UPC: Large-Scale Memory Efficient Java Primitive Collections

Ray Hylock*
Department of Health Services and Information Management, East Carolina University, Greenville, North Carolina, USA.

* Corresponding author. Tel.: 252-744-6184; email: hylockr@ecu.edu
Manuscript submitted October 13, 2015; accepted December 24, 2015.
doi: 10.17706/jsw.11.3.251-271

Abstract: The Java programming language is versatile and robust, however, for large-scale applications with memory intensive processes, much is still to be desired. The public Java Development Kit (JDK) supports neither forcible object destruction nor large collection sizes (limited to 231-1) as found in choice computational languages such as Fortran and C/C++. Within Java's Hotspot, however, the non-public sun.misc.Unsafe class allows such features through basic off-heap functionality. The submitted UPC collections are fully integrated with Unsafe to provide large (i.e., 263-1), destructible arrays, lists, hash sets, hash maps, and matrices, consistent with advanced computational needs. Additionally, a customized version of OpenJDK 1.9 with bulk-operation Unsafe support and companion collections are provided. These tools are compared to Java and extent third-party primitive collections. Testing indicates UPC performs in a consistent to superior manner compared to the state-of-the-art, with greater improvements realized by way of the modified virtual machine.

Key words: Destructible, java, long-indexed, OpenJDK enhancements, primitive collections.

1. Introduction

Java is one of the world's most prolific programming languages. According to Oracle, Inc. [1], over 3 billion mobile devices, 125 million televisions, and 97% of enterprise desktops are Java enabled, with more than 5 billion Java Cards currently in use. Java is a leading tool for embedded, web-based, and distributed software due to its rich feature set, which notably includes: its object-oriented paradigm; platform independence (portability); automatic garbage collection (decreasing code complexity and minimizing memory leaks); type safety; built-in networking and multithreading; extensive runtime, error checking, and application profiling to support development; and the *Just-in-Time* (JIT) compiler of the *Java Virtual Machine* (JVM), which approaches the speed of natively compiled code [2]-[15]. The developer community is continually expanding – with current estimates at 9 million – and an increasing number of universities offer Java in lieu of, for example, C/C++ [1], [12], [16]-[19]. Since 2001, the TIOBE software index, which uses search engine queries to indicate programming language popularity, has consistently ranked Java in the top two, alternating with C [20]. Trendy Skills (a website that has extracted and aggregated IT skills from job posting sites in 14 countries since 2012) indicates 1-in-5.55 IT job descriptions (number 1 by far) mention Java; 1-in-8.33 C#, 1-in-14.29 C, and 1-in-16.67 C++ [21]. With all of these benefits and large pool of developers, it is no wonder Java is transitioning into a computation language as well [13], [14], [22], [23],

[24].

Java is slowing making its way into the ranks of *scientific* and *high performance computing* (HPC) with its ever expanding capabilities. Much of the language's criticism in this environment revolves around its lack of support for basic HPC features. Whereas some traits are noticeably absent (e.g., memory management and large collections), the community has, in fact, been actively increasing Java's capabilities. A few examples are as follows:

- MPI Java implementations such as MPJ [25], mpiJava [26], MPJExpress [11], and OpenMPI (since version 1.7) [27]
- OpenMP-like packages such as JOMP [28] and JaMP [29]
- Parallel Java – advanced tools for built-in Java threads [30]
- Java Fast Sockets – high-speed socket support (e.g., InfiniBand and Myrinet) [31]
- Java bindings for accelerators such as CUDA (e.g., JCUDA [32], JaCuda [33], and java-gpu [34]) and OpenCL (e.g., jocl.org [35] and JavaCL [36])
- Java HPC and parallel computing benchmark suites like Java Grade [22] and NAS Parallel for MPJ [37]

However, as previously mentioned, for memory intensive applications, many developers are reluctant to employ Java due to its large memory footprint and lack of direct system controls in the public *Application Programming Interface* (API). A solution is found, in part, in the non-public sun.misc.Unsafe class, which provides basic support for off-heap (C heap) access. This class, however, is rudimentary, with simple operations such as allocation, destruction, getting, and setting. Thus, for one to make use of this class, entire collections must be constructed with it as its foundation. The submitted UPC collections take this approach.

The contributions of this paper are two-fold. First, we submit a suite of Java primitive collections utilizing sun.misc.Unsafe. Supported collections include arrays, lists, hash sets, hash maps, and two-dimensional matrices; each with the ability to destroy and assign 2^{63}-1 elements, and employ primitive data types. Second, we propose enhancements to sun.misc.Unsafe to offer additional low-level support for these collections. The posited modifications are implemented in OpenJDK 1.9 and a subsequent set of collections created, integrating this new functionality. Both libraries are compared to extent Java and third-party tools. These experiments include basic performance analysis as well as large-scale, realistic implementations.

The remainder of this paper is as follows. Section 2 places the problem into context. The UPC collections and modifications are disclosed in Section 3. Section 4 introduces the third party collections used for comparison. Experimental results are covered in Section 5, with concluding remarks and future work presented in Section 6.

