STAC-A2™ Benchmark on POWER8®

Bishop Brock, Frank Liu, Karthick Rajamani
IBM Research, Austin Texas
{bcbrock,frankliu,karthick}@us.ibm.com

Workshop for High-Performance Computational Finance
WHPCF'15
November 20, 2015

With thanks to Kenneth Hill of the University of Florida and Julian Demouth of NVIDIA
Summary

• Workload characterization and performance analysis demonstrate that STAC-A2™ is a well-rounded single-system HPC benchmark for risk analytics

• POWER8-based Linux® systems demonstrate consistently high performance at every scale (and we explain why)

→ Current POWER8-based systems are capable and competitive platforms for computational finance
POWER8 Processor

Technology
- 22nm SOI, eDRAM, 15 ML 650mm²

Cores
- 12 cores (SMT8)
- 8 dispatch, 10 issue, 16 exec pipe
- 2X internal data flows/queues
- Enhanced prefetching
- 64K data cache, 32K instruction cache

Accelerators
- Crypto & memory expansion
- Transactional Memory
- VMM assist
- Data Move / VM Mobility

Caches
- 512 KB SRAM L2 / core
- 96 MB eDRAM shared L3
- Up to 128 MB eDRAM L4 (off-chip)

Memory
- Up to 230 GB/s sustained bandwidth

Bus Interfaces
- Durable open memory attach interface
- Integrated PCIe Gen3
- SMP Interconnect
- CAPI (Coherent Accelerator Processor Interface)

Energy Management
- On-chip Power Management Micro-controller
- Integrated Per-core VRM
- Critical Path Monitors
Simplified POWER8 Core: SMT2, SMT4, SMT8

Execution units are split between UniQs (thread sets)

Thread sets only dispatch to 'their' UniQ

Register file/renames shared equally by thread set

Up to 8 threads in 2 thread sets;
Dispatch/complete up to 3 non-branch + 1 branch per cycle per thread set
STAC® and the STAC-A2™ Benchmark

- Securities Technology Analysis Center (STAC)
  - Coordinates the STAC Benchmark Council™
  - Publishes and audits several benchmarks covering technologies important to capital markets

- STAC-A2 Benchmark: Risk Management
  - Computes numerous sensitivities (Greeks) of a multi-asset, exotic call option (lookback, best-of) with early exercise
  - Modeled by Monte Carlo simulation of the Heston stochastic volatility model
  - Priced using the Longstaff-Schwartz algorithm
  - Greeks computed using finite differences

The STAC-A2 benchmark suite measures the performance, scalability, quality and energy efficiency of any hardware/software system that is capable of implementing the benchmark specification. In this presentation we focus only on the performance and scalability benchmarks.
STAC-A2 Benchmark Dimensions and Challenges

- **Performance/Scalability Dimensions**
  - *Assets*: The number of correlated assets: $O(assets^2)$
  - *Timesteps*: Granularity of discretization: $O(timesteps)$
  - *Paths*: Number of Monte Carlo Paths: $O(paths)$

- **Challenges**
  - Unit-normal random number generation
  - Dense matrix operations (custom correlation routine, beaucoup ILP)
  - SQRT/DIVIDE-intensive Monte Carlo kernel (little ILP)
  - Cache-efficient data management
  - Efficient numerical methods (custom SVD routine)
  - Bandwidth-limited performance in certain benchmarks
  - Single-system parallel load balancing
IBM STAC-A2 Solution Structure

Monte Carlo simulation produces a number of *paths x timesteps* arrays (scenarios) which must be stored for later analysis.

Finite difference example:  

\[ \Theta = \frac{y_{\Delta t} - y}{\Delta t} \]

- \( y \) is the unmodified option value scenario
- \( y_{\Delta t} \) is the modified expiration scenario

A master thread (a) spawns multiple worker threads that perform Monte Carlo simulation in parallel (b). Simulation is partitioned by paths; Each thread creates the same path-partition of every array. Scenarios are (generated and) priced by cohorts of 1 or more threads (c). Finally the master thread computes the finite differences (d).
### Key “Greeks” Workloads + Characterization

