminate merge actor, for otherwise, any system of data flow actors is both determinate and repeatable. Figure 5 shows a schema that is not determinate, yet represents a repeatable computation. If f and g are the same function, then the two possible output sequences of the schema, $f(x), g(x)$ and $g(x), f(x)$, will be the same.

IV. Task Parallelism

For discussing the determinacy and repeatability of computer programs, we will use the Habanero-Java (HJ) language [15] as an exemplar of task-parallel languages. However, similar ideas can be applied to other task-parallel programming models including Cilk [9] and OpenMP 3.0 [16], and to global-view PGAS languages including X10 [17] and Chapel [18].

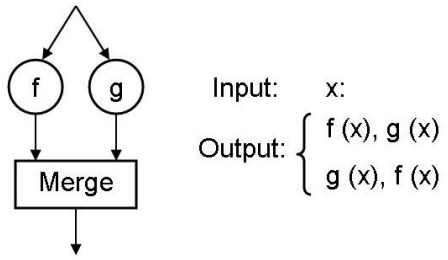

Fig. 5. A data flow schema, using a non-determinate merge actor, that is repeatable if the functions f and g are the same. The non-determinate merge passes values to its output port, making an arbitrary choice if values are present at both input ports.

In fact, HJ was derived from X10, with extensions to X10's task-parallel model that include data-driven tasks [19], finish accumulators [20], phasers [21], and phaser accumulators [22].

A. Async/Finish statements in HJ

The basic primitives of task parallelism relate to creation and termination of tasks. In HJ, this support is provided by the `async` and `finish` statements of the X10 language.

> **async:** The statement "**async** ⟨*stmt*⟩" causes the parent task to create a new child task to execute ⟨*stmt*⟩ *asynchronously* with (*i.e.,* before, after, or in parallel with) the remainder of the parent task.
>
> **finish:** The statement "**finish** ⟨*stmt*⟩" causes the parent task to execute ⟨*stmt*⟩ and then wait until all `async` tasks created within ⟨*stmt*⟩ have completed, including transitively spawned tasks.

Each dynamic instance T_A of an `async` task has a unique *Immediately Enclosing Finish* (IEF) instance F of a `finish` statement during program execution, where F is the dynamic innermost `finish` containing T_A [21]. The IEF of T_A is always contained in an ancestor task of T_A. There is an implicit `finish` scope surrounding the body of the program, so program execution will only end after all `async` tasks have completed.

Figure 6 shows an HJ program for multiplying a matrix A by vector X using async and finish to compute the dot products in separate concurrent tasks. We may convert an HJ program into a Karp-Miller parallel program schema by replacing statements with operators. For our example a possible schema for the HJ program is shown in Figure 7.

In this schema, operators C, D, and E model operations on variable `i` for the outer `for` loop, and operators P, Q, and R model operations on variable `j` for the inner loop; note that operators D and Q have two outcomes: true and false. The remaining operators model the actual computations performed on elements of matrix A and vector X and have a single outcome.

It is evident that the definitions of final value determinate and repeatable in Section II apply to schemata of HJ programs.

```
int[] multiplyByVector (int [][] A, int []  X) {
   int m = A.length;
   int n = X.length;
   finish {
      int [] Y = new int [ m ];
      for (int i = 0; i < m; i++) {
      async {
         int sum = 0.;
         for (int j = 0; j < n; j++) {
            sum += A [ i ][ j ] * X [ j ];
         }
         Y [ i ] = sum;
      } // async
   } // finish
   return Y;
}
```

Fig. 6. HJ program for matrix-vector multiplication. The use of `finish` here is redundant because an HJ program is treated as though it is enclosed in a `finish` statement.

```
int[] multiplyByVector (
         int [][] A, int [] X) {
   B ( -> m, n);
   finish {
      L ( -> Y);
      // scheme of the outer 'for' statement
      C ( -> i);
      outer:
      if (D (i, m)) {
         async {
            M ( -> sum);
            // scheme of the inner 'for' statement
            P ( -> j);
            inner:
            if (Q (i,n) ) {
               F(A, X, i, j, sum -> sum)
               R ( j -> j);
               goto inner;
            }
            G(Y, i, sum -> Y);
         }
         E (i -> i);
         goto outer;
      }
   }
   return Y;
}
```

Fig. 7. Schema of the HJ program in Figure 6.

