A Design and Implementation of Clocked Variables in X10

Daniel Atkins, Alex Potanin, Lindsay Groves

What is X10?

- X10 is a language currently being developed by IBM
- It is designed to be used in highly concurrent environments, such as on large clusters
- It contains many language features that make writing and maintaining concurrent code much easier
- Seems to be mostly for computation based problems

What is X10?

- Similar to Java in a lot of ways
  - Class-based OO
  - Similar syntax for the most part
- Compiler doesn't translate *directly* to executable byte-code / machine code
  - Can translate X10 code into Java, C++, or CUDA.
- Java back-end runs on the JVM and uses special X10 Runtime Libraries

X10 – Sample Code

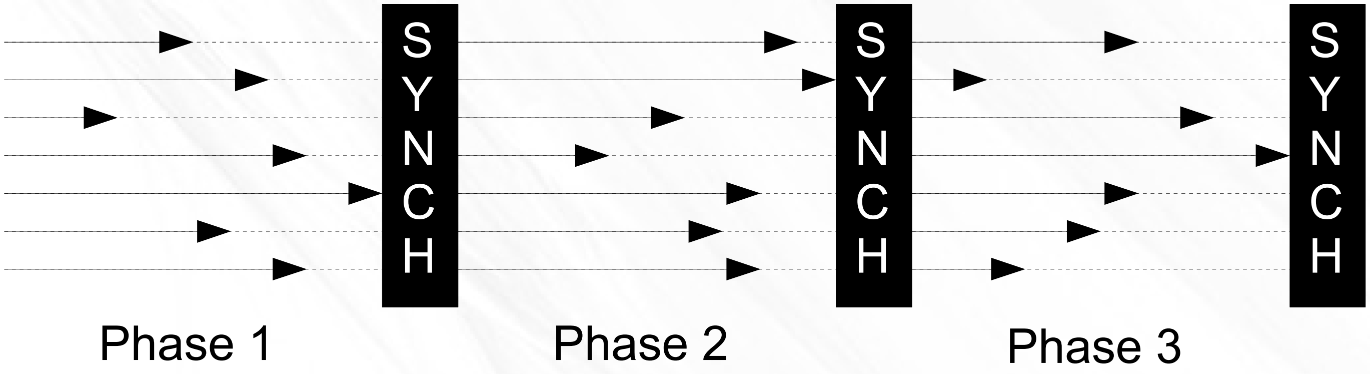
```
public class Summer{

    val data:Array[Integer](1) = {…}

    public static def main( args:Array[String](1) ) {
        var sum:Integer = 0;
        finish{
            for(var i in 0..4){
                val j = i;
                var local:Integer = 0;
                async {
                    for(var x = j; x < j + data.length()/4;x++)
                            local += data(x);
                    atomic{sum += local;}
                }
            }
        }
        Console.OUT.println(“The sum is: “ + sum);
    ...
```

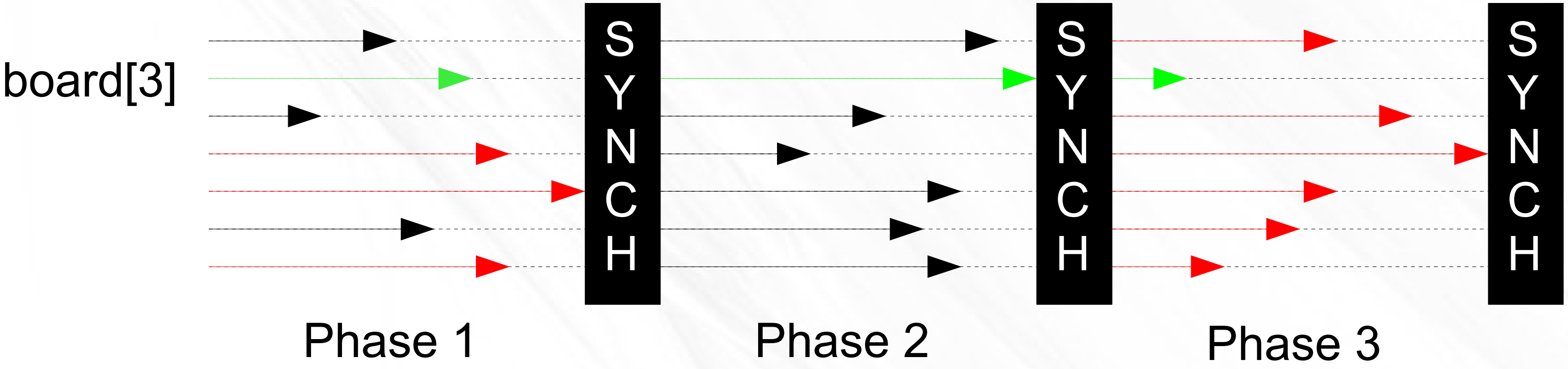
X10's Clocks

- X10 provides the Clock class to allow synchronization between threads.
- It functions as a barrier: threads that wish to continue with execution are blocked until **all** threads are ready to continue.



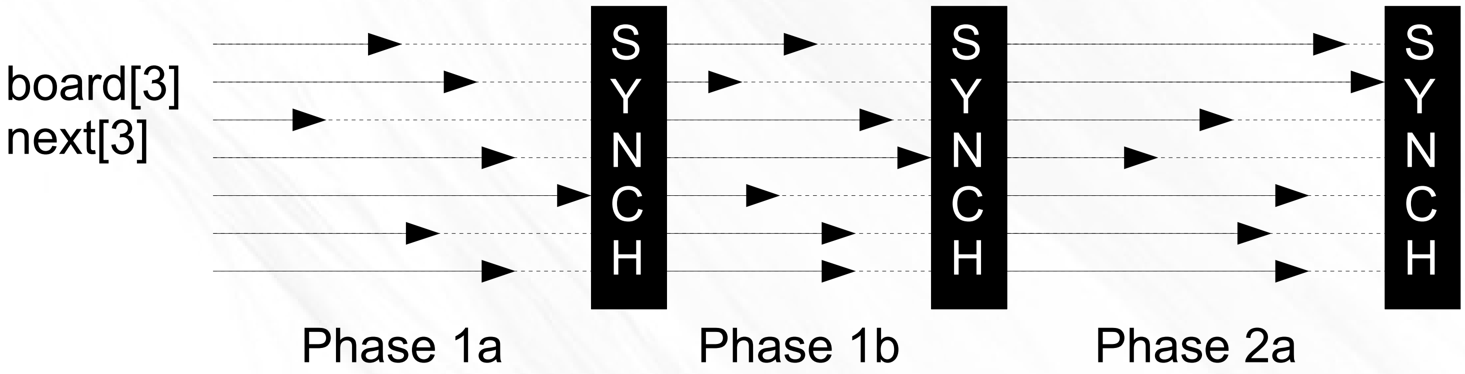
Clocked Variables

- Threads share memory!
- What if we have a variable we want all the threads to be able to read, but one of the threads updates it? (Example: Board State)
- This introduces race conditions!