<table>
<thead>
<tr>
<th>Workload</th>
<th>Goal</th>
<th>Assets</th>
<th>Timesteps</th>
<th>Paths</th>
<th>Result</th>
<th>Scenarios</th>
<th>Memory*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>Speed</td>
<td>5</td>
<td>252</td>
<td>25K</td>
<td>0.317 s</td>
<td>93</td>
<td>2.5 GiB</td>
</tr>
<tr>
<td>Large Problem</td>
<td>Speed</td>
<td>10</td>
<td>1260</td>
<td>100K</td>
<td>28.9 s</td>
<td>308</td>
<td>143 GiB</td>
</tr>
<tr>
<td>Asset Scaling</td>
<td>Max Assets</td>
<td>*</td>
<td>252</td>
<td>25K</td>
<td>78</td>
<td>15642*</td>
<td>309 GiB</td>
</tr>
<tr>
<td>Path Scaling</td>
<td>Max Paths</td>
<td>5</td>
<td>252</td>
<td>*</td>
<td>28M</td>
<td>93</td>
<td>2.8 TiB</td>
</tr>
</tbody>
</table>

* Values unique to our solution

Approximate Profile Breakdown by Application Phase, POWER8 S824*

- Longstaff-Schwartz Pricing
- Array Transpose
- Monte Carlo (Heston Model)
- RNG + Correlation

* 2 x POWER8 @ 3.52 GHz (nominal)/ 3.925 GHz (turbo); 24 total cores; 1 TiB memory; IBM XL C/C++; RHEL 7 Big-endian
Memory Bandwidth Sensitivity

POWER8 S824 Relative Performance vs. Available Memory Bandwidth*

- **Baseline**
- **Large Problem**
- **Asset Scaling (65 Assets)**
- **Path Scaling (8M Paths/STC)**

- Full: 192GB/sec per socket
- Half: 96GB/sec per socket
- Quarter: 48GB/sec per socket

*Not a STAC-A2 benchmark
Performance Comparisons – STAC-A2 Benchmarks

**Legend**

<table>
<thead>
<tr>
<th>Legend</th>
<th>STAC SUT*</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM150305</td>
<td>2 x IBM POWER8 @ 3.52 GHz; 24 total cores; 1 TiB Memory</td>
<td></td>
</tr>
<tr>
<td>INTC151028</td>
<td>2 x Intel Xeon E5-2697 v3 @ 2.6 GHz + 2 x Intel Xeon PHI 7120P @ 1.24 GHz; 28 Haswell cores + 122 PHI cores; 256 GiB Memory</td>
<td></td>
</tr>
<tr>
<td>INTC150811</td>
<td>4 x Intel Xeon E7-8890 v3 @ 2.50 GHz; 72 total cores; 1 TiB Memory</td>
<td></td>
</tr>
<tr>
<td>NVDA141116</td>
<td>1 x NVIDIA Tesla K80 GPU; Supermicro SYS-2027GR-TRFH Host; 128 MiB Memory</td>
<td></td>
</tr>
<tr>
<td>INTC140815</td>
<td>2 x Intel Xeon E5-2699 v3 @ 2.3 GHz + 1 x Intel Xeon PHI 7120A @ 1.24 GHz; 36 Haswell cores + 61 PHI cores; 256 GiB Memory</td>
<td></td>
</tr>
<tr>
<td>INTC140814</td>
<td>2 x Intel Xeon E5-2699 v3 @ 2.3 GHz; 36 total cores; 256 GiB Memory</td>
<td></td>
</tr>
</tbody>
</table>

*For details see [www.stacresearch.com](http://www.stacresearch.com)<STAC SUT>*

Intel, Xeon, PHI, NVIDIA, Tesla and Supermicro are trademarks or registered trademarks of their respective owners and are used for identification purposes only.
Performance by SMT Mode

SMT Mode Performance Relative to SMT1, STAC-A2 Baseline Greeks Workload*

*Not a STAC-A2 benchmark
System-Level Scalability Comparisons

Scaling Single Core Performance to Half/Full System Performance*
(Higher is Better, 1.0 = Ideal)

*Data is from the official STAC-A2 audit reports for these systems, however these are not considered STAC-A2 benchmarks. Please see page 10 for full details of the systems being compared here.
Optimizing Transpose for Monte Carlo Simulation

Straightforward (i.e., not using complex blocking schemes), cache-efficient Longstaff-Schwartz pricing effectively requires transposing time-major data into path-major storage.

