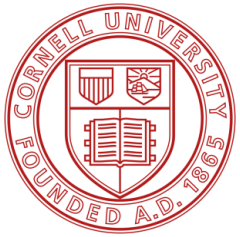


High-Performance Computing

Marcos López de Prado, Ph.D.

Advances in Financial Machine Learning

ORIE 5256

Electronic copy available at: https://ssrn.com/abstract=3274354

What are we going to learn today?

- Supercomputing
  - What is a supercomputer?
  - How are supercomputers useful in finance?
  - Supercomputing code
- Parallelism
  - Vectorization
  - Multi-threading
  - Asynchronicity
- Scheduling
  - Hierarchical parallelism
  - Slave / master scheduling

Electronic copy available at: https://ssrn.com/abstract=3274354

Section I

Supercomputing

Electronic copy available at: https://ssrn.com/abstract=3274354

What is a supercomputer?

- A supercomputer is a computer that can perform a large number of floating-point operations per second (FLOPS), relative to a general-purpose computer.
- This is generally achieved through one of two paths:
  - **Distributed Computing**: Thousands of computers distributed across a network solve tasks independently, and report the results to a central repository. An effective way to use idle computers when their users are logged-off.
  - **High performance computing cluster (HPC)**: Thousands of dedicated processors in close proximity form a cluster, and solve the tasks in coordination. This often requires:
    - An architecture that enables massive parallelization of tasks
    - Fast communication between the processors
    - Large amounts of RAM and ROM
    - Efficient cooling systems
    - Dedicated power supplies
- Berkeley Lab operates some of the fastest HPCs.

Electronic copy available at: https://ssrn.com/abstract=3274354

How are supercomputers useful in finance?

- Some reasons why supercomputers are essential tools in finance:
  - Financial markets are complex dynamic and adaptive systems.
    - Problems often involve non-linear, high-dimensional relations
    - Regime switching, state-space methods, hidden variables
    - Weather forecasting is trivial compared to markets forecasting. Models fail because they are too simplistic
  - Machine learning involves NP-Complete and NP-Hard problems.
  - Big data, experimental math require heavy machinery.
- Examples of problems that require supercomputers are:
  - Dynamic portfolio optimization.
  - Scenario analysis, modelling path-dependent systems.
  - Large scale classification and clustering: Bagging & boosting.
- One researcher operating an HPC can accomplish what dozens of researchers with powerful servers could not.

Electronic copy available at: https://ssrn.com/abstract=3274354

Supercomputing code

- Supercomputing code is designed to extract as many FLOPS as possible out of a supercomputer.
  - Different hardware architectures will require different coding structures.
  - Understand well the algorithmic complexity of your problem before you code.
- Generally speaking, desired features in HPC code are:
  - **Vectorization**: Replace virtually all explicit iterators (like For…loops) with matrix algebra operations, compiled iterators, generators, etc.
  - **Multi-threading**: When task B is not contingent on the outcome of task A, both should be solved in parallel.
  - **Asynchronicity**: Outputs may not return in their called sequence.
  - **Scheduling**: Split your jobs in ways that all resources are maximized.
  - **Race-condition resilience**: Expect your program will not have an accurate representation of the variables states.
  - **Redundancies**: It is impossible to anticipate what will go wrong. Assume tasks will fail, and resubmit jobs on the fly as needed.

Electronic copy available at: https://ssrn.com/abstract=3274354

Section II

Vectorization, multi-threading, asynchronicity

Electronic copy available at: https://ssrn.com/abstract=3274354

Vectorization example: Cartesian of a dict of lists

Bad

```
# Cartesian product of dictionary of lists
dict0={'a':['1','2'],'b':['+','*'],'c':['!','@']}
for a in dict0['a']:
    for b in dict0['b']:
        for c in dict0['c']:
            print {'a':a,'b':b,'c':c}
```

Better

```
# Cartesian product of dictionary of lists
from itertools import izip,product
dict0={'a':['1','2'],'b':['+','*'],'c':['!','@']}
jobs=(dict(izip(dict0,i)) for i in product(*dict0.values()))
for i in jobs:print i
```

- The code to the right is preferable, because:
  - It is vectorized:
    - Nested For…loops were replaced with iterators
  - It is flexible:
    - The code infers the dimensionality of the mesh from the dimensionality of dict0
    - We could run 100 variables without having to modify the code, or needing 100 For…loops
  - It is faster:
    - Under the hood, python is running operations in C++
- This code snippet is very useful for unit testing, brute force searches and scenario analysis. Can you see why?

Electronic copy available at: https://ssrn.com/abstract=3274354

Multi-threading & asynchronicity (1/2)

```
import numpy as np
#---------------------------------------------------------------------------
def main0():
    # Path dependency: Sequential implementation
    r=np.random.normal(0,.01,size=(1000,10000))
    t=barrierTouch(r)
    return
#---------------------------------------------------------------------------
def barrierTouch(r,width=.5):
    # find the index of the earliest barrier touch
    t,p={},np.log((1+r).cumprod(axis=0))-np.log(1)
    for j in xrange(r.shape[1]): # go through columns
        for i in xrange(r.shape[0]):
            if p[i,j]>=width or p[i,j]<=-width:
                t[j]=i
                continue
    return t
#---------------------------------------------------------------------------
if __name__=='__main__':
    import timeit
    print min(timeit.Timer('main0()',setup='from __main__ import main0').repeat(5,10))
```

This code finds the earliest time 10,000 Gaussian processes of length 1,000 touch a symmetric double barrier of width 50x std. In a server with 16 Intel Xeon 3.30GHz processors, the fastest 10-run out of 5 took 55.1 secs.