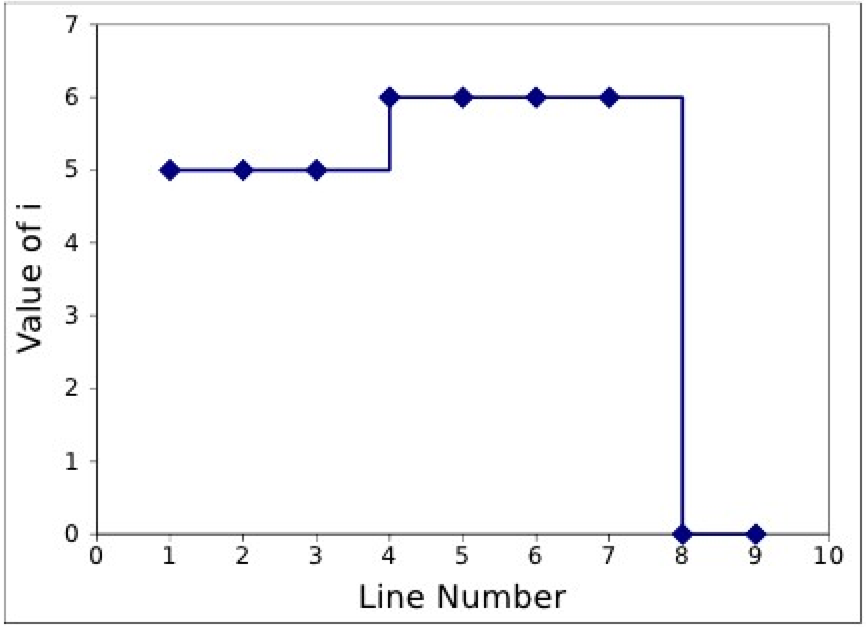
Clocked Variables

- Could solve this problem by requiring all threads to finish processing before updating
- Requires writing more code, and having extra variables to store new state before we update.



- Can we do this automatically?

Clocked Variables



```
1    clocked var i:Int = 5;
2    i = 6;
3    Console.OUT.println(i); //Prints 5
4    Clock.advanceAll();
5    Console.OUT.println(i); //Prints 6
6    i = 0;
7    Console.OUT.println(i); //Prints 6
8    Clock.advanceAll();
9    Console.OUT.println(i); //Prints 0
```

- Within a single clock phase, the value of a clocked variable should remain constant
- If a clocked variable is updated, this change should occur when the clock phase changes.

Clocked Variables – Primitive Types

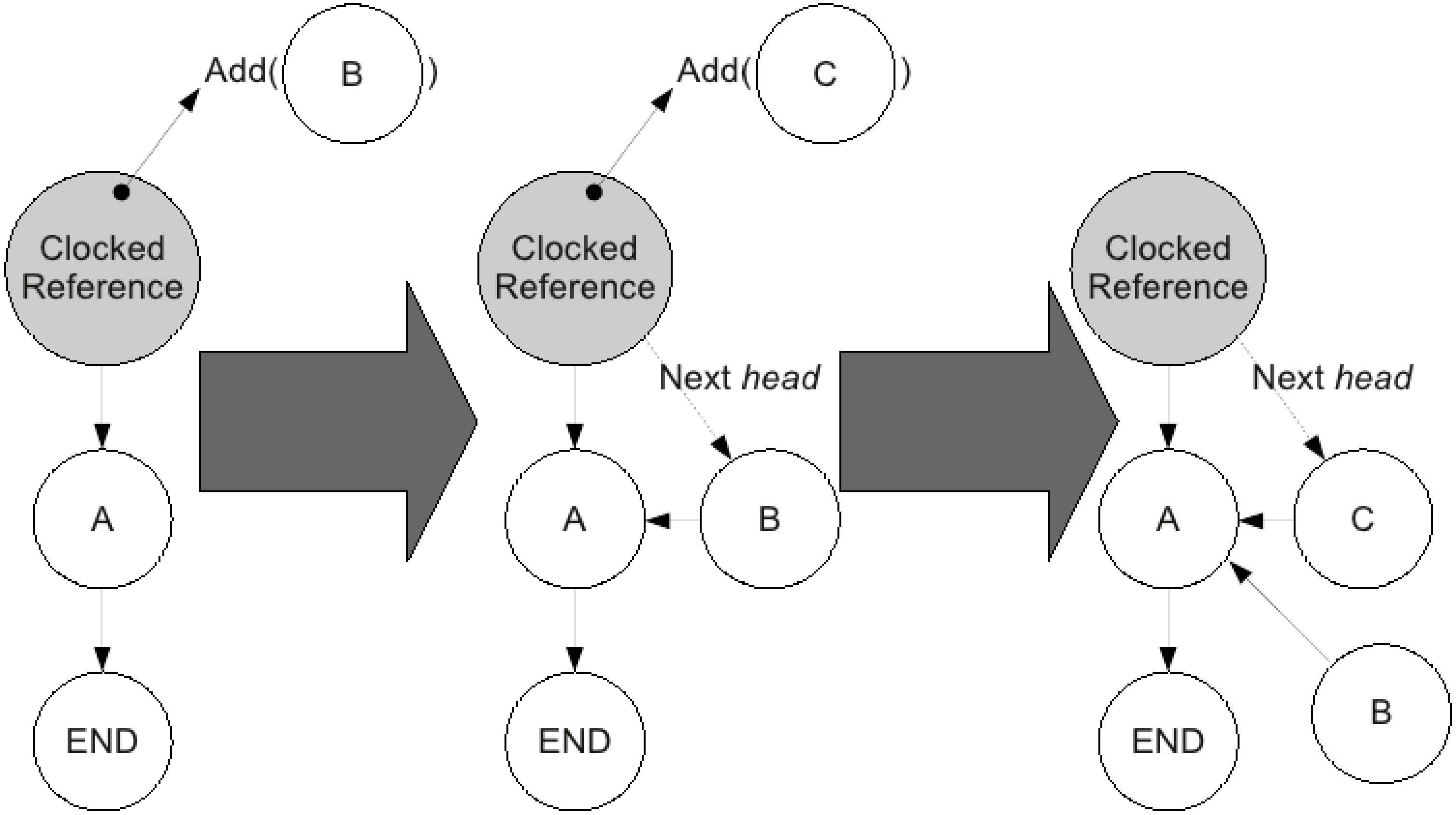
```
var x:ClockedInt = new ClockedInt(5);   //Create a new Clocked Integer
x.register();                           //Register x on the current clock
x() = 10;                               //WRITE 10 into x.
Console.OUT.println(x() + 1);           //Prints 6
Clock.advanceAll();                     //Advance the clock
Console.OUT.println(x());               //Prints 10
```

