Beyond Simple Monte-Carlo: Parallel Computing with QuantLib

Klaus Spanderen

E.ON Global Commodities

November 14, 2013

Symmetric Multi-Processing

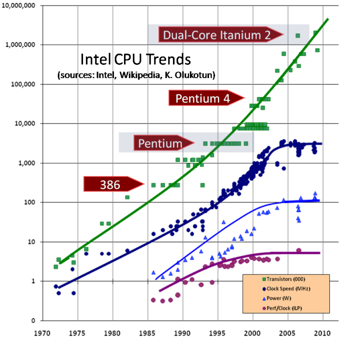
Graphical Processing Units

Message Passing Interface

Conclusion

Symmetric Multi-Processing: Overview

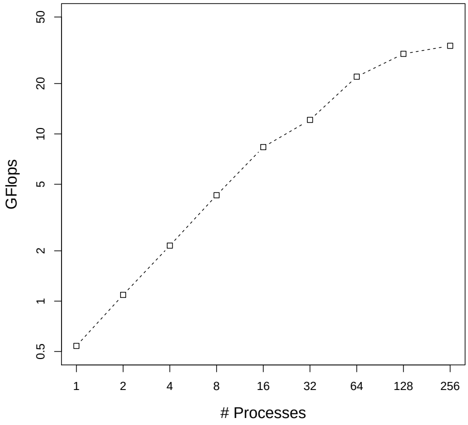
- Moore's Law: Number of transistors doubles every two years.
- Leaking turns out to be the death of CPU scaling.
- Multi-core designs helps processor makers to manage power dissipation.
- Symmetric Multi-Processing has become a main stream technology.



Herb Sutter: "The Free Lunch is Over: A Fundamental Turn Toward Concurrency in Software."

Multi-Processing with QuantLib

Divide and Conquer: Spawn several independent OS processes



The QuantLib benchmark on a 32 core (plus 32 HT cores) server.

Multi-Threading: Overview

- QuantLib is per se not thread-safe.
- Use case one: really thread-safe QuantLib (see Luigi's talk)
- Use case two: multi-threading to speed-up single pricings.
  - Joesph Wang is working with Open Multi-Processing (OpenMP) to parallelize several finite difference and Monte-Carlo algorithms.
- Use case three: multi-threading to parallelize several pricings, e.g. parallel pricing to calibrate models.
- Use case four: Use of QuantLib in C#,F#, Java or Scala via SWIG layer and multi-threaded unit tests.
- Focus on use case three and four:
  - Situation is not too bad as long as objects are not shared between different threads.

Multi-Threading: Parallel Model Calibration

C++11 version of a parallel model calibration function

```
Disposable<Array>
  CalibrationFunction::values(const Array& params) const {
  model_->setParams(params);

  std::vector<std::future<Real> > errorFcts;
  std::transform(std::begin(instruments_), std::end(instruments_),
                 std::back_inserter(errorFcts),
                 [](decltype(*begin(instruments_)) h) {
                   return std::async(std::launch::async,
                      &CalibrationHelper::calibrationError,
                      h.get());});

  Array values(instruments_.size());
  std::transform(std::begin(errorFcts), std::end(errorFcts),
    values.begin(), [](std::future<Real>& f) { return f.get();});

  return values;
}
```

Multi-Threading: Singleton

- Riccardo's patch: All singletons are thread local singletons.

```
template <class T>
T& Singleton<T>::instance() {
  static boost::thread_specific_ptr<T> tss_instance_;
  if (!tss_instance_.get()) {
    tss_instance_.reset(new T);
  }
  return *tss_instance_;
}
```

- C++11 Implementation: Scott Meyer Singleton

```
template <class T>
T& Singleton<T>::instance() {
  static thread_local T t_;
  return t_;
}
```

Multi-Threading: Observer-Pattern

- Main purpose in QuantLib: Distributed event handling.
- Current implementation is highly optimized for single threading performance.
- In a thread local environment this would be sufficient, but ...
- ... the parallel garbage collector in C#/F#, Java or Scala is by definition not thread local!
- Shuo Chen article "Where Destructors meet Threads" provides a good solution ...
- ... but is not applicable to QuantLib without a major redesign of the observer pattern.

