Bond Portfolio Analysis with Parallel Collections in Scala

Ron Coleman
Computer Science Department
Marist College
Poughkeepsie, NY, United States

Udaya Ghattamanei
Computer Science Department
Marist College
Poughkeepsie, NY, United States

Mark Logan
Computer Science Department
Marist College acronyms
Poughkeepsie, NY, United St

***Abstract*— In this paper, we report the results of new experiments that test the performance of Scala parallel collections to find the fair value of riskless bond portfolios using commodity multicore platforms. We developed four algorithms, each of two kinds in Scala and ran them for one to 1024 portfolios, each with a variable number of bonds with daily to yearly cash flows and 1 year to 30 year. We ran each algorithm 11 times at each workload size on three different multicore platforms. We systematically observed the differences and tested them for statistical significance. All the parallel algorithms exhibited super-linear speedup and super-efficiency consistent with maximum performance expectations for scientific computing workloads. The first-order effort or "naïve" parallel algorithms were easiest to write since they followed directly from the serial algorithms. We found we could improve upon the naïve approach with second-order efforts, namely, fine-grain parallel algorithms, which showed the overall best, statistically significant performance, followed by coarse-grain algorithms. To our knowledge these results have not been presented elsewhere.**

Keywords- parallel functional programming; parallel processing; multicore processors; Scala; computational finance.

I. INTRODUCTION

A review of the high performance computing literature suggests opportunities and challenges to exploit parallelism to solve compute-intensive problems. [1] [2] [3]. Proponents of functional programming have long maintained that elaboration of the lambda calculus lends itself to mathematical expressiveness and avoids concurrency hazards (e.g., side-effects, managing threads, etc.) that are the bane of shared-state parallel computing. [4] Yet parallel functional programming has remained largely outside the mainstream programming community. [5] One could conceivably argue that parallel functional programming was ahead of its time and the era of inexpensive multicore processors in which some investigators have observed that the "free lunch is over" since clock speeds have been decreasing or at least not increasing significantly, necessitating a turn toward parallel programming. [6]

Enter Scala [7], a relatively new, general-purpose language which runs on the Java Virtual Machine (JVM) and hence, desktops, browsers, servers, cell phones, tablets, set-tops, and lately, GPUs [8] [9] [10], a related topic we do not explore here (see the section, "Conclusions and Future Directions"). Scala blends object-oriented and functional styles with shared-nothing, task-level parallelism based on the actor model. [7] Parallel collections [11] [12] are recent additions that provide data-level parallelism [3] through a simple, functional extension of the ordinary, non-parallel collections of Scala. While the use of parallel collections has potential to improve programmer productivity and greatly facilitate a transition to parallel programming, no independent study has investigated whether parallel collections scale in terms of run-time performance on commodity hardware, taking into account furthermore end-to-end processing that involves I/O which is typically a prerequisite for and often the bottleneck of practical applications.

Coleman, et al., conducted end-to-end experiments to find the fair value of riskless bond portfolios using task-level parallelism via map-reduce. [13] [14] In this paper, we take a new, different tack on the same problem that applies data-level parallelism via parallel collections. We were motivated to use bond portfolio analysis, first, because computational finance workloads can be very large. [15] Second, bond portfolio pricing theory is fairly transparent. [16] Finally, bonds inform or are closely related to other financial instruments, including annuities, mortgage securities, bond derivatives, and interest rate swaps, which are among the most heavily traded financial contracts in the world. [17] Thus, computational methods and performance results from this class of problem would likely have implications beyond bonds and finance.

Indeed, the experiments with Scala parallel collections using eight algorithms on three different hardware platforms show super-linear speedup and super-efficiency are consistent with the maximum performance expectations for scientific computing workloads. While the data suggests that the more modern processors are also more efficient, overall fine-grain algorithms significantly outperform others in runtime, which interests and surprises us considering the presumed overhead of this approach. The coarse-grain algorithms are next best, followed by the "naïve" algorithms. The findings we report here using parallel collections are new and have not been reported elsewhere or by others. All the source code is available online for review, download, and testing (see section, "Appendix – Source Code").

II. METHODS

A. Parallel collections – a primer

Scala has standard, template data structures called *collections*, which include lists, arrays, ranges, and vectors, among others. Scala collections are different from the ones it also inherits from the Java standard library in that the Scala

versions are typically immutable with methods to operate on the data elements using functional objects. For instance, to multiply every element of a range collection by two using the map method, we have the snippet below (where "scala>" is the Scala interactive shell prompt):

```
scala> (1 to 5).map(x => x * 2)
Vector(2, 4, 6, 8, 10)
```

Snippet 1. Maps sequential range.

The parameter, x => x * 2, an *anonymous function literal object*, receives each element of the range collection as an immutable value parameter, x, multiplies it by two, and map copies the result into a new collection, Vector.

The methods of a parallel collection as accessed in the same way except the method name is preceded by .par as the snippet below suggests:

```
scala> (1 to 5).par.map(x => x * 2)
ParVector(10, 6, 2, 8, 4)
```

Snippet 2. Maps the parallel range.

Here map invokes the function literal object on the range using the machine's parallel resources. The parallel collections map method returns a parallel vector, ParVector, in which the ordering of the return results is unspecified because of the asynchronous nature of parallel execution. From a programmer's point of view, virtually no effort is involved to parallelize the code. There are no new programming constructs to learn and apply and algorithm redesign and code refactoring are not demanded. There is furthermore no need to write special test cases to verify the results since in principle the serial (non-parallel) implementation is the test case. While the result ordering may need to be addressed, in general, parallel collections are a potential windfall for programmer productivity and transitioning to parallel programming.

