Reversible Computation in Optimistic Parallel Discrete Event Simulation

Workshop on Program Synthesis for Scientific Computing – Online (organized by ANL/LLNL)

CASC
Center for Applied Scientific Computing

Markus Schordan

Joint work with Tomas Oppelstrup, Michael Kirkedal Thomsen, Robert Glück, David R. Jefferson, Peter D. Barnes, Dan Quinlan, August 4-5, 2020

LLNL-PRES-812897

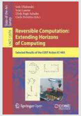
This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Lawrence Livermore National Laboratory

Reversible Computation

Paradigm Reversible Computation

- Extends traditional forwards-only mode of computation
- Computation can run backwards as easily as forwards
- Aims to



  - Deliver novel computing devices and software
  - Enhance existing systems by equipping them with reversibility
  - New book: *Reversible Computation - Extending Horizons of Computing*, Springer, 237 pages, 2020.

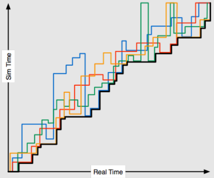
Applications

- Database transactions, fault detection/tolerance, debugging
- Parallel computing and synchronization (Optimistic PDES)
- Revolutionary reversible logic gates and circuits

Use Case for Reversibility

Optimistic Parallel Discrete Event Simulation (PDES)

- Time Warp Algorithm
  - Published 1985 TOPLAS, David Jefferson.
  - Optimistic algorithm
  - Key feature: distributed asynchronous rollback
    - Global virtual time increases monotonically
    - Allows to commit (="clean up") data stored for old events
    - Simulation can run arbitrarily long
  - World record 2013 (504 billion events per second (PHOLD benchmark)) at LLNL.



- Requires reversibility of event computation for rollback
  - When events are detected to be in conflict, the effect of previous messages must be reversed.
- Applications
  - Network, traffic, particle simulations, etc.

Reverse Computation - Computing backwards

Reverse Computation

Establishes a previous program state by computing backwards

Reversible and irreversible programs

- Reversible code
  - e.g. forward: `a=a+1;` reverse: `a=a-1;`
- Irreversible code: destroys information
  - e.g.: `a=a*a;`
- Transform irreversible code into reversible code:
  - add code to store information that is destroyed otherwise

Reversible Computation Approaches - Pros/Cons

Paradigm Reversible Computation - Software Reversibility

1. Reversible languages (Janus, CoreFun, ...)
   - Functions can be called to compute forwards or backwards
   - Programs written to be forward/backward deterministic
   - Forward: conditionals+assertions, Backward: assertions+conditionals
2. Reverse code generators (Reverse C Compiler)
   - Generates reverse C code from given C forward code
   - Requires additional memory only when information destroyed
   - In: Perumalla 2013, Introduction to Reversible Computing
3. Incremental checkpointing (C/C++ Backstroke)
   - Can be applied to C++ language (reversible assignment ops)
   - Templates, Virtual methods, exceptions, etc.
   - Always requires additional memory
   - Works with any data type (e.g. floating point types)

Crout Matrix Multiply in Janus vs. Backstroke

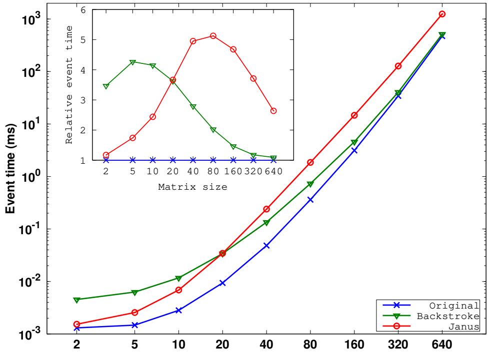
Listing 1: Janus implementation of matrix multiplication [RevComp2020]

```
procedure matrix_mult(int A[][], int B[][], int n)
  call crout(B, n)   // In-place LDU decomposition of B
  call multLD(A, B, n) // A := A*LD in place
  call multU(A, B, n) // A := A*U in place
  uncall crout(B, n) // Revert LDU decomposition to recover B
```

Listing 2: Backstroke-Instrumented (reversible) C++ Forward Code

```
template<typename myuint>
void matmul(int n, myuint A[], myuint B[], myuint AB[]) {
  for(int i = 0; i<n; i++) {
    for(int j = 0; j<n; j++) {
      myuint s = 0;
      for(int k = 0; k<n; k++) {
        s = s + A[i*n+k]*B[k*n+j];
      }
      (xpdes::avpushT(AB[i*n+j])) = s;
} } }
```

Matrix Multiply - Janus vs. Backstroke



Benchmark model using matrix multiplication. Janus: in-place and step-wise reversible. 5/3 times more arithmetic operations than standard multiply. To compute $A := A \times B$: Crout algorithm for LDU decomposition, $B = L \times D \times U$ in place, then the sequence $A := A \times L$, $A := A \times D$, $A := A \times U$. Revert LDU decomposition in place, to recover B. 8000 LPs seq. [RevComp 2020 (LNCS 12070)].

