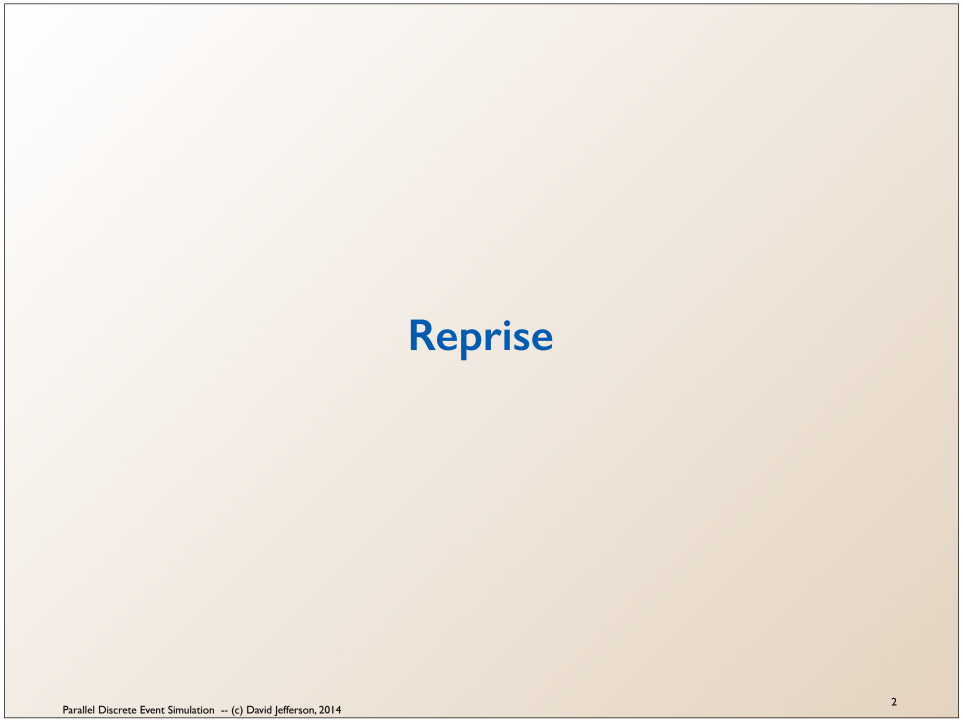
Parallel Discrete Event Simulation Course #13

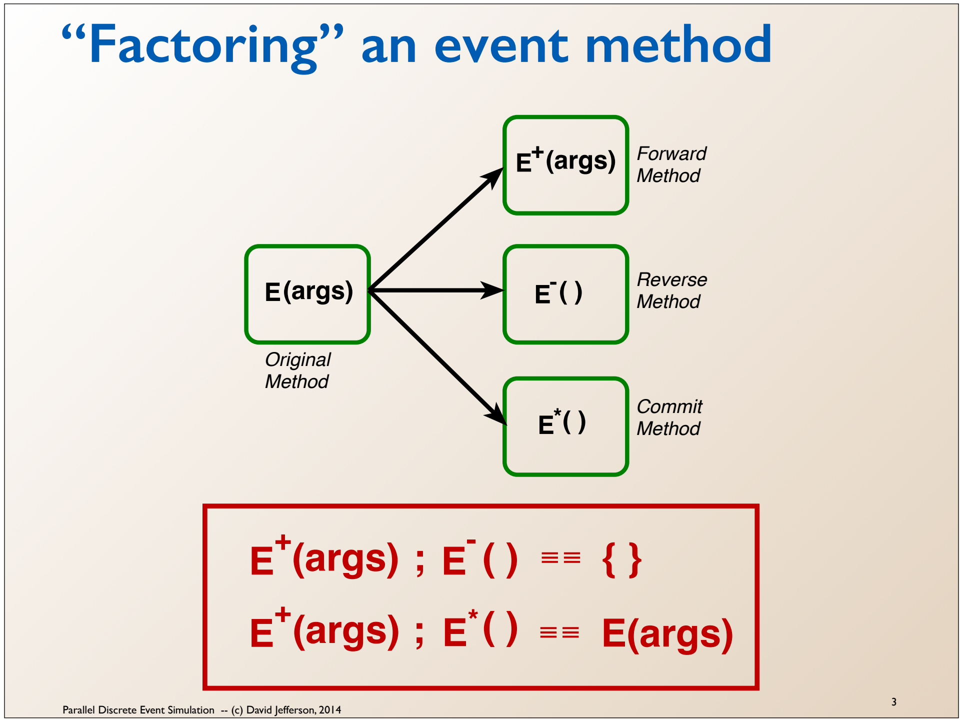
David Jefferson
Lawrence Livermore National Laboratory
2014

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Release Number: LLNL-PRES-653913



Reprise



“Factoring” an event method

E^{+}(args) Forward Method

E(args) Original Method

E^{-}( ) Reverse Method

E^{*}( ) Commit Method

$$E^{+}(\text{args})\ ;\ E^{-}(\,)\ \equiv\equiv\ \{\,\}$$

$$E^{+}(\text{args})\ ;\ E^{*}(\,)\ \equiv\equiv\ E(\text{args})$$

The fundamental idea is to take all of the event methods **E(args)** in a parallel discrete event simulation and “factor” each of them into three parts: **E+(args)**, **E-()**, and **E*()**.

E+(args) is executed in place of E(args) in the simulation and is instrumented to save all information destroyed by the forward execution of **E(args)** so as to preserve the option after **E(args)** completes of restoring the initial state of an object before it executed.

$\mathbf{E}^{-}$**()** uses the information stored by $\mathbf{E}^{+}$**()** and also the object’s state information to exactly reconstruct the state before $\mathbf{E}^{+}$**()** executed. It in effect reverses all of the side effect of $\mathbf{E}^{+}$**()** and exactly accomplishes rollback of the event.

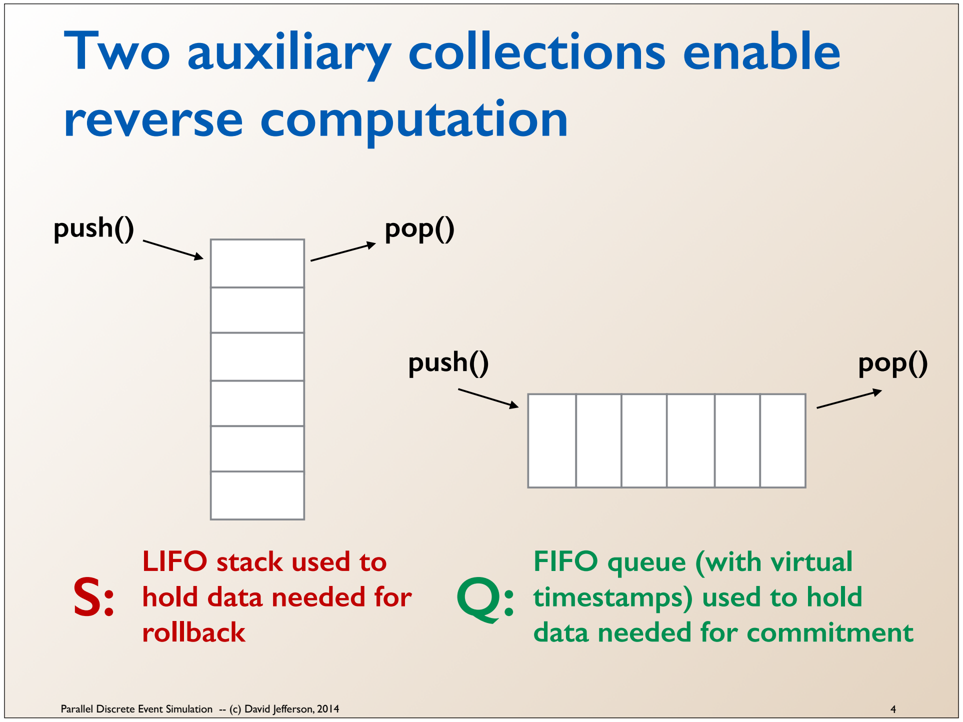
$\mathbf{E}^{*}$**()** is executed at the time event **E(args)** is committed, and deals with actions specified in $\mathbf{E}^{+}$**(args)** that really cannot be rolled back, such as output, or the freeing of dynamically allocated storage.

The two equations boxed in red are properties that the three methods must satisfy.

The first one says the $\mathbf{E}^{-}$**()** really does reverse all of the side effects of $\mathbf{E}^{+}$**(args)**, *and does nothing else*, so that executing $\mathbf{E}^{+}$**(args)** followed by $\mathbf{E}^{-}$**()** is a no-op.

The second one says that executing $\mathbf{E}^{+}$**(args**) followed by $\mathbf{E}^{*}$**()** is equivalent to executing the original method E(args).

Since every time $\mathbf{E}^{+}$**(args)** is executed it will either be rolled back or committed, then either $\mathbf{E}^{-}$**()** or $\mathbf{E}^{*}$**()** will be executed after it and the net effect will either be a no-op (in the case of a rollback) or it will be as if **E(args)** executed (in the case of commitment).



Two auxiliary collections enable reverse computation

S: LIFO stack used to hold data needed for rollback

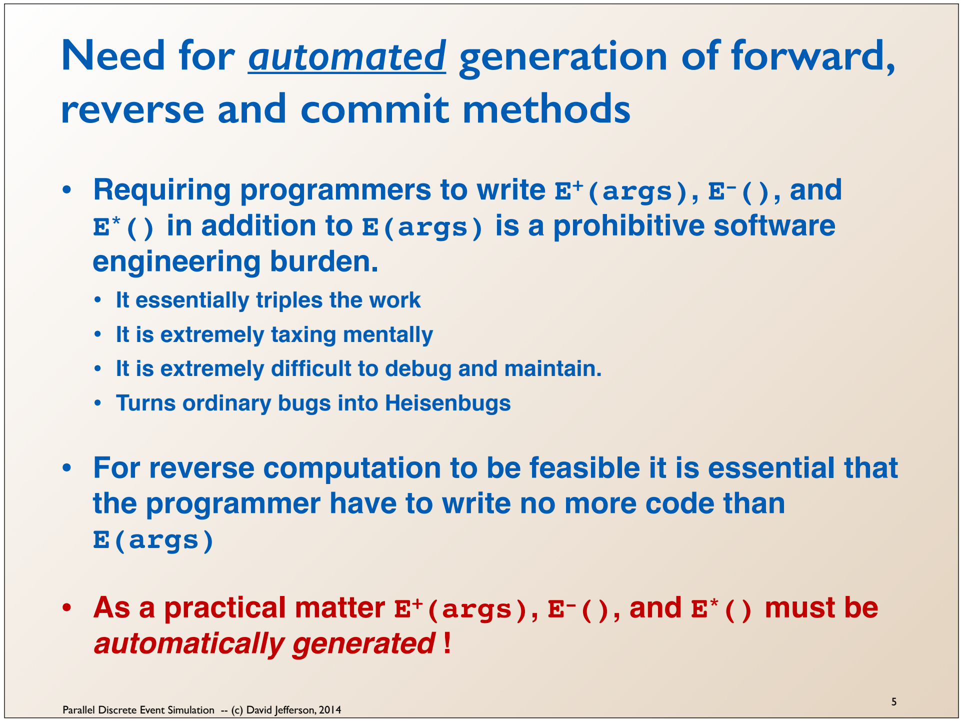
Q: FIFO queue (with virtual timestamps) used to hold data needed for commitment

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 4

In factoring an event method we generally use two auxiliary data structures. One, which we call S in these slides, is a LIFO stack that is used to hold data needed for rollback. The LIFO structure is natural, because the data that we need for executing $\mathbf{E}^{-}\mathbf{()}$ is need in the reverse order of the order in which it was put in during forward execution of $\mathbf{E}^{+}\mathbf{()}$.

The other we are calling Q, and it is a FIFO queue of data to be saved for $\mathbf{E}^{*}\mathbf{()}$ at commitment time. Since the things done at commitment time (e.g. output) have to be done in the same order as specified during execution of $\mathbf{E}^{+}\mathbf{()}$, it is natural for a FIFO queue to be employed.

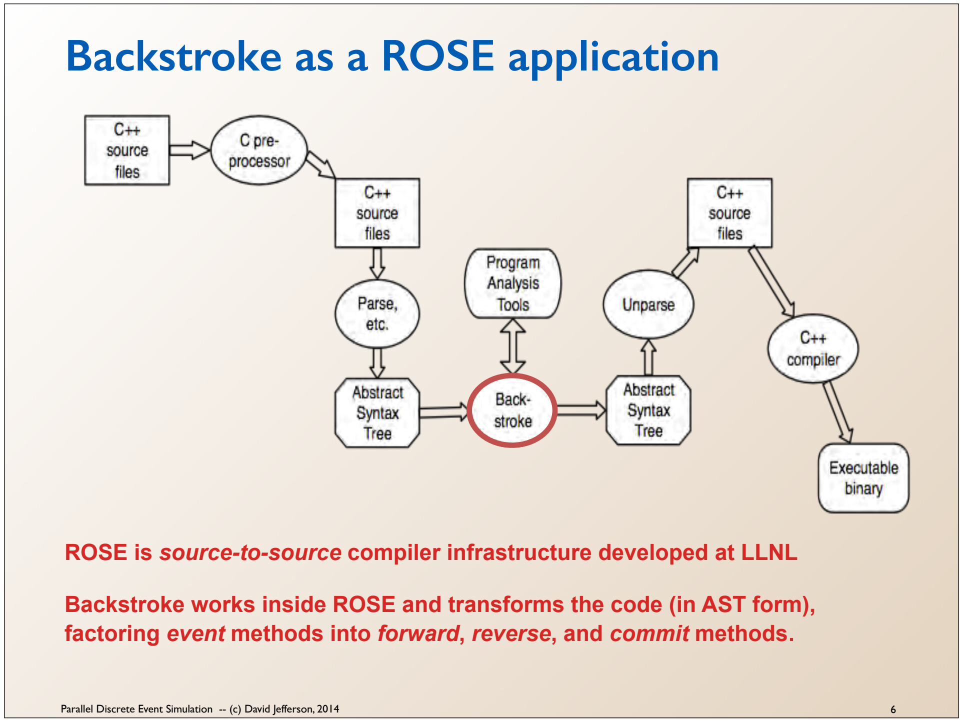
The "**`push`**", "**`pop`**" and later "**`top`**" and "**`front`**" terminology are from the C++ STL.



Need for *automated* generation of forward, reverse and commit methods

- Requiring programmers to write E^{+}`(args)`, E^{-}`()`, and E^{*}`()` in addition to `E(args)` is a prohibitive software engineering burden.
  - It essentially triples the work
  - It is extremely taxing mentally
  - It is extremely difficult to debug and maintain.
  - Turns ordinary bugs into Heisenbugs
- For reverse computation to be feasible it is essential that the programmer have to write no more code than `E(args)`
- As a practical matter E^{+}`(args)`, E^{-}`()`, and E^{*}`()` must be *automatically generated* !

LLNL has a project called Backstroke that is intended to automatically generate reverse code for almost any code written in C++.



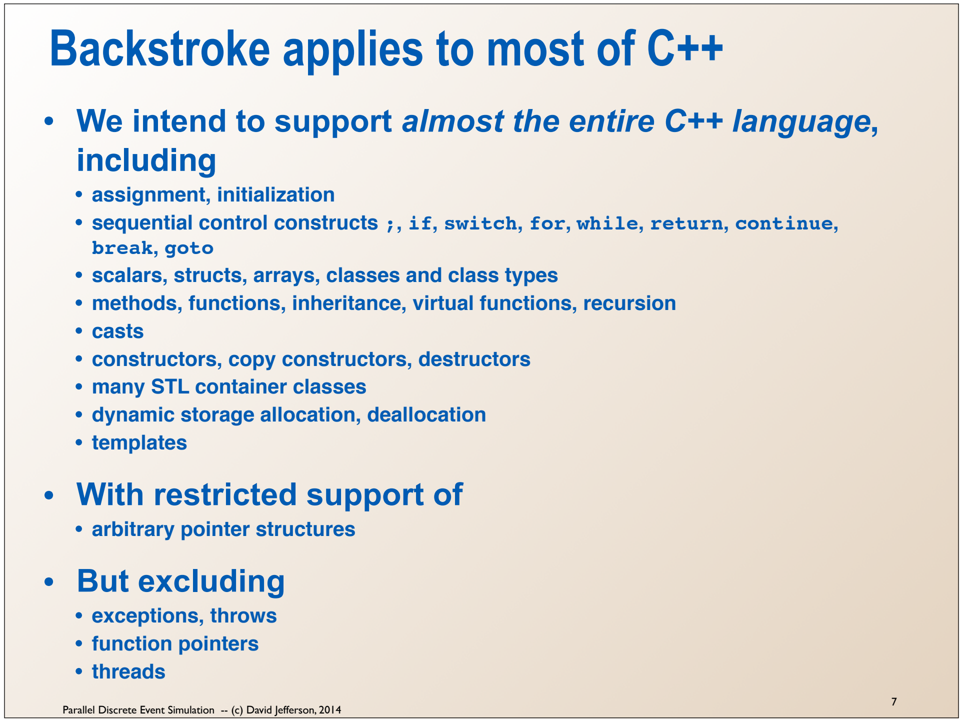
Backstroke as a ROSE application

ROSE is *source-to-source* compiler infrastructure developed at LLNL

Backstroke works inside ROSE and transforms the code (in AST form), factoring *event* methods into *forward*, *reverse*, and *commit* methods.

This is a diagram of the workflow in ROSE, a general source-to-source program transformation system and compiler. Backstroke is the component in **red** that uses ROSE's powerful program analysis tools and adds forward, reverse and commit routines for all event methods in a ROSS-compatible simulation.

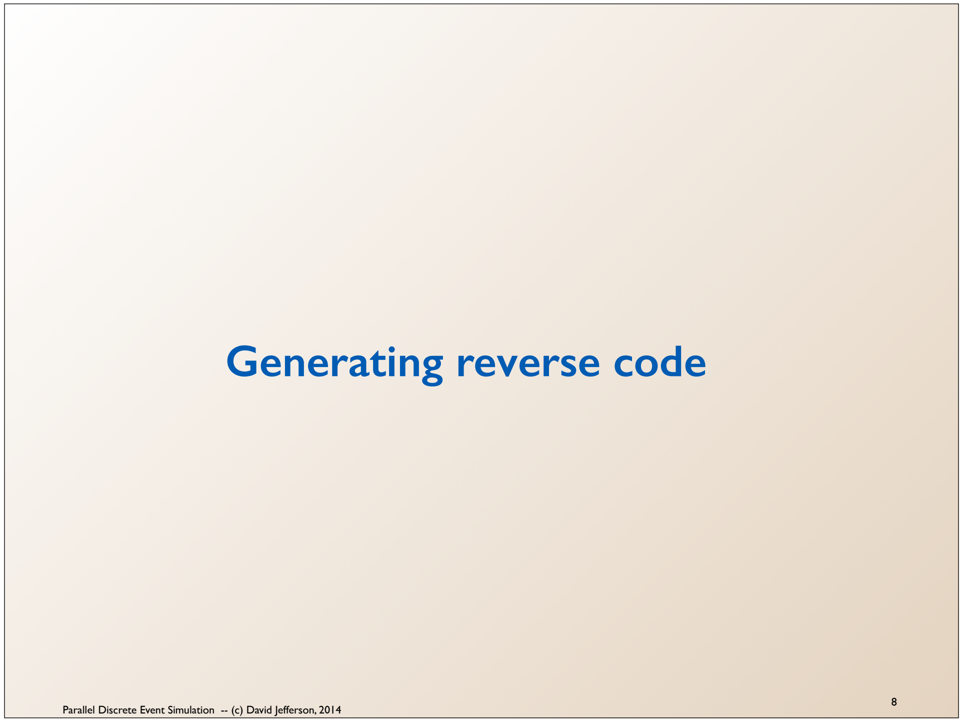
It is not absolutely essential to construct a reverse code generator this way. You could go directly from the revised Abstract Syntax Tree to executable binary. (The LORAIN project at RPI, based on LLVM, is structured that way.) But by going back through source code, the programmer can see what the forward, reverse, and commit methods look like, and perhaps learn how to improve their performance.