Electronic copy available at: https://ssrn.com/abstract=3274354

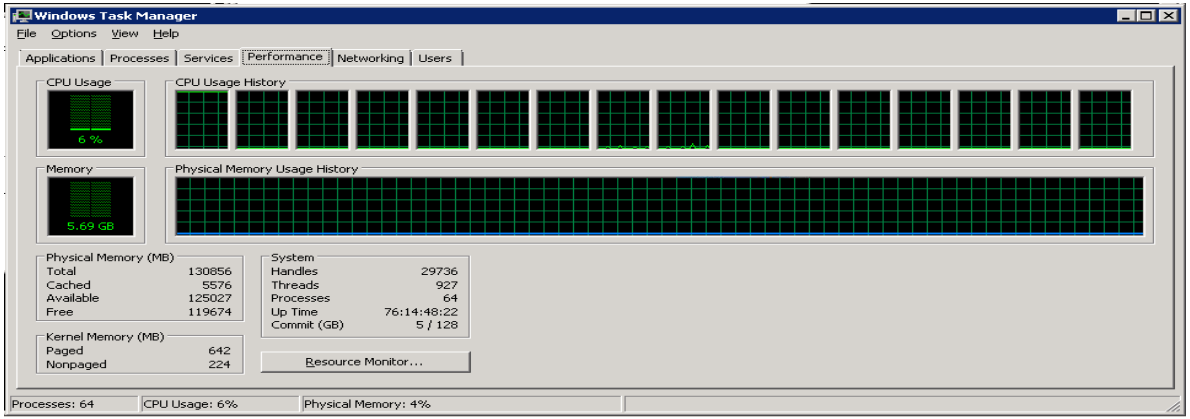
Multi-threading & asynchronicity (2/2)

```
import numpy as np
import multiprocessing as mp
#---------------------------------------------------------------------------
def main1():
   # Path dependency: Multi-threaded implementation
   r,numThreads=np.random.normal(0,.01,size=(1000,10000)),16
   parts=np.linspace(0,r.shape[0],min(numThreads,r.shape[0])+1)
   parts,jobs=np.ceil(parts).astype(int),[]
   for i in xrange(1,len(parts)):
      jobs.append(r[:,parts[i-1]:parts[i]]) # parallel jobs
   pool,out=mp.Pool(processes=numThreads),[]
   outputs=pool.imap_unordered(barrierTouch,jobs)
   for out_ in outputs:out.append(out_) # asynchronous response
   pool.close();pool.join()
   return
#---------------------------------------------------------------------------
if __name__=='__main__':
   import timeit
   print min(timeit.Timer('main1()',setup='from __main__ import main1').repeat(5,10))
```

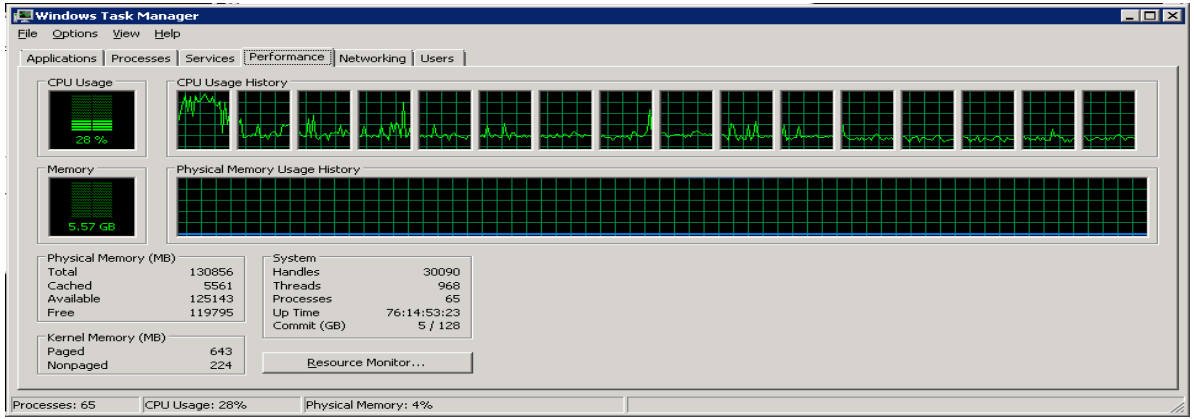
This code splits the previous problem into 16 tasks, one per processor. The tasks are then run asynchronously in parallel. The fastest 10-run out of 5 took 9.6 secs.
(Question for later: Where is the slave scheduler in the above snippet?)

Electronic copy available at: https://ssrn.com/abstract=3274354

CPU workload balance



System performance when the job runs in (standard) single thread: 6%. One core does all the work, while the rest remain idle.



System performance after parallelization: 28%. Work is balanced across all cores. Further parallelization would utilize 100% CPU.

Electronic copy available at: https://ssrn.com/abstract=3274354

Multi-threading with callback

```
def expandCall(kargs):
    # Expand the arguments of a callback function, kargs['func']
    func=kargs['func']
    del kargs['func']
    out=func(**kargs)
    return out
#----------------------------------------------------------------------------
def processJobs(jobs, numThreads=24):
    # jobs must contain a 'func' callback, for expandCall
    pool=mp.Pool(processes=numThreads)
    outputs,out=pool.imap_unordered(expandCall,jobs),[]
    # Process asynchronous output, collect outputs
    for i,out_ in enumerate(outputs,1):out.append(out_)
    pool.close();pool.join() # needed to prevent memory leaks
    return out
```

This snippet generalizes the multi-threading method to a generic function with unknown arguments.