<table>
<thead>
<tr>
<th>Data generated</th>
<th>Simulation</th>
<th>t&lt;sub&gt;0&lt;/sub&gt;</th>
<th>t&lt;sub&gt;timesteps-1&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>time major</td>
<td>path 0</td>
<td></td>
<td>path N-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data analyzed</th>
<th>Pricing</th>
<th>t&lt;sub&gt;0&lt;/sub&gt;</th>
<th>t&lt;sub&gt;timesteps-1&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>path major</td>
<td>path 0</td>
<td></td>
<td>path N-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Our Solution:

A blocked, in-place matrix transpose using no additional storage, taking advantage of the fact that (post-processed) simulation data is created one path (pair) at a time

Simple and relatively efficient despite:

- Moderately poor locality
- Requires a second pass over the data

Simple heuristics based on working set size improve performance for large working sets

2% - 9% of total benchmark run time

Finally
Fast, Parallel Longstaff-Schwartz Algorithm

- Least-Squares Monte Carlo (LSMC or Longstaff-Schwartz) is an algorithm for valuing early-exercise options.

- LSMC operates serially, backwards in time.

- At each time step, LSMC estimates the optimal exercise value by least squares regression using a cross section of simulated data, approximating this value as a linear combination of basis functions.

- We describe a fast, easily parallelizable LSMC method based on a QR factorization algorithm specific to row Vandermonde matrices.

- This approach is used for “microbenchmarks” and for Path Scaling, where utilizing every thread optimizes performance.

- This approach was inspired by NVIDIA’s description of their LSMC algorithm for STAC-A2.

- Paths are partitioned by thread similar to Monte Carlo simulation

- Threads synchronize (twice per timestep here) and complete each timestep in lock-step

- Scalability is limited by synchronization and communication overheads

- For the Baseline case (25,000 paths) performance improves for up to 16 – 20 SMT4 threads per scenario
Future Work

- Exploiting the high-bandwidth CPU ↔ GPU NVLINK™ unique to future OpenPOWER systems

*Current OpenPOWER Foundation Gold Members

Special Notices

Information in this document concerning non-IBM products was obtained from the suppliers of these products or other public sources. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. Send license inquires, in writing, to IBM Director of Licensing, IBM Corporation, New Castle Drive, Armonk, NY 10504-1785 USA.

All statements regarding IBM future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only.

The information contained in this document has not been submitted to any formal IBM test and is provided "AS IS" with no warranties or guarantees either expressed or implied.

All examples cited or described in this document are presented as illustrations of the manner in which some IBM products can be used and the results that may be achieved. Actual environmental costs and performance characteristics will vary depending on individual client configurations and conditions.

Any performance data contained in this document was determined in a controlled environment. Actual results may vary significantly and are dependent on many factors including system hardware configuration and software design and configuration. Some measurements quoted in this document may have been made on development-level systems. There is no guarantee these measurements will be the same on generally-available systems. Some measurements quoted in this document may have been estimated through extrapolation. Users of this document should verify the applicable data for their specific environment.

A full list of U.S. trademarks owned by IBM may be found at: http://www.ibm.com/legal/copytrade.shtml.

Other company, product and service names may be trademarks or service marks of others and are used for identification purposes only.
Backup
POWER8 Simultaneous Multi-threading (SMT)

- SMT modes represent different partitioning of core resources: Fetch/Dispatch; Reg. Files; Execution Pipes

- SMT mode is determined by the number of active hardware threads/core, regardless of the number of threads configured per core
  - SMT mode switch is automatic as threads enter/exit idle states
  - An enhancement over POWER7

- SMT modes: SMT1, SMT2, SMT4 and SMT8 entered when 1, 2, 3 – 4, 5 – 8 threads are active respectively

- Most STAC-A2 benchmarks run in SMT4; Some SMT2
  - SMT4 here means we bind 4 software threads to 4 of 8 configured HW threads per core
Simplified POWER8 Core: SMT1

Execution units: Pairs of Double Precision Floating Point pipes implement 2-way DP SIMD; 1 Fixed point Unit, 1 Load-Store Unit, 1 Load Unit per side.

Note each DP FP can also function as 2 x Single-Precision FP

Unified issue queues; BRanch and Condition Register have private queues; Crypto issued from either UniQ

Register files/renames are mirrored in SMT1 allowing use of either UniQ

Dispatch/Complete up to 6 non-branch + 2 branch per cycle
Optimizing Data Transpose for Monte Carlo Sim.

Straightforward (i.e., not using complex blocking schemes), cache-efficient Longstaff-Schwartz pricing effectively requires transposing time-major data into path-major storage. Traditional out-of-place rectangular transpose requires too much memory. Traditional in-place transpose is too slow.