B. Execution Sequences

To gain further understanding of the semantics of task parallelism in HJ, it is useful to define the execution sequences an HJ program may generate in execution.

An *execution sequence* is a sequence of events $s_0, s_1, \ldots$ where each event is one of the following kinds of events that can occur during execution of an HJ program.

- A statement: An assignment statement that evaluates application of an operator to values from a domain of memory locations and updates a range of memory locations.
- Beginning of the scope of a finish statement (including the implicit finish statement that encloses the entire program).
- Start of execution of an async statement.

- End of execution of an async statement.
- End of the scope of a finish statement.

Note that we are no longer using the term "operator" in the sense of "transformation of memory", as in the work on program schemata, but in the usual mathematical sense of simply a function from a domain of values to a range of values. This distinction between "operator" and "statement" will be important for the discussion in Section V.

The execution sequence for a run of an HJ program for specific values of input variables contains all events that occur in each task during execution, arbitrarily interleaved into a single sequence while maintaining the order in which they occur in the tasks they came from. A large number of distinct execution sequences are possible. Instead of exhibiting one, we will introduce the *computation graph* which represents the collection of execution sequences of an HJ program or schema.

C. Computation Graphs for HJ Programs

A *Computation Graph* is a directed acyclic graph (DAG) that captures the meaning of execution of a program as a *partial order*. Specifically, a Computation Graph (CG) for an HJ program consists of:

- A set of *nodes*, where each node represents an event occurring in some task during execution of the HJ program.
- A set of *directed edges* that represent ordering constraints [23]:
  1) *Continue* edges represent the ordering of events within a task, drawn as vertical bars.
  2) A *spawn* edge connects an async begin event in a parent task to the start of the sequence of events in a new child `async` task. A spawn edge is drawn as a horizontal arrow from the async begin event in the parent task to the start of the event sequence of the child task.
  3) A *join* edge connects the termination of a task to the task executing its immediately enclosing finish (IEF) statement.

A computation graph for an HJ program may be constructed for any execution sequence w by following this procedure:

- Draw a vertical bar to represent the outermost (top level) task.
- Consider the next event in sequence w. It is either a statement, or one of the four concurrency control events:
  - statement: Place its operator symbol on the current task bar.
  - begin `finish`: Place a "begin `finish`" marker on the current task bar containing a count of the number of async tasks for which this task is the IEF.
  - end `finish`: If the termination count of the matching "begin `finish`" event is greater than zero, begin a dashed section of the task bar indicating a waiting period.
  - begin `async`: Place a "begin async" mark on the current task bar; draw a right horizontal arrow to the start of a new bar representing a child of the current task; make the child task the current task.
  - end `async`: End the current task bar and draw a left horizontal arrow to the bar representing the parent task; decrement the termination counter of the IEF at its begin marker; if the termination count is now zero, and the parent task is not waiting, place an "end `finish`" marker on the parent task bar and make it the current task; otherwise: make the parent task of the async task the current task.

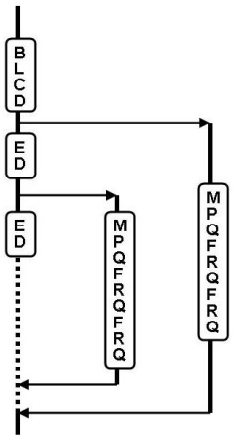

Fig. 8. Computation Graph for example HJ program of Figure 6

In general, many execution sequences of an HJ program will generate the same computation graph due to the variety of interleavings of events that task parallelism allows. Computation graphs will differ for execution sequences in which the action of control operators for conditionals and loops alter the pattern of task creation. For example the outer loop in the program schema of figure 7 determines the number of tasks created to execute the inner loop.