The `jobs` variable is a list of dictionaries. Each dictionary contains a `func` key, that assigns a callback function to every job. The outputs are collected in the `out` list, and returned.

Electronic copy available at: https://ssrn.com/abstract=3274354

Section III

Scheduling

Electronic copy available at: https://ssrn.com/abstract=3274354

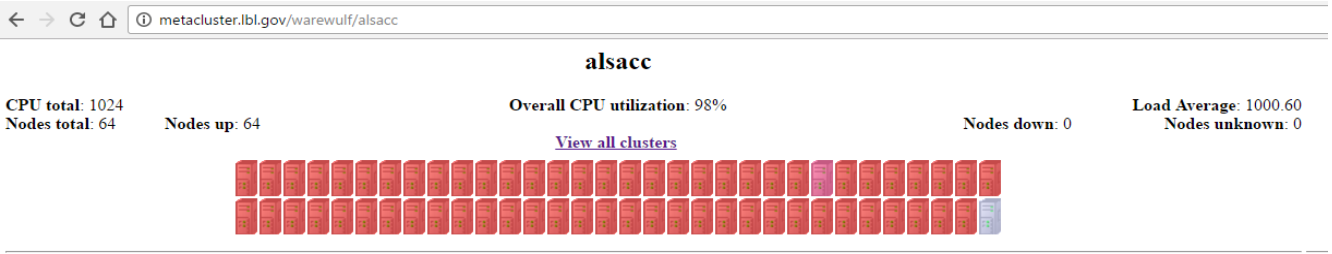
Hierarchical parallelisms

- Ideally, we want to balance the workload across all nodes & cores.
  - Tasks that share the same data should be bundled together and sent to the same node, to minimize expensive I/O operations and network traffic.
- I personally find the following hierarchical approach useful. Divide a large problem into:
  - **Atoms**: This is the basic task that can be *parallelized* within the same node.
    - E.g., given some input data, fit a SVM using joblib or multiprocessing
  - **Molecules**: A collection of atoms that are best worked by the same node *sequentially*, because they share some dependency.
    - E.g., fit various SVMs (atoms) on the same RAM-loaded dataset
  - **Groups**: The collection of asynchronous molecules that solve the problem.
    - E.g., a complete backtest results from stitching together all molecules in a group
- This gives us two levels of parallelization:
  - 1st level: Multi-core parallelization of atomic tasks.
  - 2nd level: Multi-node parallelization of molecular tasks.

Electronic copy available at: https://ssrn.com/abstract=3274354

Slave / master scheduling

- To make sense of this organized chaos, write two schedulers:
  - Slave scheduler:
    - This is a program that coordinates the solution of a *molecule*, and reports the outputs to the master
    - Every node runs this program (2nd level parallelization)
  - Master scheduler:
    - One node (the master) runs this program, telling all the other (slave) nodes what to do
    - This program also executes the final process on the *group* of outputs
    - The master should not be overloaded with work, or the slaves will be underutilized
    - The master must be resilient to race-conditions
    - The master must be able to resubmit failed jobs and rebalance workload on the fly, in response to unpredictable slave behavior
    - (Funny story: How the slaves killed the master)



A HPC cluster operated with 1 master and 63 slaves. The master has plenty of spare resources, so that it can handle pending tasks, exceptions, and outputs.

Electronic copy available at: https://ssrn.com/abstract=3274354

Structure of the master scheduler (1/2)

```
def getAtomsInMol(atoms,molNum,molsPerGroup):
   # Select the atoms associated with 0<=molNum<=molsPerGroup-1
   # A molecule is a slice of the atoms list, as partitioned by parts
   parts=np.ceil(np.linspace(0,len(atoms),min(molsPerGroup,len(atoms))+1)).astype(int)
   if molNum+1>=len(parts):return [] # empty molecule, because len(atoms)<molsPerGroup
   mol=atoms[parts[molNum]:parts[molNum+1]]
   return mol
#--------------------------------------------------------------------------
def initializeAllMols(pathIn,molsPerGroup,maxIters):
   # get all molecules from all groups (rows) in the params file
   numGroups=sum(1 for i in open(pathIn))-1
   pending={(i,j):maxIters for i in xrange(numGroups) for j in xrange(molsPerGroup)}
   return pending,numGroups
#--------------------------------------------------------------------------
def isJobRunning(jobId): …  ← Boolean indicating whether that job ID is still running or not
def runJobs(jobName): …  ← Execute the script that submits a molecule for execution
def prepareMolScript(pathOut,slaveName,groupNum,molNum,molsPerGroup,timeLimit=None): …  ← mol script
def getJobsStatus(pathOut,pending,jobIds,run): …  ← identify pending,submit,done,jobIds
def processGroups(mod,pathOut,molsPerGroup,done): …  ← combine molecules from a group, after all done
#--------------------------------------------------------------------------
```

Function `getAtomsInMol` binds together a slice of the atoms, to form a molecule.

Functions `isJobRunning`, `runJobs` and `prepareMolScript` are system specific, and their code will depend on the particular cluster's API.