- Primitives are easy!
- Have a wrapper object that has two fields (*current* and *next*) and write only into next, and read only from current.
- Clock calls update method, which just copies the value of next into current.

Clocked Variables – Reference Types

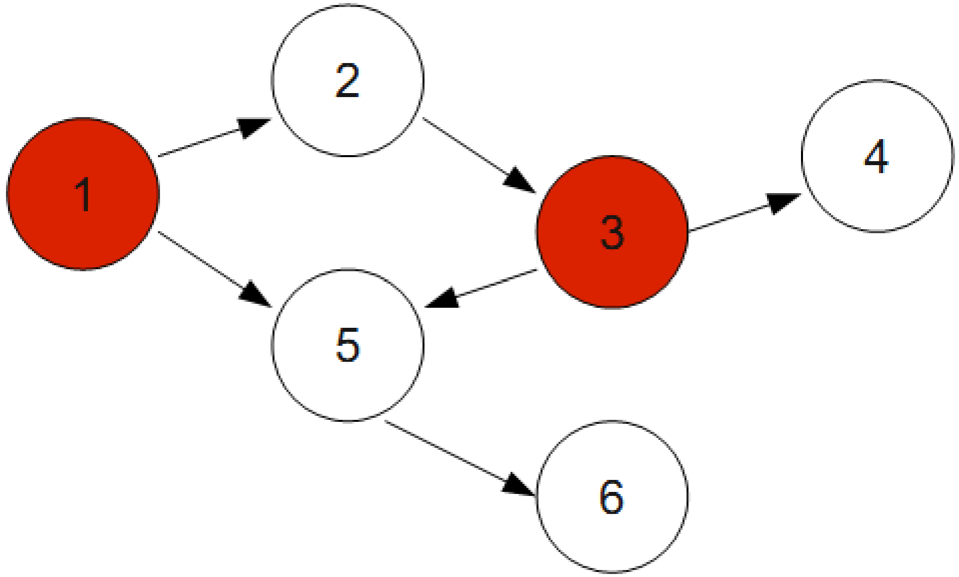
- Reference Types are a little trickier, as there is a lot more state to deal with—we can't just copy the value.
- Current solution uses a naïve method: deep-cloning the object graph of the *next* field and replacing the *current* field with the clone.
- Slow and inefficient, but it works and establishes a *baseline* for comparing possible improvements. Future work will deal with finding a better way to do it.
- There are some other problems...

Clocked Variables – Multiple Writes?



To avoid situations like this, we limit the variable to only allowing **one** write per phase.

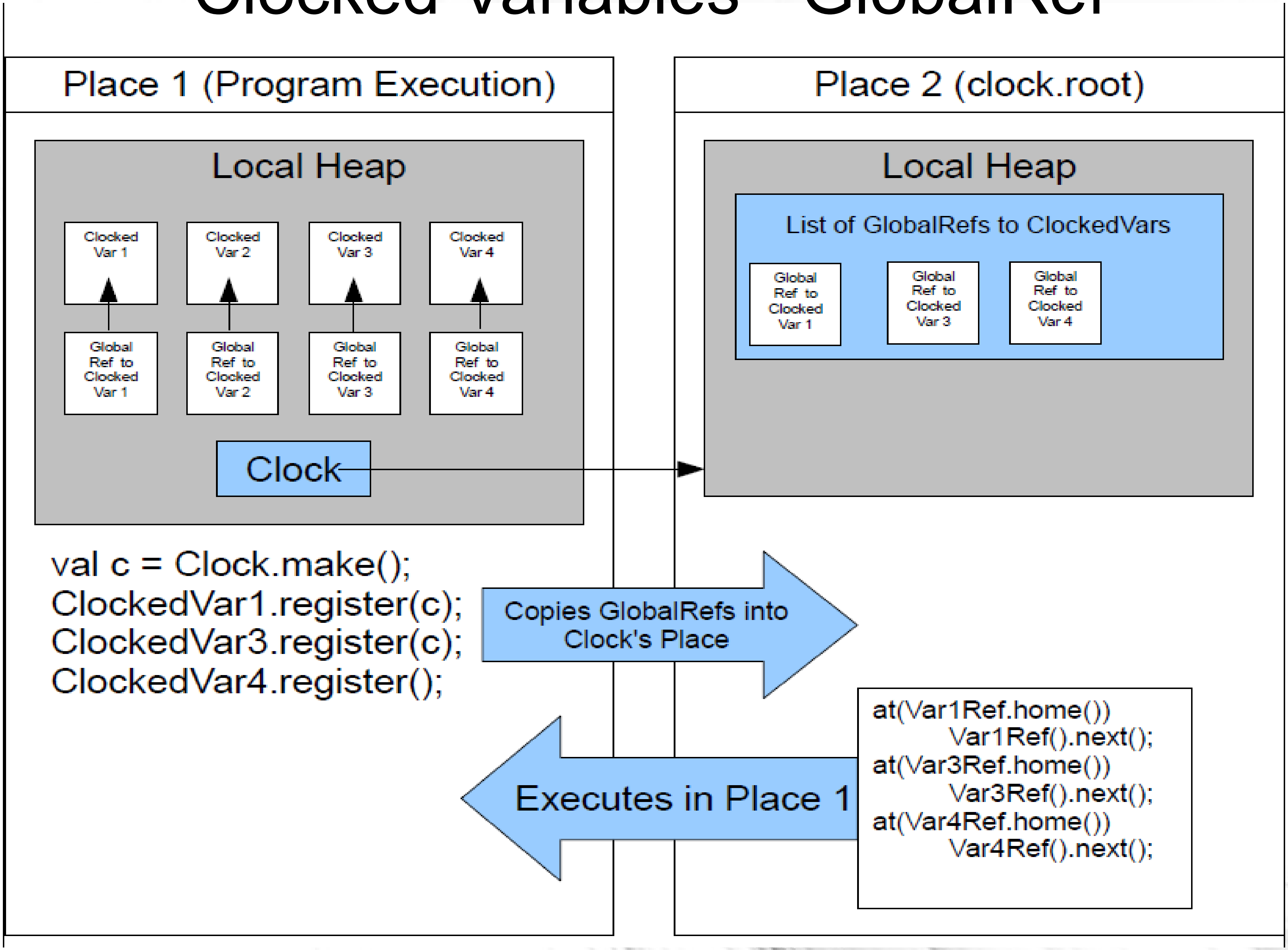
Clocked Variables – Memory Usage?