Figure 8 shows a computation graph for this HJ program when executed with inputs $m = 2$ and $n = 2$. In fact, the computation graph in the figure is unique for this input. (This property holds for HJ programs with parallelism restricted to async and finish, and a further condition that no data race occurs for the given input [24].)

D. Function Composition Graphs

A *Function Composition Graph* (FCG) is a DAG showing the composition of function applications resulting from program execution.

Each operator node of the DAG represents an application of the operator of a statement. To construct the FCG for an execution sequence of an HJ program, we will use a map M that assigns an operator node to each memory location, specifically, the operator responsible for the most recent value in the location. The method constructs the graph G as follows:

- Step 0. Initialize G with a single root node r. The initial map M assigns root node r to every memory location.

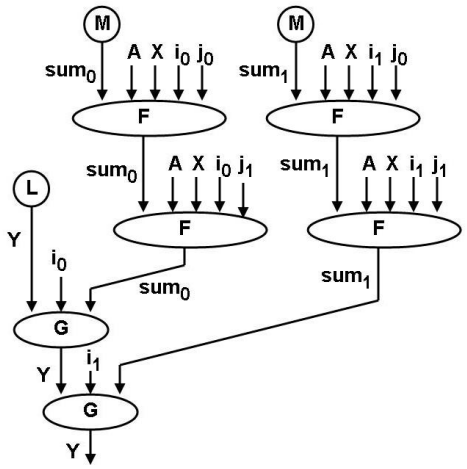

Fig. 9. A function composition graph for the HJ program of Figure 6

- Step 1. Choose the next operator s in the execution sequence.
- Step 2. Add a node labeled s to G. Add a dependence arc to s from each node of FCG that is the mapping by M of any location in the domain of s.
- Step 3. Update M to map each location in the range of s to node s.
- Step 4. If there are further operators in the program execution sequence, return to Step 1. Otherwise the construction is complete.

Figure 9 shows a function execution graph for our example HJ program with the same input values as for the computation graph of Figure 8. This is one of two FCGs that correspond to the computation graph. The other FCG differs from the one shown in the order in which the two instances of the `G` operator update the array variable `Y`.

We see that although the chosen input values lead to a unique computation graph, they do not necessarily produce output values that are a unique function of input values. Nevertheless the computed result computed by the HJ Program is the same in both cases because the addition operator is commutative and associative. Thus, this program is repeatable even though it is not determinate (because other interpretations of the addition operator might not be associative and commutative).

V. Conditions for Determinacy and Repeatability

As we have noted, determinacy of a parallel program implies execution of the program is repeatable, regardless of the interpretation given to the operators; for any arbitrary input and arbitrary interpretation of operators, all runs of the program will produce the same result. However, as our program examples have shown, repeatable interpreted programs need not be determinate. We would like to take advantage of this observation to devise rules that would ensure repeatability of task parallel programs.

In 1990 Guy Steele published his paper "Making Asynchronous Parallelism Safe for the World" containing the following statement:

> "The minimal restriction that guarantees that the unpredictability of ordering will not affect the behavior of a program is then that, for any possible serialization order for the operations performed by a program, any two consecutive operations of the program that are not causally related must commute with respect to the memory state that precedes the first. (Note that exchanging such commutative operations produces another possible serialization order.)"

He goes on to note that requiring commutativity with respect to all possible memory states simplifies the condition to "any two operations that are not causally related must commute." The paper later discusses situations where commutativity can be used in safe parallel programming.

It is possible to formulate a relaxed condition for repeatability by exploiting the idea of commuting statements. Statements `x = f(x,u)` and `x = g(x,v)` *commute*, where `x` is a shared variable and `u` and `v` are private variables of concurrent tasks containing the two statements, respectively, if their operators `f` and `g` satisfy the following condition:

```
let   x' = f(u,g(v,x))
      x'' = g(v,f(u,x))
then  x'' = x'
```

That is, it makes no difference for the ending state of the shared variable `x` in which order the two statements containing `f` and `g` are executed.