Electronic copy available at: https://ssrn.com/abstract=3274354

Structure of the master scheduler (2/2)

```
def processAllMols(pathOut,slaveName,molsPerGroup,maxNodes,timeLimit,sleepSecs=10):
   jobIds={}
   pending,numGroups=initializeAllMols(pathIn,molsPerGroup,maxIters)
   while len(pending)>0:
      #1) adjust tasks
      run=filter(lambda i:isJobRunning(i),jobIds) # hpc call
      run={i:jobIds[i] for i in run} # jobs still running
      pending,submit,done,jobIds=getJobsStatus(pathOut,pending,jobIds,run)
      #2) submit,resubmit jobs
      for groupNum,molNum in submit[:min(len(submit),maxNodes-len(run))]:
         prepareMolScript(pathOut,slaveName,groupNum,molNum,molsPerGroup,timeLimit)
         jobId=runJobs(slaveName) # hpc call
         jobIds[jobId]=(groupNum,molNum)
      #3) process outcomes
      processed=processGroups(mod,pathOut,molsPerGroup,done)
      if len(processed)==0:time.sleep(sleepSecs)
   return
```

The state of the molecules is tracked through these variables:

- **pending**: dictionary of pending molecules (some submitted, some not), with the count of remaining trials
- **jobIds**: submitted molecules
- **run**: submitted molecules currently running
- **submit**: pending – run (this includes jobs that were run and failed)
- **done**: completed molecules still unprocessed

Electronic copy available at: https://ssrn.com/abstract=3274354

Structure of the slave scheduler

```
from masterScheduler import getAtomsInMol
#-------------------------------------------------------------------------
def defineAtoms(kargs): …   ← Define the indivisible tasks that will be multi-threaded
def atomicTask(atom): …   ← function that solves the task associated with a particular atom
#-------------------------------------------------------------------------
def main0(pathOut,molNum,molsPerGroup,groupNum):
   # slave executes molNum from groupNum, knowing that there are molsPerGroup
   #1) Define atoms,mol
   atoms=defineAtoms(kargs)
   mol=getAtomsInMol(atoms,molNum,molsPerGroup)
   #2) run atoms in mol
   out={}
   for atom in mol:out[atom]=atomicTask(atom)
   return
```

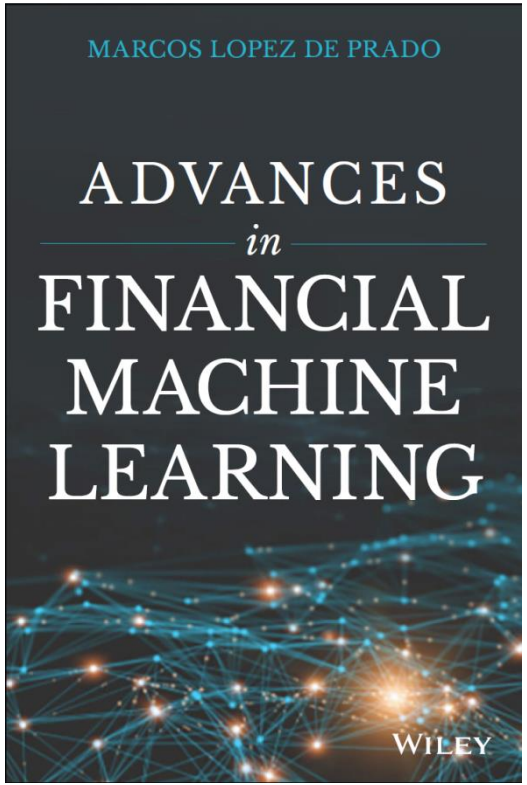
The master scheduler invokes the slave scheduler, passing as arguments the triplet that identifies a particular molecule: (`molNum`,`molsPerGroup`,`groupNum`).

With that information, the slave scheduler executes each atom within the molecule, with multi-threading (1st level of parallelization).

Each node in the cluster runs the slave scheduler, hence the 2nd level of parallelization.

Electronic copy available at: https://ssrn.com/abstract=3274354

For Additional Details



The first wave of quantitative innovation in finance was led by Markowitz optimization. Machine Learning is the second wave and it will touch every aspect of finance. López de Prado's Advances in Financial Machine Learning is essential for readers who want to be ahead of the technology rather than being replaced by it.
— Prof. **Campbell Harvey**, Duke University. Former President of the American Finance Association.

Financial problems require very distinct machine learning solutions. Dr. López de Prado's book is the first one to characterize what makes standard machine learning tools fail when applied to the field of finance, and the first one to provide practical solutions to unique challenges faced by asset managers. Everyone who wants to understand the future of finance should read this book.
— Prof. **Frank Fabozzi**, EDHEC Business School. Editor of The Journal of Portfolio Management.

Electronic copy available at: https://ssrn.com/abstract=3274354

THANKS FOR YOUR ATTENTION!

Electronic copy available at: https://ssrn.com/abstract=3274354

Disclaimer

- The views expressed in this document are the authors' and do not necessarily reflect those of the organizations he is affiliated with.
- No investment decision or particular course of action is recommended by this presentation.
- All Rights Reserved. © 2017-2019 by True Positive Technologies, LP

Electronic copy available at: https://ssrn.com/abstract=3274354

Multi-Period Integer Portfolio Optimization Using a Quantum Annealer

Marcos López de Prado

Lawrence Berkeley National Laboratory
Computational Research Division

Key Points

- The problem: For large portfolio managers, a sequence of single-period optimal positions is rarely multi-period optimal.
- A major cause: Transaction costs can prevent large portfolio managers from monetizing most of their forecasting power.
- The solution: Compute the trading trajectory that comes sufficiently close to the single-period optima while minimizing turnover costs.
- A few solutions exist that compute trading trajectories, under rather idealistic/simplistic scenarios.
- Computing a trading trajectory in general terms is a NP-Complete problem.
- In this presentation we will illustrate how quantum computers can solve this problem in the most general terms.