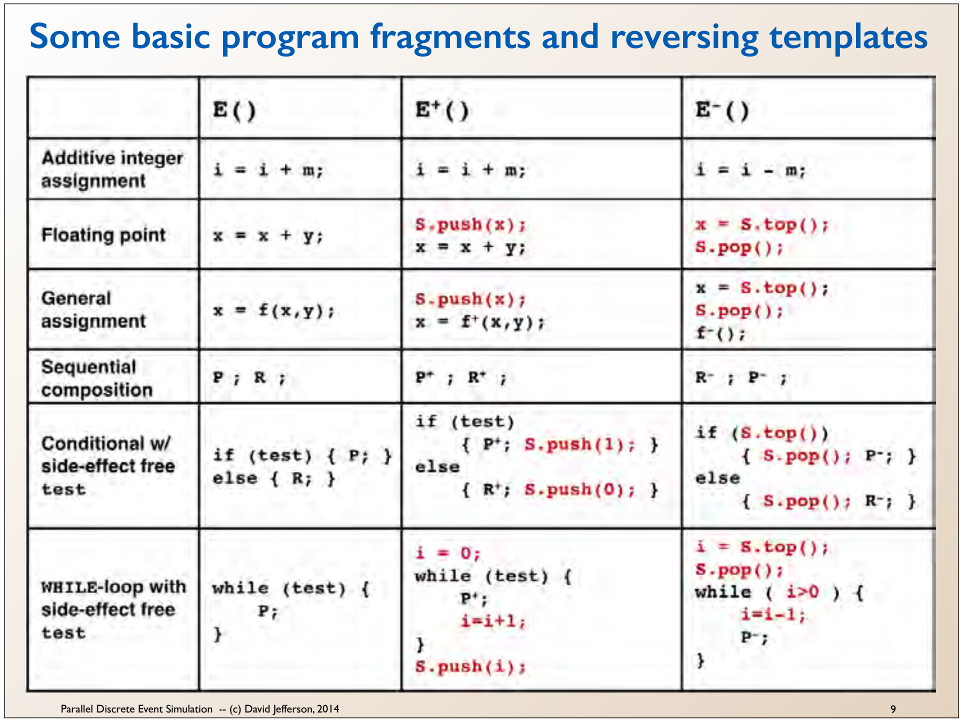
Backstroke applies to most of C++

- We intend to support *almost the entire C++ language*, including
  - assignment, initialization
  - sequential control constructs `;`, `if`, `switch`, `for`, `while`, `return`, `continue`, `break`, `goto`
  - scalars, structs, arrays, classes and class types
  - methods, functions, inheritance, virtual functions, recursion
  - casts
  - constructors, copy constructors, destructors
  - many STL container classes
  - dynamic storage allocation, deallocation
  - templates
- With restricted support of
  - arbitrary pointer structures
- But excluding
  - exceptions, throws
  - function pointers
  - threads

Some parts of the C++ language are straightforward to support. Some are quite tricky. Some may benefit greatly from programmer advice. and some are so difficult to support that it will never be worth the effort to try. If generating reverse code for legacy code we may need hand work to rewrite those parts that use features of the language that are not conducive to reversibility. For new code we need to advise programmers to avoid certain language constructs that will preclude automatic generation of reverse code.



Generating reverse code



Some basic program fragments and reversing templates

	E()	E^+()	E^-()
Additive integer assignment	i = i + m;	i = i + m;	i = i - m;

Floating point	x = x + y;	S.push(x);
		x = x + y;
			x = S.top();
			S.pop();

General assignment	x = f(x,y);	S.push(x);
		x = f^+(x,y);
			x = S.top();
			S.pop();
			f^-();

Sequential composition	P ; R ;	P^+ ; R^+ ;	R^- ; P^- ;

Conditional w/ side-effect free test	if (test) { P; }
	else { R; }
		if (test)
		{ P^+; S.push(1); }
		else
		{ R^+; S.push(0); }
			if (S.top())
			{ S.pop(); P^-; }
			else
			{ S.pop(); R^-; }

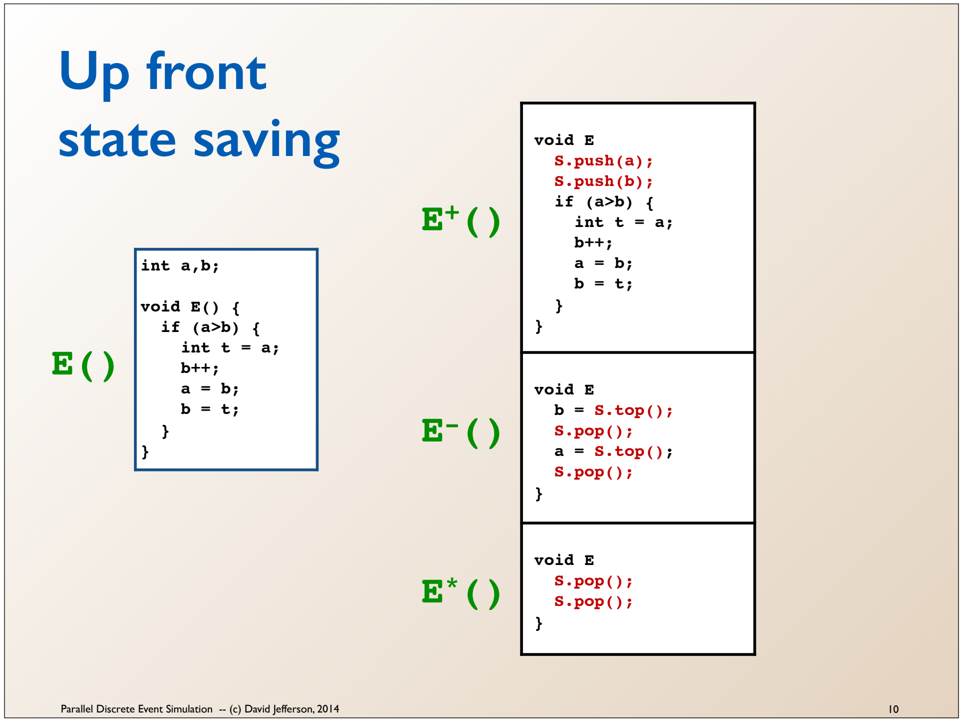
WHILE-loop with side-effect free test	while (test) {
	P;
	}
		i = 0;
		while (test) {
		P^+;
		i=i+1;
		}
		S.push(i);
			i = S.top();
			S.pop();
			while (i>0) {
			i=i-1;
			P^-;
			}

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 9

S is the global stack onto which all saved data is pushed that is required for rolling back forward execution.

For the tests in both the conditional and the while loop, we assume there are no side effects. If there are, then the reverse code can to be easily adjusted.

Note that integer increments / decrements and sequential composition require no data to be stored on the stack **S**.



Up front state saving

E()

```
int a,b;

void E() {
  if (a>b) {
    int t = a;
    b++;
    a = b;
    b = t;
  }
}
```

E$^+$()

```
void E
  S.push(a);
  S.push(b);
  if (a>b) {
    int t = a;
    b++;
    a = b;
    b = t;
  }
}
```

E$^-$()

```
void E
  b = S.top();
  S.pop();
  a = S.top();
  S.pop();
}
```

E*()

```
void E
  S.pop();
  S.pop();
}
```

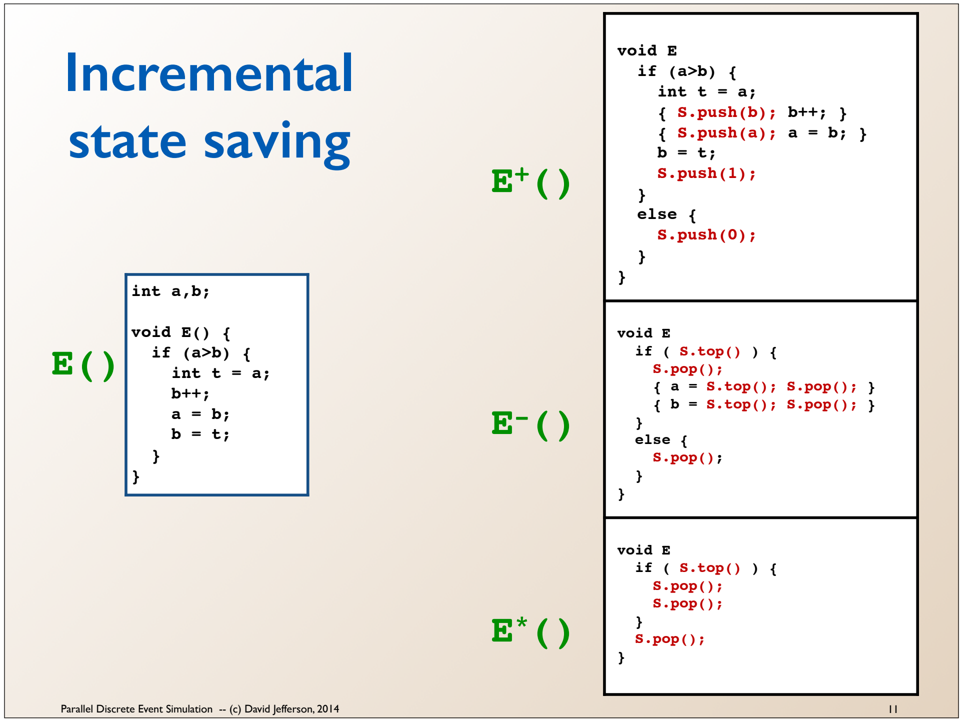
Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

10

With up front state saving we identify all of the state variables that *might* change and push their values on the stack at the beginning of the forward routine. We do not have to record which branch of the conditional was taken, and if there were a loop in the method body we would not need to record how many times the body was executed. In this case both state variables do change. But if there were more variables in the state and we could not determine statically that some of them would not change during execution of **E()** we would just introduce code in **E**$^+$**()** and **E**$^-$**()** to save and restore them all. In **E**$^-$**()** we simply restore the values of **a** and **b** and pop the stack.. We do not have to pay any attention to the algorithm used in **E()**.

The commit method **E*****()** must also pop any values off the stack that were pushed there by the forward method because the commit method is called if and only if the reverse method **E**$^-$**()** is *not* called.

Note also that variable **t** here is only a temp. It is not a state variable in the simulation, and hence it does not have to be restored during rollback.



Incremental state saving

E()

```
int a,b;

void E() {
  if (a>b) {
    int t = a;
    b++;
    a = b;
    b = t;
  }
}
```

E+()

```
void E
  if (a>b) {
    int t = a;
    { S.push(b); b++; }
    { S.push(a); a = b; }
    b = t;
    S.push(1);
  }
  else {
    S.push(0);
  }
}
```

E-()

```
void E
  if ( S.top() ) {
    S.pop();
    { a = S.top(); S.pop(); }
    { b = S.top(); S.pop(); }
  }
  else {
    S.pop();
  }
}
```

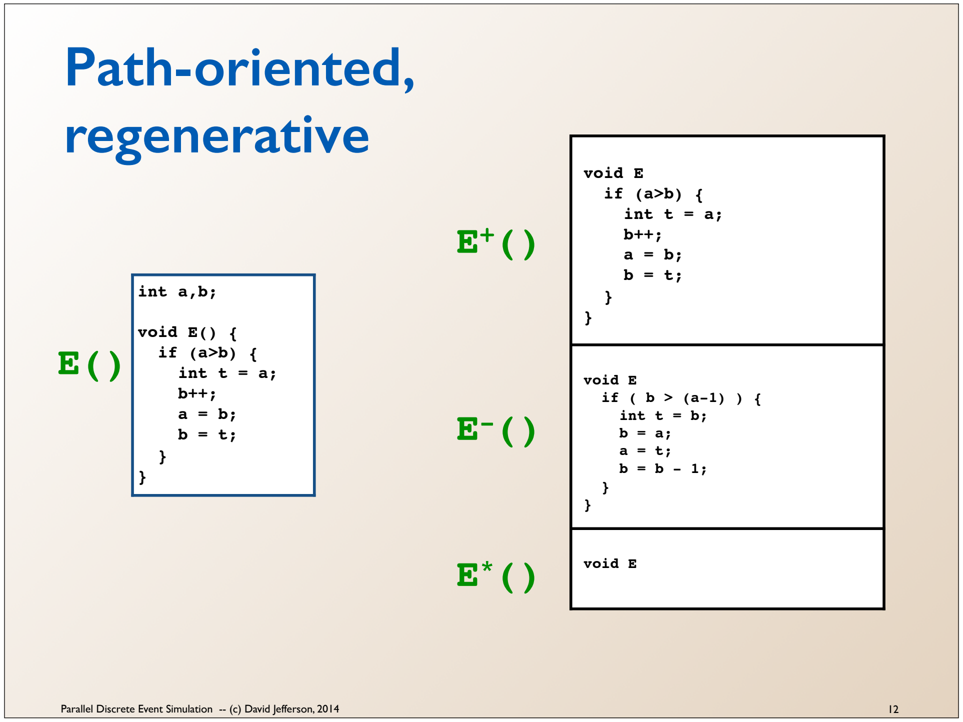
E*()

```
void E
  if ( S.top() ) {
    S.pop();
    S.pop();
  }
  S.pop();
}
```

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 11

With Incremental State Saving we instrument the forward routine $E^{+}()$ to save the value of a state variable the very first time it is overwritten or, if we cannot determine that statically, then we save the value every time it is overwritten that *might possibly be the first time*. In the forward method $E^{+}()$ we save its value only the first time if we can because of course in the reverse method we only need to restore it to its *initial* value. In this example the variable b is overwritten twice, but we push its value onto the stack only the first time. Of course we also have to push a boolean indicating which branch of the conditional was taken. The reverse method, $E^{-}()$, only restores the variable to their original values and pops the stack.

The commit method $E^{*}()$ must also pop any values off the stack that were pushed there by the forward method because the commit method is called if and only if the reverse method $E^{-}()$ is *not* called.



Path-oriented, regenerative

$E()$

```
int a,b;

void E() {
  if (a>b) {
    int t = a;
    b++;
    a = b;
    b = t;
  }
}
```

$E^{+}()$

```
void E
  if (a>b) {
    int t = a;
    b++;
    a = b;
    b = t;
  }
}
```

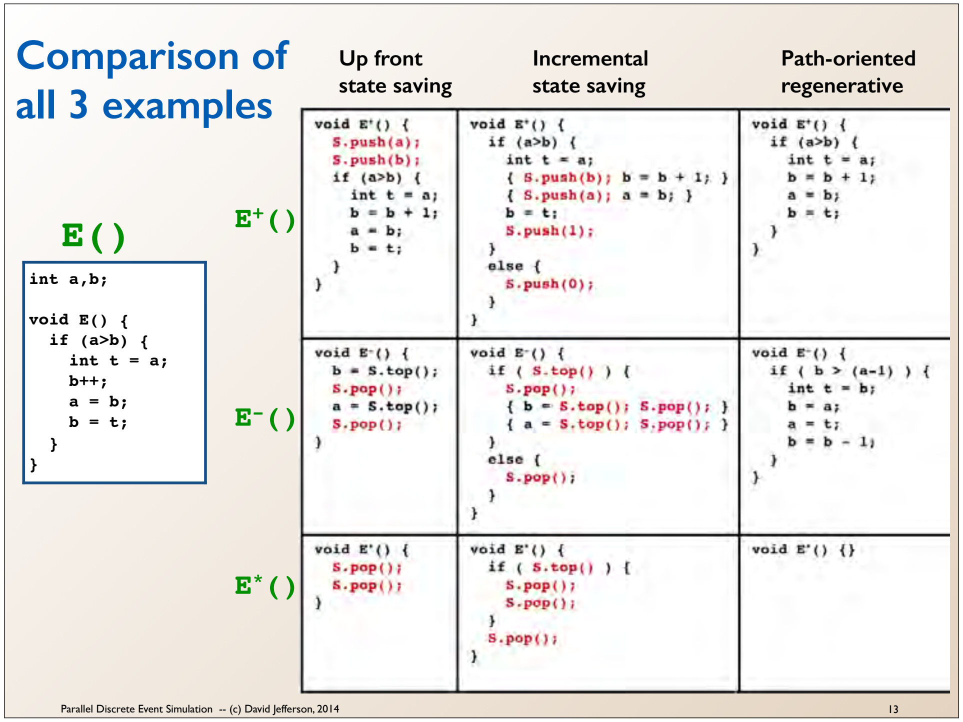
$E^{-}()$

```
void E
  if ( b > (a-1) ) {
    int t = b;
    b = a;
    a = t;
    b = b - 1;
  }
}
```

$E^{*}()$

```
void E
```

In this case the path-oriented and regenerative style of reverse code generation manages to produce *perfectly reversible code*, with nothing pushed onto or popped from the stack. Not that the forward $\mathbf{E}^{+}\mathbf{()}$ routine in this case is identical to **E()** and the commit routine $\mathbf{E}^{*}\mathbf{()}$ is a no-op. We did not even have to introduce any additional variables. Perfect reversibility is not usually achievable for a whole event method, but it often is for at least some regions of the code and/or some variables. It is the ideal of reverse computation.