- Nodes 1 and 3 are clocked.
- That means there are two *complete* copies of each in memory.

- But node 3 is also part of node 1's graph!
- So there are two copies of node 3's graph *per copy of node 1's*!
- Many duplicates. This explodes as you add more clocked sub-graphs.

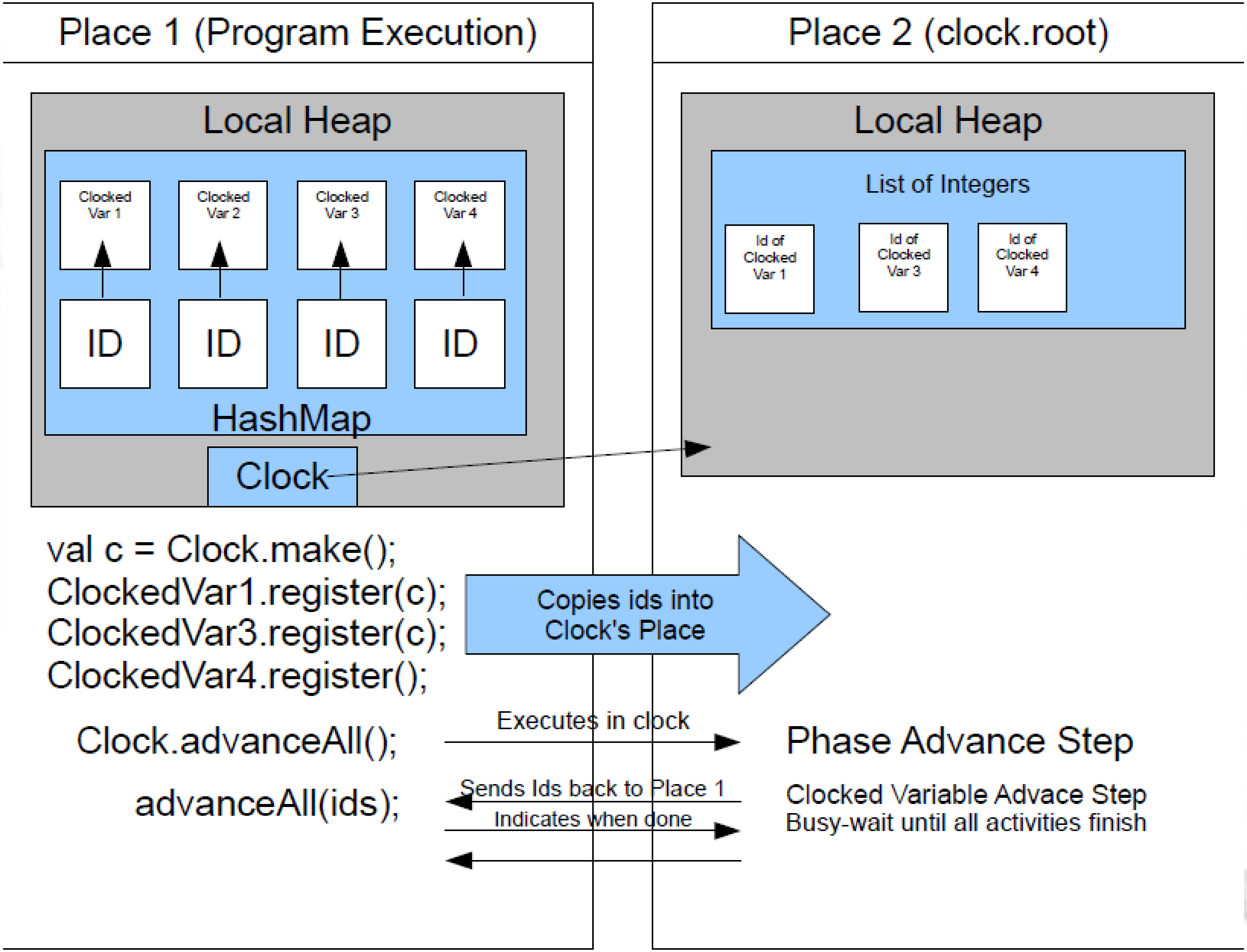
Two Possible Ways to Implement Clocked Variables - GlobalRef