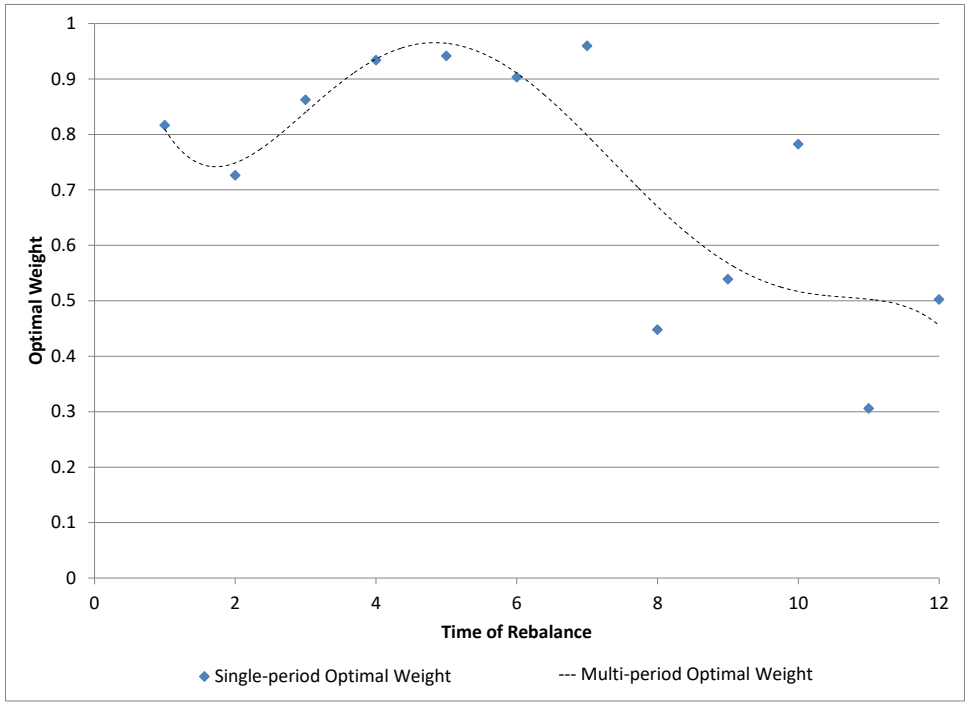
Electronic copy available at: https://ssrn.com/abstract=3274354

SECTION I
The Trading Trajectory Problem

Electronic copy available at: https://ssrn.com/abstract=3274354

The Trading Trajectory Problem

- Large portfolio managers need to optimize their portfolio over multiple forecasted horizons, in order to properly account for significant market impact (temporary & permanent).
- Some positions can only be traded in blocks (e.g., ETF units, real estate, private offerings, etc.), thus requiring integer solutions.

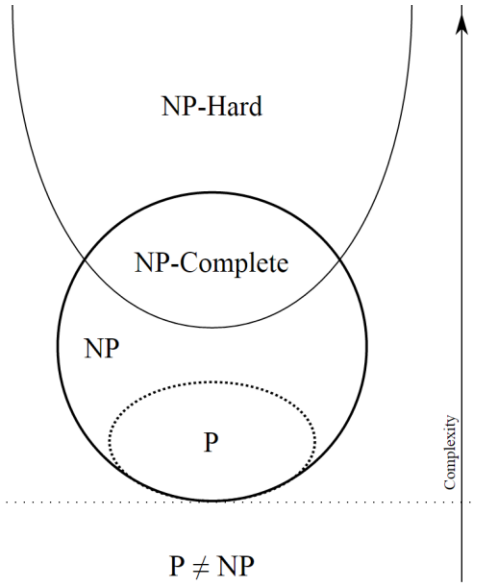


- Rebalancing the portfolio to align it with each single-step optimum (blue dots) is often prohibitively expensive.
- We could instead compute a trajectory that maximizes overall risk-adjusted performance after all transaction costs (black line).

Electronic copy available at: https://ssrn.com/abstract=3274354

How “hard” is this problem?

- **Single-period**:
  - Jobst et al. [2001] showed that the efficient frontier of the *integer* problem is discontinuous.
  - Bonami and Lejeune [2009] have solved the single-period problem with *integer* holdings without transaction costs.
  - The single-period *integer* portfolio optimization problem has been shown to be NP-complete, regardless of the risk measure used (Kellerer, Mansini, Speranza [2000]).



- **Multi-period**:
  - Garleanu and Pedersen [2013] solved a *continuous* multi-period problem for a Brownian motion via dynamic programming (Bellman equations), deriving a closed-form solution when the covariance matrix is positive definite and transaction costs are proportional to market risk.
  - Lopez de Prado [2014] showed that the *integer* problem is, in its most general formulations, amenable to quantum computing.