The research question is whether use of .par scales, enabling speed-up and efficiency on a non-trivial problem on commodity hardware. For bond portfolio analysis, the functional nature of parallel collections makes implementation of the pricing equations straightforward. In the "naïve" case, we simply reuse the pricing function object from the serial algorithm with no other changes to the code other than to apply .par, just as we did in the above snippet. However, we go further and explore whether we can obtain further improvements using fine-grain and course-grain algorithms.

B. Pricing theory

For purposes of this paper, we are considering only simple bonds [16] b_i, defined by the five-tuple:

$$b_i = [i, C, n, T, M] \tag{1}$$

i is an integer which plays no part in bond pricing except to uniquely identify the bond in an inventory which we describe below; C is the coupon amount paid one or more times; n is the payment frequency of coupons per annum; T is the time to maturity in years; and M is the face value due at maturity. The sum of the net present value of these cash flows, C and M, is the fair value of the bond. Thus, the fair value, $P(b_{i,}r)$, of a bond, b_i, is the net-present value of its cash flows which functionally defined as:

$$P(b_i, r) = \sum_{t=1}^{n \times T} \frac{C}{(1+r_t)^{t/n}} + \frac{M}{(1+r_T)^T} \tag{2}$$

The parameter, r, is the time-dependent yield curve, the general discussion of which is beyond the scope of this paper. Without loss of generality, we use the United States Treasury on-the-run bond yield curve, which we observe once. We interpolate between the tenors (i.e., Treasury maturity dates) using polynomial curve fitting, the coefficients of which we cache and apply for all bonds in the inventory.

A portfolio is a collection instruments, in our case, bonds. The fair value, $P(\phi_j)$, of a portfolio, ϕ_j, with a basket of Q bonds is functionally defined as follows:

$$P(\phi_j) = \sum_{q=1}^{Q} P(b_{\phi(j,q)}, r) \tag{3}$$

C. Bond portfolio generation

We generate simple bonds that model a wide range of computational scenarios. The goals are to 1) produce a sufficient number of bonds to mimic realistic fixed-income portfolios and 2) avoid biases in commercial-grade bonds that depend on prevailing market conditions. Specifically, we have the collections, $\vec{n}$ ={1, 4, 12, 52, 365), $\vec{T}$ ={1, 2, 3, 4, 5, 7, 10, 30}, and $\vec{\delta}$ ={0.005, 0.01, 0.02, 0.03, 0.04, 0.05}, where the elements of $\vec{n}$ are payment frequencies, $\vec{T}$ are maturities, and $\vec{\delta}$ are coefficients. We derive the parameters for a bond object from the bond generator equations below:

$$M = 1000 \tag{4a}$$
$$n = \vec{n}\,[\bullet] \tag{4b}$$
$$T = \vec{T}\,[\bullet] \tag{4c}$$
$$C = \mathrm{M} / \mathrm{T} \times \vec{\delta}\ [\bullet] \tag{4d}$$

where • is an integer uniform random deviate in the range of [0, s-1]; and s is the size of the respective collection. We invoke Equations 4a - 4d a total of 5,000 times to produce the bond inventory, V, which we store in an indexed persistent database that we describe below.

We generate a portfolio by first selecting its size, that is, the number of bonds, Q, per the equation below.

$$Q = v + \sigma \times \eta \tag{5}$$

η is a Gaussian deviate with mean of zero and one standard deviation. v and σ are configurable parameters set to 60 and 20, respectively. Finally, we construct a basket of size, Q, bonds for a portfolio, ϕ_j. We use the equation below to specify a bond id or primary key,

$$i = \bullet \tag{6}$$

where • is an integer uniform random deviate in the range of [1,|V|] and |V|=5,000 is the size of the bond inventory. We generate a universe, U, of bond portfolios where |U|=100,000.

The bond portfolios are also store in a database indexed by *j*, a unique portfolio id.

D. Database design

We store the bonds, b_i, portfolios, ϕ_j (which also contains the result of Equation 3) in MongoDB, an indexed, document-oriented, client-server database. [18] As we noted above, ϕ_j does not contain bond objects, b_i, but the bond primary key, *i*. In MongoDB parlance, the bonds are linked to portfolios rather than embedded by them. In other words, the database is organized in third-object normal (3ONF) form. [19] Thus, to evaluate Equation 3, a total of 2+Q accesses are necessary: one access to fetch ϕ_j; *Q* fetches to retrieve each b_i; and finally, one store to update the portfolio, ϕ_j, with its price. The figure below gives the class diagram, as it is stored in the document repository.

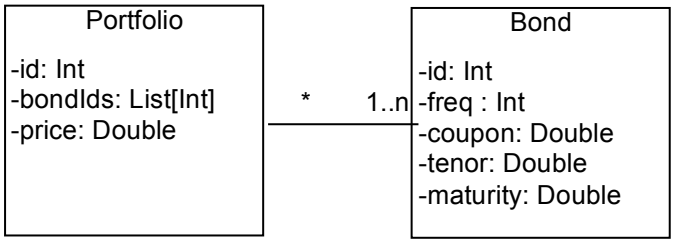

Figure 1. Third normal object form (3ONF) of the database

Although this design is consistent with best practices for data modeling, we could reduce the number of database accesses at the expense of redundancy through denormalization. However, we decided to forgo this optimization in the interests of establishing a baseline of performance for future reference.

E. Algorithms

We develop two classes of algorithms: serial and parallel. There are three types of parallel algorithms, "naïve," fine-grain, and coarse-grain. Each serial and parallel algorithm comes in two kinds: composite and memory-bound. The composite kind, represented by the notation, {io+compute}, overlaps access to the database while evaluating Equation 2 and Equation 3. The memory-bound kind, represented by the notation, {io}+{compute}. In other words, we measure I/O ({io}) and compute ({compute}) runtimes separately, first caching all the bonds by portfolio into memory and only then evaluating Equation 2 and Equation 3. I/O ({io}) and compute ({compute}) runtimes furthermore provide insight into the maximum compute and IO performance potentials. In each case, the algorithms evaluate the same collection of portfolios, $U' \subset U$, which has been randomly sampled from the database. We give here only snippets from the source code. See the appendix to access the complete source.