Place 1 (Program Execution)
Place 2 (clock.root)
Local Heap
Clocked Var 1
Clocked Var 2
Clocked Var 3
Clocked Var 4
Global Ref to Clocked Var 1
Global Ref to Clocked Var 2
Global Ref to Clocked Var 3
Global Ref to Clocked Var 4
Clock
Local Heap
List of GlobalRefs to ClockedVars
Global Ref to Clocked Var 1
Global Ref to Clocked Var 3
Global Ref to Clocked Var 4
Copies GlobalRefs into Clock's Place
Executes in Place 1

```
val c = Clock.make();
ClockedVar1.register(c);
ClockedVar3.register(c);
ClockedVar4.register();
```

```
at(Var1Ref.home())
        Var1Ref().next();
at(Var3Ref.home())
        Var3Ref().next();
at(Var4Ref.home())
        Var4Ref().next();
```

Two Possible Ways to Implement Clocked Variables - HashMap



Place 1 (Program Execution)

Place 2 (clock.root)

Local Heap

Clocked Var 1

Clocked Var 2

Clocked Var 3

Clocked Var 4

ID

ID

ID

ID

HashMap

Clock

Local Heap

List of Integers

Id of Clocked Var 1

Id of Clocked Var 3

Id of Clocked Var 4

```
val c = Clock.make();
ClockedVar1.register(c);
ClockedVar3.register(c);
ClockedVar4.register();
```