Electronic copy available at: https://ssrn.com/abstract=3274354

Integer Optimization Formulation

- This multi-period *integer* optimization problem may be written as

$$w = \text{argmax}_w \left\{ \sum_{t=1}^{T} \left(\underbrace{\mu_t^T w_t}_{\text{returns}} - \frac{\gamma}{2} \underbrace{w_t^T \Sigma_t w_t}_{\text{risk}} - \underbrace{\Delta w_t^T \Lambda_t \Delta w_t}_{\substack{\text{direct costs,} \\ \text{temp.impact}}} - \underbrace{\Delta w_t^T \Lambda_t' \Delta w_t}_{\text{perm.impact}} \right) \right\}$$

s.t.:

$$\forall t: \sum_{n=1}^{N} w_{n,t} \leq K \; ; \forall t, \forall n: w_{n,t} \leq K'$$

Symbol	Type	Description
K	$\mathbb{N}_1$	Number of units to be allocated at each time step
K'	$\mathbb{N}_1$	Largest allowed holding for any asset
N	$\mathbb{N}_1$	Number of assets
T	$\mathbb{N}_1$	Number of time steps
μ	$\mathbb{R}^{N \times T}$	Forecast mean returns of each asset at each time step
γ	$\mathbb{R}$	Risk aversion
Σ	$\mathbb{R}^{T \times N \times N}$	Forecast covariance matrix for each time step
c'	$\mathbb{R}^{N \times T}$	Permanent market impact coefficients for each asset at each time step
c	$\mathbb{R}^{N \times T}$	Transaction cost coefficients for each asset at each time step
w_0	$\mathbb{N}_0^N$	Initial holdings for each asset
w_{T+1}	$\mathbb{N}_0^N$	Final holdings for each asset
w	$\mathbb{N}_0^{N \times T}$	Holdings for each asset at each time step (the trading trajectory)

Temporary t-costs c are represented using the tensor Λ, where $\Lambda_{t,n,n'} = c_{n,t}\delta_{n,n'}$, and similarly for the permanent price impact c' and Λ'.

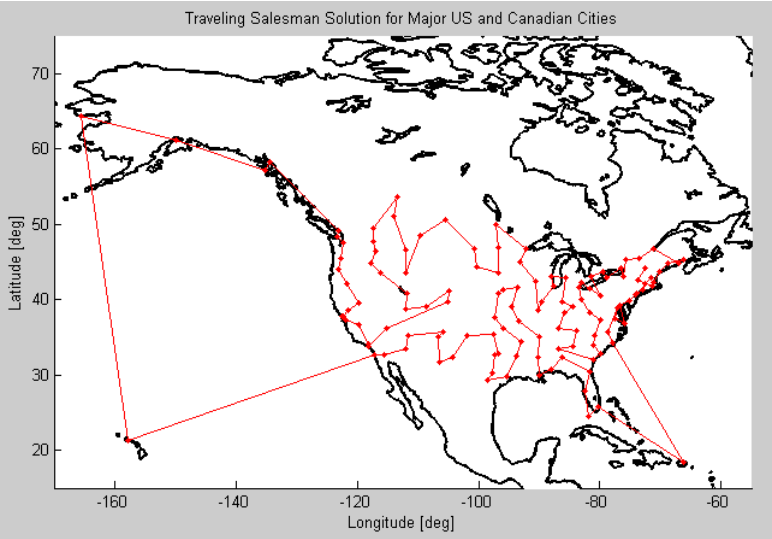
Electronic copy available at: https://ssrn.com/abstract=3274354

SECTION II
How does a Quantum Annealer solve this problem?

Electronic copy available at: https://ssrn.com/abstract=3274354

What is combinatorial optimization?

- Combinatorial optimization problems can be described as problems where there is a finite number of feasible solutions, which result from combining the discrete values of a finite number of variables.
- As the number of feasible combinations grows, an exhaustive search becomes impractical.
- The traveling salesman problem is an example of a combinatorial optimization problem that is known to be NP-hard.



- Digital computers evaluate and store feasible solutions *sequentially*.
  - The bits of a standard computer can only adopt one of two possible states ({0,1}) at once.
- This is a major disadvantage for finding path-dependent solutions.

Electronic copy available at: https://ssrn.com/abstract=3274354

How do Quantum Computers work? (1/2)

- Quantum computers rely on **qubits**, which are memory elements that may hold a linear superposition of both states ({0,1}).
  - This allows them to evaluate and store all feasible solutions simultaneously.

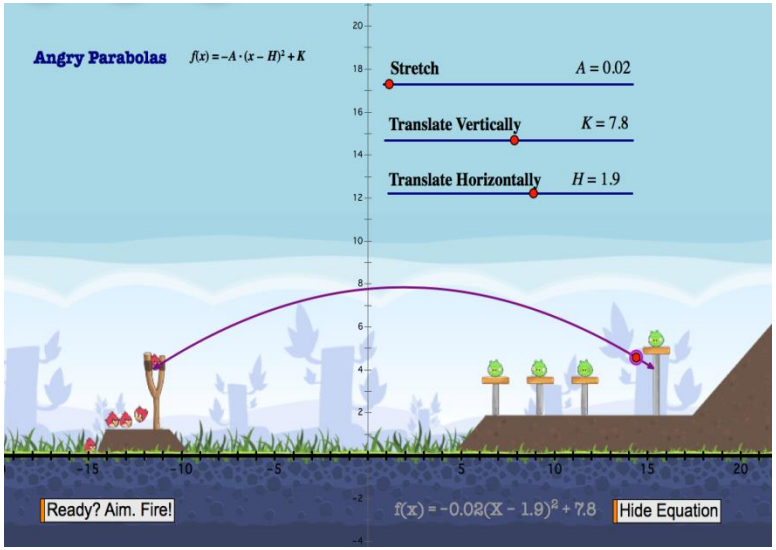
Superposition	States
δ	$\|(1,1)>$
γ	$\|(0,1)>$
β	$\|(1,0)>$
α	$\|(0,0)>$

- The state of a 2-bit digital system is determined by 2 binary digits.
- The state of a 2-bit quantum system is determined by 4 coefficients: $(\alpha, \beta, \gamma, \delta)$, normalized.
- N-qubits contain 2^N units of classical information.