Comparison of all 3 examples

`E()`

```
int a,b;

void E() {
  if (a>b) {
    int t = a;
    b++;
    a = b;
    b = t;
  }
}
```

	Up front state saving	Incremental state saving	Path-oriented regenerative

`E`$^+$`()`	`void E+() {`
	`S.push(a);`
	`S.push(b);`
	`if (a>b) {`
	`int t = a;`
	`b = b + 1;`
	`a = b;`
	`b = t;`
	`}`
	`}`
		`void E+() {`
		`if (a>b) {`
		`int t = a;`
		`{ S.push(b); b = b + 1; }`
		`{ S.push(a); a = b; }`
		`b = t;`
		`S.push(1);`
		`}`
		`else {`
		`S.push(0);`
		`}`
		`}`
			`void E+() {`
			`if (a>b) {`
			`int t = a;`
			`b = b + 1;`
			`a = b;`
			`b = t;`
			`}`
			`}`

`E`$^-$`()`	`void E-() {`
	`b = S.top();`
	`S.pop();`
	`a = S.top();`
	`S.pop();`
	`}`
		`void E-() {`
		`if ( S.top() ) {`
		`S.pop();`
		`{ b = S.top(); S.pop(); }`
		`{ a = S.top(); S.pop(); }`
		`}`
		`else {`
		`S.pop();`
		`}`
		`}`
			`void E-() {`
			`if ( b > (a-1) ) {`
			`int t = b;`
			`b = a;`
			`a = t;`
			`b = b - 1;`
			`}`
			`}`

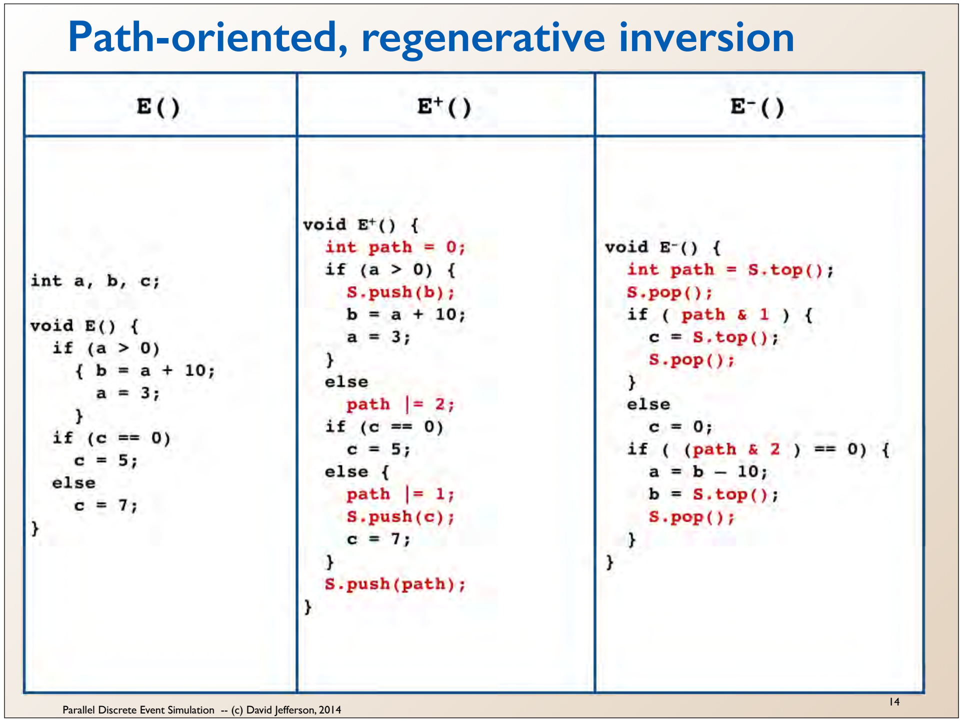
`E`*`()`	`void E*() {`
	`S.pop();`
	`S.pop();`
	`}`
		`void E*() {`
		`if ( S.top() ) {`
		`S.pop();`
		`S.pop();`
		`}`
		`S.pop();`
		`}`
			`void E*() {}`

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 13

This slide just summarizes the last three examples. The original code for **`E()`** is in a box on the left, and three different ways of factoring it into **`E`**$^+$**`()`**, **`E`**$^-$**`()`**, and **`E`*****`()`** are recorded in the next three columns of the table.

The comparison shows that in this particular example, if the condition **`(a>b)`** is true, then there is more overhead with incremental state saving then there is with up front state saving, but if **`(a>b)`** is false then the reverse is true. This is not a general statement, however.

Also in this case the path-oriented regenerative methods generate perfect reverse code that does not need to save any data on the stack at all. In this case the code it produces is clearly superior to either of the other methods.



Path-oriented, regenerative inversion

E()

```
int a, b, c;

void E() {
  if (a > 0)
    { b = a + 10;
      a = 3;
    }
  if (c == 0)
    c = 5;
  else
    c = 7;
}
```

E^+()

```
void E+() {
  int path = 0;
  if (a > 0) {
    S.push(b);
    b = a + 10;
    a = 3;
  }
  else
    path |= 2;
  if (c == 0)
    c = 5;
  else {
    path |= 1;
    S.push(c);
    c = 7;
  }
  S.push(path);
}
```

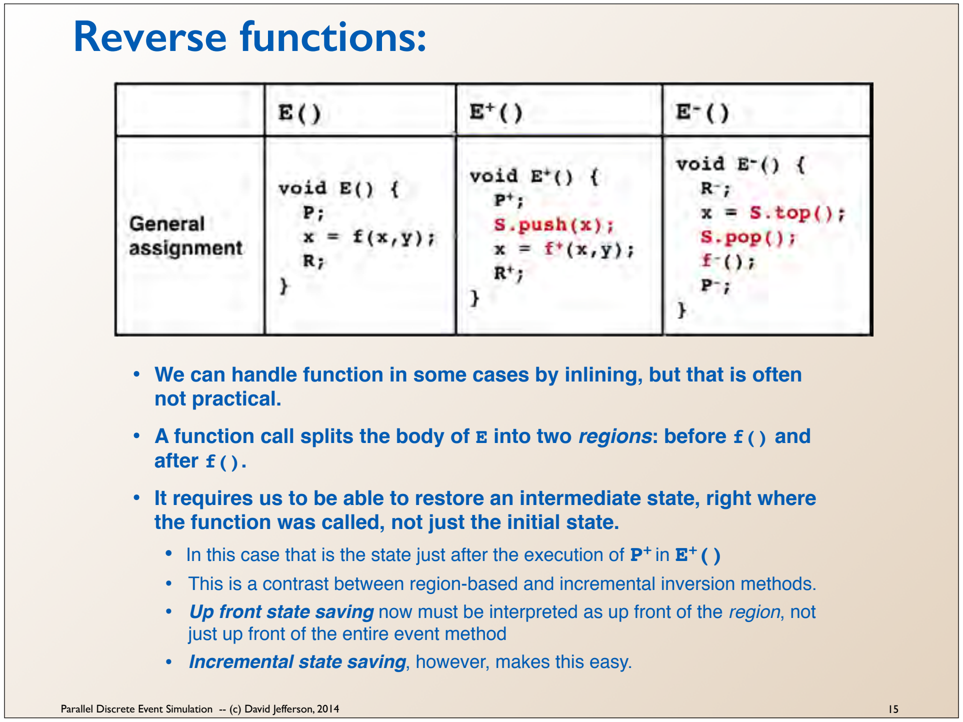
E^-()

```
void E-() {
  int path = S.top();
  S.pop();
  if ( path & 1 ) {
    c = S.top();
    S.pop();
  }
  else
    c = 0;
  if ( (path & 2 ) == 0) {
    a = b – 10;
    b = S.top();
    S.pop();
  }
}
```

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

14

This is another example application of Backstroke's path-oriented regenerative inversion algorithm. The variables **`a`**, **`b`**, and **`c`** are state variables. The forward code is instrumented in red to keep track of the dynamic path taken, and the reverse code uses path information to restore variable values. Note that the variable **`path`** that is introduced in the forward and reverse methods should be viewed as a bit mask that records for each conditional which branch on the conditional was taken. In the forward routine the low order bit of **`path`** records is **`0`** if the **`then`**-branch of the second conditional is taken, and is a **`1`** if it is not. The second bit records the same thing for the first conditional. In the reverse method the corresponding conditionals are reversed in order, so the low order bit indicates which branch to take in the first conditional, and the second bit indicates which branch to take with the second conditional.



Reverse functions:

	$E()$	$E^{+}()$	$E^{-}()$

General assignment	`void E() {`
	`P;`
	`x = f(x,y);`
	`R;`
	`}`
		void E^{+}() {
		P^{+};
		`S.push(x);`
		x = f^{+}(x,y);
		R^{+};
		}
			void E^{-}() {
			R^{-};
			`x = S.top();`
			`S.pop();`
			f^{-}();
			P^{-};
			}

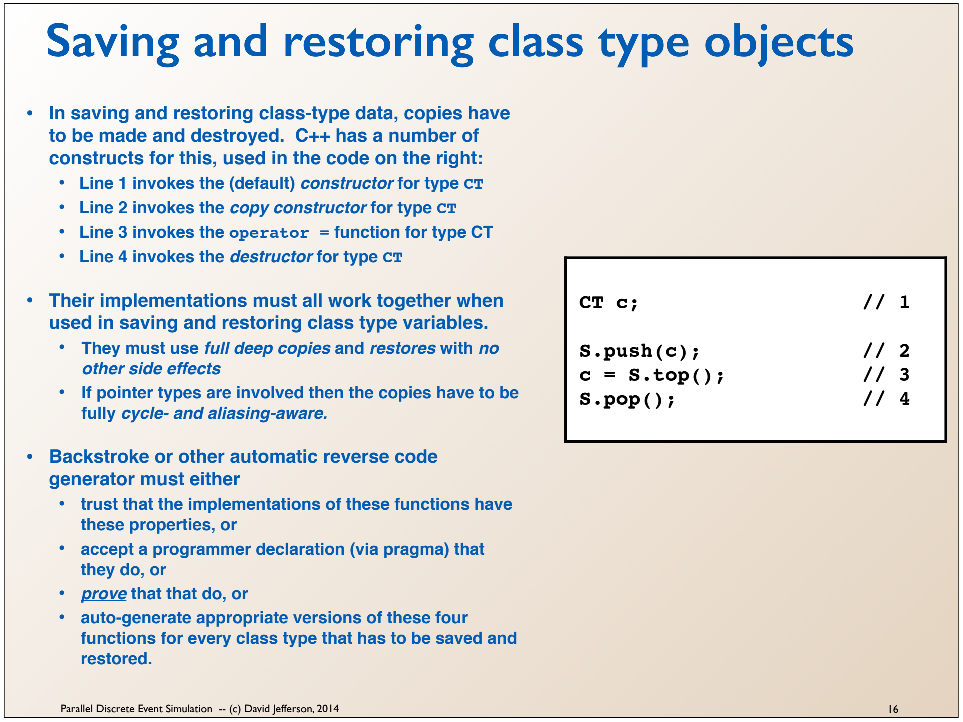
- **We can handle function in some cases by inlining, but that is often not practical.**
- **A function call splits the body of `E` into two *regions*: before `f()` and after `f()`.**
- **It requires us to be able to restore an intermediate state, right where the function was called, not just the initial state.**
  - In this case that is the state just after the execution of P^{+} in $E^{+}()$
  - This is a contrast between region-based and incremental inversion methods.
  - ***Up front state saving*** now must be interpreted as up front of the *region*, not just up front of the entire event method
  - ***Incremental state saving***, however, makes this easy.

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 15

`S` is the global stack onto which all saved data is pushed that is required for rolling back forward execution.

For the tests in both the conditional and the while loop, we assume there are no side effects. If there are, then the reverse code can to be easily adjusted.

Note that integer addition / subtraction and sequential composition require no data to be stored on the stack **`S`**.



Saving and restoring class type objects

- In saving and restoring class-type data, copies have to be made and destroyed. C++ has a number of constructs for this, used in the code on the right:
  - Line 1 invokes the (default) *constructor* for type `CT`
  - Line 2 invokes the *copy constructor* for type `CT`
  - Line 3 invokes the `operator =` function for type CT
  - Line 4 invokes the *destructor* for type `CT`
- Their implementations must all work together when used in saving and restoring class type variables.
  - They must use *full deep copies* and *restores* with *no other side effects*
  - If pointer types are involved then the copies have to be fully *cycle- and aliasing-aware.*
- Backstroke or other automatic reverse code generator must either
  - trust that the implementations of these functions have these properties, or
  - accept a programmer declaration (via pragma) that they do, or
  - *prove* that that do, or
  - auto-generate appropriate versions of these four functions for every class type that has to be saved and restored.

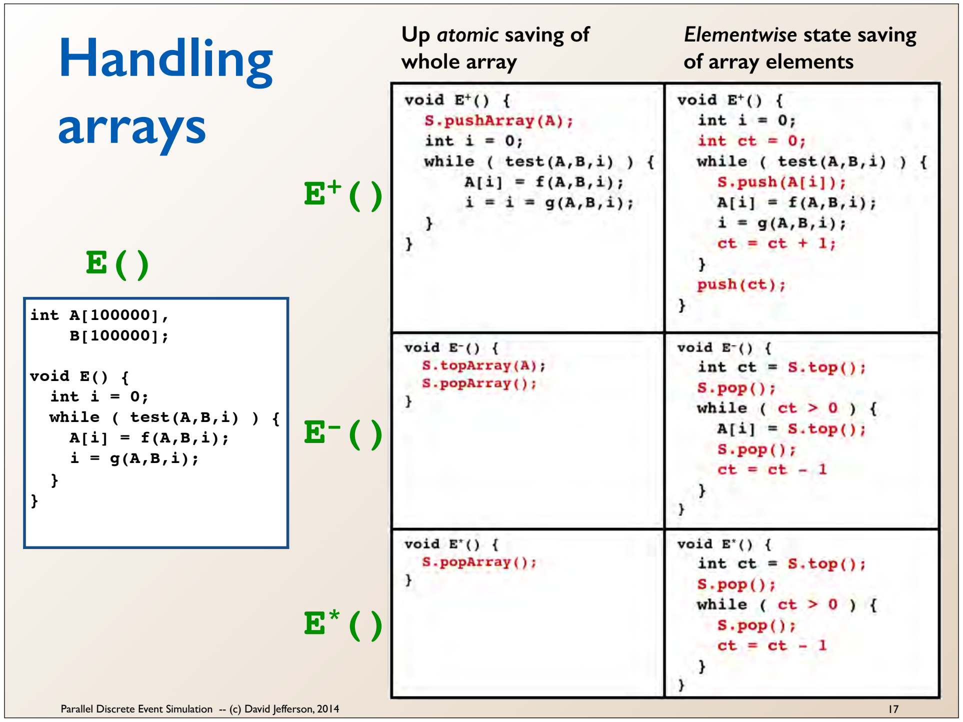
```
CT c;                       // 1

S.push(c);                  // 2
c = S.top();                // 3
S.pop();                    // 4
```

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 16

The semantics of C++ constructors, copy constructors, destructors, and assignment operators all play a fundamental role in the way reverse computation is implemented when class-type values are involved. We cannot assume that the destructor is the reverse of the constructor, and we cannot assume that they all play well together and have the properties needed in the general case for reverse computation, namely that full, deep copies are made that are aliasing- and cycle-aware and that they have no other side effects except those required for perfect copies.

In C++ it is also useful in some patterns to create uncopyable or unassignable types, and those simply should not be used as the values of state variables in a simulation.



Handling arrays

$E()$

```
int A[100000],
    B[100000];

void E() {
  int i = 0;
  while ( test(A,B,i) ) {
    A[i] = f(A,B,i);
    i = g(A,B,i);
  }
}
```

Up *atomic* saving of whole array

Elementwise state saving of array elements

$E^+()$

```
void E+() {
  S.pushArray(A);
  int i = 0;
  while ( test(A,B,i) ) {
      A[i] = f(A,B,i);
      i = i = g(A,B,i);
  }
}
```

```
void E+() {
  int i = 0;
  int ct = 0;
  while ( test(A,B,i) ) {
    S.push(A[i]);
    A[i] = f(A,B,i);
    i = g(A,B,i);
    ct = ct + 1;
  }
  push(ct);
}
```

$E^-()$

```
void E-() {
  S.topArray(A);
  S.popArray();
}
```

```
void E-() {
  int ct = S.top();
  S.pop();
  while ( ct > 0 ) {
    A[i] = S.top();
    S.pop();
    ct = ct - 1
  }
}
```

$E^*()$

```
void E*() {
  S.popArray();
}
```

```
void E*() {
  int ct = S.top();
  S.pop();
  while ( ct > 0 ) {
    S.pop();
    ct = ct - 1
  }
}
```

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

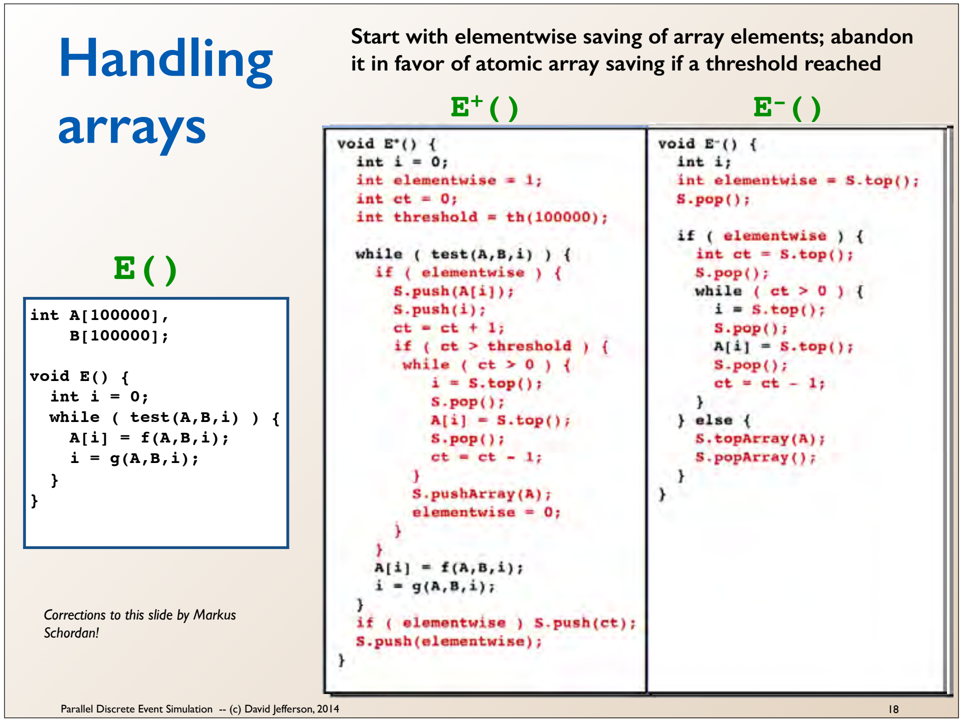
17

In this example the **A** and **B** arrays are state variables, but **i** is not. And the functions **test**, **f**, and **g** are side-effect free.

For the purposes of exposition on this slide we have assumed the existence of methods **S.pushArray(A)**, **topArray(A)**, and **S.popArray()** that do for array arguments the same things as **S.push(n)**, **S.top()**, and **S.pop()** do for integer arguments.

This exemplifies the tradeoffs that come with handling arrays and other collections. If the **while**-loop is executed only a few times, then it is much faster to do elementwise saving of the few elements of the array that are overwritten and restore them one at a time in case of rollback than to save and restore the entire 100,000-element array. But if the loop is executed many times, and many elements of the array are overwritten, then it is faster to simply save the entire array as an atomic data structure and restore the same way on rollback.

But we may not be able to determine statically which of those is the case, which leaves the reverse code generator in a quandary. Which kind of reverse code should it generate? One approach is to allow the programmer to offer advice in the form of a pragma or specially formatted comment indicating which choice to use. Another approach is illustrated on the next slide.