F. Serial algorithms

We invoke the composite serial algorithm as the snippet below suggests.

```
val outputs = inputs.map(price)
```

Snippet 3. Maps input of randomly sampled portfolio key ids to price results

The object, inputs, is a collection of portfolio ids and outputs is a collection of portfolio prices. (The "val" declaration means that outputs is an immutable value object.) The parameter, price, is a *named function object* with the declaration:

```
def price(input: Data): Data
```

Snippet 4. Price the collection of randomly sampled portfolio ids serially

This means price receives a Data object as an input parameter and returns a Data object. We wrote the Data object for use by all the algorithms of this study. It contains the portfolio id, a list of bonds, and a result object which itself contain the portfolio price and diagnostic information about the run. On input in this case, the Data object has set only the portfolio id. On output, Data has the portfolio id and the result object defined.

The function object, price, accesses the 3ONF repository to retrieve a portfolio by its id and then retrieve the bond objects, pricing them according to Equation 2, then according to Equation 3 summing the prices using the foldLeft method. (For readers who may be unfamiliar with functional programming, "folding" is a common operation in functional programming for aggregating elements. The foldLeft method is a serial aggregator, traversing the collection, left-to-right, that is, from the element at index zero to the element of the last index. The analogous foldRight traverses the collection from right-to-left using tail-recursion. We prefer foldLeft as opposed to foldRight to avoid the problem of stack overflow.)

The serial memory-bound algorithm is virtually identical to the composite algorithm as the snippet below suggests.

```
val inputs = loadPortfsFoldLeft(n)
val outputs = inputs.map(price)
```

Snippet 5. Serially load the bonds in memory, then price portfolios serially

The method, loadPortfsFoldLeft, loads a random sample of n portfolios from the database and uses foldLeft to aggregate the corresponding bonds. Thus, in this case, the inputs value is a collection of Data objects, each containing a list of bond objects. The parameter, price, is a function object, the same one used in the composite serial algorithm.

G. Naïve algorithms

The naïve algorithms are so-called because, as a first-order effort, they "naively" use .par. They are virtually identical to the serial algorithms. That is, we have the snippet below for the composite case.

```
val outputs = inputs.par.map(price)
```

Snippet 6. Price the collection of randomly sampled portfolio ids in parallel

We have the snippet below for the memory-bound kind.

```
val inputs = loadPortfsParFold(n)
    val outputs = inputs.par.map(price)
```

Snippet 7. First, load the bonds into memory in parallel by portfolio id, then prices the portfolios in parallel

Notice that the memory-bound kind uses loadPortfsParFold (i.e., rather than loadPortfsFoldLeft), which accesses the database and loads the portfolios in parallel using a parallel collection. It uses Scala's par.fold method. This method aggregates like its serial version, foldLeft, except par.fold does so in parallel with non-deterministic ordering.

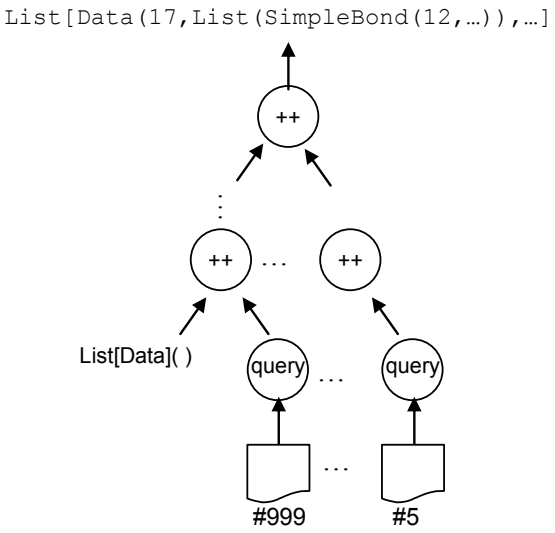

Figure 2. Parallel IO query-merge tree using the par.fold method

The figure above shows how loadPortfsParFold works. Namely, we start with an empty List collection. Here for the sake of demonstration, portfolios, #999 and #5, are being loaded into memory from the database by the "query" operation. The "++" nodes are binary operations that merge partial lists of bond objects until a complete list is merged at the root in O(log N) time. At the top of the merge tree we have the fully merged in-memory List collection of portfolio data objects. In this depiction, the value, 17, represents a portfolio id chosen for demonstration purposes. Thus, the outer list contains portfolio data objects, each of which contains a list of bond objects. Note that this parallel memory-caching algorithm is not "embarrassingly parallel" as the data lists must be merged.

H. Fine-grain algorithms

In a second-order effort to improve the naïve application of .par, we developed fine-grain algorithms, composite and memory-bound kinds. Unlike the naïve algorithm, the fine-grain algorithm uses a parallel collection within the pricing function object. In other words, we have a parallel collection within a parallel collection.

The inner parallel collection has a bondPrice function object to price the bonds by their id (i.e., it makes a query to the database) per Equation 2 using par.map and a sum function object to reduce (i.e.., accumulate) the bond prices in parallel using par.reduce. In effect, we have the snippet below of the price function.

```
val output = input.bondsIds.par.
    map(bondPrice).par.reduce(sum)
```

Snippet 8. Price bonds in parallel by their ids then reduce prices in parallel.

Bond prices flow directly to their reduction in an O(log N) processing tree. Thus, like parallel I/O, the workload is not "embarrassingly parallel" as the figure below suggests.