Simple Insertion of time-major data into a path-major array is a disaster!

- Cache lines only partially populated when touched
- Line may be evicted to memory before being touched again
- Each line is in a unique virtual page

Instead we use the blocked approach illustrated on the next slide.

Etc.
Illustration of In-Place Blocked Transpose for a 16x16 Block-Row

Path pairs (original + antithetic) are simulated together and inserted into the blocked (16 x 16) and padded destination array block-row such that the blocks are transposed. (Eventually) transposing the blocks then brings the data into the correct orientation.
Any technique other than the Simple Insertion gives good performance, and all techniques benefit from manual prefetching. 2% - 9% of total benchmark run time is currently spent doing this transpose (varies by workload).

If the working set (16 paths per simultaneously generated array, per thread, per core) appears to fit in the 8 MiB L3, it is advantageous to transpose each block-row as soon as it is filled, otherwise it is better to defer the final block transpose until the entire array is filled.

Our latest results also suggest that Blocked Insertion with Immediate Transpose is more efficient than using a library out-of-place transposition routine (DGETMO) for L3-contained working sets. [Each thread transposes the section of the array it created.]

† Not a STAC-A2 Benchmark  * Discussed in the paper
Parallel Longstaff-Schwartz Algorithm

- After simulation each (complex) scenario is reduced to a single value using the Longstaff-Schwartz algorithm

  \[
  \text{Scenario Array} \quad (\text{paths} \times \text{timesteps}) \quad \xrightarrow{\text{Longstaff-Schwartz}} \quad 3.276517
  \]

- Multi-threaded parallelization is needed in cases where the number of threads exceeds the number of scenarios
  - “Microbenchmarks”, e.g., Delta, where 96 threads price 10 scenarios
  - Path Scaling: 93 scenarios are divided into 5 partitions due to limited memory; Partitions contain 10 to 40 scenarios

- We may also use parallelization heuristically to improve load balancing, even when \text{scenarios} > \text{threads}
Longstaff-Schwartz (Least-Squares Monte Carlo [LSMC]) in a Nutshell

- Prior to expiration the holder will exercise an in-the-money option if the future discounted cash flow from holding the option is expected to be less than the current value.

- The value of the option is maximized if the exercise happens as soon as this is true.

- LSMC estimates the future value by least squares regression using a cross section of simulated data, approximating this value as a linear combination of basis functions.

- LSMC is robust to the choice of basis functions; STAC-A2 specifies the use of polynomial basis functions.

Analysis performed for each timestep. The Singular Value Decomposition (SVD) approach to least-squares is preferred for numerical stability.
Longstaff-Schwartz Parallelization Schemes

By Timestep

- Threads “leapfrog” backwards in time, computing SVD in parallel
- Final evaluation is still serial: Thread must wait for next timestep to complete before evaluating the regression
- Scalability severely limited by ratio $T_{SVD} : T_{eval}$

By Path-Partition

- Paths are partitioned by thread similar to Monte Carlo simulation
- Threads synchronize (twice per timestep here) and complete each timestep in lock-step
- Scalability is limited by synchronization and communication overheads
- For the Baseline case (25,000 paths) performance improves for up to 16 – 20 SMT4 threads per scenario
Path-Parallel Least Squares Solution from QR Factorization and SVD

\[
A_{\text{paths} \times n} = \begin{bmatrix}
1 & p_i & p_i^2 & \cdots & p_i^{n-1} \\
\vdots & & & & \\
\end{bmatrix}
\]

\[
A_{\text{paths} \times n} = Q_{\text{paths} \times n} R_{n \times n}
\]

\[R = U \Sigma V^T\]

\[A = (QU) \Sigma V^T\]

\[x^T = b^T[(QU) \Sigma^{-1} V^T] = b^T_{\text{paths}} W_{\text{paths} \times n}\]

\[x^T = [b^T(1)b^T(2)\ldots b^T(N)] \times \begin{bmatrix}
W(1)_1 & W(1)_2 & \cdots & W(1)_n \\
W(2)_1 & W(2)_2 & \cdots & W(2)_n \\
\vdots & \vdots & \ddots & \vdots \\
W(N)_1 & W(N)_2 & \cdots & W(N)_n
\end{bmatrix}\]

\[\text{• Design matrix } A: \text{ STAC-A2 specifies polynomial basis functions, hence } A \text{ is a row Vandermonde matrix}\]