Handling arrays

Start with elementwise saving of array elements; abandon it in favor of atomic array saving if a threshold reached

E()

```
int A[100000],
    B[100000];

void E() {
  int i = 0;
  while ( test(A,B,i) ) {
    A[i] = f(A,B,i);
    i = g(A,B,i);
  }
}
```

E^{+}()

```
void E+() {
  int i = 0;
  int elementwise = 1;
  int ct = 0;
  int threshold = th(100000);

  while ( test(A,B,i) ) {
    if ( elementwise ) {
      S.push(A[i]);
      S.push(i);
      ct = ct + 1;
      if ( ct > threshold ) {
       while ( ct > 0 ) {
          i = S.top();
          S.pop();
          A[i] = S.top();
          S.pop();
          ct = ct - 1;
        }
        S.pushArray(A);
        elementwise = 0;
      }
    }
    A[i] = f(A,B,i);
    i = g(A,B,i);
  }
  if ( elementwise ) S.push(ct);
  S.push(elementwise);
}
```

E^{-}()

```
void E-() {
  int i;
  int elementwise = S.top();
  S.pop();

  if ( elementwise ) {
    int ct = S.top();
    S.pop();
    while ( ct > 0 ) {
      i = S.top();
      S.pop();
      A[i] = S.top();
      S.pop();
      ct = ct - 1;
    }
  } else {
    S.topArray(A);
    S.popArray();
  }
}
```

Corrections to this slide by Markus Schordan!

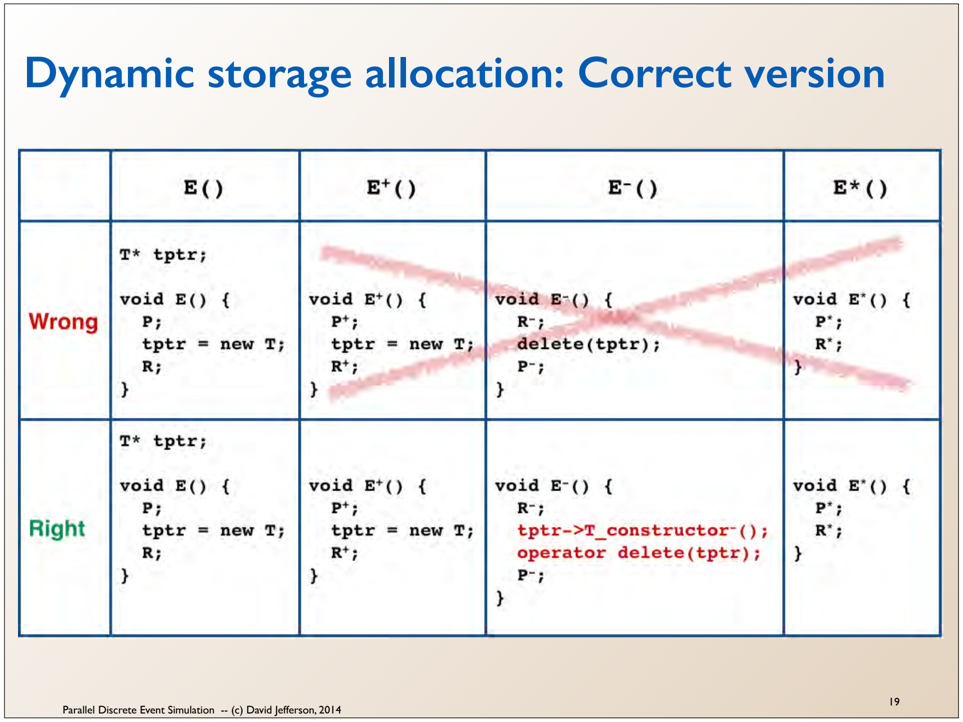
Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 18

Multiple corrections on this slide thanks to Markus Schordan.

In this example the **`A`** and **`B`** arrays are state variables, but **`i`** is not. The functions **`th`**, **`test`**, **`f`**, and **`g`** are all side-effect free. The function **`S.pushArray(A)`** is intended to push the entire array **`A`** onto the stack, even though this is not strictly correct C++; likewise **`S.topArray(A)`** is intended to copy the entire array value from the top of the stack into **`A`**. **`S.popArray()`** just pops off the entire array.

Here we do something sophisticated. We do not decide decide statically whether to use elementwise or atomic saving of array **`A`**. *Instead, we make some measurements at runtime and decide then.* Whether this method will prove practical or not (i.e. whether we can automatically generate code like this) is an open research question, but this example illustrates the depth and complexity of the options we have in generating good forward and reverse code.

In the forward method **`E`**$^{+}$**`()`** we don't know how many times the loop will be executed, and so we start out assuming that we will be doing incremental saving of array elements as they are modified, keeping count of the number of loop executions and also pushing array elements onto stack **`S`** as they are modified. But after a certain threshold number of elements have been pushed onto the stack, it becomes probable the loop will cycle many times and that a large fraction of all of the array elements will be modified. At that point we abandon elementwise saving and pop everything off of the stack that we pushed onto it. We then start over, pushing the entire array **`A`** onto the stack all at once. If, at the end of the loop, we did not abandon elementwise saving of array **`A`**, then we push the loop count **`ct`** on the stack to prepare for elementwise restoration. Either way, the last thing we push on the stack is the boolean **`elementwise`** to tell the reverse method **`E`**$^{-}$**`()`** which mechanism to use to restore **`A`**.



Dynamic storage allocation: Correct version

	E()	E^+()	E^-()	E*()

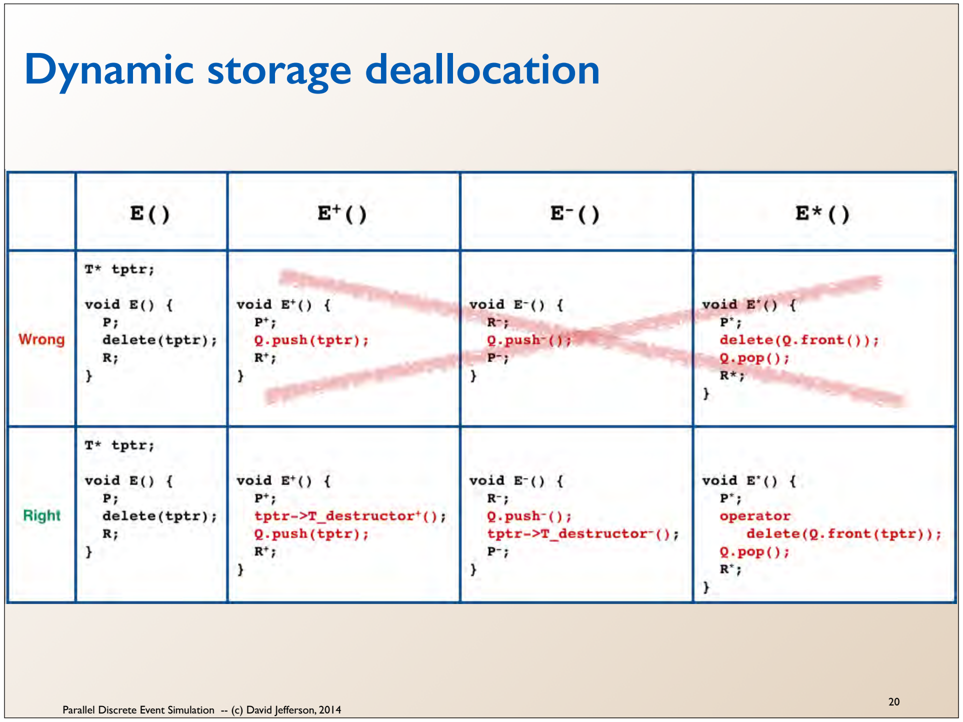
Wrong	T* tptr;
	void E() {
	P;
	tptr = new T;
	R;
	}
		~~void E^+() {~~
		~~P^+;~~
		~~tptr = new T;~~
		~~R^+;~~
		~~}~~
			~~void E^-() {~~
			~~R^-;~~
			~~delete(tptr);~~
			~~P^-;~~
			~~}~~
				~~void E^*() {~~
				~~P^*;~~
				~~R^*;~~
				~~}~~

Right	T* tptr;
	void E() {
	P;
	tptr = new T;
	R;
	}
		void E^+() {
		P^+;
		tptr = new T;
		R^+;
		}
			void E^-() {
			R^-;
			tptr->T_constructor$^-$();
			operator delete(tptr);
			P^-;
			}
				void E^*() {
				P^*;
				R^*;
				}

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

19

Calling **new T** in the forward method is OK. But we must realize that this invokes a constructor for type **T** in C++, (which in turn invokes a whole hierarchy of other constructors for members in tyre and parent types, all called in canonical order). The problem with treating **delete()** as its reverse is that **delete()** calls the *destructor* for the type of data being deleted, *but the destructor is not generally the reverse of the constructor!* In general, programmer-defined constructors can have arbitrary side-effects on other variables, which may or may not be exactly reversed by the matching destructor. What we need is for each constructor to have a *reverse constructor* for T, which we denote here by **T_constructor**$^-$**()**. We call that first, and the use the obscure C++ construct **operator delete()** to finally free the storage allocated by **new**. In C++ **operator delete()** does not call a destructor.



Dynamic storage deallocation

	E()	E^{+}()	E^{-}()	E*()

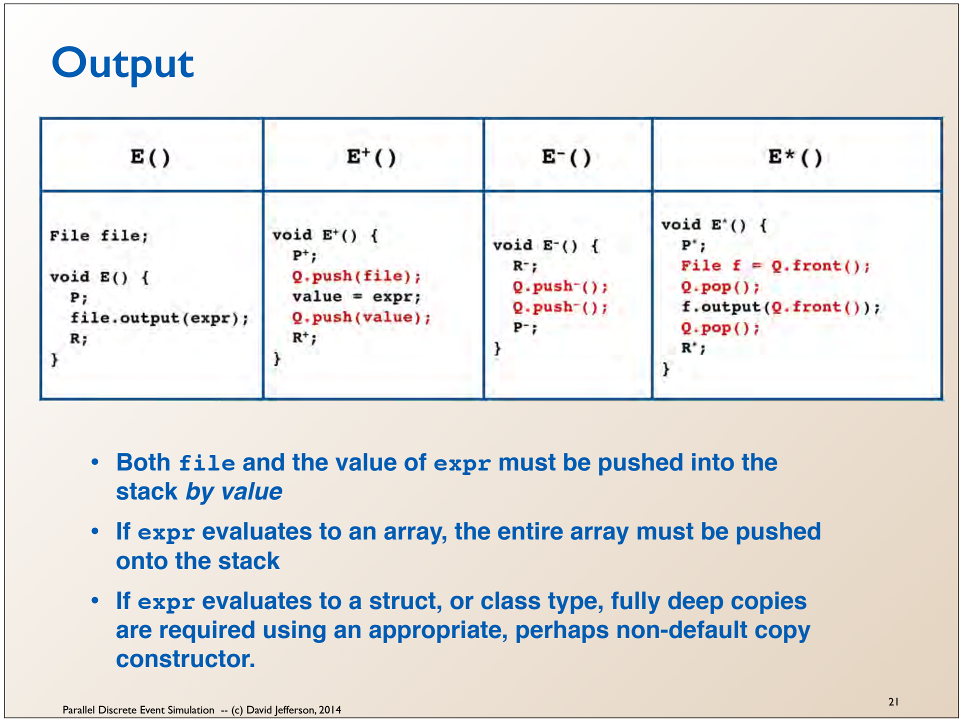
Wrong	T* tptr;
	
	void E() {
	P;
	delete(tptr);
	R;
	}
		~~void E^{+}() {
		P^{+};
		Q.push(tptr);
		R^{+};
		}~~
			~~void E^{-}() {
			R^{-};
			Q.$push^{-}$();
			P^{-};
			}~~
				~~void E^{*}() {
				P^{*};
				delete(Q.front());
				Q.pop();
				R*;
				}~~

Right	T* tptr;
	
	void E() {
	P;
	delete(tptr);
	R;
	}
		void E^{+}() {
		P^{+};
		tptr->T_$destructor^{+}$();
		Q.push(tptr);
		R^{+};
		}
			void E^{-}() {
			R^{-};
			Q.$push^{-}$();
			tptr->T_$destructor^{-}$();
			P^{-};
			}
				void E^{*}() {
				P^{*};
				operator
				delete(Q.front(tptr));
				Q.pop();
				R^{*};
				}

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

20

As with new, a similar problem arises in reversing **delete()** in that **delete()** invokes the destructor for the type of object being destroyed. We need to invoke that destructor in the forward routine, because the side effects of the destructor on other variables can be felt in subsequent statements (**R**). However, we need to run not the original destructor, **tptr->T_destruct()**, but its forward instrumented version, **tptr->T_destruct+()**. And we need to be able to reverse the effects of **tptr->T_destruct+()** in the **E^{-}()** by calling **tptr->T_destruct-()**. But while we have to reverse the effects of the destructor in **E^{-}()**, we cannot really free the storage associated with the object in the **E^{-}() routine**, because if we do the storage could be re-allocated and overwritten, and the overwriting would make it impossible for us to reverse the action in **E^{-}()**. So we delay the actual freeing of the storage until the commit routine, and then we do it using the **operator delete()** construct.



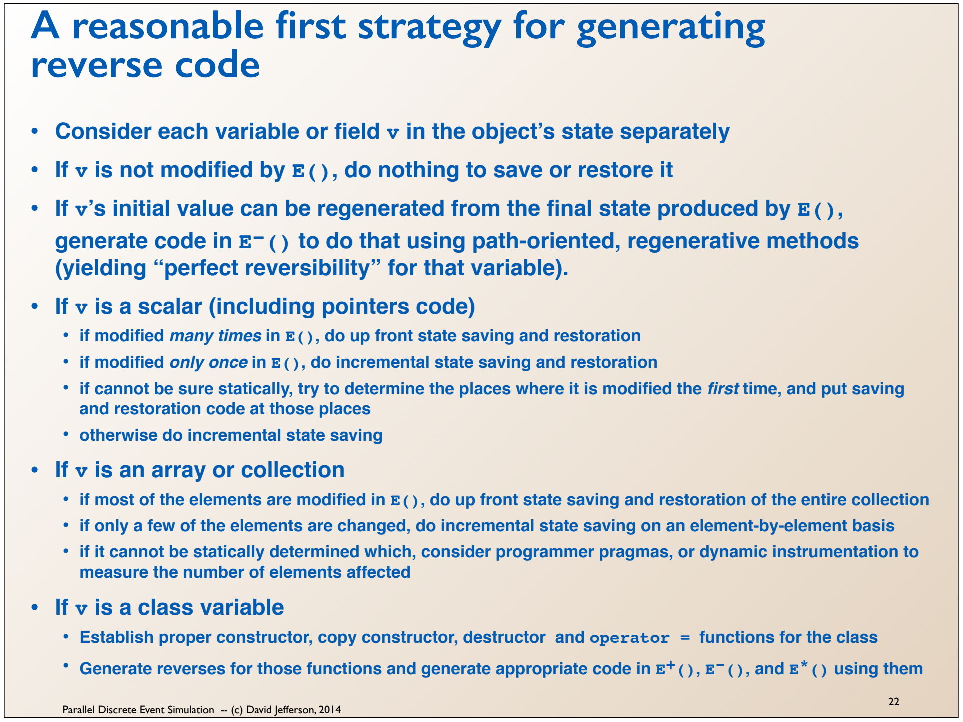
Output

E()	E^{+}()	E^{-}()	E^{*}()

File file;
 	
void E() {
  P;
  file.output(expr);
  R;
}
	void E^{+}() {
	  P^{+};
	  Q.push(file);
	  value = expr;
	  Q.push(value);
	  R^{+};
	}
		void E^{-}() {
		  R^{-};
		  Q.push^{-}();
		  Q.push^{-}();
		  P^{-};
		}
			void E^{*}() {
			  P^{*};
			  File f = Q.front();
			  Q.pop();
			  f.output(Q.front());
			  Q.pop();
			  R^{*};
			}

- Both `file` and the value of `expr` must be pushed into the stack *by value*
- If `expr` evaluates to an array, the entire array must be pushed onto the stack
- If `expr` evaluates to a struct, or class type, fully deep copies are required using an appropriate, perhaps non-default copy constructor.

Note: **`expr`** is a side-effect free expression. When we push data onto the queue **`Q`** for processing at commit time, we must push *values,* not expressions to be evaluated.