A sufficient condition for repeatability can now be stated as:

> A parallel program is repeatable if every pair of concurrent statements commute.

One example of the sufficient condition for repeatability is as follows. Consider a parallel loop in which each iteration i computes a value and writes it as the i^{th} element of an array. This computation is repeatable because statements writing values into an array commute so long as the writes are for different index values. A simpler example of commuting statements is two concurrent writes of the same value to the same location. Other examples of commutative operators include data parallel blocks that combine results using a commutative and associative reduction operator, and multiple threads making contributions to building a histogram.

However, as indicated earlier, these properties of commuting statements cannot be used to establish determinacy, since they constrain the interpretation assigned to operators.

A. Data Races and Atomicity

In practice it may be desired to use a compound statement to code an action that is expected to commute with other such actions to ensure repeatability of a parallel program. Consider the following HJ code fragment as an example:

```
async {r1 = X; r1 = r1 + 1; X = r1;} // A1
```

```
async {r2 = X; r2 = r2 + 1; X = r2;} // A2
```

In this example, actions $A1$ and $A2$ conflict with each other because they are concurrent and both read and write location X. Since actions $A1$ and $A2$ commute, we might conclude that the program is repeatable because of the sufficient condition just discussed. However, even a beginner in parallel programming recognizes that this program is not repeatable because the instructions executed within actions $A1$ and $A2$ can be arbitrarily interleaved.

A solution to this problem is to ensure that such interleaving is not permitted by making actions $A1$ and $A2$ *atomic*. Often this is done with locks, which are prone to misuse. In HJ this can be accomplished by defining each of the above `async` tasks to be `isolated` [15]. Such an approach precludes the possibility of interleavings of statement execution discussed above.

B. Language Design for Parallel Programming

A sound goal for language design for parallel computation is that following some simple syntactic rules should provide the programmer with a guarantee that a program is repeatable. It was observed in [15] that any HJ program that restricts its use of parallelism to the `async`, `finish`, `future`, `phaser`, and `data-driven task` parallel constructs is guaranteed to be determinate if the program is guaranteed to be data-race-free [24] for all inputs i.e., if all concurrent steps are guaranteed to be conflict-free. Like many other publications in the field, the term "determinism" was used in [15] without making a distinction between determinacy and repeatability. However the intended use was for "determinism" to represent determinacy, since the computation graph and function composition graphs are guaranteed to be the same for the same input, for any program in this data-race-free subset, regardless of the interpretation given to the steps [25]. The addition of isolated statements (and actors) to the set of HJ programs under consideration breaks this determinacy guarantee.

A limited form of the commutativity principle was applied in the design and implementation of the Sisal programming language [26] where the data parallel `for` expression permits parallel reductions to be expressed using commutative/associative operators. Here is an example.

```
A, s = for i in 0, m  1 do
   x = calculate (i);
   return array of x, value of sum x;
endfor
```

This `for` expression constructs a tuple of values from a set values of the variable `x` calculated for values of `i` over a contiguous range. The result tuple consists of the vector `A` containing the x's as elements and the sum `s` of all of the x's.

Sisal supports construction of a class of parallel programs that includes much of what one desires from going beyond strictly determinate programs. However, Sisal is incomplete in its lack of support for expressing the full possibilities of producer/consumer patterns and dynamic DAGs of tasks such as supported by Habanero Java. Sisal evolved from the Val language [27] which already included the data parallel `for` expression as described above.

VI. Related Work

In the main body of this paper, we have already introduced past work on determinacy and repeatability that directly influenced the main subject of this paper as well as their related citations. The concept of "determinism" of parallel programs and parallel systems has received a lot of attention, with multiple proposals for formal and informal definitions of deterministic execution. The importance of this concept has been increasingly well recognized, and the challenges have been forecasted and studied in [28], [29], [30]. Many forms of deterministic parallelism have been described and investigated. A good source of references related to determinism can be found in a recent publication by Blelloch et al [8].