The memory-bound algorithm is similar except, it uses the parallel IO query-tree to access the database and cache the bonds in memory.

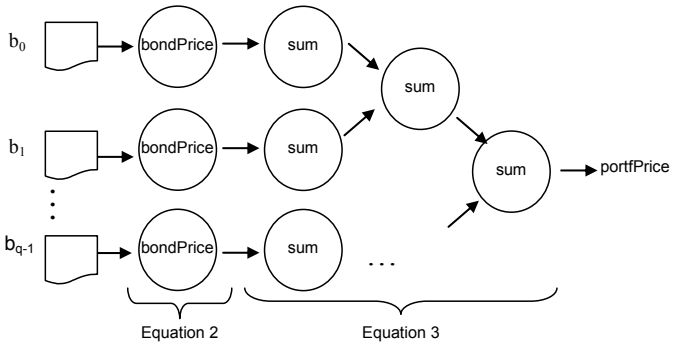

Figure 3. Accessing and pricing bonds then reducing prices in parallel.

I. Coarse-grain algorithms

The other algorithms created a parallel collection of input portfolios whose size was independent of the number of processor cores. The idea of the parallel coarse-grain algorithm is to "chunk" the portfolios as a second-order effort to the naïve application of .par. That is, we create a parallel collection whose size is proportional to the number of processors.

The design of parallel collections does not provide a direct way to bind the pricing function to a core. This is part of parallel collections design philosophy: the programmer focuses on the functional specification and the parallel collection distributes it across the cores.

Nevertheless, the programmer can control the chunk size by making the input collection a List of a List of portfolio ids. For example, for a four-core platform like the W3540 we study, the containing List has eight List elements.

Each element has |U'|/c portfolios where *c* is the number of cores. For *u*=1024 portfolios, each element in the containing is a List of 128 portfolios. The pricing function object is then passed this list with 128 portfolios, which it processes serially to evaluate Equation 2 and Equation 3.

To compute the size of the contained List, we use the Java class, Runtime. It has a method, availableProcessors( ). However, this method returns the number of hyperthreads, not the number of cores. As far as we know there is no way to get the number of core except manually from the OEM datasheets, which we rely on for calculating the efficiency (see below). Otherwise, programmatically we use the Runtime class.

The coarse-grain composite algorithm loads the bond objects by portfolio just as the naïve algorithm except it does in "chunks" on-demand. The memory-bound algorithm, like its naïve and fine-grain counterparts, uses the parallel IO query tree to cache the bonds in memory.

III. EXPERIMENTAL DESIGN

A. Environment

The test environment consisted of three hardware platforms of different Intel multicore processors. The table below shows the system configurations, with the clock speed in GHz and years of introduction by the Intel Corporation.

TABLE I. EXPERIMENTAL ENVIRONMENT

CPU	Clock	Cores	Threads	RAM	Year
W3540	2.93	4	8	4 GB	2009
i7-2670M	3.20	2	4	4 GB	2011
i3-370M	2.40	2	4	2 GB	2010

All platforms run Microsoft Windows 7. The code was compiled by Eclipse 3.7.1 using the Scala IDE plugin version 2.0.0. The code was executed with the 64-bit JVM. We used MongoDB, version 1.8.3. Although MongoDB is accessed through TCP/IP, the database server runs on the same host as the Scala code. We indexed the portfolios and bonds documents on their key ids.

B. Runs and trials

We instrument the code and make the following measurements.

TABLE II. MEASUREMENTS SOURCES

#	Algorithm kind	Measurement (T)
1	Composite	{io + compute}
2	Memory-bound	{io}
3	Memory-bound	{compute}
4	Memory-bound	{io} + {compute}

For each algorithm by its kind in Table 2, we make a total of 11 trial invocations of the code to obtain stable run-time statistics following. [20] Each trial starts a new JVM, the code of which allocates new JVM objects and opens new database connections. The trial ends when the algorithm ends and the code exits, terminating the JVM, which closes the database connections and causes the operating system to recycle the JVM objects. A given set of trials, taken together, we call a *run*. There is a run for $u=2^x$ portfolios (i.e., the problem size) where $x \in [0..10]$. The run, $u=1024$, is we call the *terminal run*. Note: #4 in Table 2 is not an actual run; it is derived by adding the measurements for #2 and #3 for the respective runs. For each run at a given problem size, we analyze the measurements for statistical significance as we describe below. We also graph the run-times using the median value of the run.

C. Speed-up and efficiency calculations

T_1 is the serial time of a serial algorithm. T_N is the time using parallel collections.

Given T_1 and T_N where N is the number of cores, we have the speedup, R:

$$R = T_1 / T_N \tag{8}$$

The efficiency, e, is

$$e = R / N \tag{9}$$

In this case, N is the number of cores, which we got from the OEM datasheets online. [21] [22] [23]

D. Statistical significance calculations

After obtaining the runtimes, we observe the differences and test them for statistical significance in the indicated direction. That is, if the median runtime of algorithm, A, is less than the median runtime of algorithm, B, we have the null hypothesis H_0:

$$H_0 : E(T^A) \geq E(T^B) \tag{10}$$

where E is expectation. To conservatively estimate the p value, we used the one-tailed Mann-Whitney test. [24] We report (see the appendix) the rank sum statistic, S,

$$S = \sum R(T_i) \tag{11}$$

where $R(T_i)$ is the rank of runtime, T_i. Since there are 11 observations for each algorithm, the one-tailed threshold for p=0.05 is the rank sum, $S_{.05}$=101. This value can be found in Table A7 in [24]. Thus, for $S < S_{.05}$, we reject H_0.