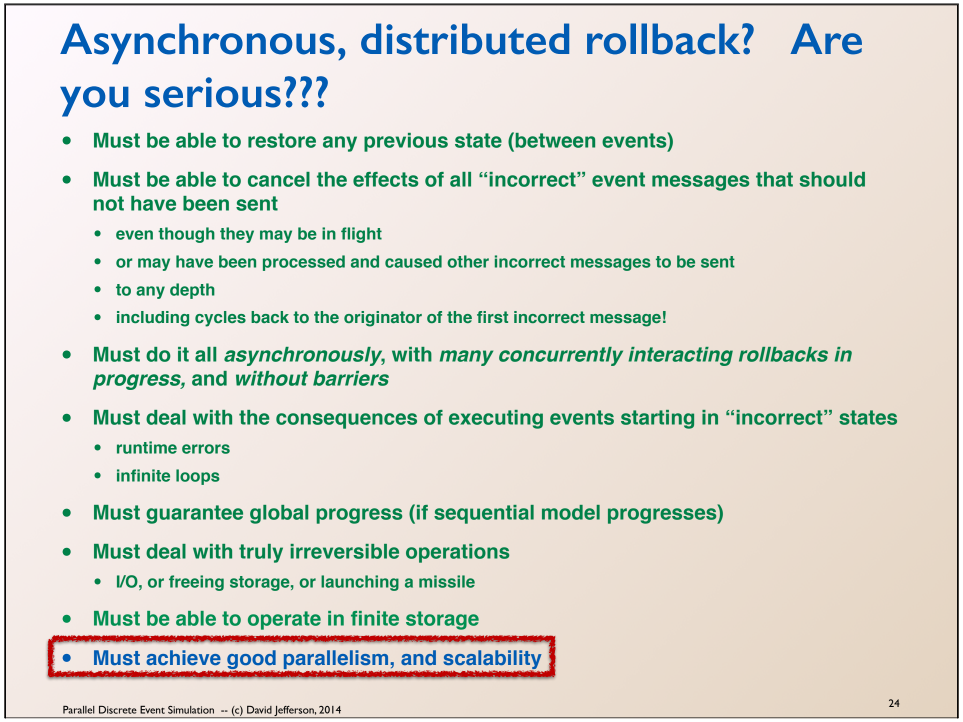
A reasonable first strategy for generating reverse code

- Consider each variable or field `v` in the object's state separately
- If `v` is not modified by `E()`, do nothing to save or restore it
- If `v`'s initial value can be regenerated from the final state produced by `E()`, generate code in $E^{-}()$ to do that using path-oriented, regenerative methods (yielding "perfect reversibility" for that variable).
- If `v` is a scalar (including pointers code)
  - if modified *many times* in `E()`, do up front state saving and restoration
  - if modified *only once* in `E()`, do incremental state saving and restoration
  - if cannot be sure statically, try to determine the places where it is modified the *first* time, and put saving and restoration code at those places
  - otherwise do incremental state saving
- If `v` is an array or collection
  - if most of the elements are modified in `E()`, do up front state saving and restoration of the entire collection
  - if only a few of the elements are changed, do incremental state saving on an element-by-element basis
  - if it cannot be statically determined which, consider programmer pragmas, or dynamic instrumentation to measure the number of elements affected
- If `v` is a class variable
  - Establish proper constructor, copy constructor, destructor and `operator =` functions for the class
  - Generate reverses for those functions and generate appropriate code in $E^{+}()$, $E^{-}()$, and $E^{*}()$ using them

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

22

End Reprise



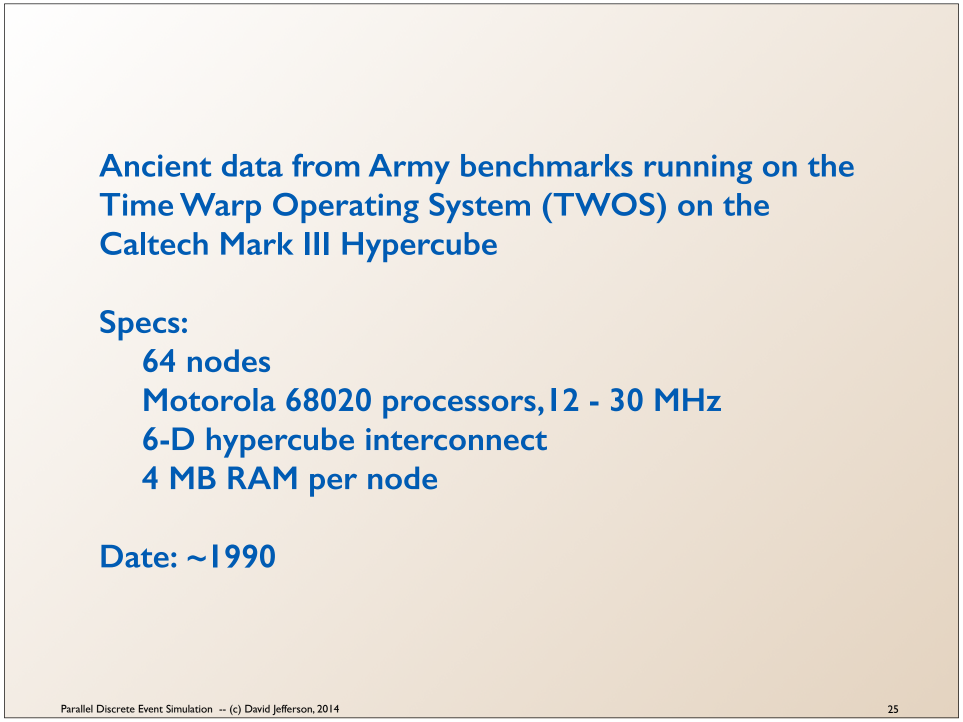
Asynchronous, distributed rollback? Are you serious???

- Must be able to restore any previous state (between events)
- Must be able to cancel the effects of all “incorrect” event messages that should not have been sent
  - even though they may be in flight
  - or may have been processed and caused other incorrect messages to be sent
  - to any depth
  - including cycles back to the originator of the first incorrect message!
- Must do it all *asynchronously*, with *many concurrently interacting rollbacks in progress,* and *without barriers*
- Must deal with the consequences of executing events starting in “incorrect” states
  - runtime errors
  - infinite loops
- Must guarantee global progress (if sequential model progresses)
- Must deal with truly irreversible operations
  - I/O, or freeing storage, or launching a missile
- Must be able to operate in finite storage
- Must achieve good parallelism, and scalability

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

24

At this point in the course we have demonstrated the Optimistic (rollback-based_ methods have all of the good properties shown in green. But we have not yet demonstrated that good performance can be achieved at large scale. We will be doing that in the next few dozen slides.

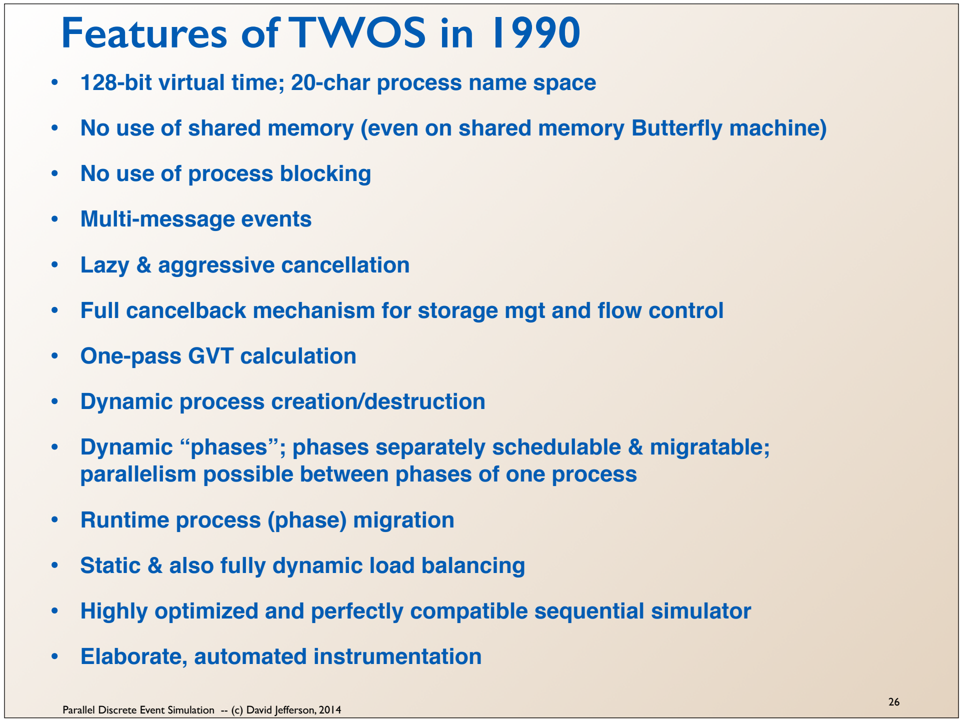


Ancient data from Army benchmarks running on the Time Warp Operating System (TWOS) on the Caltech Mark III Hypercube

Specs:

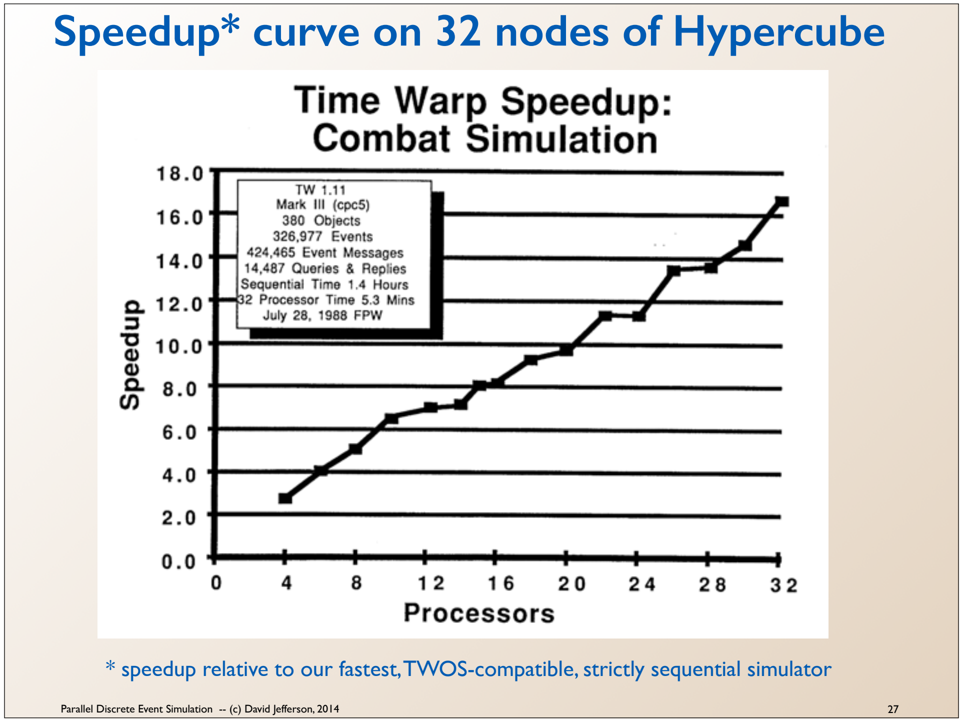
- 64 nodes
- Motorola 68020 processors, 12 - 30 MHz
- 6-D hypercube interconnect
- 4 MB RAM per node

Date: ~1990



Features of TWOS in 1990

- 128-bit virtual time; 20-char process name space
- No use of shared memory (even on shared memory Butterfly machine)
- No use of process blocking
- Multi-message events
- Lazy & aggressive cancellation
- Full cancelback mechanism for storage mgt and flow control
- One-pass GVT calculation
- Dynamic process creation/destruction
- Dynamic "phases"; phases separately schedulable & migratable; parallelism possible between phases of one process
- Runtime process (phase) migration
- Static & also fully dynamic load balancing
- Highly optimized and perfectly compatible sequential simulator
- Elaborate, automated instrumentation



Speedup* curve on 32 nodes of Hypercube

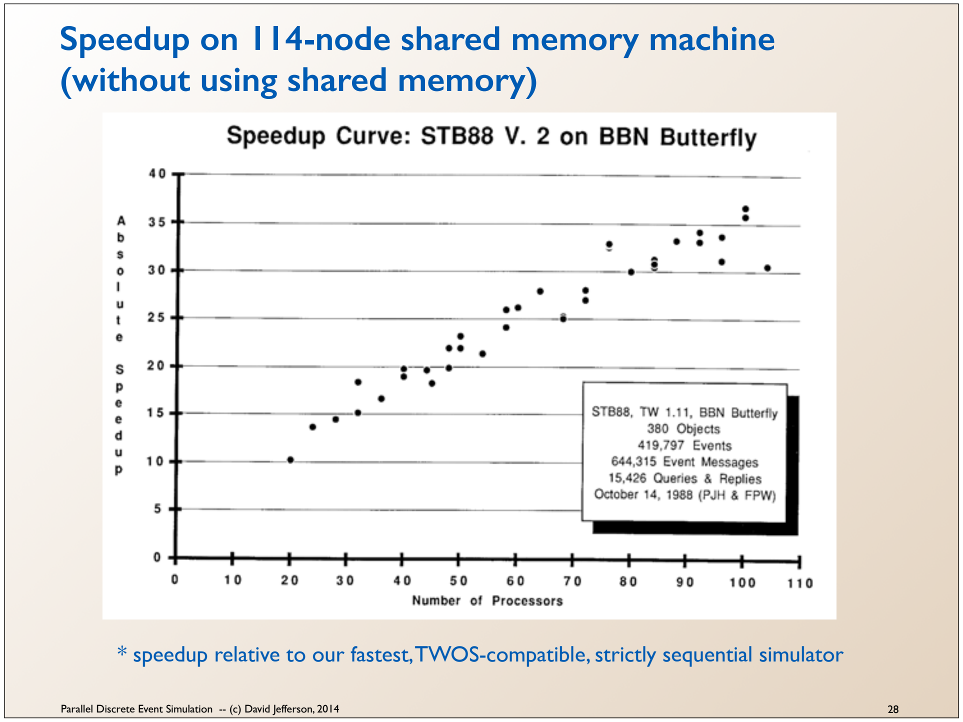
* speedup relative to our fastest, TWOS-compatible, strictly sequential simulator

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 27

Combat simulation -- mini JCATS
JPL Mark III Hypercube, 32 processors, Motorola 68020, 4 MB RAM per node
July 1988

This is a strong scaling speedup curve, and the speedup is compared to an optimized sequential simulator, not against a one-node execution of Time Warp.

Linear speedup up to 32 nodes, almost 17x speedup



Speedup on 114-node shared memory machine (without using shared memory)

Speedup Curve: STB88 V. 2 on BBN Butterfly

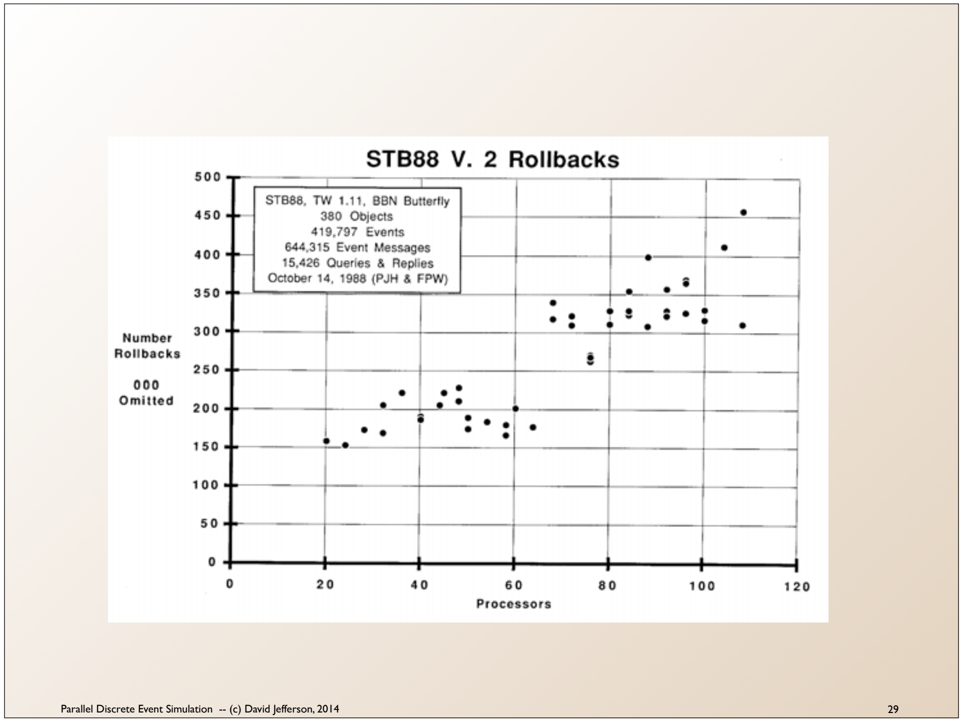
STB88, TW 1.11, BBN Butterfly
380 Objects
419,797 Events
644,315 Event Messages
15,426 Queries & Replies
October 14, 1988 (PJH & FPW)

* speedup relative to our fastest, TWOS-compatible, strictly sequential simulator

A few months later in October. Same combat model (modified since July), same size, run for a little more time.
BBN Butterfly (shared memory, NUMA), 108 processors

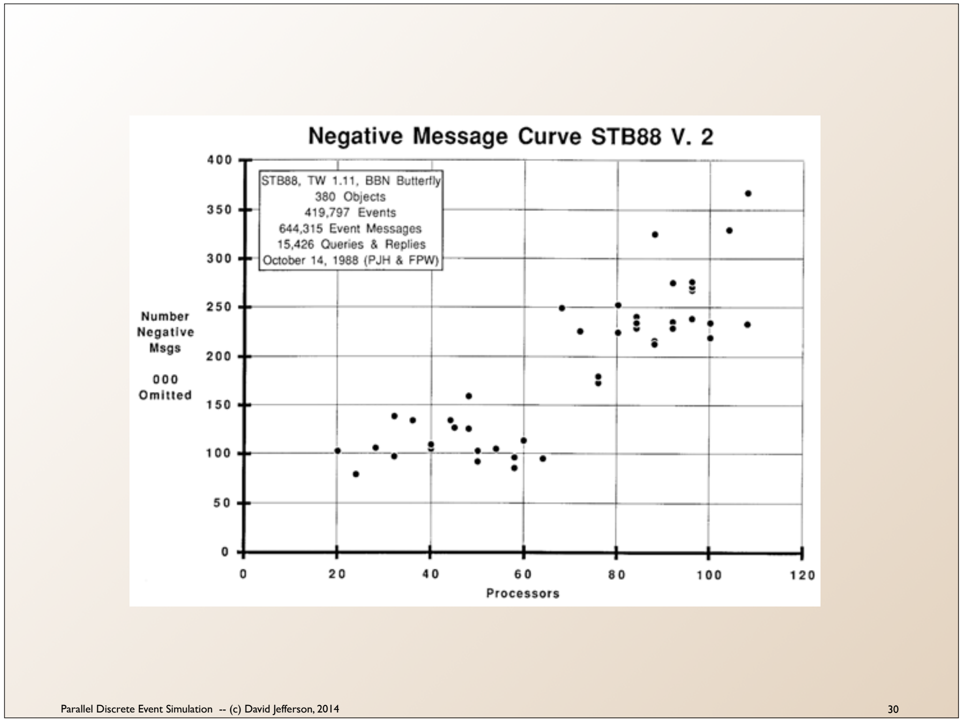
Linear speedup to about 80 nodes, then probably some flattening of performance.

Note performance variations in multiple runs of exact same configuration.



Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 29

Same runs as on the previous slide with number of rollbacks plotted as function of number of processors.
Note that as more processors are used (strong scaling) the number of rollbacks trends upward!
But the previous graph showed that we we still getting good scaling results in spite of that!
This shows that it is not true that we should have as a tuning goal of TW the minimization of the number of rollbacks.
Most these rollbacks (hundreds of thousands of them) occur *off the critical path*, so they do not slow down the progress of the simulation.