This paper explores application of the observations of Stelle to nested parallel programs to enlarge the class of computation that can be expressed with assurance of repeatable behavior. Their "control flow DAG" and "trace" concepts are closely related to our Parallel Execution Graph and Function Composition Graph respectively. According to the Blelloch paper, a program is "internally deterministic" if all runs of the program for given input generate the same control flow DAG and trace. This property is at once weaker than determinacy and stronger than repeatability. It is weaker than determinacy because it applies to a specific interpretation of operators, whereas a parallel program is determinate if it is internally deterministic for all interpretations of operators of the program. On the other hand, being internally deterministic is a sufficient but not necessary condition for a program to be repeatable, so it is stronger than repeatability. The authors of [8] apply their concepts to show how commuting operators may be used to write repeatable parallel programs for several challenging applications.

VII. Conclusion

The revolution in computer systems arising from the transition to multicore and many-core processors has created an urgent need for programming models that support massively parallel computation. Of particular interest are means for avoiding the hazards of concurrency that lead to non-repeatable behavior of programs and the attendant difficulties of debugging and establishing correctness. In this paper we have sought to clarify the meanings commonly expressed in the terms "deterministic" and "nondeterministic", which frequently appear in discussions of problems of writing correct programs for parallel computation without precise definition. We have shown that the two properties "determinacy" and "repeatability" can be given precise meaning for data flow and task-parallel program schemata, and are useful in the study of strategies for structuring and transforming parallel programs to avoid harmful data races.

Determinacy requires that a program schema produce the same results for given input regardless of the specific functions

assigned to its operators. A program, with specified operations, is *repeatable* if any run of the program for a given input produces the same result. Determinacy implies repeatability for any interpretation of the operators, but the converse does not hold. We note that use of commuting operators is a powerful tool for building parallel programs that are repeatable even though the schema of the program is not determinate.

To avoid confusion, we strongly recommend that authors of future papers related to parallel program behavior use the terms "determinate" and "repeatable" when referring to those precise properties, and avoid "deterministic" when there is ambiguity about whether it is intended to denote determinacy or repeatability in a given context.

ACKNOWLEDGMENTS

The authors would like to express their sincere appreciation to Dr. Chen Chen, post-doctoral researcher at the University of Delaware, for his assistance in researching prior work, especially for his efforts in working with us to achieve a deep understanding of past work by Karp and Miller. We also thank Robert Pavel from University of Delaware for his assistance in preparing this document, and Raghavan Raman from Rice University for clarification about properties related to determinacy in his Ph.D. dissertation [25].

This work was supported in part by the National Science Foundation through grant CCF-0937832, CCF-0833122, CCF-0937907, and OCI-0904534. This work was also partly supported by European FP7 project TERAFLUX, id. 249013.

REFERENCES

[1] R. M. Karp and R. E. Miller, "Properties of a model for parallel computations: Determinacy, termination, queueing," IBM Research Center, Tech. Rep., 1964, RC 1285.

[2] R. Karp and R. Miller, "Properties of a model for parallel computations: Determinancy, termination, queueing," *SIAM Journal on Applied Mathematics*, pp. 1390–1411, 1966.

[3] R. Karp and R. Miller, "Parallel program schemata," *Journal of Computer and system Sciences*, vol. 3, no. 2, pp. 147–195, 1969.

[4] R. M. Karp and R. E. Miller, "Parallel program schemata: A mathematical model for parallel computation," in *SWAT (FOCS)*, 1967, pp. 55–61.

[5] Y. Ianov, "The logical schemes of algorithms," *Problems of cybernetics*, vol. 1, pp. 82–140, 1960.

[6] D. Luckham, D. Park, and M. Paterson, "On formalised computer programs," *Journal of Computer and System Sciences*, vol. 4, no. 3, pp. 220–249, 1970.

[7] M. Paterson and C. Hewitt, "Comparative schematology," in *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*. ACM, 1970, pp. 119–127.