We compare each of our eight algorithms relative to one another and test the differences for statistical significance. To make the report more accessible, we give the frequency count for the number of times an algorithm is found to be statistically significantly faster than another algorithm. Again, the rank sums, S, algorithm by algorithm for each hardware platform, can be found in the appendix.

We present graphical evidence for performance over the range of u mentioned above for each algorithm on each platform. We assess the statistical significance and present tabular data only for the terminal run, u=1024.

IV. RESULTS

The table below gives the kind of algorithms symbolized in the graphs and tables that follow.

TABLE III. KIND OF PROCESSING

◆	Composite
●	Memory-bound
✳	Compute-only
△	IO-only

A. Naïve results

The results for the naïve treatments are summarized in the next three graphs, one for the W3540, i7, and i3, respectively.

The number of portfolios or problem size, is $u=2^x$.

The speedup, R, is on the left axis, and the efficiency, e, is on the right axis.

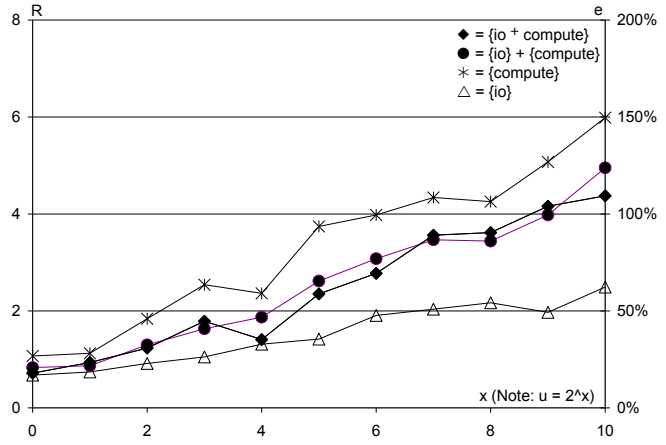

Figure 4. W3540 naïve results

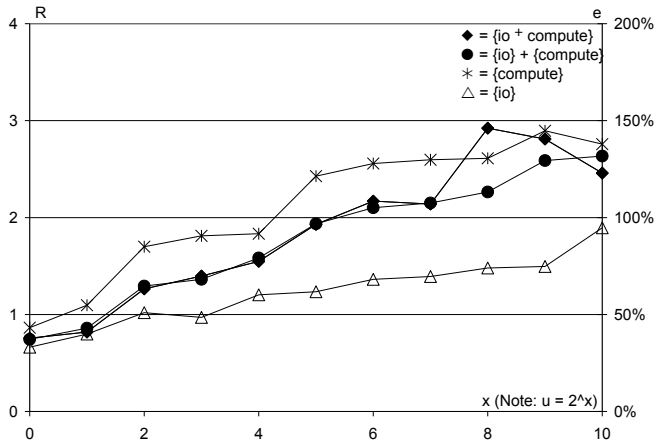

Figure 5. i7 naïve results

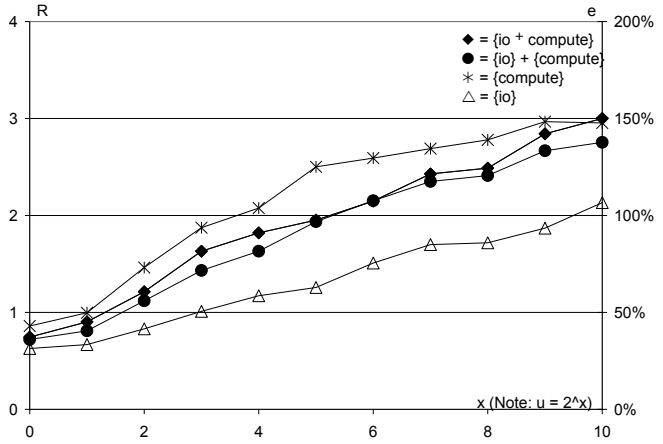

Figure 6. i3 naïve results

The table below gives the terminal run results. T_N is the median run-time in seconds.

TABLE IV. TERMINAL RUN, U=1,024, BOND PORTFOLIOS

	W3540		i7		i3	
	T_N	R	T_N	R	T_N	R
◆	14.80	4.37	19.36	2.46	23.19	3.00
●	12.39	4.95	16.93	2.63	24.17	2.75
✳	8.74	5.66	13.90	2.76	18.74	2.95
△	3.52	2.49	3.07	1.89	5.39	2.13

Note: In general, the median operator does not distribute. Namely, median({io} + {compute}) ≠ median({io}) + median({compute}). For example, for the W3540, T_N({io} + {compute}) = 12.39 whereas T_N({io}) + T({compute}) = 8.74 + 3.52 = 12.26.

B. Fine-grain results

The results for the fine-grain algorithms are summarized in the next three graphs, one for the W3540, i7, and i3, respectively.