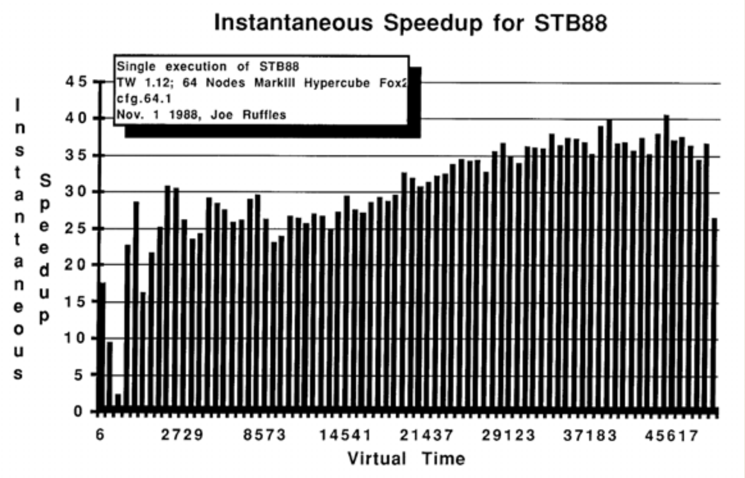


Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

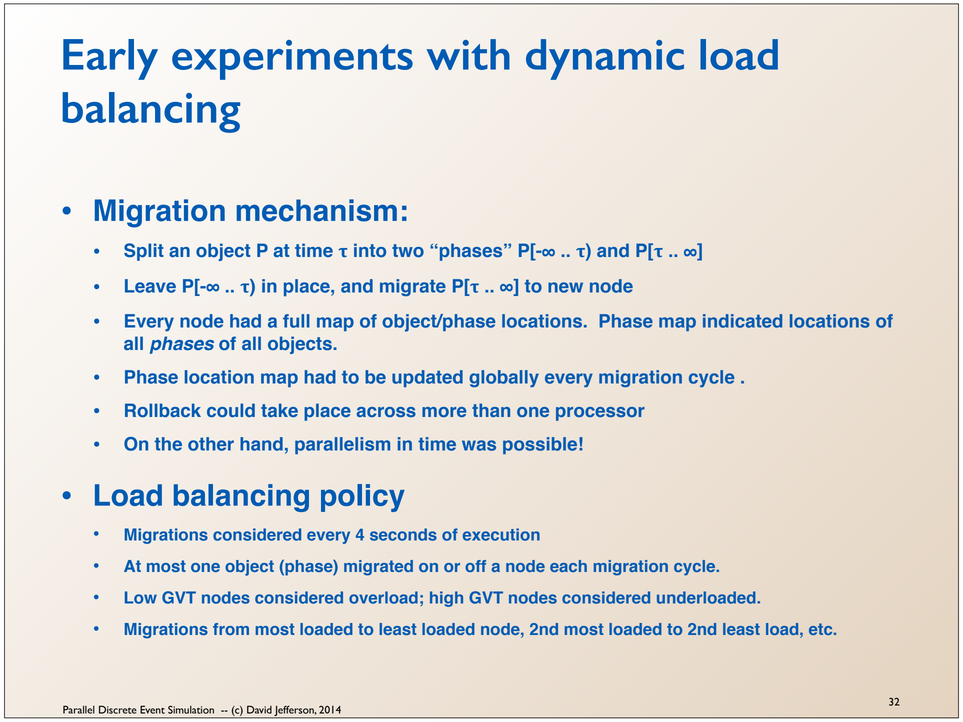
30

Again, same runs, this time with number of antimessages transmitted plotted as a function of number of nodes.
Again, number of antimessages increases, even as speed increases (see two slides previous).
Note that antimessages is approximately proportional to the number of rollbacks from the previous slide.

“Instantaneous” speedup

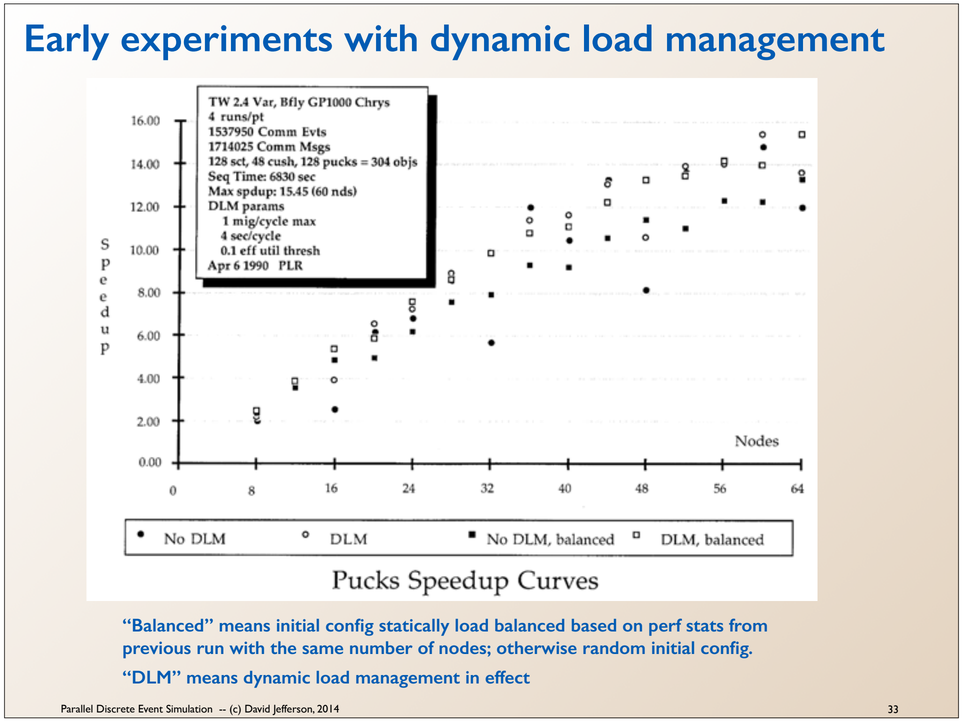


Short term speedup calculated at many virtual times during the simulation.



Early experiments with dynamic load balancing

- **Migration mechanism:**
  - Split an object P at time τ into two “phases” $P[-\infty .. \tau)$ and $P[\tau .. \infty]$
  - Leave $P[-\infty .. \tau)$ in place, and migrate $P[\tau .. \infty]$ to new node
  - Every node had a full map of object/phase locations. Phase map indicated locations of all *phases* of all objects.
  - Phase location map had to be updated globally every migration cycle .
  - Rollback could take place across more than one processor
  - On the other hand, parallelism in time was possible!
- **Load balancing policy**
  - Migrations considered every 4 seconds of execution
  - At most one object (phase) migrated on or off a node each migration cycle.
  - Low GVT nodes considered overload; high GVT nodes considered underloaded.
  - Migrations from most loaded to least loaded node, 2nd most loaded to 2nd least load, etc.



Early experiments with dynamic load management

Pucks Speedup Curves

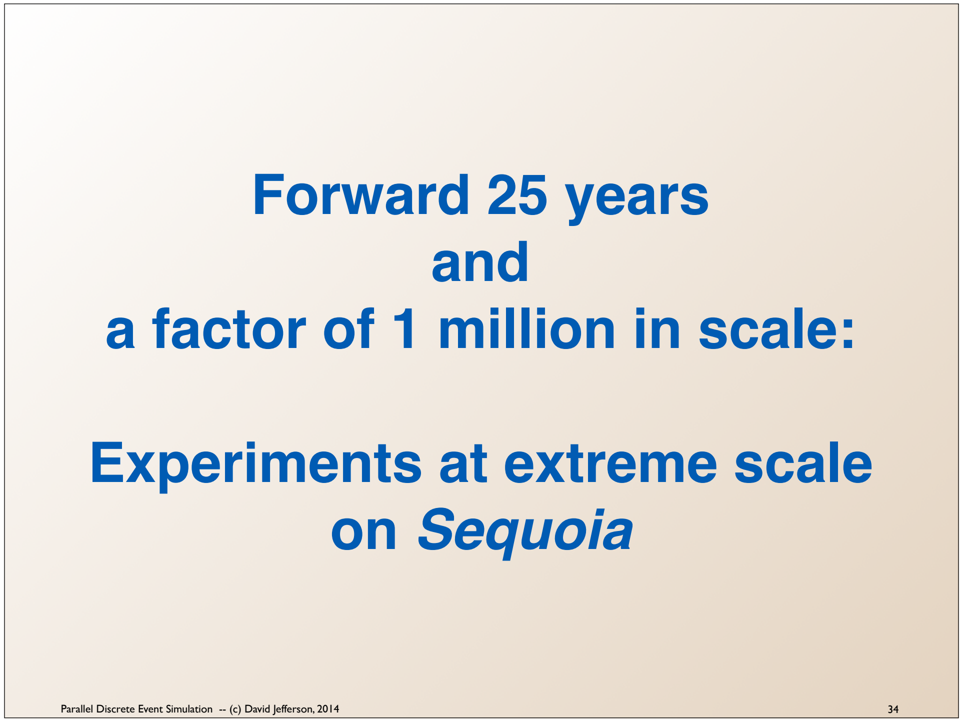
"Balanced" means initial config statically load balanced based on perf stats from previous run with the same number of nodes; otherwise random initial config.

"DLM" means dynamic load management in effect

Different model: "Pucks", a bunch of frictionless hockey pucks colliding with the sides of the rink and with each other

Dynamic load management based on "effective utilization" concept

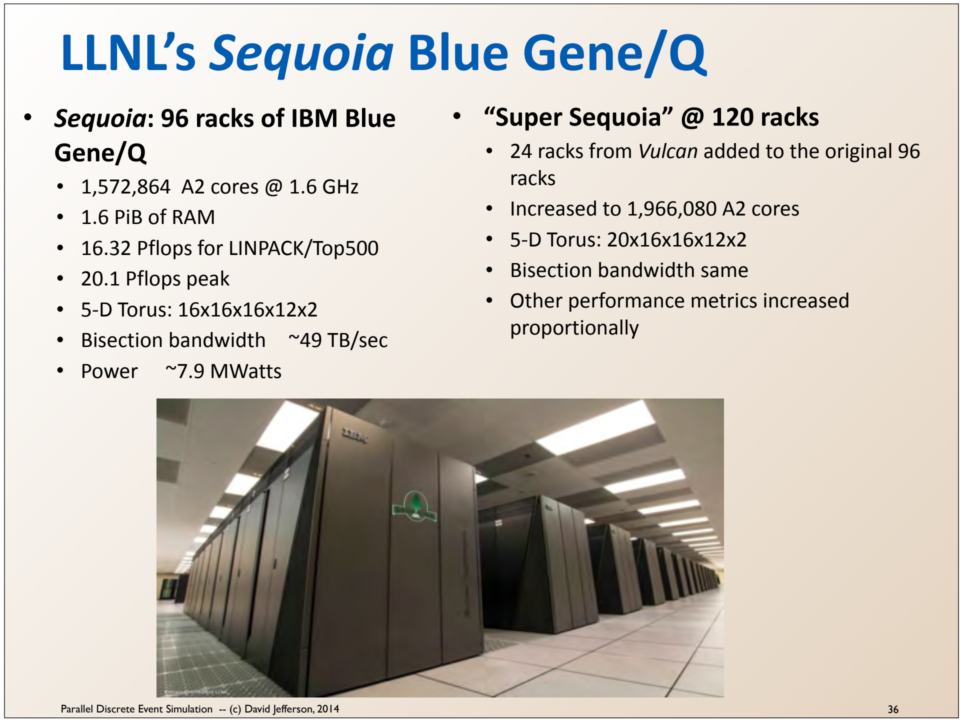
No DLM:	Random initial arrangement of Objects on nodes; no dynamic load management
DLM:	Random initial arrangement of Objects on nodes, and with dynamic load management
No DLM, balanced:	Statically balanced initial arrangement of nodes, but no DLM
DLM, balanced:	Statically balanced initial arrangement of nodes, with DLM in addition



Forward 25 years
and
a factor of 1 million in scale:

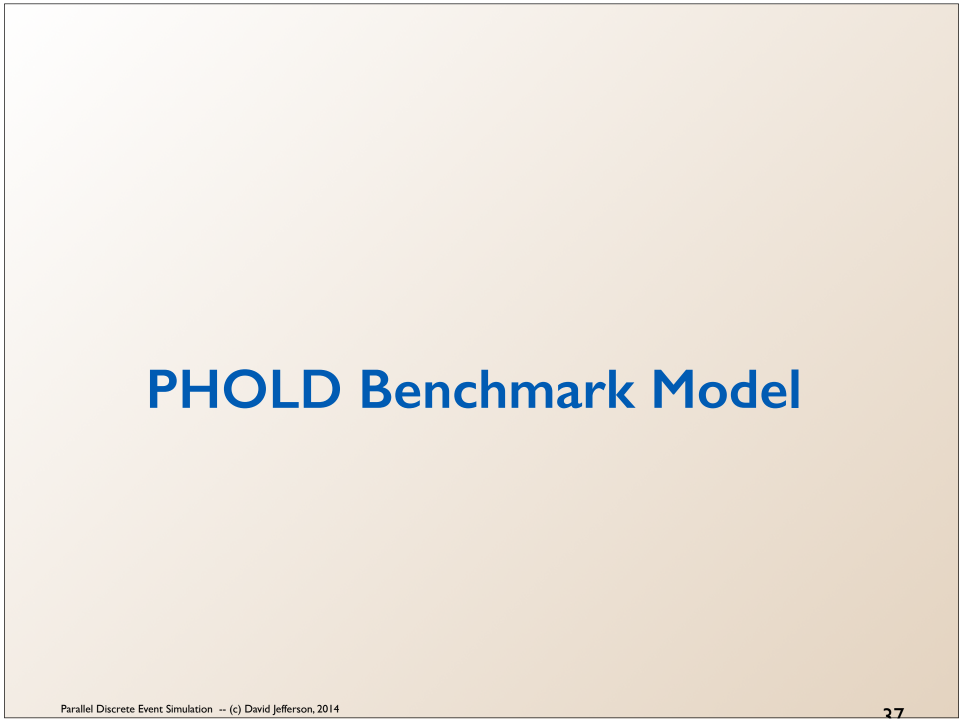
Experiments at extreme scale
on *Sequoia*

Sequoia Blue Gene/Q



LLNL's *Sequoia* Blue Gene/Q

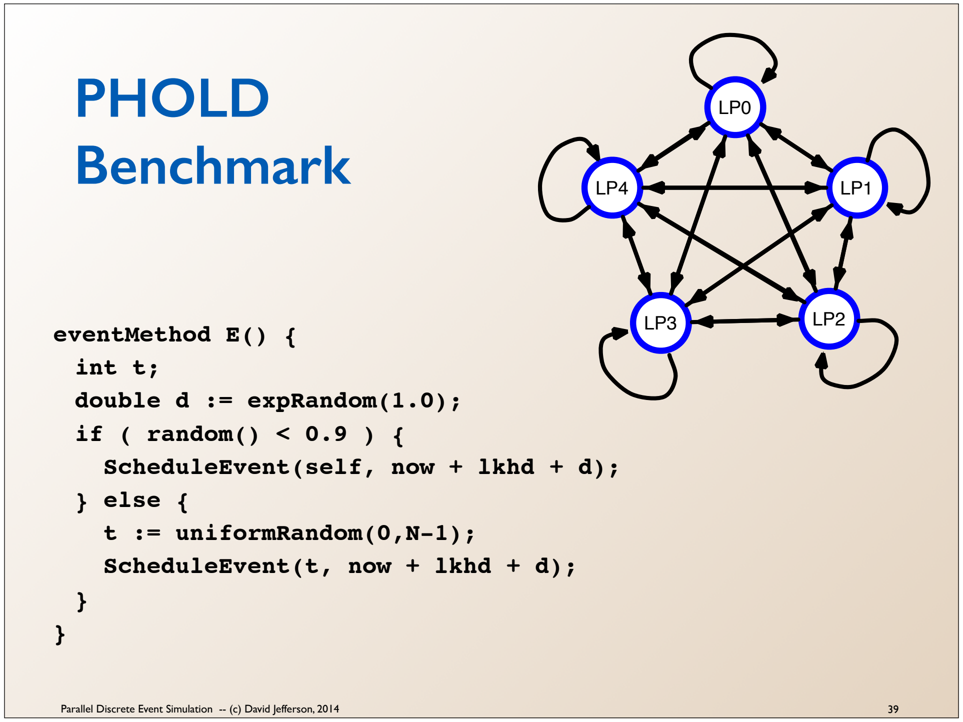
- ***Sequoia*: 96 racks of IBM Blue Gene/Q**
  - 1,572,864 A2 cores @ 1.6 GHz
  - 1.6 PiB of RAM
  - 16.32 Pflops for LINPACK/Top500
  - 20.1 Pflops peak
  - 5-D Torus: 16x16x16x12x2
  - Bisection bandwidth ~49 TB/sec
  - Power ~7.9 MWatts
- **"Super Sequoia" @ 120 racks**
  - 24 racks from *Vulcan* added to the original 96 racks
  - Increased to 1,966,080 A2 cores
  - 5-D Torus: 20x16x16x12x2
  - Bisection bandwidth same
  - Other performance metrics increased proportionally



PHOLD Benchmark Model

PHOLD Benchmark model

- Defined by Richard Fujimoto
- *De facto* standard for comparing parallel discrete event simulators to one another
  - In use for >20 years
- Artificial model representing challenging simulation dynamics
  - Very fine-grained events
  - No communication locality
  - Symmetric and (statistically) load balanced, steady state equilibrium
  - Almost all simulator overhead
- We did an exact *strong scaling study*:
  - Same number of LPs, same random seeds, same events, and same exact results at all scales
  - Only variable is the number of nodes (racks) of Sequoia used



PHOLD Benchmark

LP0
LP1
LP2
LP3
LP4

```
eventMethod E() {
  int t;
  double d := expRandom(1.0);
  if ( random() < 0.9 ) {
    ScheduleEvent(self, now + lkhd + d);
  } else {
    t := uniformRandom(0,N-1);
    ScheduleEvent(t, now + lkhd + d);
  }
}
```

N = number of LPs in the simulation

self = the name of the LP executing

uniformRandom(m,n) draws an integer in the inclusive range **[m..n]**

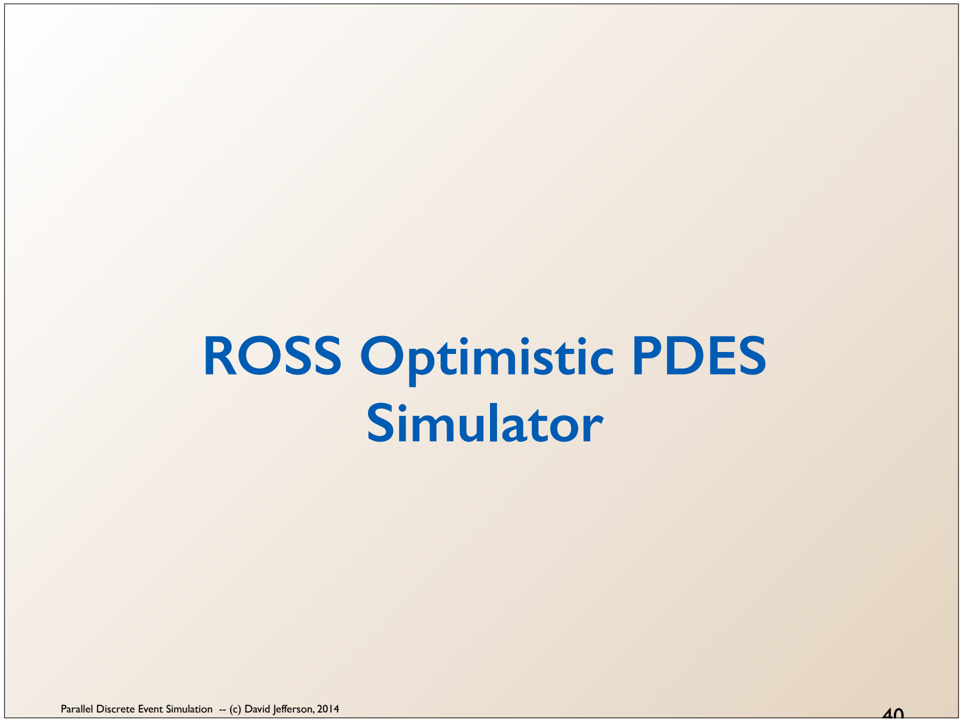
expRandom(λ) draws a random real number exponentially distributed with mean **λ**

random() draws a real number uniformly distributed in the range [0.0 .. 1.0)

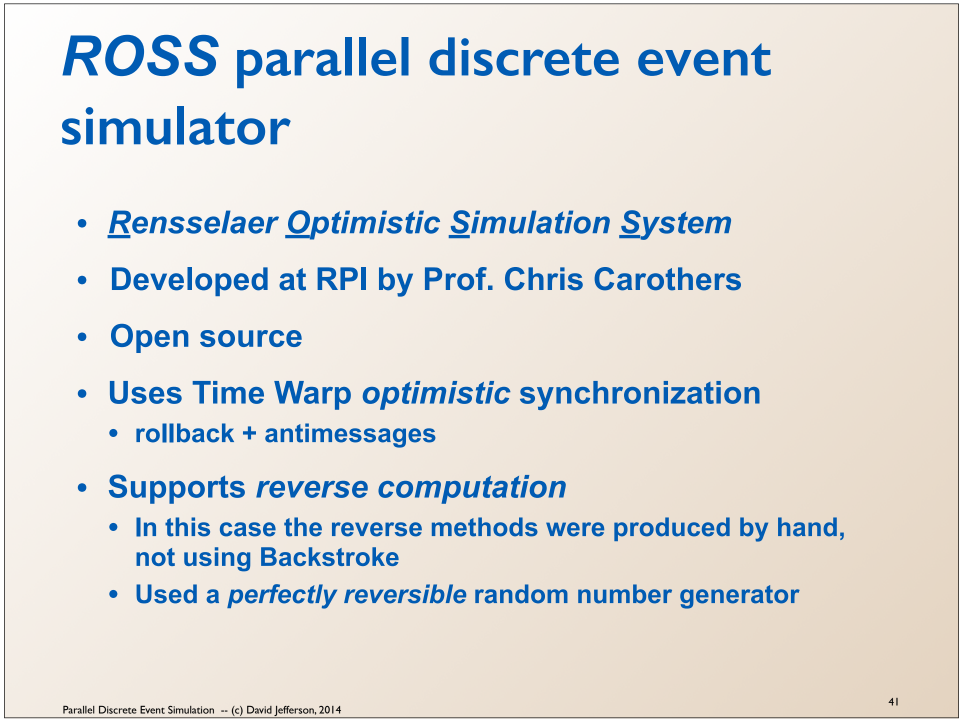
now is the current simulation time in the calling object

lkhd is the lookahead value.