Multi-Threading: Observer-Pattern

Scala example fails immediately with spurious error messages

- pure virtual function call
- segmentation fault

```
import org.quantlib.{Array => QArray, _}
object ObserverTest {
  def main(args: Array[String]) : Unit = {
    System.loadLibrary("QuantLibJNI");
    val aSimpleQuote = new SimpleQuote(0)

    while (true) {
      (0 until 10).foreach(_ => {
          new QuoteHandle(aSimpleQuote)
          aSimpleQuote.setValue(aSimpleQuote.value + 1)
      })
      System.gc
    }
  }
}
```

Multi-Threading: Observer-Pattern

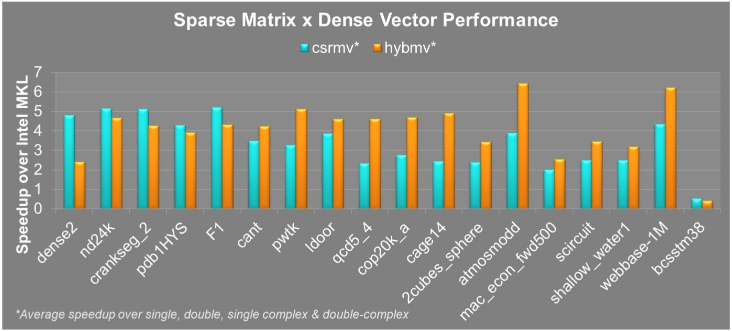
- The observer pattern itself can be solved using the thread-safe boost::signals2 library.
- Problem remains, an observer must be unregistered from all observables before the destructor is called.
- Solution:
  - QuantLib enforces that all observers are instantiated as boost shared pointers.
  - The preprocessor directive BOOST_SP_ENABLE_DEBUG_HOOKS provides a hook to every destructor call of a shared object.
  - if the shared object is an observer then use the thread-safe version of Observer::unregisterWithAll to detach the observer from all observables.
- Advantage: this solution is backward compatible, e.g. test suite can now run multi-threaded.

Finite Differences Methods on GPUs: Overview

- Performance of Finite Difference Methods is mainly driven by the speed of the underlying sparse linear algebra subsystem.
- In QuantLib any finite difference operator can be exported as boost::numeric::ublas::compressed_matrix<Real>
- boost sparse matrices can by exported in Compressed Sparse Row (CSR) format to high performance libraries.
- CUDA sparse matrix libraries:
  - cuSPARSE: basic linear algebra subroutines used for sparse matrices.
  - cusp: general template library for sparse iterative solvers.

Spare Matrix Libraries for GPUs

Performance pictures from NVIDIA (https://developer.nvidia.com/cuSPARSE)

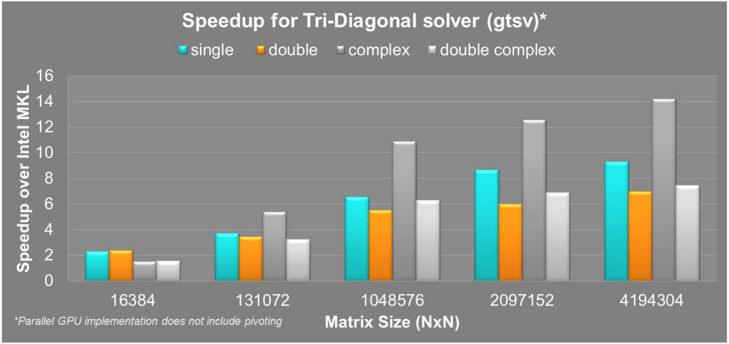


- cuSPARSE 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz

•Performance may vary based on OS ver. and motherboard config.

Spare Matrix Libraries for GPUs

Performance pictures from NVIDIA



- cuSPARSE 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz
- Performance may vary based on OS ver. and motherboard config.

Speed-ups are smaller than the reported "100x" for Monte-Carlo

Example I: Heston-Hull-White Model on GPUs

SDE is defined by

$$
\begin{aligned}
dS_t &= (r_t - q_t)S_t dt + \sqrt{v_t} S_t dW_t^S \\
dv_t &= \kappa_v(\theta_v - v_t)dt + \sigma_v \sqrt{v_t} dW_t^v \\
dr_t &= \kappa_r(\theta_{r,t} - r_t)dt + \sigma_r dW_t^r \\
\rho_{Sv} dt &= dW_t^S dW_t^v \\
\rho_{Sr} dt &= dW_t^S dW_t^r \\
\rho_{vr} dt &= dW_t^v dW_t^r
\end{aligned}
$$

Feynman-Kac gives the corresponding PDE:

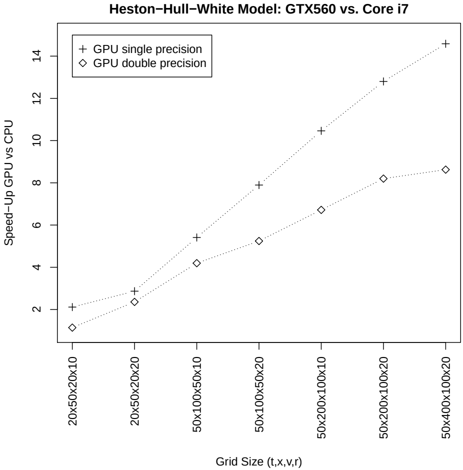
$$
\begin{aligned}
\frac{\partial u}{\partial t} &= \frac{1}{2} S^2 \nu \frac{\partial^2 u}{\partial S^2} + \frac{1}{2}\sigma_\nu^2 \nu \frac{\partial^2 u}{\partial \nu^2} + \frac{1}{2}\sigma_r^2 \frac{\partial^2 u}{\partial r^2} \\
&+ \rho_{S\nu}\sigma_\nu S \nu \frac{\partial^2 u}{\partial S \partial \nu} + \rho_{Sr}\sigma_r S \sqrt{\nu} \frac{\partial^2 u}{\partial S \partial r} + \rho_{vr}\sigma_r \sigma_\nu \sqrt{\nu} \frac{\partial^2 u}{\partial \nu \partial r} \\
&+ (r - q) S \frac{\partial u}{\partial S} + \kappa_v(\theta_v - \nu)\frac{\partial u}{\partial \nu} + \kappa_r(\theta_{r,t} - r)\frac{\partial u}{\partial r} - ru
\end{aligned}
$$

Example I: Heston-Hull-White Model on GPUs

- Good new: QuantLib can build the sparse matrix.
- An operator splitting scheme needs to be ported to the GPU.

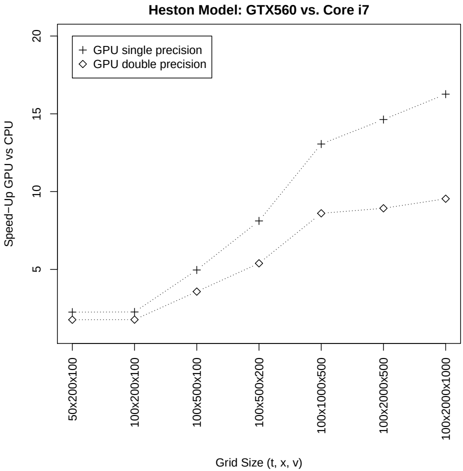
```
void HundsdorferScheme::step(array_type& a, Time t) {
  Array y = a + dt_*map_->apply(a);
  Array y0 = y;

  for (Size i=0; i < map_->size(); ++i) {
    Array rhs = y - theta_*dt_*map_->apply_direction(i, a);
    y = map_->solve_splitting(i, rhs, -theta_*dt_);
  }

  Array yt = y0 + mu_*dt_*map_->apply(y-a);
  for (Size i=0; i < map_->size(); ++i) {
    Array rhs = yt - theta_*dt_*map_->apply_direction(i, y);
    yt = map_->solve_splitting(i, rhs, -theta_*dt_);
  }
  a = yt;
}
```

Example I: Heston-Hull-White Model on GPUs



Speed-ups are much smaller than for Monte-Carlo pricing.

Example II: Heston Model on GPUs



Speed-ups are much smaller than for Monte-Carlo pricing.

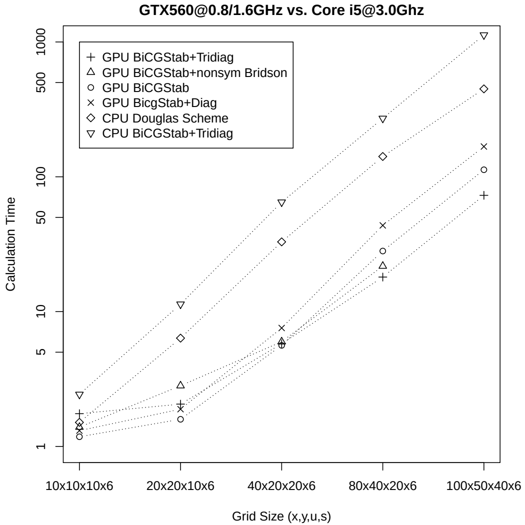
Example III: Virtual Power Plant

Kluge model (two OU processes plus jump diffusion) leads to a three dimensional partial integro differential equation:

$$
\begin{aligned}
rV &= \frac{\partial V}{\partial t} + \frac{\sigma_x^2}{2}\frac{\partial^2 V}{\partial x^2} - \alpha x \frac{\partial V}{\partial x} - \beta y \frac{\partial V}{\partial y} \\
&+ \frac{\sigma_u^2}{2}\frac{\partial^2 V}{\partial u^2} - \kappa u \frac{\partial V}{\partial u} + \rho \sigma_x \sigma_u \frac{\partial^2 V}{\partial x \partial u} \\
&+ \lambda \int_{\mathbb{R}} \left(V(x, y+z, u, t) - V(x, y, u, t) \right) \omega(z) dz
\end{aligned}
$$

Due to the integro part the equation is not truly a sparse matrix.

Example III: Virtual Power Plant



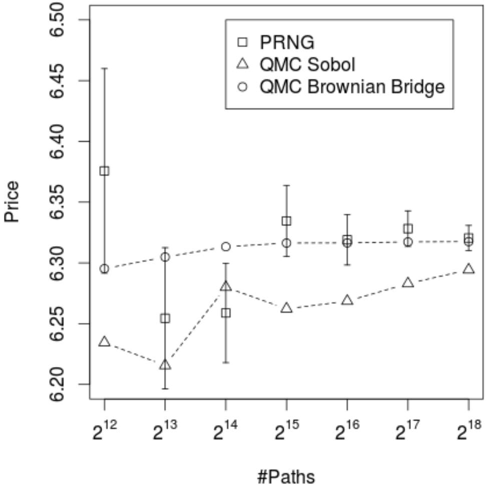
Quasi Monte-Carlo on GPUs: Overview

- Koksma-Hlawka bound is the basis for any QMC method:

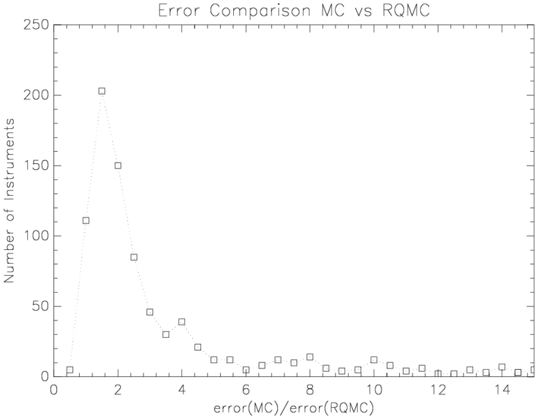
$$\begin{aligned}
\left|\frac{1}{n}\sum_{i=1}^{n} f(x_i) - \int_{[0,1]^d} f(u)du\right| &\leq V(f)D^*(x_1,...,x_n) \\
D^*(x_1,...,x_n) &\geq c\frac{(\log n)^d}{n}
\end{aligned}$$

- The real advantage of QMC shows up only after $N \sim e^d$ drawing samples, where d is the dimensionality of the problem.
- Dimensional reduction of the problem is often the first step.
- The Brownian bridge is tailor-made to reduce the number of significant dimensions.

Quasi Monte-Carlo on GPUs: Arithmetic Option Example



Quasi Monte-Carlo on GPUs: Exotic Equity Options

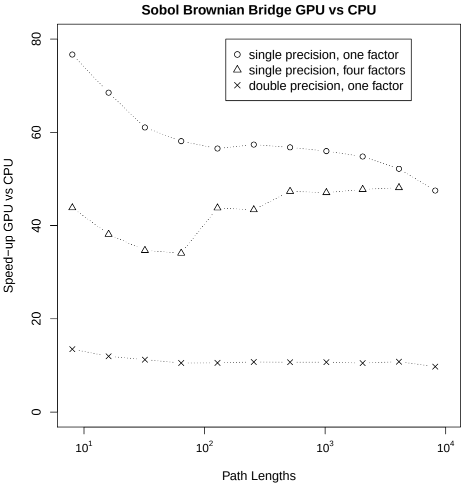


Accelerating Exotic Option Pricing and Model Calibration Using GPUs, Bernemann et al in High Performance Computational Finance (WHPCF), 2010, IEEE Workshop on, pages 17, Nov. 2010.

Quasi Monte-Carlo on GPUs: QuantLib Implementation

- CUDA supports Sobol random numbers up to the dimension 20,000.
- Direction integers are taken from the JoeKuoD7 set.
- On comparable hardware CUDA Sobol generators are approx. 50 times faster than MKL.
- Weights and indices of the Brownian bridge will be calculated by QuantLib.