Copies ids into Clock's Place

```
Clock.advanceAll();
```

Executes in clock

Phase Advance Step

```
advanceAll(ids);
```

Sends Ids back to Place 1

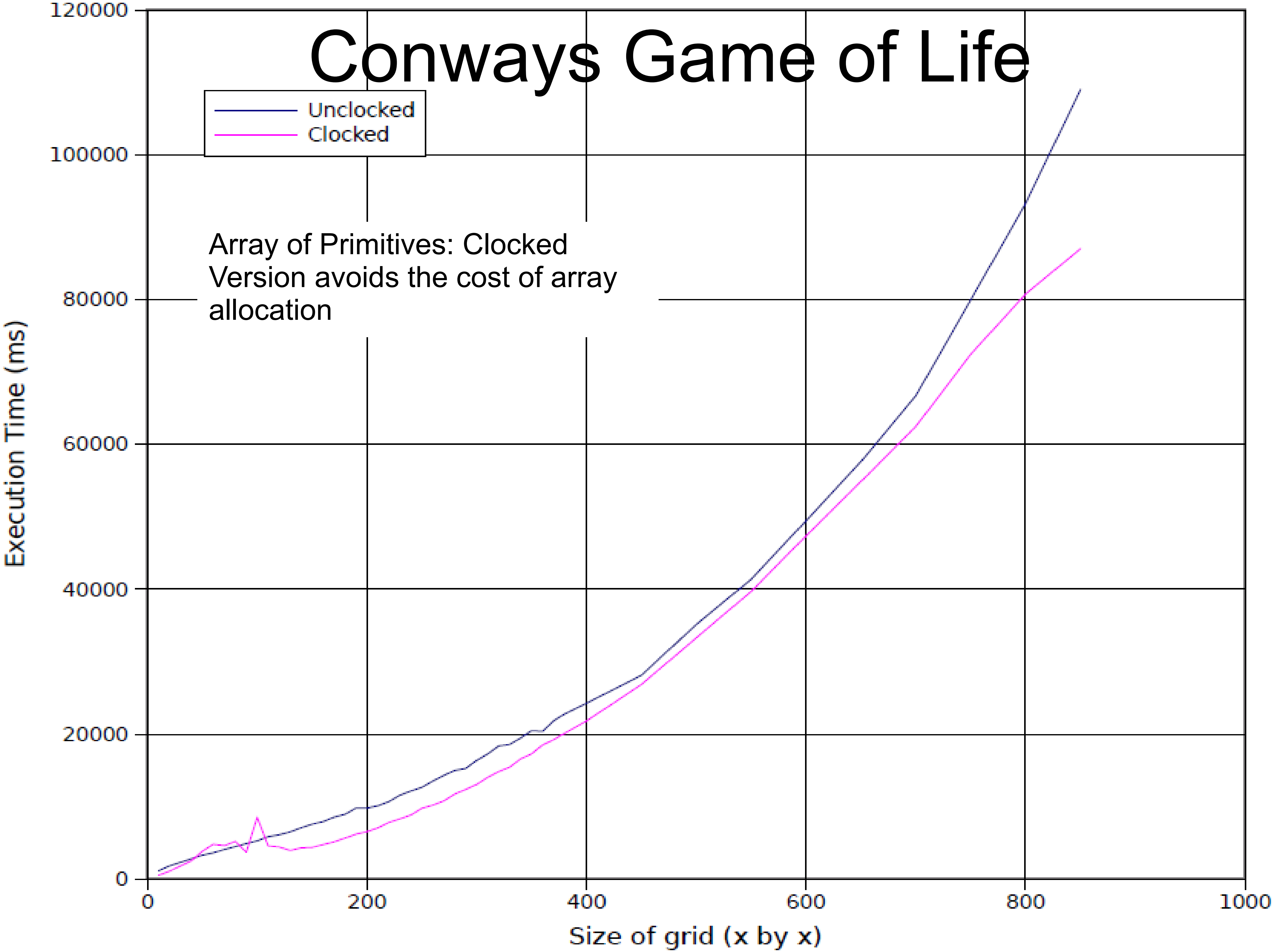
Indicates when done

Clocked Variable Advace Step
Busy-wait until all activities finish

Clocked Variables – Benchmarks

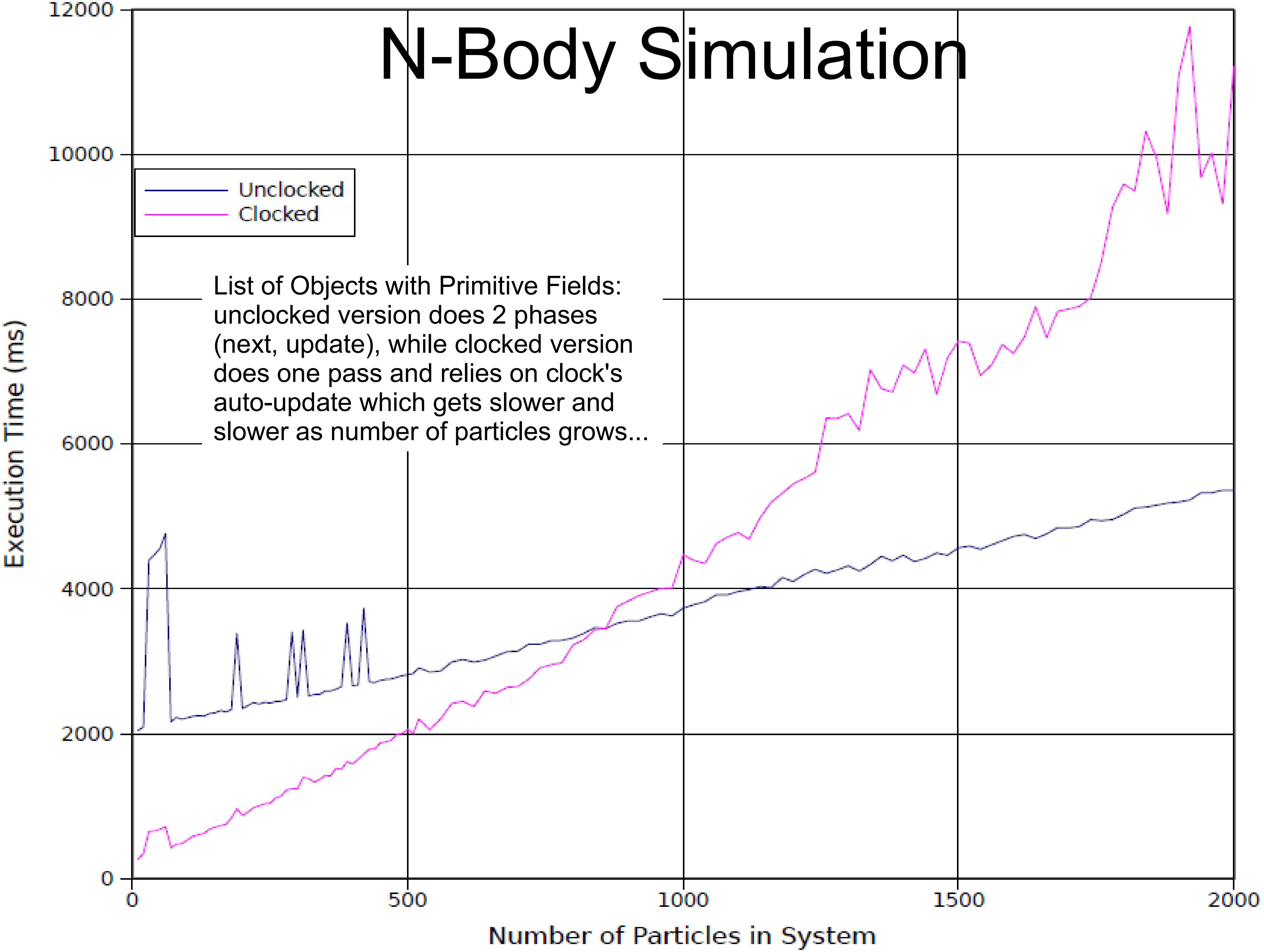
- Four fairly simple benchmarks
  - Conway's “Game of Life” (array of primitives)
  - An N-Body Simulator (objects with primitive fields only)
  - Sparse Matrix (complex linked object structure)
  - Linked List (typical linked object structure)

Conways Game of Life



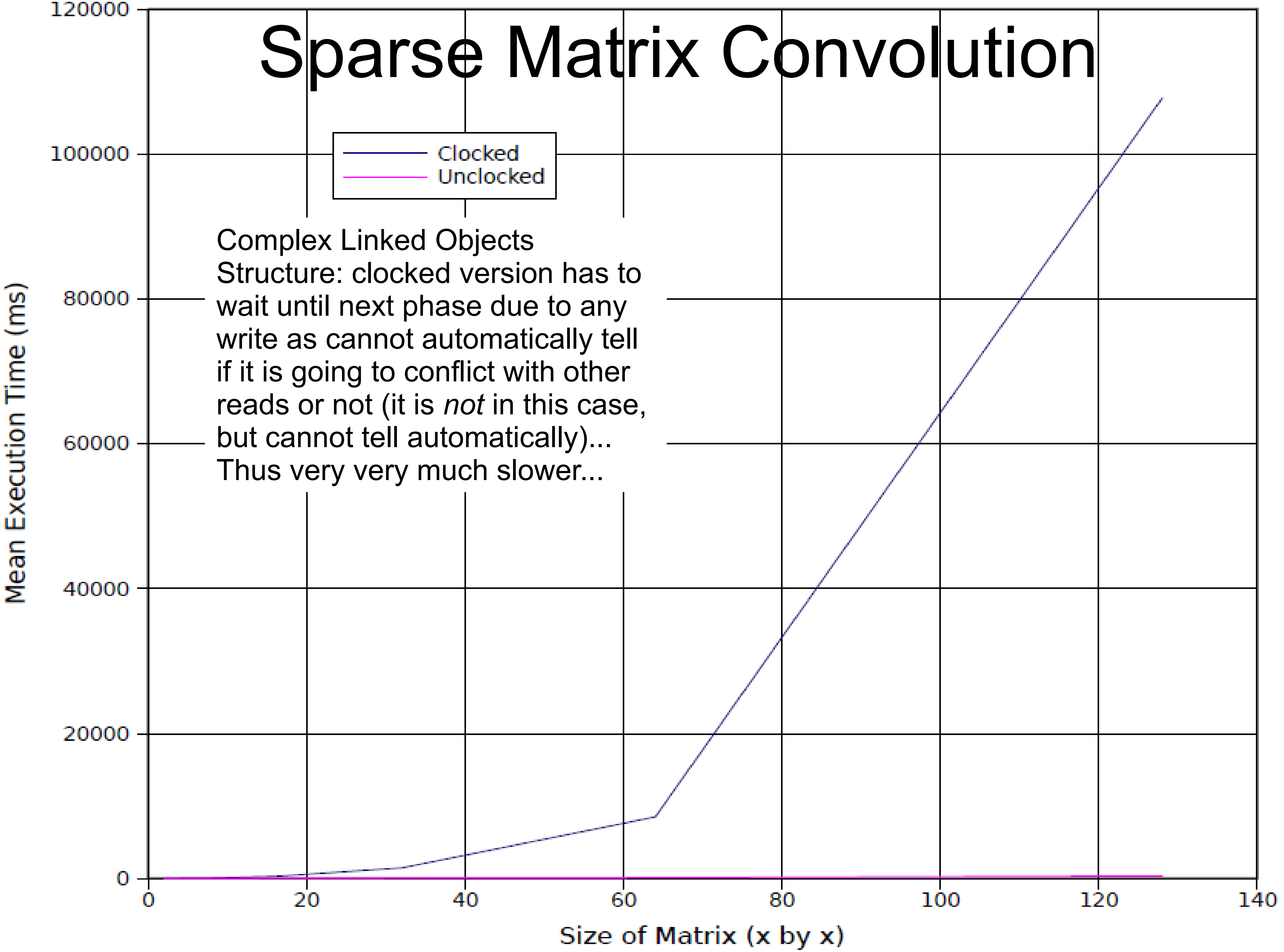
Array of Primitives: Clocked Version avoids the cost of array allocation