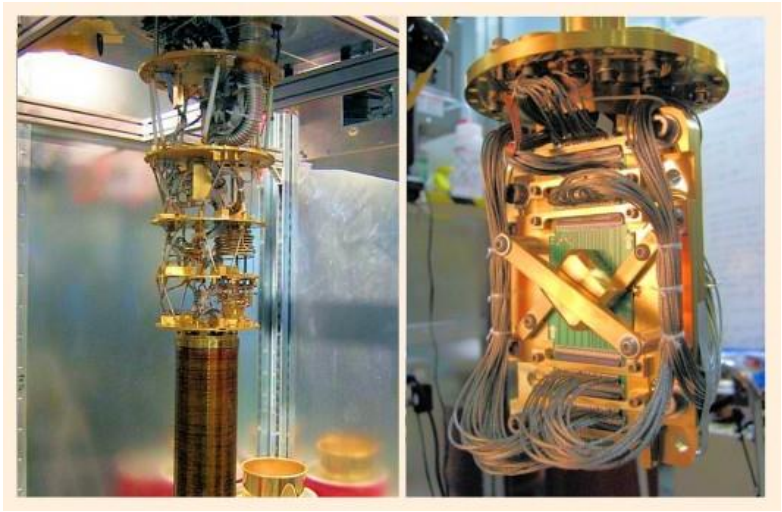
- Quantum computers may require an exponentially smaller number of operations to reach a solution.
- Quantum effects such as *tunneling* and *entanglement*, may help speed up the convergence of the procedure.

Electronic copy available at: https://ssrn.com/abstract=3274354

How do Quantum Computers work? (2/2)



- The digital-computer game "Angry Birds" uses mathematics to solve a physics problem.
  - A PDE library solves classical mechanics problems as demanded by the user.
- Conversely, Quantum Computers use physics to solve a mathematical problem.
- The Quantum Computing analogue of "Angry Birds" is to program the chip to behave like gravity, toss the bird, measure how it bounces.



- In order to keep external disturbances to a minimum, the D-Wave machine is cooled to 15 mK (about 180 times colder than interstellar space), and operates in a near vacuum shielded from electromagnetic fields.
- This makes the D-Wave machine one of the coolest things in the Galaxy… quite literally!

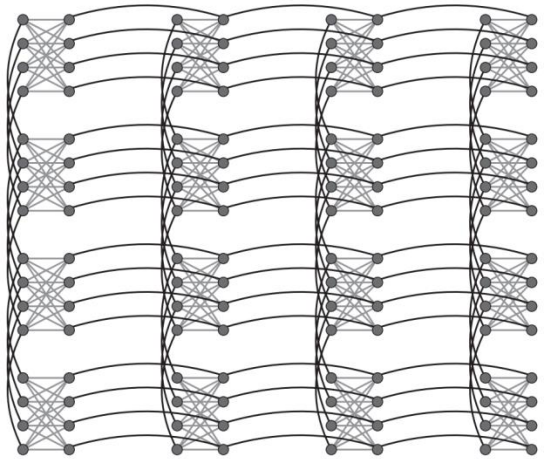
Electronic copy available at: https://ssrn.com/abstract=3274354

How does a Quantum Annealer work? (1/2)

- A Quantum Annealer is a quantum computer that solves optimization problems that can be encoded as a *Hamiltonian*.
- D-Wave Systems has developed a device which minimizes unconstrained binary quadratic functions of the form

$$\min_{x} x^T Q x$$
$$s.t.: x \in \{0,1\}^N$$

where $Q \in \mathbb{R}^{N \mathrm{x} N}$.



- The connectivity of the qubits in D-Wave's quantum annealer is currently described by a squared *Chimera graph* (left figure, a 128-Qubit architecture).
- The hardware graph is sparse and in general does not match the problem graph, which is defined by the adjacency matrix of *Q*.
- Minor embedding: Several physical qubits can be treated as a single logical qubit.

Electronic copy available at: https://ssrn.com/abstract=3274354

How does a Quantum Annealer work? (2/2)

- Superposition is useful to reach a solution quickly, however the final result must be representable in terms of the basis states.
- Encoding: Integer variables are recast as binary variables.

TABLE II: Comparison of the four encodings described in Section II-B. The column "Variables" refers to the number of binary variables required to represent a particular problem. The column "Largest integer" refers to a worst-case estimate of the largest integer that could be represented based on the limitation imposed by the noise level ϵ and the ratio between the largest and smallest problem coefficients δ and $n \equiv 1/\sqrt{\epsilon\delta}$.

Encoding	Variables	Largest integer	Notes
Binary	$TN\log_2(K'+1)$	$\lfloor 2n \rfloor - 1$	Most efficient in number of variables; allows representing of the second-lowest integer.
Unary	TNK'	No limit	Biases the quantum annealer due to differing redundancy of code words for each value; encoding coefficients are even, giving no dependence on noise, so it allows representing of the largest integer.
Sequential	$\frac{1}{2}TN\left(\sqrt{1+8K'}-1\right)$	$\frac{1}{2}\lfloor n \rfloor (\lfloor n \rfloor + 1)$	Biases the quantum annealer (but less than unary encoding); second-most-efficient in number of variables; allows representing of the second-largest integer.
Partition	$\leq T\binom{K+N-1}{N-1}$	$\lfloor n \rfloor$	Can incorporate complicated constraints easily; least efficient in number of variables; only applicable for problems in which groups of variables are required to sum to a constant; allows representing the lowest integer.