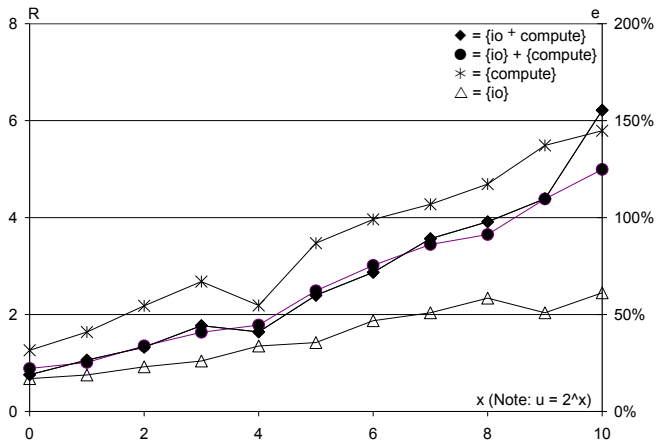

Figure 7. W3540 fine-grain results

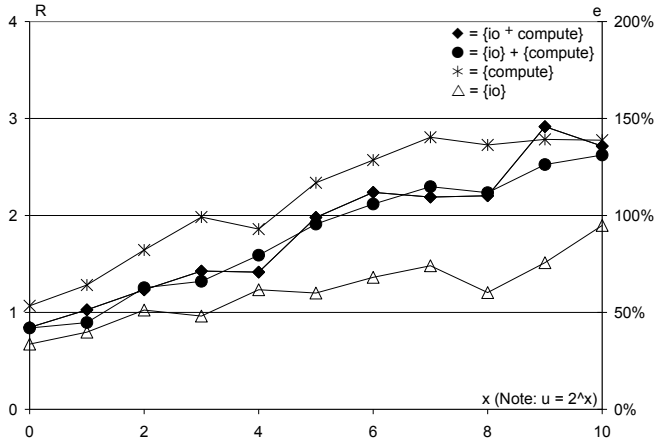

Figure 8. i7 fine-grain results

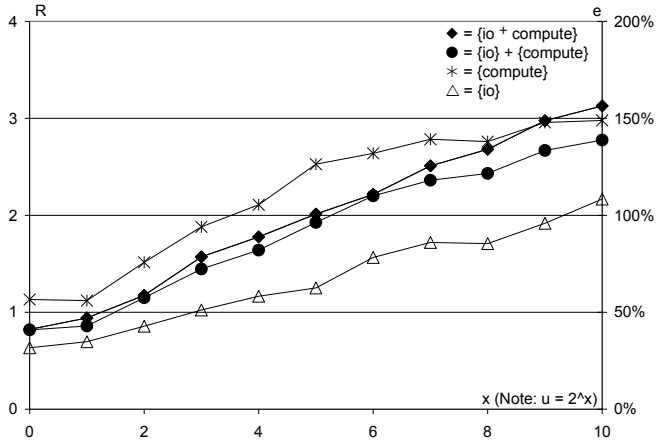

Figure 9. i3 fine-grain results

The table below gives the results for the terminal run.

TABLE V. FINE-GRAIN TERMINAL RUN, U=1,024, BOND PORTFOLIOS

	W3540		i7		i3	
	T_N	R	T_N	R	T_N	R
◆	10.42	6.21	17.53	2.72	22.24	3.13
●	12.29	4.99	17.00	2.62	23.98	2.78
✳	9.04	5.79	13.81	2.78	18.58	2.98
△	3.58	2.45	3.06	1.90	5.30	2.17

C. Coarse-grain results

The results for the coarse-grain algorithms are summarized in the next three graphs, one for the W3540, i7, and i3, respectively. Note that the algorithms are not defined for portfolios less than the number of hyper-threads.

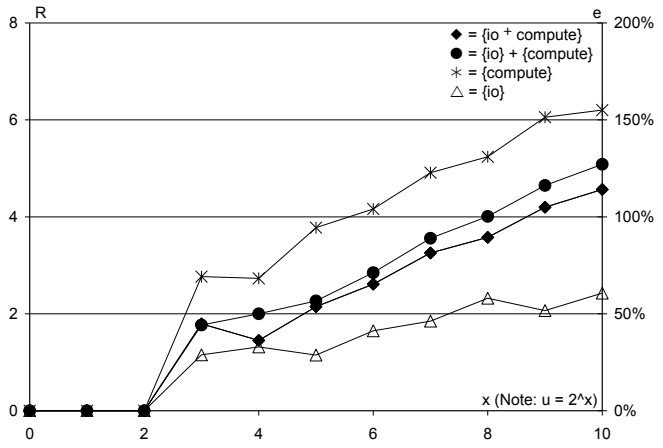

Figure 10. W3540 coarse-grain results

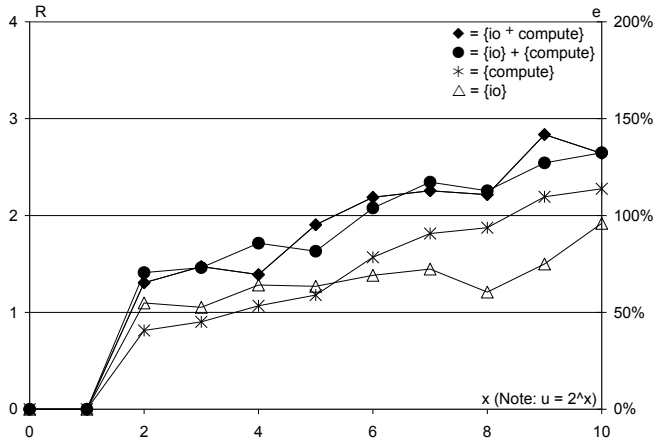

Figure 11. i7 coarse-grain results

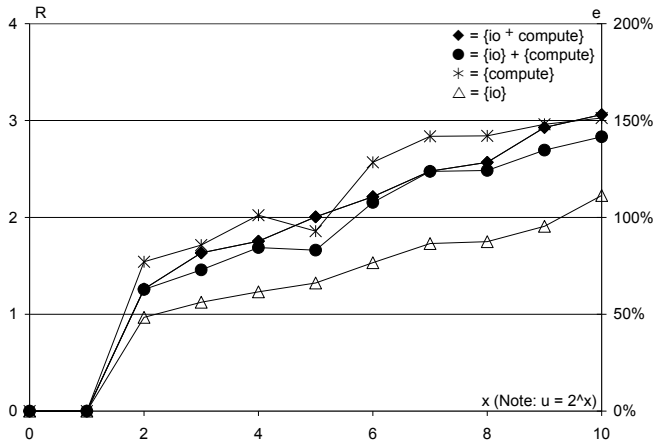

Figure 12. i3 coarse-grain results

The table below gives results for the terminal run.