[8] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun, "Internally deterministic parallel algorithms can be fast," in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York, NY, USA: ACM, 2012, pp. 181–192. [Online]. Available: http://doi.acm.org/10.1145/2145816.2145840

[9] R. D. Blumofe *et al.*, "CILK: An efficient multithreaded runtime system," *PPoPP'95*, pp. 207–216, Jul. 1995.

[10] J. Dennis, "First version of a data flow procedure language," in *Lecture Notes in Computer Science 19: Programming Symposium*. Springer, 1974, pp. 362–376.

[11] J. Dennis, J. Fosseen, and J. Linderman, "Data flow schemas," in *International Symposium on Theoretical Programming*. Springer, 1974, pp. 187–216.

[12] W. R. Sutherland, "The On-Line Graphical Specification of Computer Procedures," Ph.D. dissertation, Massachusetts Institute of Technology, 1966.

[13] J. Rodriguez, "A graph model for parallel computations," Ph.D. dissertation, MIT; Electronic Systems Lab, 1969.

[14] S. Patil, "Closure properties of interconnections of determinate systems," in *Record of the Project MAC conference on concurrent systems and parallel computation*. ACM, 1970, pp. 107–116.

[15] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-Java: the New Adventures of Old X10," in *PPPJ'11: Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java*, 2011.

[16] "OpenMP Application Program Interface, version 3.0, May 2008." [Online]. Available: http://www.openmp.org/mp-documents/spec30.pdf

[17] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *OOPSLA*, NY, USA, 2005, pp. 519–538. [Online]. Available: http://doi.acm.org/10.1145/1094811.1094852

[18] C. Inc., "The Chapel language specification version 0.4," Cray Inc., Tech. Rep., Feb. 2005.

[19] S. Taşırlar and V. Sarkar, "Data-Driven Tasks and their Implementation," in *ICPP'11: Proceedings of the International Conference on Parallel Processing*, Sep 2011.

[20] J. Shirako, V. Cavé, J. Zhao, and V. Sarkar, "Finish Accumulators: a Deterministic Reduction Construct for Dynamic Task Parallelism," in *WoDet'13: Proceedings of The 4th Workshop on Determinism and Correctness in Parallel Programming*, Mar 2013.

[21] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer, "Phasers: a unified deadlock-free construct for collective and point-to-point synchronization," in *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*. New York, NY, USA: ACM, 2008, pp. 277–288.

[22] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer, "Phaser accumulators: A new reduction construct for dynamic parallelism," in *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–12.

[23] Y. Guo, "A Scalable Locality-aware Adaptive Work-stealing Scheduler for Multi-core Task Parallelism," Ph.D. dissertation, Rice University, Aug 2010.

[24] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav, "Scalable and precise dynamic datarace detection for structured parallelism," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 531–542. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254127

[25] R. Raman, "Dynamic Data Race Detection for Structured Parallelism," Ph.D. dissertation, Rice University, August 2012.

[26] J. McGraw, S. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas, "SISAL: Streams and Iteration in a Single Assignment Language Reference Manual Version 1.2," Lawrence Livermore National Laboratory, Tech. Rep., 1985, no. M-146, Rev. 1.

[27] W. Ackerman and J. Dennis, "Val: Value-oriented algorithmic language: Preliminary reference manual," Cambridge, MA, USA, Tech. Rep., 1979.

[28] P. B. Gibbons, "A more practical PRAM model," in *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, ser. SPAA '89. New York, NY, USA: ACM, 1989, pp. 158–168. [Online]. Available: http://doi.acm.org/10.1145/72935.72953

[29] R. Halstead, JR., "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Transactions of Programming Languages and Systems*, vol. 7, no. 4, pp. 501–538, October 1985.

[30] F. Putze, P. Sanders, and J. Singler, "Mcstl: the multi-core standard template library," in *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '07. New York, NY, USA: ACM, 2007, pp. 144–145. [Online]. Available: http://doi.acm.org/10.1145/1229428.1229458