N-Body Simulation



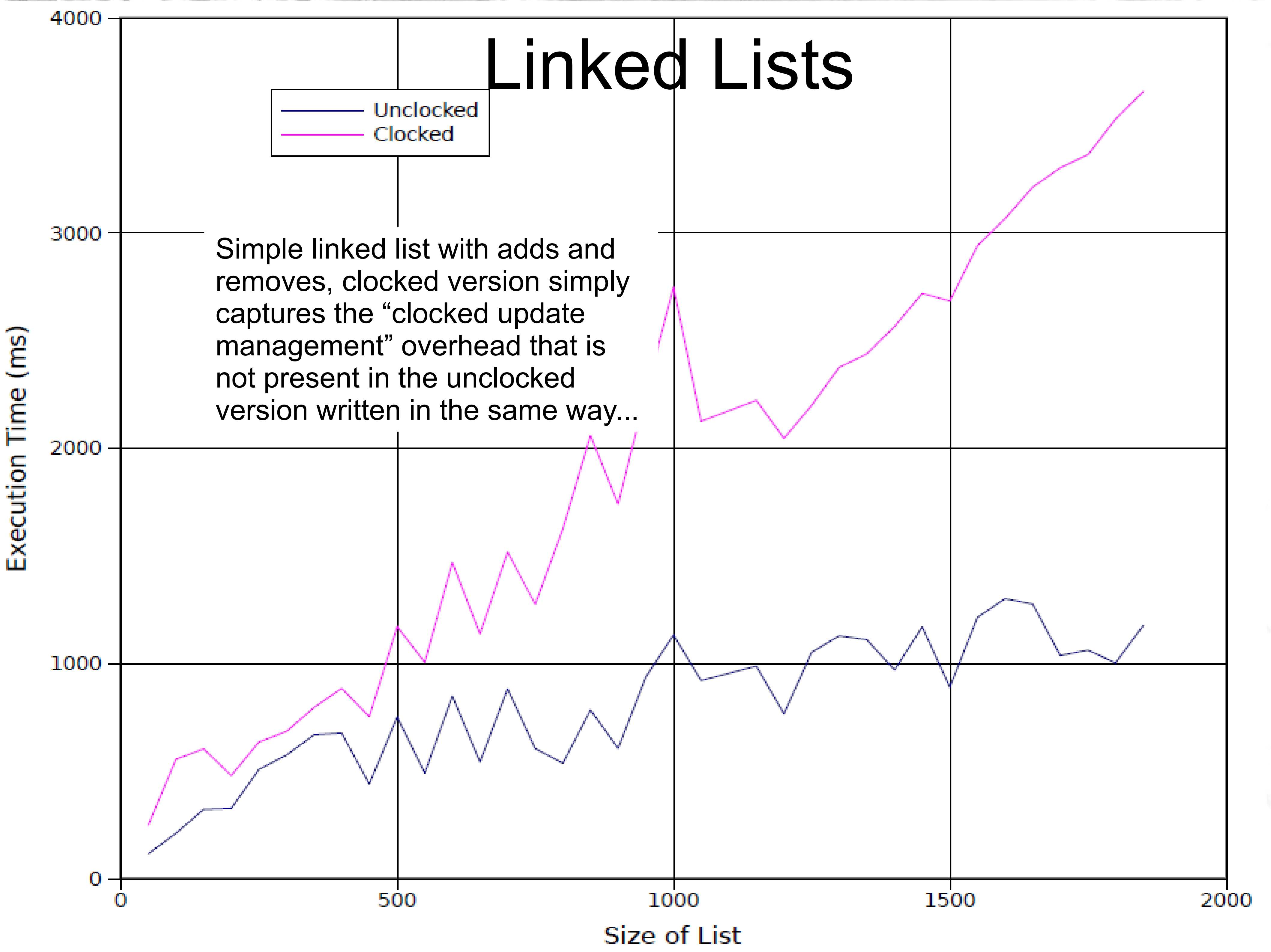
List of Objects with Primitive Fields: unclocked version does 2 phases (next, update), while clocked version does one pass and relies on clock's auto-update which gets slower and slower as number of particles grows...

Sparse Matrix Convolution



Complex Linked Objects Structure: clocked version has to wait until next phase due to any write as cannot automatically tell if it is going to conflict with other reads or not (it is *not* in this case, but cannot tell automatically)... Thus very very much slower...