TABLE VI. COARSE-GRAIN TERMINAL RUN, U=1,024, BOND PORTFOLIOS

	W3540		i7		i3	
	T_N	R	T_N	R	T_N	R
◆	14.18	4.56	18.01	2.64	22.73	3.06
●	12.07	5.08	16.84	2.65	23.50	2.83
✳	8.44	6.20	16.84	2.28	18.28	3.03
△	3.61	2.43	3.03	1.92	5.16	2.23

D. Statistical significance results

The table below gives the counts in which an algorithm is statistically significantly faster than another algorithm and kind. The details underlying this table are in the appendix, "Sorted Rank Sums."

TABLE VII. STATISTICALLY SIGNIFICANTLY COUNTS MEASURED BY FASTER RUNTIMES

Kind	Algorithm	W3540	i7	i3	Totals
◆	Serial	0	0	0	0
	Naive	2	2	2	6
	Fine	4	3	2	9
	Coarse	2	2	2	6
●	Serial	1	1	1	3
	Naive	3	2	2	7
	Fine	2	2	6	10
	Coarse	2	2	6	10
	Totals	16	14	21	

To read the above table, choose the kind of algorithm (composite vs. memory-bound) and read across for type of algorithm. For example, the composite serial algorithm ran slower than every other algorithm on the W3540, i7, or i3 platforms.

Hence, there are zero (0) values across the composite serial row. The memory-bound naive algorithm ran faster than three algorithms on the W3540 and two algorithms the i7 and i3, respectively. The memory-bound serial algorithm outperformed one algorithm on each platform: these slower algorithms were the composite serial algorithms. Evidently loading on all the portfolios into memory significantly improves even the serial performance.

See the appendix, "Sorted Rank Sums" for the specific counts.

V. Discussion

The graphs, Figures 4 – 11, show that for larger problem sizes, *u*, the composite and memory-bound algorithms performed better than I/O processing alone which is the least efficient but worse than compute by itself which is the most efficient. The slopes of these graphs generally point toward increasing speedup and efficiency for larger *u*.

Tables IV – VI show evidence for high levels of overlap between compute and I/O. For instance, the ratios of T{compute} /T {io + compute} and T{compute} / (T{io}+T{compute}) found in these tables are often around 80% or higher.

Table VII nevertheless indicates that the memory-bound algorithms tend generally to give statistically significant better runtimes compared to the composite algorithms. In other words, caching the portfolios in memory upfront seems to give better performance than loading them, as they are needed.

Table VII also suggests that the algorithms on a given platform tend to run with significantly more efficiency on the i3 across all the algorithms, followed respectively by the W3540 and the i7.

Finally, the data in Table VI show the fine-grain algorithms give statistically significantly better runtimes followed respectively by coarse-grain and the naïve algorithms across different platforms.

VI. Conclusions and Future Directions

This study has found that bond portfolio analysis using parallel collections achieve super-linear speedup and super-efficiency with as few as *u*=64 portfolios across different multicore processors. The data suggests that the "naïve" application of parallel collections can be improved significantly, foremost with the fine-grain algorithm, which we find interesting. That is, portfolio analysis is "embarrassingly parallel," but not for the fine-grain or the I/O parallel algorithms which contain inherent dependencies that necessitated the use of parallel merge-trees.

The data points toward greater speed up and efficiency for larger problem sizes, *u*>1024. The terminal run analyzed only about 1% of the portfolios. Additional research could consider how to harness multiple hosts and/or GPUs to price all portfolios.

Future work might also compare and contrast map-reduce versus parallel collections as well as possibly consider how to improve the I/O performance.

Acknowledgements

This research has been funded in part by grants from the National Science Foundation, Academic Research Infrastructure award number 0963365 and Major Research Instrumentation award number 1125520.

References

[1] Patterson, J., and Hennessy, D., Computer Architecture, Morgan Kaufman, 2006

[2] Hill, M., Marty, M., Amdahl's Law in the Multicore Era, IEEE Computer Society, Vol. 41, Issue 7, 2008

[3] Hager, G., and Wellein, G.,, Introduction to High Performance Computing for Scientists and Engineers, CRC, 2010

[4] Michaelson, G., Introduction to Functional Programming through Lamda Calculus, Dover, 2011

[5] McKenney, P. Is Parallel Programming Hard, And, If So, What Can You Do About It?, 2011, http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html, accessed: 8 April, 2012

[6] Sutter, H., The Free Lunch is Over: The Fundamental Turn Toward Concurrency in Software, Dr. Dobbs Journal, vol. 30, no. 3, 2005

[7] Odersky, M., et al., Programming in Scala, Artima, Mountain View, 2011

[8] Owens, J., et al., GPU Computing, Proceedings of the IEEE, Vol. 96, No. 5, May 2008

[9] Nystrom, N., et al., Filepile: Run-time Compilation for GPUs in Scala, GPCE '11. October 22-23, 2011, Portland, OR., U.S., 2011

[10] Das, K., GPU Parallel Collections for Scala, M.S. Thesis, University of Texas Arlington, 2011

[11] Prokopec, A., et al., A Generic Parallel Collection Framework, EPFL, InfoScience 2011, 2010-7-31, 2011

[12] Lester, B., Data parallel programming in Scala. Scala Days 2010, EPFL, Lausanne, Switzerland, 15 - 16 April 2010.