> **lkhd** can be 0 for optimistic simulations, but is generally made nonzero so that comparisons can be made with conservative algorithms (which need nonzero lookahead). It was 0.1 throughout this study.



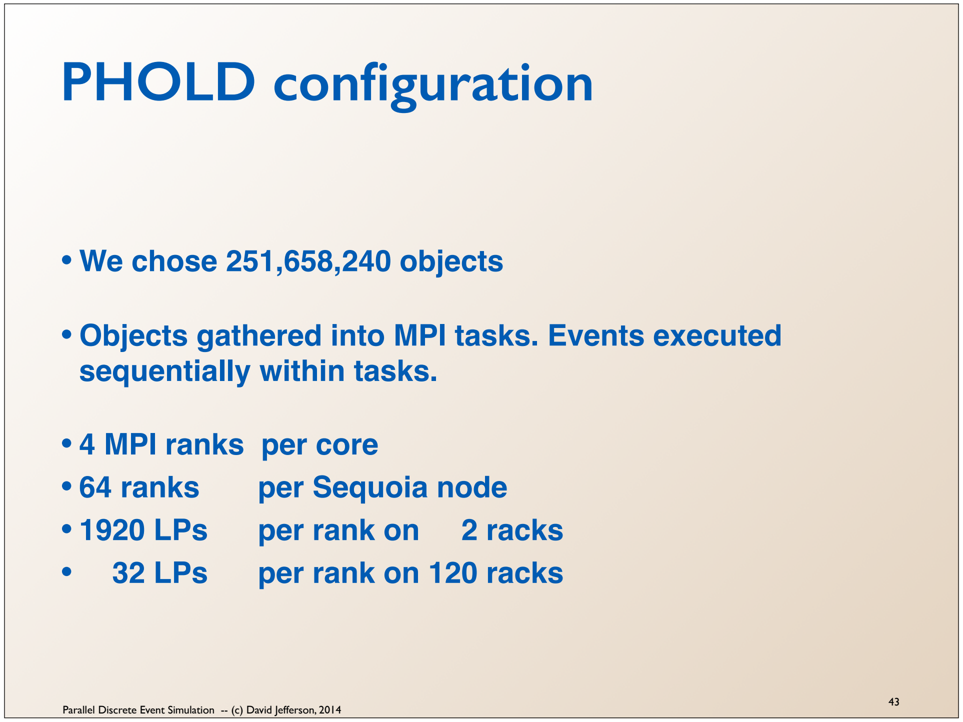
ROSS Optimistic PDES Simulator



ROSS parallel discrete event simulator

- *Rensselaer Optimistic Simulation System*
- Developed at RPI by Prof. Chris Carothers
- Open source
- Uses Time Warp *optimistic* synchronization
  - rollback + antimessages
- Supports *reverse computation*
  - In this case the reverse methods were produced by hand, not using Backstroke
  - Used a *perfectly reversible* random number generator

New “World Records” for Speed and Scale

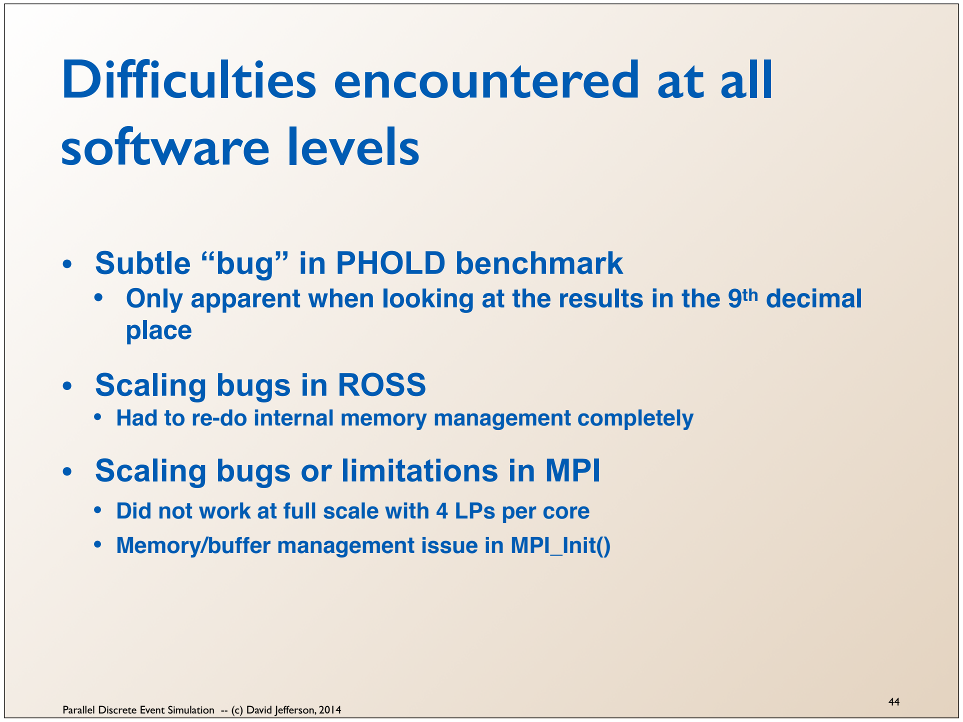


PHOLD configuration

- We chose 251,658,240 objects

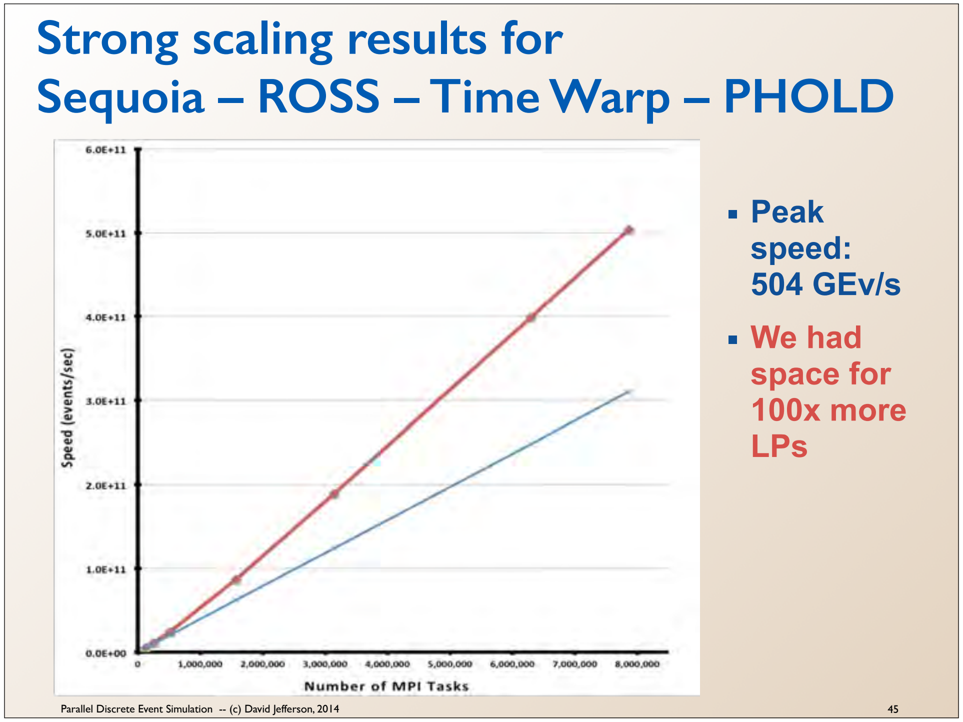
- Objects gathered into MPI tasks. Events executed sequentially within tasks.

- 4 MPI ranks per core
- 64 ranks per Sequoia node
- 1920 LPs per rank on 2 racks
- 32 LPs per rank on 120 racks



Difficulties encountered at all software levels

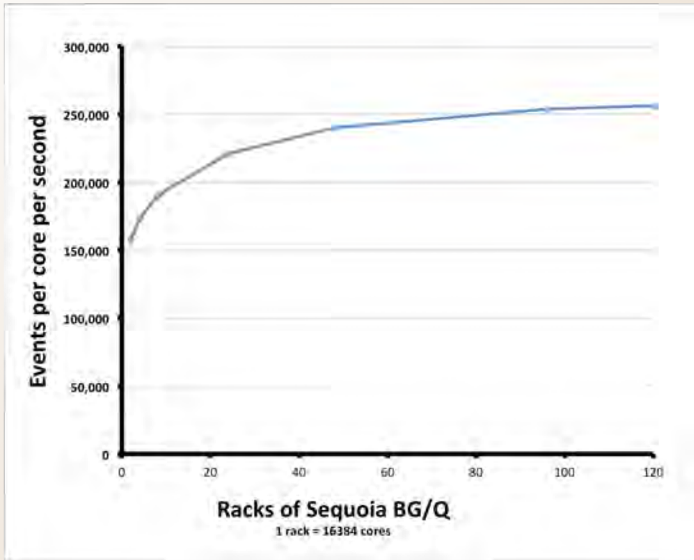
- Subtle “bug” in PHOLD benchmark
  - Only apparent when looking at the results in the 9th decimal place
- Scaling bugs in ROSS
  - Had to re-do internal memory management completely
- Scaling bugs or limitations in MPI
  - Did not work at full scale with 4 LPs per core
  - Memory/buffer management issue in MPI_Init()



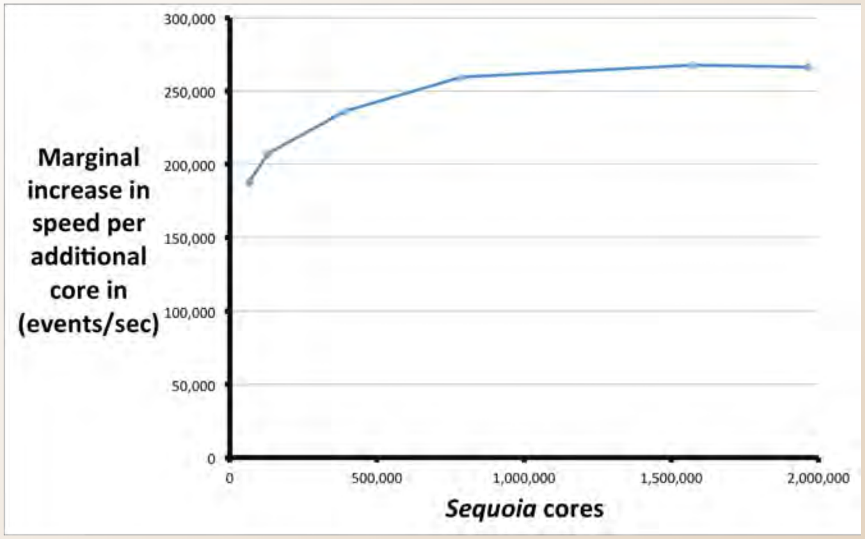
Strong scaling results for Sequoia – ROSS – Time Warp – PHOLD

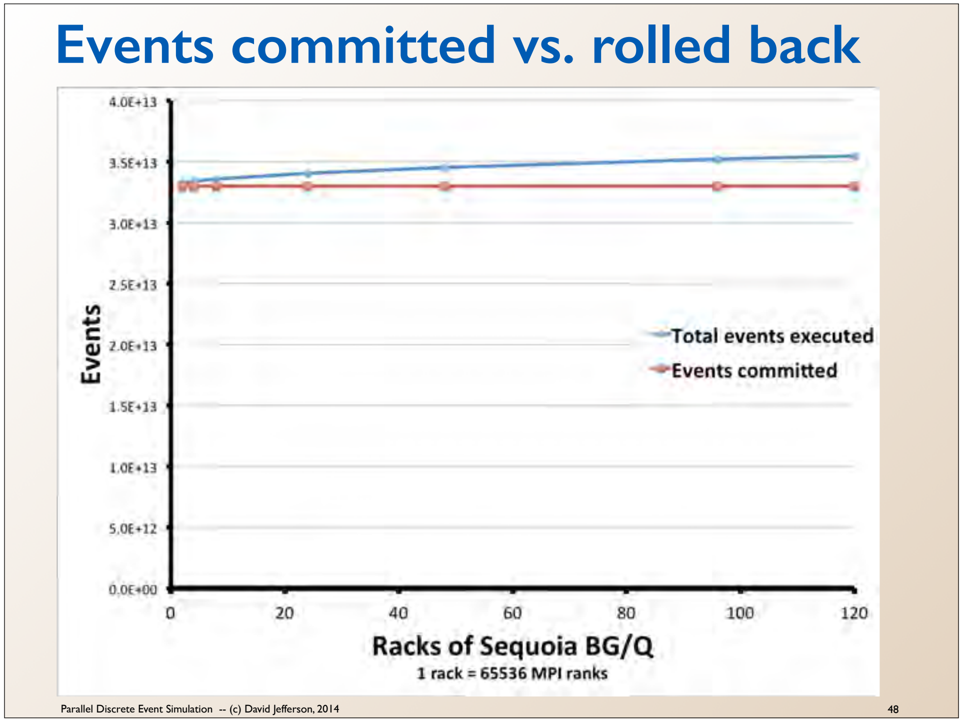
- **Peak speed: 504 GEv/s**
- **We had space for 100x more LPs**

Committed Events per Core-Sec as a Function of Scale



Diminishing returns as more cores added

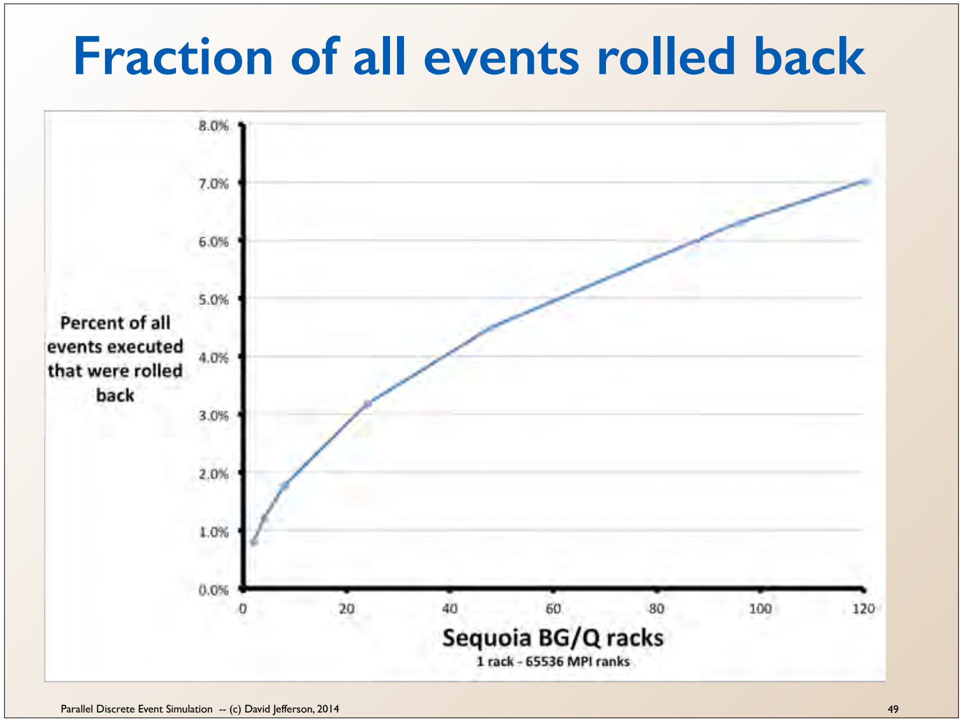




Events committed vs. rolled back

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 48

The difference between the two curves is the number of events rolled back.
Events committed is exactly constant because this is a strong scaling study, and so we are committing exactly the same events at all scales.

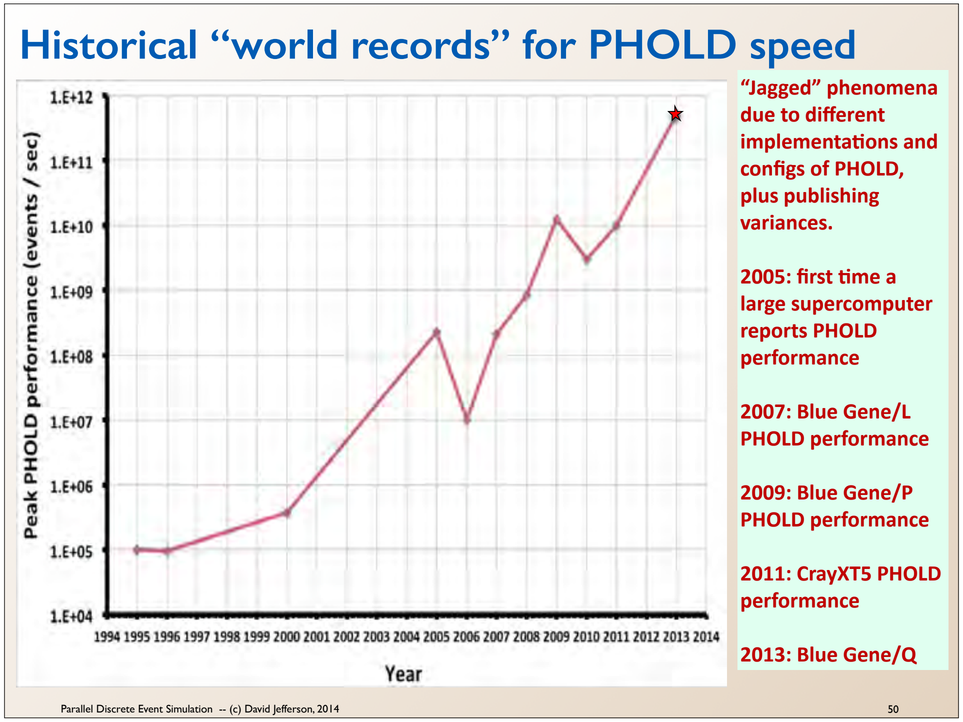


Fraction of all events rolled back

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

49

Of all of the events executed (committed + rolled back) this is the fraction rolled back, i.e. rolledback/(rolledback+committed).