Electronic copy available at: https://ssrn.com/abstract=3274354

SECTION III
Experimental Results

Electronic copy available at: https://ssrn.com/abstract=3274354

Experiment Design

1. Generate a random multi-period integer optimization problem.
2. Using a digital computer:
   a) Compute the exact solution by brute force.
   b) Compute the variance of that solution by perturbating the covariance matrix.
3. Solve the problem using the Quantum Annealer.
4. Evaluate whether the Quantum solution falls within the exact solution's margin of error.

Why do we need to repeat the procedure multiple times?:

- Remember, **Quantum Computers will not return the exact solution**, because of hardware limitations.
- They are non-deterministic devices that search for the optimal linear combinations of basis states!
- In addition, they are physical devices, and the solution is recovered with a measurement error.

Electronic copy available at: https://ssrn.com/abstract=3274354

Accuracy

TABLE IV: Results using D-Wave's quantum annealer (with 200 instances per problem): N is the number of assets, T is the number of time steps, K is the number of units to be allocated at each time step and the maximum allowed holding (with $K' = K$), "encoding" refers to the method of encoding the integer problem into binary variables, "vars" is the number of binary variables required to encode the given problem, "density" is the density of the quadratic couplers, "qubits" is the number of physical qubits that were used, "chain" is the maximum number of physical qubits identified with a single binary variable, and $S(\alpha)$ refers to the success rate given a perturbation magnitude $\alpha\%$ (explained in the text).

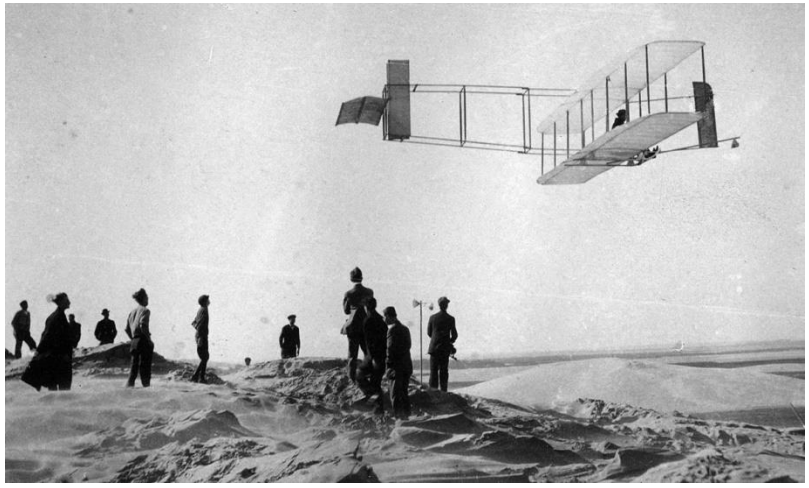
N	T	K	encoding	vars	density	qubits	chain	$S(0)$	$S(1)$	$S(2)$
2	3	3	binary	12	0.52	31	3	100.00	100.00	100.00
2	2	3	unary	12	0.73	45	4	97.00	100.00	100.00
2	4	3	binary	16	0.40	52	4	96.00	100.00	100.00
2	3	3	unary	18	0.53	76	5	94.50	100.00	100.00
2	2	7	binary	12	0.73	38	4	90.50	100.00	100.00
2	5	3	binary	20	0.33	63	4	89.00	100.00	100.00
2	6	3	binary	24	0.28	74	4	50.00	100.00	100.00
3	2	3	unary	18	0.65	91	6	38.50	80.50	95.50
3	3	3	binary	18	0.45	84	5	35.50	80.50	96.50
3	4	3	binary	24	0.35	106	6	9.50	89.50	100.00

- **The Quantum Annealer's solution was within the margin of error in a large majority of experiments.**

Electronic copy available at: https://ssrn.com/abstract=3274354

Why is this relevant?

- Our approach is general and scalable:
  - It computes integer (discrete) solutions (a NP-complete problem) that are globally optimal over multiple-periods.
  - We don't impose a particular returns process.
  - The procedure accepts very general transaction cost functions, and there is no need to impose simplifying assumptions such as costs proportional to risk.
  - The covariance matrix does not need to be positive definite. The solution does not even require covariance invertibility!



The size of problems currently solvable by Quantum computers is relatively small.

Numerous challenges need to be addressed.

The purpose of our work is to show that, in the near future, Quantum computers will solve many currently intractable problems, and render obsolete some existing approaches.

Electronic copy available at: https://ssrn.com/abstract=3274354

THANKS FOR YOUR ATTENTION!

Electronic copy available at: https://ssrn.com/abstract=3274354

SECTION IV
The stuff nobody reads

Electronic copy available at: https://ssrn.com/abstract=3274354

Notice:

The research contained in this presentation is the result of a continuing collaboration with

Peter Carr (Morgan Stanley, NYU), Phil Goddard (*1QBit),*
Poya Haghnegahdar (1QBit), Gili Rosenberg (1QBit),
Kesheng Wu (Berkeley Lab)

The full paper is available at:
http://ssrn.com/abstract=2649376

For additional details, please visit:
http://ssrn.com/author=434076
www.QuantResearch.info

Electronic copy available at: https://ssrn.com/abstract=3274354

Disclaimer

- The views expressed in this document are the authors' and do not necessarily reflect those of the organizations he is affiliated with.
- No investment decision or particular course of action is recommended by this presentation.
- All Rights Reserved. © 2017-2019 by True Positive Technologies, LP

Electronic copy available at: https://ssrn.com/abstract=3274354