\[\text{• The QR factorization of } A. \text{ This is path-parallelizable in general for any basis functions, however we use a fast QR factorization for Vandermonde matrices requiring trivial parallel communication}\]

\[\text{• The SVD of } R\]

\[\text{• The SVD of } A\]

\[\text{• Solution for coefficients } x^T \text{ minimizing } \|Ax - b\|_2\]

\[\text{where } b \text{ is the discounted future cash flow}\]

\[\text{• Both } b^T \text{ and } W \text{ can be path-partitioned between } N \text{ threads. Each thread } j \text{ computes } n \text{ partial coefficients}\]

\[x_i^T(j) = b^T(j)W(j)_i \text{ for } i=1,\ldots,n\]
High-Level Sketch – Per-thread Operations

Loop over timesteps:

\[
M_{i,j} = \sum_{k = \text{firstPath}_j}^{\text{lastPath}_j} p_k^{i-1} \quad \text{for} \quad i = 1, \ldots, 2n - 1
\]

---------- Thread Join ----------

\[
R = f(M_1, \ldots, M_{2n-1})
\]

\[
W(j) = g(A(j), \text{SVD}(R))
\]

\[
x_i^T(j) = b^T(j)W(j)_i \quad \text{for} \quad i = 1, \ldots, n
\]

---------- Thread Join ----------

Use the regression

\[
A(j)x
\]

to compute the new future value

- Each thread \(j\) computes the sums of the first \(2n-1\) powers of its share of prices for in-the-money paths.

- Each thread \(j\) combines the partial sums to construct \(R\) and its SVD, which is then used to create partial coefficients.

- In total, 3 passes are made over the price data at each timestep, stressing caches and memory bandwidth as the number of paths per partition increases.

- The working set here consists of 3 column vectors: The price, the future value, and an index vector recoding in-the-money paths.
All-Threads Cohorts vs. Small-Threads Cohorts

• All-Threads-Cohort (ATC) mode – using all threads to price each scenario – significantly improves performance and reduces memory bandwidth dependency vs. using cohorts with smaller numbers of threads (Small-Threads-Cohort, STC) for large numbers of paths.

• Speedups are primarily due to better load balancing.

• Memory effects are due to smaller cache footprints and ideal NUMA locality.

*Not a STAC-A2 Benchmark
Least Squares Fitting

- Given observed pairs \((x_i, y_i)\), find a polynomial model which describes the relationship.

\[
y = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \cdots + a_n \cdot x^n
\]  
(1)

- Expanding into matrix format

\[
\begin{bmatrix}
1 & x_1 & \cdots & x_1^n \\
1 & x_2 & \cdots & x_2^n \\
\vdots & \vdots & \ddots & \vdots \\
1 & x_M & \cdots & x_M^n \\
\end{bmatrix}
\begin{bmatrix}
a_0 \\
a_1 \\
\vdots \\
a_n \\
\end{bmatrix}
=
\begin{bmatrix}
y_1 \\
y_2 \\
\vdots \\
y_M \\
\end{bmatrix}
\]  
(2)

- Or in matrix form:

\[
A \cdot x = b
\]  
(3)

- Least squares solution minimizes the residue

\[
\arg \min_x ||r||^2_2 = \arg \min_x ||A \cdot x - b||^2_2
\]  
(4)
Solutions to Least Squares Problems

- Requires more observed data points than order of polynomial: $M \gg n$. Overdetermined problem.
- Skipped existence and uniqueness ......
- Several methods to solve the LS problems. For example, pseudo-inverse method:
  \[ A^T A \cdot x = A^T b \]  
  \[ A^T A \] is square and fully ranked, we can invert it to compute $x$:
  \[ x = (A^T A)^{-1} A^T b \]  
- Works fine for small $n$. Issues when $n$ gets large: lots of data movement; bad condition number if not properly scaled
- Better: SVD (singular value decomposition)
  \[ A = U \Sigma V^T \]  
- Both $U$ and $V$ are unitary, and $\Sigma$ is diagonal. Much more stable:
  \[ x = V \Sigma^{-1} U \cdot b \]
Singular Value Decomposition via QR decomposition

- One path of SVD is through the QR decomposition

\[
A = Q \begin{bmatrix} R \\ 0 \end{bmatrix}
\]  

(9)