Historical “world records” for PHOLD speed

“Jagged” phenomena due to different implementations and configs of PHOLD, plus publishing variances.

2005: first time a large supercomputer reports PHOLD performance

2007: Blue Gene/L PHOLD performance

2009: Blue Gene/P PHOLD performance

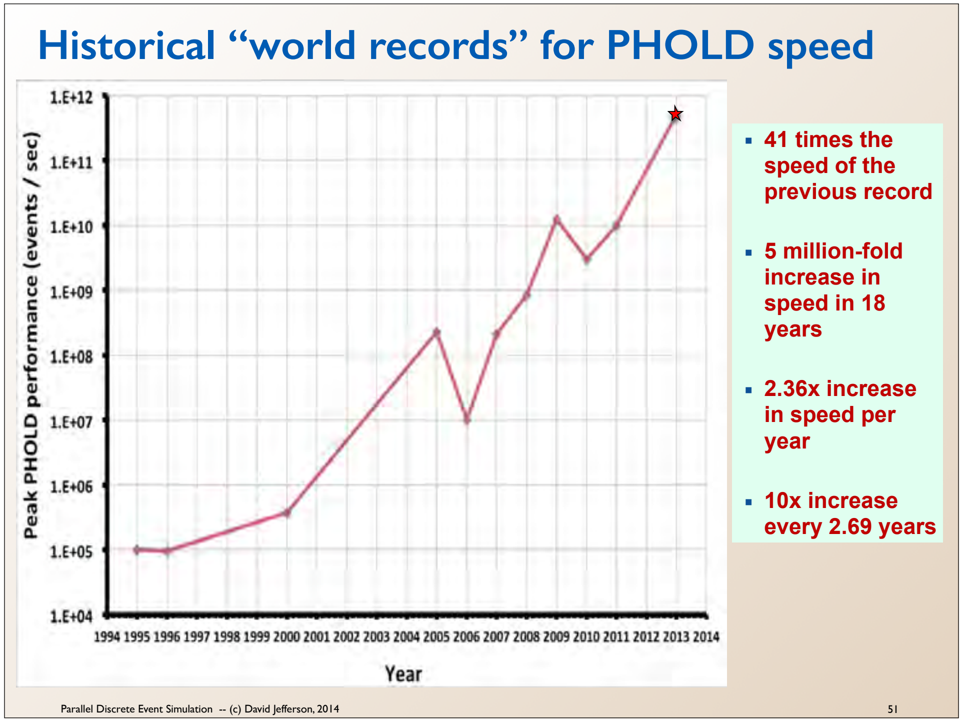
2011: CrayXT5 PHOLD performance

2013: Blue Gene/Q

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 50

This logarithmic plot is the historical record of PHOLD benchmark speeds published. Each point plots the highest speed published on this benchmark in the year, regardless of platform. In most cases the authors just configured the fastest PHOLD instance they could on the biggest machine they had access to and reported the (among much other detail) the resulting speed in events per second.

The “jagged” look of the graph has a mixture of causes: The implementations, parameters, and configurations of PHOLD were not identical from year to year. And some years no one beat the previous records, but since they (quite appropriately) published their results anyway, we plotted them.



Historical “world records” for PHOLD speed

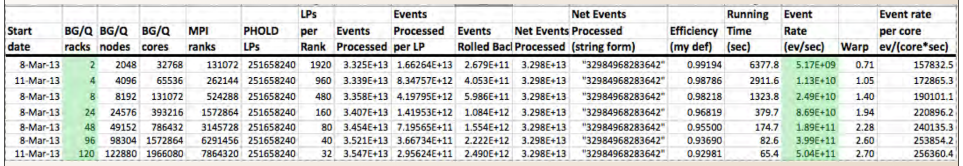
- 41 times the speed of the previous record
- 5 million-fold increase in speed in 18 years
- 2.36x increase in speed per year
- 10x increase every 2.69 years

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 51

This logarithmic plot is the historical record of PHOLD benchmark speeds published. Each point plots the highest speed published on this benchmark in the year, regardless of platform. In most cases the authors just configured the fastest PHOLD instance they could on the biggest machine they had access to and reported the (among much other detail) the resulting speed in events per second.

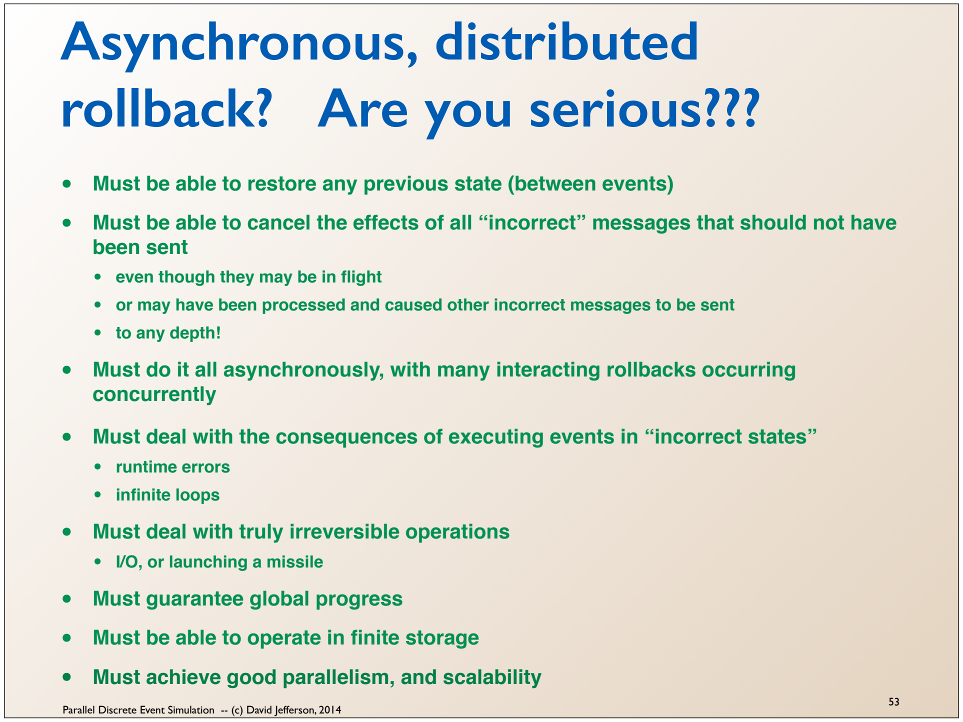
The “jagged” look of the graph has a mixture of causes: The implementations, parameters, and configurations of PHOLD were not identical from year to year. And some years no one beat the previous records, but since they (quite appropriately) published their results anyway, we plotted them.

SEQUOIA / ROSS / Time Warp / PHOLD Performance Data



						LPs		Events			Net Events		Running	Event		Event rate
Start	BG/Q	BG/Q	BG/Q	MPI	PHOLD	per	Events	Processed	Events	Net Events	Processed	Efficiency	Time	Rate		per core
date	racks	nodes	cores	ranks	LPs	Rank	Processed	per LP	Rolled Bacl	Processed	(string form)	(my def)	(sec)	(ev/sec)	Warp	ev/(core*sec)
8-Mar-13	2	2048	32768	131072	251658240	1920	3.325E+13	1.66264E+13	2.679E+11	3.298E+13	"32984968283642"	0.99194	6377.8	5.17E+09	0.71	157832.5
11-Mar-13	4	4096	65536	262144	251658240	960	3.339E+13	8.34757E+12	4.053E+11	3.298E+13	"32984968283642"	0.98786	2911.6	1.13E+10	1.05	172865.3
8-Mar-13	8	8192	131072	524288	251658240	480	3.358E+13	4.19795E+12	5.986E+11	3.298E+13	"32984968283642"	0.98218	1323.8	2.49E+10	1.40	190101.1
8-Mar-13	24	24576	393216	1572864	251658240	160	3.407E+13	1.41953E+12	1.084E+12	3.298E+13	"32984968283642"	0.96819	379.7	8.69E+10	1.94	220896.2
8-Mar-13	48	49152	786432	3145728	251658240	80	3.454E+13	7.19565E+11	1.554E+12	3.298E+13	"32984968283642"	0.95500	174.7	1.89E+11	2.28	240135.3
8-Mar-13	96	98304	1572864	6291456	251658240	40	3.521E+13	3.66734E+11	2.222E+12	3.298E+13	"32984968283642"	0.93690	82.6	3.99E+11	2.60	253854.2
11-Mar-13	120	122880	1966080	7864320	251658240	32	3.547E+13	2.95624E+11	2.490E+12	3.298E+13	"32984968283642"	0.92981	65.4	5.04E+11	2.70	256360.4

Part of the original data collect for our Sequoia / ROSS/ Time Warp / PHOLD study.

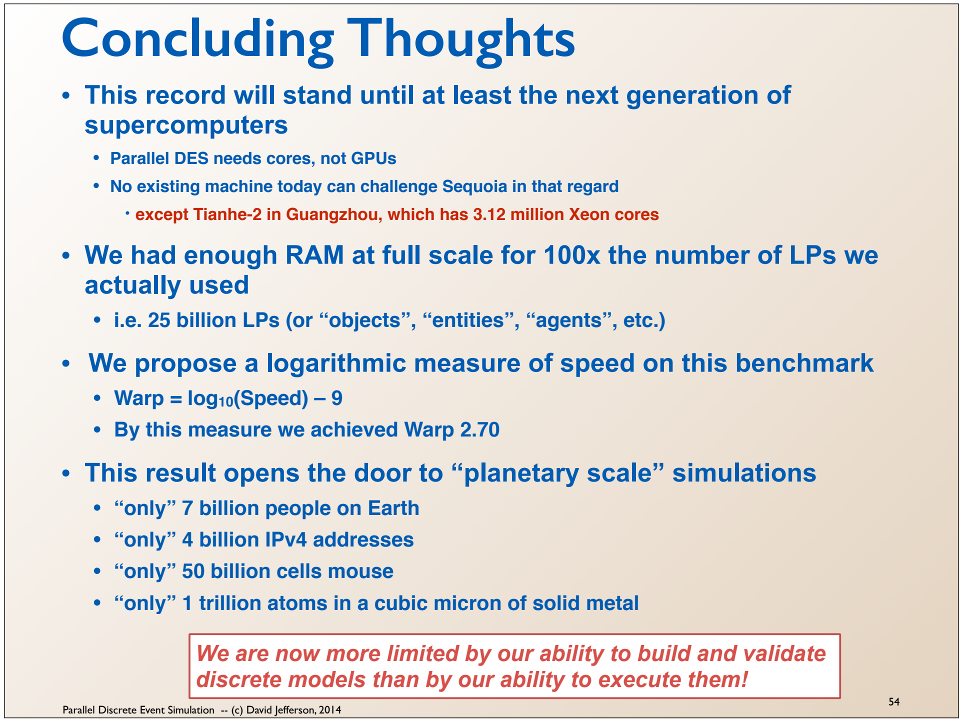


Asynchronous, distributed rollback? Are you serious???

- Must be able to restore any previous state (between events)
- Must be able to cancel the effects of all “incorrect” messages that should not have been sent
  - even though they may be in flight
  - or may have been processed and caused other incorrect messages to be sent
  - to any depth!
- Must do it all asynchronously, with many interacting rollbacks occurring concurrently
- Must deal with the consequences of executing events in “incorrect states”
  - runtime errors
  - infinite loops
- Must deal with truly irreversible operations
  - I/O, or launching a missile
- Must guarantee global progress
- Must be able to operate in finite storage
- Must achieve good parallelism, and scalability

This shows the list of challenges we have to overcome for the Time Warp algorithm, or any optimistic PDES algorithm, to be practical.

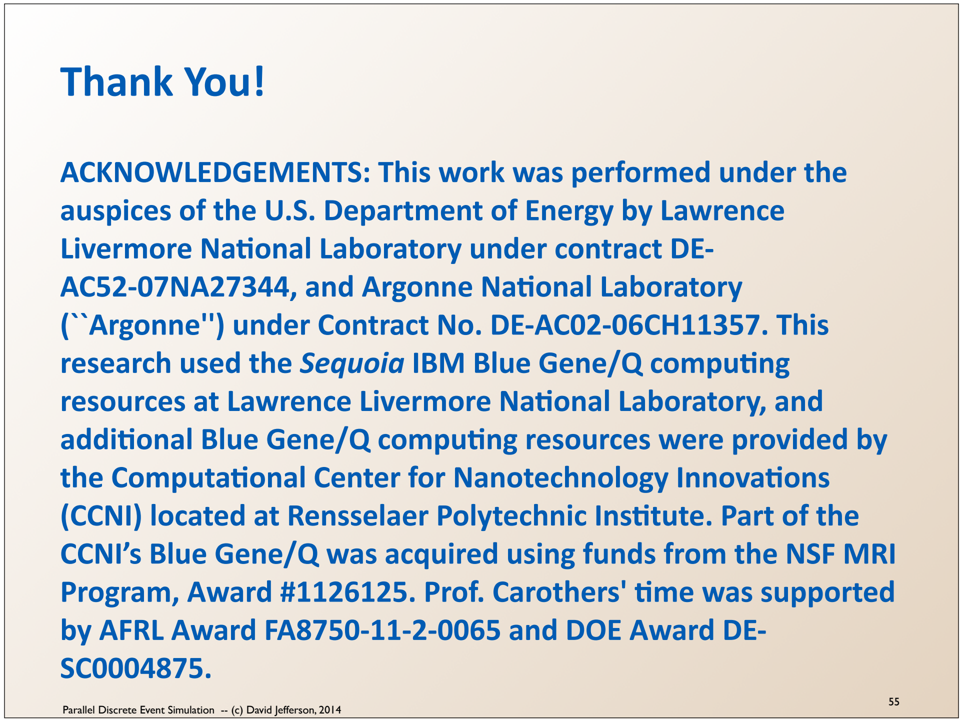
I have now completed the list.



Concluding Thoughts

- This record will stand until at least the next generation of supercomputers
  - Parallel DES needs cores, not GPUs
  - No existing machine today can challenge Sequoia in that regard
    - except Tianhe-2 in Guangzhou, which has 3.12 million Xeon cores
- We had enough RAM at full scale for 100x the number of LPs we actually used
  - i.e. 25 billion LPs (or “objects”, “entities”, “agents”, etc.)
- We propose a logarithmic measure of speed on this benchmark
  - Warp = $\log_{10}$(Speed) – 9
  - By this measure we achieved Warp 2.70
- This result opens the door to “planetary scale” simulations
  - “only” 7 billion people on Earth
  - “only” 4 billion IPv4 addresses
  - “only” 50 billion cells mouse
  - “only” 1 trillion atoms in a cubic micron of solid metal

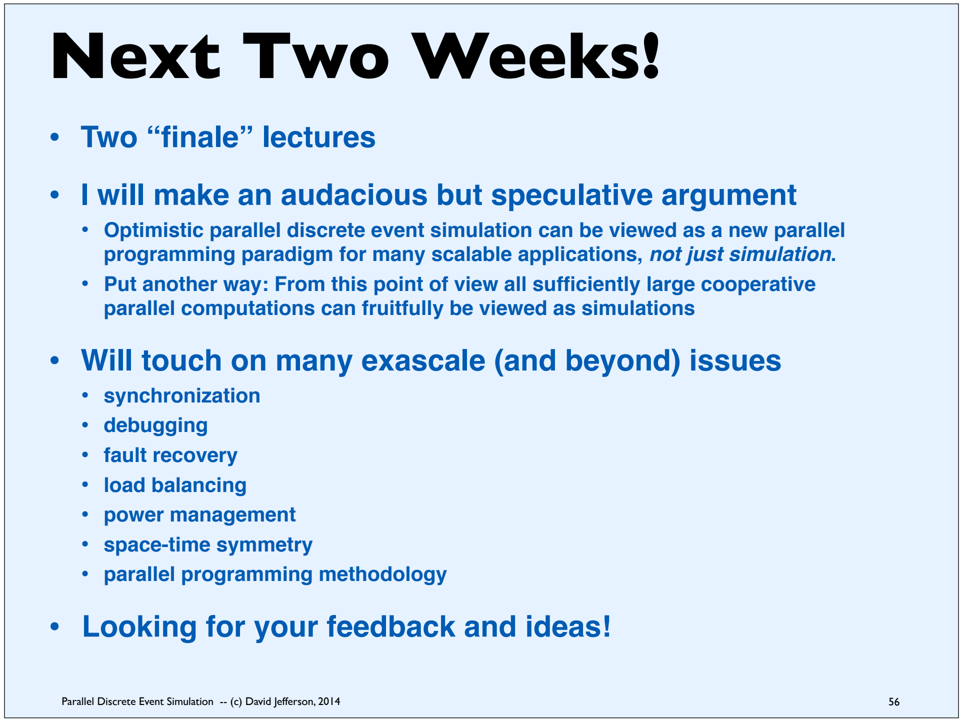
We are now more limited by our ability to build and validate discrete models than by our ability to execute them!



Thank You!

ACKNOWLEDGEMENTS: This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344, and Argonne National Laboratory (``Argonne'') under Contract No. DE-AC02-06CH11357. This research used the *Sequoia* IBM Blue Gene/Q computing resources at Lawrence Livermore National Laboratory, and additional Blue Gene/Q computing resources were provided by the Computational Center for Nanotechnology Innovations (CCNI) located at Rensselaer Polytechnic Institute. Part of the CCNI's Blue Gene/Q was acquired using funds from the NSF MRI Program, Award #1126125. Prof. Carothers' time was supported by AFRL Award FA8750-11-2-0065 and DOE Award DE-SC0004875.

A word from our sponsors.



Next Two Weeks!

- Two “finale” lectures
- I will make an audacious but speculative argument
  - Optimistic parallel discrete event simulation can be viewed as a new parallel programming paradigm for many scalable applications, *not just simulation*.
  - Put another way: From this point of view all sufficiently large cooperative parallel computations can fruitfully be viewed as simulations
- Will touch on many exascale (and beyond) issues
  - synchronization
  - debugging
  - fault recovery
  - load balancing
  - power management
  - space-time symmetry
  - parallel programming methodology
- Looking for your feedback and ideas!

End