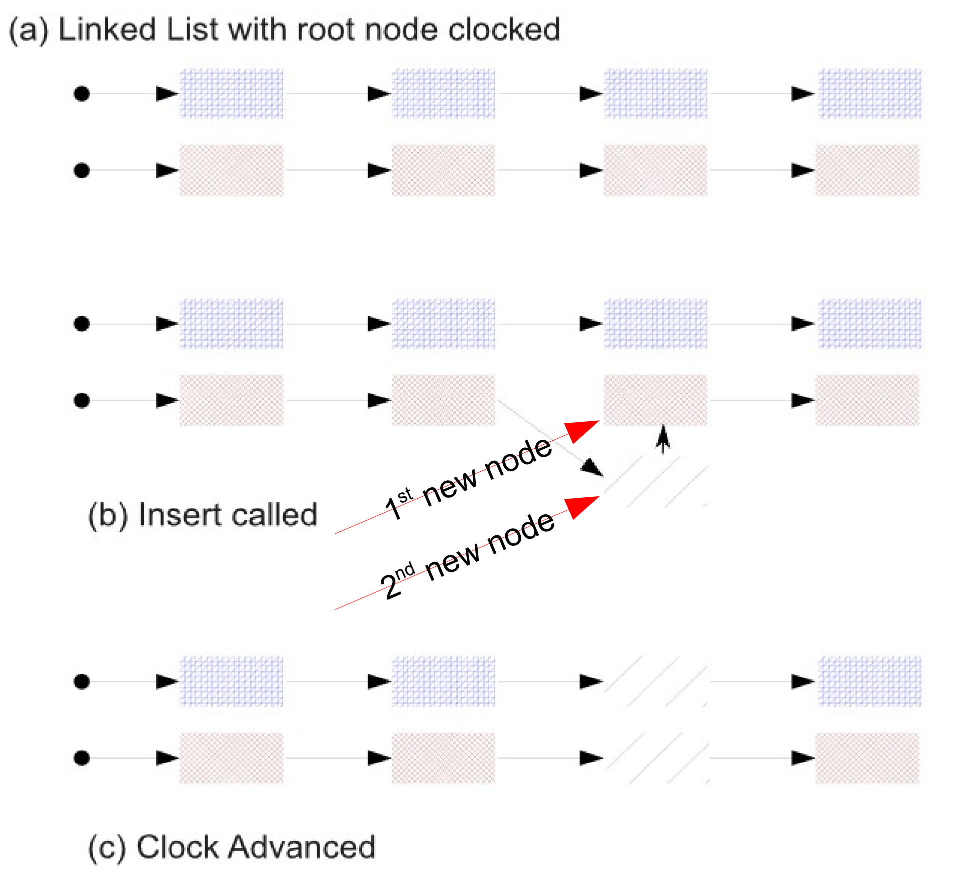
Linked Lists

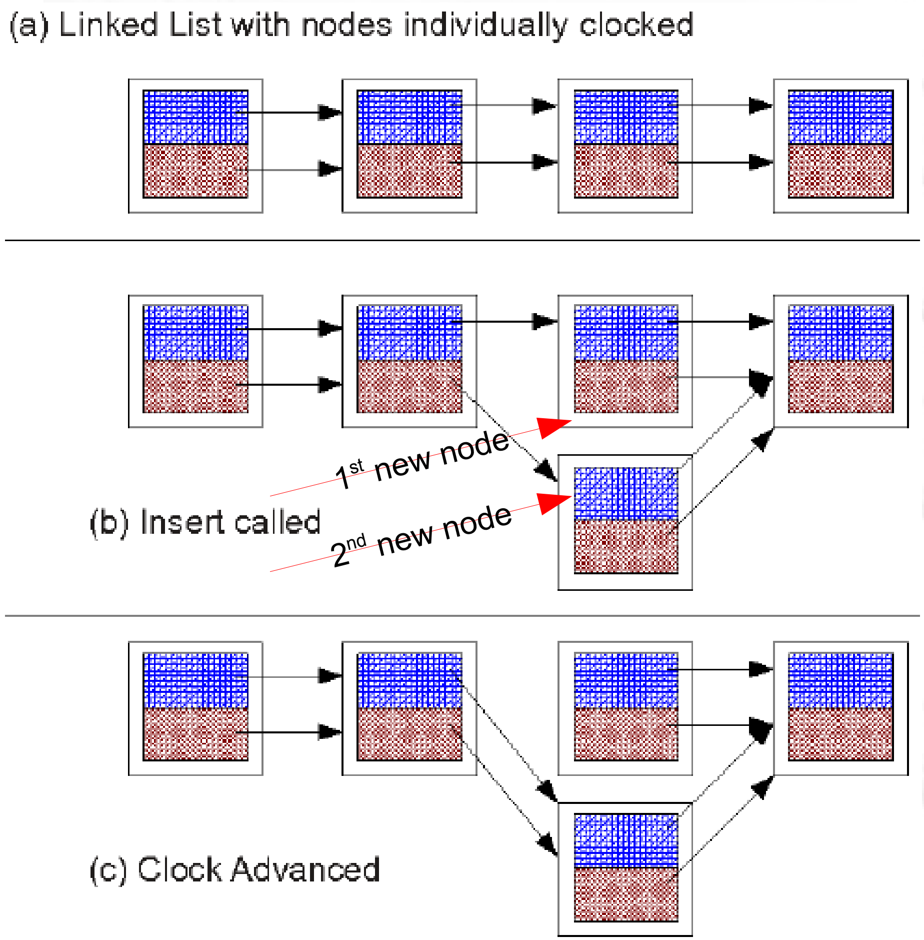


Simple linked list with adds and removes, clocked version simply captures the “clocked update management” overhead that is not present in the unclocked version written in the same way...

Alternative Clocking Mechanisms

(here we are *inserting 2 nodes*)





Currently: Clocking Whole List (Full Cloning)

Possible Approach: Clocking Individual Objects *but* Manually Managing Concurrency Problems :-(

Conclusion

Clocked Primitives are the most viable form of clocked variable. The benefit gained from using them is a cleaner way of updating dual-state variables often found inside concurrent code.

Clocked References, however, presented more of a challenge. While our initial attempts at solving this problem were not entirely successful, we have presented our results and offered insights into what could be done to solve this issue. There are many options for future work with Clocked References, and many new avenues to explore.

http://ecs.vuw.ac.nz/~atkinsdani1/x10-clocked.tar.gz