- \(Q\) is orthogonal and \(R\) is upper triangular
- Classic methods: Householder transformation or Givens rotation. Similar to Gaussian elimination for matrix factorization, but use orthogonal transformations
- Very much a serial operation

\[
\begin{bmatrix}
a_{11} & a_{12} & \cdots & a_{1n} \\
a_{21} & a_{22} & \cdots & a_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
a_{n1} & a_{n2} & \cdots & a_{nn}
\end{bmatrix}
\rightarrow
\begin{bmatrix}
a_{11}^* & a_{12}^* & \cdots & a_{1n}^* \\
0 & a_{22}^* & \cdots & a_{2n}^* \\
\vdots & \vdots & \ddots & \vdots \\
0 & a_{n2}^* & \cdots & a_{nn}^*
\end{bmatrix}
\]  

(10)
Demeure’s QR Factorization Method

- Take advantage of the special structure of the $A$ to speed-up the QR factorization.
- For LS problems, the $A$ matrix is a Vandermonde matrix.
- Sketch of the strategy
  - The QR factorization is
  
  $A = Q\Sigma B^T$  

  with $Q$ orthogonal, $B^T$ upper triangular with 1 on diagonal, $\Sigma$ diagonal with real values.
  - From Eqn. (11), if $E$ is inverse of $B^T$.

  $AE = Q\Sigma$  

  (12)

  - If $H = A^TA$:

  $HE = B\Sigma^2$  

  (13)

  - Also

  $A^TQ = B\Sigma$  

  (14)

  - From Eqn. (13), recursively compute $E$ as column space of $H$; use $E$ to compute $Q$ from Eqn. (12); use $Q$ to compute $B$ from Eqn. (14)
Matrix represented by vector $p$

$$A = \begin{bmatrix}
1 & p_1 & p_1^2 & \cdots & p_1^{n+1} \\
1 & p_2 & p_2^2 & \cdots & p_2^{n+1} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & p_M & p_M^2 & \cdots & p_M^{n+1}
\end{bmatrix} = \begin{bmatrix}
1 & p & p^2 & \cdots & p^{n+1}
\end{bmatrix} \quad (15)$$

1: procedure \textbf{VANDERMONDEQR} \hfill $\triangleright$ initialization
2: $\sigma_1 \leftarrow m$
3: for $j$ in 2, \ldots, $2n - 1$ do
4: $B_{j,1} \leftarrow \|p^{j-1}\|_1/\sigma_1$
5: $\mu_1 \leftarrow B_{21}$
6: $\nu_1 \leftarrow B_{31}$
7: $\sigma_2 \leftarrow \sigma_1(\nu_1 - \mu_1^2)$
Flow (cont’d)

1: procedure VandermondeQR

2: for $j$ in $3,\ldots,2n-2$ do

3: \[ B_{j,2} \leftarrow (\sigma_1/\sigma_2)(B_{j+1,1} - \mu_1B_{j,1}) \]

4: \[ Q::,1 = 1/\sqrt{\sigma_1} \]

5: \[ Q::,2 = (p - \mu_1)/\sqrt{\sigma_2} \]

6: for $k$ in $3,\ldots,n$ do

7: \[ \mu_{k-1} \leftarrow B_{k,k-1} \]

8: \[ \nu_{k-1} \leftarrow B_{k+1,k-1} \]

9: \[ \sigma_k \leftarrow \sigma_{k-1}(\nu_{k-1} - \nu_{k-2} + \mu_{k-1}(\mu_{k-2} - \mu_{k-1})) \]

10: for $j$ in $k+1,\ldots,2n-k$ do

11: \[ B_{j,k} \leftarrow (\sigma_{k-1}/\sigma_k)(B_{j+1,k-1} - B_{j,k-2} + \]

\[ + (\mu_{k-2} - \mu_{k-1})B_{j,k-1}) \]

12: \[ Q::,k \leftarrow (\sqrt{\sigma_{k-1}/\sigma_k}) \cdot ((p + \mu_{k-2} - \mu_{k-1}) \cdot Q::,k-1 - \]

\[ - \sqrt{\sigma_{k-1}/\sigma_{k-2}} \cdot Q::,k-2) \]

13: \[ \Sigma \leftarrow \text{diag}(\sqrt{\sigma_1},\sqrt{\sigma_2},\ldots,\sqrt{\sigma_n}) \]

14: \[ B \leftarrow B_{1:n,:} \]
Characteristics

- Many operations are local, no large amount of data movements
- Can be parallelized easily