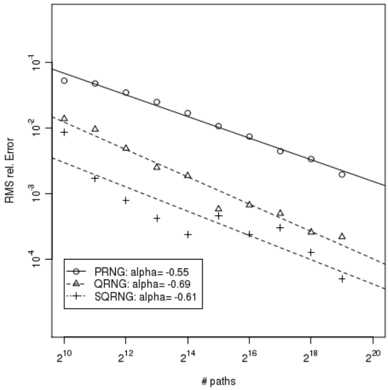
Quasi Monte-Carlo on GPUs: Performance



Comparison GPU (GTX 560@0.8/1.6Ghz) vs. CPU (i5@3.0GHz)

Quasi Monte-Carlo on GPUs: Scrambled Sobol Sequences

- In addition CUDA supports scrambled Sobol sequences.
- Higher order scrambled sequences are a variant of randomized QMC method.
- They achieve better root mean square errors on smooth integrands.
- Error analysis is difficult. A shifted (t,m,d)-net does not need to be a (t,m,d)-net.



RMSE for a benchmark portfolio of Asian options.

Message Passing Interface (MPI): Overview

- De-facto standard for massive parallel processing (MPP).
- MPI is a complementary standard to OpenMP or threading.
- Vendors provide high performance/low latency implementations.
- The roots of the MPI specification are going back to the early 90s and you will feel the age if you use the C-API.
- Favour Boost.MPI over the original MPI C++ bindings!
- Boost.MPI can build MPI data types for user-defined types using the Boost.Serialization library.

Message Passing Interface (MPI): Model Calibration

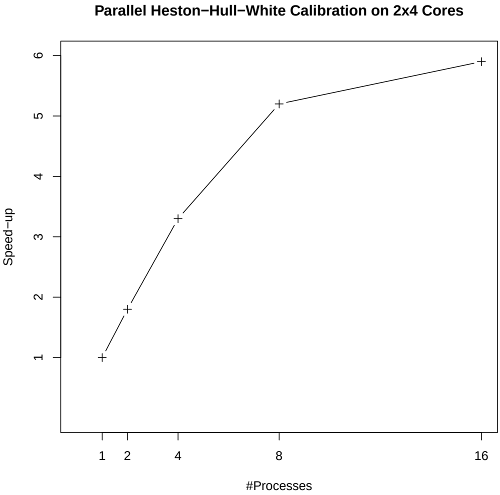
- Model calibration can be a very time-consuming task, e.g. the calibration of a Heston or a Heston-Hull-White model using American puts with discrete dividends → FDM pricing
- Minimal approach: introduce a MPICalibrationHelper proxy, which "has a" CalibrationHelper.

```
class MPICalibrationHelper : public CalibrationHelper {
  public:
    MPICalibrationHelper(
      Integer mpiRankId,
      const Handle<Quote>& volatility,
      const Handle<YieldTermStructure>& termStructure,
      const boost::shared_ptr<CalibrationHelper>& helper);
    ....
  private:
    std::future<Real> modelValueF_;
    const boost::shared_ptr<boost::mpi::communicator> world_;
    ....
};
```

Message Passing Interface (MPI): Model Calibration

```
void MPICalibrationHelper::update() {
  if (world_->rank() == mpiRankId_) {
    modelValueF_ = std::async(std::launch::async,
                              &CalibrationHelper::modelValue, helper_);
  }
  CalibrationHelper::update();
}

Real MPICalibrationHelper::modelValue() const {
  if (world_->rank() == mpiRankId_) {
   modelValue_ = modelValueF_.get();
  }
  boost::mpi::broadcast(*world_, modelValue_, mpiRankId_);

  return modelValue_;
}

int main(int argc, char* argv[])  {
  boost::mpi::environment env(argc, argv);
  ....
}
```

Message Passing Interface (MPI): Model Calibration



Conclusion

- Often a simple divide and conquer approach on process level is sufficient to "parallelize" QuantLib.
- In a multi-threading environment the singleton- and observer-pattern need to be modified.
  - Do not share QuantLib objects between different threads.
  - Working solution for languages with parallel garbage collector.
- Finite Difference speed-up on GPUs is rather 10x than 100x.
- Scrambled Sobol sequences in conjunction with Brownian bridges improve the convergence rate on GPUs.
- Boost.MPI is a convenient library to utilise QuantLib on MPP systems.