[13] Coleman, R., et al, Computational Finance with Map-Reduce in Scala, Conference on Parallel and Distributed Processing (PDPTA '12), CSREA, 2012

[14] Dean, J., and Ghemawat, S., Simplified Data Processing on Large Clusters, OSDI, 2004

[15] Tsang, E. and Martinez-Jaramillio, Computational Finance, IEEE Computational Intelligence Society Newsletter, August 2004, 3-8, 2004

[16] Fabozzi, F., and Mann, S., Introduction to Fixed Income Analytics, 2nd ed., Wiley, 2010

[17] Hull, J. Options, Futures, and Other Derivatives and DerivaGem CD Package, 8th ed. Prentice Hall, 2011

[18] Chodorow, K. and Dirolf, M., MongoDB: The Definitive Guide, O'Rielly, 2010

[19] Merunka, V., et al., Normalization Rules of the Object-Oriented Data Model, EOMAS '09 Proceedings of the International Workshop on Enterprises & Organizational Modeling and Simulation, 2009

[20] Georges, A., et al., Statistically Rigorous Java Performance Evaluation, *OOPSLA '07*, October 21-25, Montreal, Quebec, Canada, 2007

[21] Intel Corp., Intel Xenon Processor W3540, 2009, http://ark.intel.com/products/39719/Intel-Xeon-Processor-W3540-(8M-Cache-2_93-GHz-4_80-GTs-Intel-QPI) accessed: 17-April-2012

[22] Intel Corp., Core i3-370M Processor, http://ark.intel.com/products/49020/Intel-Core-i3-370M-Processor-(3M-cache-2_40-GHz), 2010, accessed: 17 April 2012

[23] Intel Corp. (2011). Core i7-2670QM Processor, 2011, http://ark.intel.com/products/53469, accessed: 17-April-2012

[24] Conover, J., Practical Non-Parametric Statistics, Wiley, 1999

Appendix -- Sorted Rank Sums

The three tables below give the sorted rank sums of runtimes according to Equation 12 for the terminal (*u*=1,024)

run. Algorithm A which has the smaller median runtime compared to algorithm B. Smaller rank sums (*S*) imply greater statistical significance. Since there are 11 trials for each algorithm, the minimum rank sum is *S*=1+2+3…11=66 (i.e., all runtimes of algorithm A are less than the runtimes of algorithm B). In this case, $p < 0.001$. The threshold for statistical significance is $S<101$ in which $p \leq 0.05$. Comparisons that are not statistically significant are not included in the tables and by implication tables with more rows imply cores with greater performance.

TABLE VIII. W3540 RANK SUMS (S) ALGORITHM A(KIND) × B(KIND)

S	Algo A	Kind	Algo B	Kind
66	Naive	composite	Serial	composite
66	Naive	mem-bound	Serial	composite
66	Naive	composite	Serial	mem-bound
66	Naive	mem-bound	Serial	mem-bound
66	Fine	composite	Serial	composite
66	Fine	mem-bound	Serial	composite
66	Fine	composite	Serial	mem-bound
66	Fine	mem-bound	Serial	mem-bound
66	Coarse	composite	Serial	composite
66	Coarse	mem-bound	Serial	composite
66	Coarse	composite	Serial	mem-bound
66	Coarse	mem-bound	Serial	mem-bound
89	Fine	composite	Naive	composite
96	Serial	mem-bound	Serial	composite
98	Fine	composite	Coarse	composite
100	Naive	mem-bound	Naive	composite

TABLE IX. I7 RANK SUMS (S) ALGORITHM A(KIND) × B(KIND)

S	Algo A	Kind	Algo B	Kind
66	Naive	composite	Serial	composite
66	Coarse	composite	Serial	composite
66	Fine	composite	Serial	composite
66	Naive	mem-bound	Serial	composite
66	Fine	mem-bound	Serial	composite
66	Coarse	mem-bound	Serial	composite
66	Naive	composite	Serial	mem-bound
66	Coarse	composite	Serial	mem-bound
66	Fine	composite	Serial	mem-bound
66	Naive	mem-bound	Serial	mem-bound
66	Fine	mem-bound	Serial	mem-bound
66	Coarse	mem-bound	Serial	mem-bound
86	Serial	mem-bound	Serial	composite
99	Fine	composite	Naive	composite

TABLE X. RANK SUMS (S) ALGORITHM A(KIND) × B(KIND)

S	Algo A	Kind	Algo B	Kind
66	Naive	composite	Serial	composite
66	Coarse	composite	Serial	composite
66	Fine	composite	Serial	composite
66	Naive	mem-bound	Serial	composite
66	Fine	mem-bound	Serial	composite
66	Coarse	mem-bound	Serial	composite
66	Naive	composite	Serial	mem-bound
66	Coarse	composite	Serial	mem-bound
66	Coarse	composite	Naive	mem-bound
66	Fine	composite	Serial	mem-bound
66	Naive	mem-bound	Serial	mem-bound
66	Fine	mem-bound	Serial	mem-bound
66	Coarse	mem-bound	Serial	mem-bound
69	Coarse	composite	Fine	mem-bound
70	Serial	mem-bound	Serial	composite
71	Fine	composite	Naive	mem-bound
75	Fine	composite	Fine	mem-bound
79	Fine	composite	Coarse	mem-bound
85	Fine	composite	Naive	composite
85	Coarse	composite	Coarse	mem-bound
100	Coarse	composite	Naive	composite

APPENDIX -- SOURCE CODE

All the source code used for this project is freely available via the Scaly project home and downloadable as an Eclipse project at http://code.google.com/p/scaly/. See the ParaBond folder and the package, scaly.parabond.test. The table below gives the algorithm and its source.

TABLE XI. SOURCE FILES

Algorithm	Kind	Scala source file
Serial	Composite	NPortfolio02
	Memory-bound	NPortfolio03
Naive	Composite	Par00
	Memory-bound	Par01
Fine	Composite	Par05
	Memory-bound	Par06
Coarse	Composite	Par04
	Memory-bound	Par07