Backstroke : Kinetic Monte Carlo - C++ Model

Listing 3: Original Code

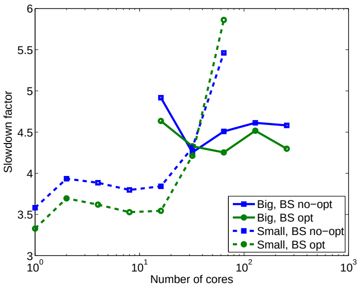
```
template <typename T,typename K> inline
T * Hash<T,K>::Insert(const K &key) {
  int idx = (int) (hash_value<K>(key) % (
        unsigned int) size);
  used = used + 1;
  table[idx] = new Link(key,table[idx]);


  return &table[idx]->data;
}
template <typename T,typename K> inline
void Hash<T,K>::Remove(const K &key) {
  int idx = (int) (hash_value<K>(key) % (
        unsigned int) size);
  Link *p = table[idx],*last = 0;
  while(p != 0 && !(p->key == key)) {
    last = p;
    p = p->next;
  }
  if(p != 0) {
    used = used - 1;
    if(last == 0)
      table[idx] = p->next;
    else
      last->next = p->next;
    delete p;
  }
}
```

Listing 4: Reversible Instrumented Code

```
template <typename T,typename K> inline
T * Hash<T,K>::Insert(const K &key) {
  int idx = (int) (hash_value<K>(key) % (unsigned int)
        size);
  (xpdes::avpushT(used)) = used + 1;
  (xpdes::avpushT(table[idx])) = (xpdes::
        registerAllocationForRollbackT(new Link(key,
        table[idx])));
  return &table[idx]->data;
}
template <typename T,typename K> inline
void Hash<T,K>::Remove(const K &key) {
  int idx = (int) (hash_value<K>(key) % (unsigned int)
        size);
  Link *p = table[idx],*last = 0;
  while(p != 0 && !(p->key == key)) {
    last = p;
    p = p->next;
  }
  if(p != 0) {
    (xpdes::avpushT(used)) = used - 1;
    if(last == 0)
      (xpdes::avpushT(table[idx])) = p->next;
    else
      (xpdes::avpushT(last->next)) = p->next;
    (xpdes::registerDeallocationForCommitT(p));
  }
}
```

Evaluation

KMC Model - Instrumentation vs. Best Known Hand-Written



- Slowdown factor the Backstroke instrumented code compared to best known hand written reverse code.
- Grain evolution sim - SPOCK: Scalable Parallel Optimistic Crystal Kinetics
- The big system consists of 768×768 spins, divided into a grid of $96 \times 96 = 9216$ LP's. The small system is 128×128 spins, divided into a grid of $16 \times 16 = 256$ LP's. The simulations are run for 1000 time units. [PADS 2016]

Conclusion

- Janus: reverseible programming language
  - deterministic in both directions
- Backstroke: incremental checkpointing
  - "Memory-modification traces" (assignment,alloc,dealloc)
  - Transactional - Forward/Reverse/Commit paradigm
- Use case: Optimistic PDES
  - Time Warp: distributed asynchronous lock-free rollback

Tool: Backstroke 2.1.4, https://github.com/LLNL/backstroke

References

Recent publications relevant to reversible computation

- [RC 2015] *Reverse Code Generation for Parallel Discrete Event Simulation.*
  Markus Schordan, David Jefferson, Peter Barnes, Tomas Oppelstrup, Daniel Quinlan. In Proc. of the 7th Conference on Reversible Computation. Reversible Computation, LNCS 9138, pp. 95-110, Springer, 2015.
- [PADS 2016] *Automatic generation of reversible C++ code and its performance in a scalable kinetic monte-carlo application.*
  Markus Schordan, Tomas Oppelstrup, David Jefferson, Peter D. Barnes Jr., and Daniel Quinlan. In Proc. of the 2016 annual ACM Conference on SIGSIM Principles of Advanced Discrete Simulation, SIGSIM-PADS 2016, Banff, Alberta, Canada, May 15-18, 2016, pp. 111–122. ACM, 2016.
- [PADS 2017] *Dealing with Reversibility of Shared Libraries in PDE.*
  Davide Cingolani, Alessandro Pellegrini, Markus Schordan, Francesco Quaglia, David R. Jefferson. Proc. of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, SIGSIM-PADS 2017, pp. 41-52, Singapore, May 24-26, 2017. ACM 2017.
- [RevComp 2020] *Reversible Languages and Incremental State Saving in Optimistic Parallel Discrete Event Simulation.*
  Markus Schordan, Tomas Oppelstrup, Michael K. Thomsen, Robert Glück. In Reversible Computation: Extending Horizons of Computing. LNCS, vol 12070, pp. 187-207, Springer, 2020.

Thank you for your attention. Any questions?

CASC
Center for Applied
Scientific Computing

Lawrence Livermore
National Laboratory