2. Context

2.1. Java Limitations

Commonly used collections for analysis include arrays, lists, hash sets, hash maps, and matrices. Languages such as Fortran and C/C++ (popular HPC languages) provide three key advantages over Java in this arena. First, these languages allow one to reclaim memory, freeing space for proceeding processes. Java has garbage collection (GC), however, it is never guaranteed to run or free memory upon execution and the collection event itself is quite slow and resource intensive [8], [38]-[43]. Second, Java limits arrays and collections to 2^{31}-1 entries (i.e., length of a Java int). This might appear to be sufficient and for Java's intended audience of desktop, web, and distributed systems developers it is, however, a limitation of roughly two billion is actually quite restrictive in scientific computing; thus the reason 2^{63}-1 is supported by other languages. Lastly, with the exception of primitive arrays, Java's collections are entirely object-based, incurring a memory storage (as objects require a reference object) and retrieval penalty (i.e., two

sequentially allocated objects are not guaranteed to be in adjoining memory locations). Popular computational languages have the ability to reserve contiguous blocks of memory for primitive types, providing both speed and size benefits.

2.2. Proposed JDK and JVM Improvements

There are several projects of interest Oracle and the OpenJDK community are actively developing in hopes of assuaging some of the aforementioned limitations. First is *Arrays 2.0*, which is supposed to handle 2^{63}-1 array elements (i.e., a Java long index). The current (and only) commit to the OpenJDK repository is a set of JDK classes employing multidimensional arrays of int-bound blocks [44], [45]. Being a JDK-only construct, it cannot be manually destroyed, and as multidimensional arrays are object-based, it consumes more memory. *Project Valhalla* (expected with Java 10 in 2018), is schedule to bring *generic specialization* and *value types* to the language [46]. Generic specialization will allow primitives (or more specifically value types) as generics, providing for the creation of primitive-like collections [47]. Value types essentially bridge the gap between primitive (e.g., int) and reference types (e.g., Integer). Integer, for example, can be a value type by removing all Object features. This allows for defining one's own "primitives" and even grants access to machine-specific data types [48]-[50]. These projects, however, are only in the "incubator" stage, with no guarantee they will end up in a released version of the JDK/JVM.

2.3. The sun.misc.Unsafe API

Within the sun.misc non-public Java API resides the Unsafe class. This provides low-level access to basic system functions such as memory allocation and destruction. As it is not part of the public API, developers are cautioned from integrating it into their work as it may change at any time without warning and might not be reverse compatible. However, it is heavily integrated into both public ((prefix of java.) lang, math, net, nio, io, and util) and non-public ((prefix of sun.) awt, cobra, font, invoke, java2d, management, misc, nio, reflect, security, and swing) JDK classes [51], and is deeply entrenched in high-performance open source and commercial tools (e.g., Apache DirectMemory [52], Heliosearch/Solr [53], VoltDB [54], ChronicleMap [55], Guava [56], and even a patent (EP2755129 A1) on creating a new type of ByteBuffer for maps [57]), indicating its necessity. An ad-hoc survey of developers using Unsafe, called for by Paul Sandoz of Oracle, indicates 63.6% (n = 316) of respondents currently use the class for off-heap memory operations to, among other things, reduce garbage collection and efficiently manage memory layout (the number one feature-based result). Additionally, 88.6% are prepared to switch from Unsafe to a safe JDK version when one presents itself [58]-[60].

Oracle has signaled its intent to remove access to Unsafe in future Java versions; enforced primarily by Project Jigsaw's modular source design [58], [59], [61]. However, its use is so critical to developers, the JDK will incrementally replace these functions with safe, internal JDK options using VarHandle's as defined in *Java Enhancement Proposal* (JEP) 193 Enhanced Volatiles for Project Valhalla [46], [58], [59], [62].

3. UPC — Unsafe Primitive Collections

The UPC library[1] offers developers large, memory-efficient, primitive collections. Achieved through sun.misc.Unsafe, UPC interacts with the C heap, allowing users to allocate and destroy long-indexed primitive arrays, lists, hash maps, hash sets, and two-dimensional matrices (stored in linearized form); thus overcoming the limitations discussed in Section 2.1. UPC is available in stock and modified (UPCM) versions.

The stock version functions on standard Java, integrating six Unsafe methods. For allocating and

[1]UPC is available from http://blog.ecu.edu/hsimcomputationallab/?p=19.

destroying collections, allocateMemeory and freeMemory are used. Bulk operations are achieved through setMemory and copyMemory. Lastly, getting and setting single entries is done through get<Type> and put<Type>, where <Type> refers to the data type – byte, char, double, float, int, long, or short. Therefore, any version of Java supporting these operations can employ the stock UPC library. For example, the authors have successfully experimented with UPC on Windows, Mac, and Linux systems operating Oracle and OpenJDK 1.7 and 1.8 – Linux includes OpenJDK 1.9 as well.

The modified version requires a customized OpenJDK implementation, which provides advanced, native support for bulk operations – detailed in Section 3.2. Its intent is to offload repetitive JNI calls to the JVM, providing a performance boost for tasks such as sorting and searching (refer to Section 5 for experimental results). These enhancements are not limited in use to UPC; they are intended as general-purpose methods, easily integrated with existing Unsafe code.

UPC provides support for the following in-built data types: boolean (stored as a byte); byte; char; double; float; int; long; short; and ASCII_8, ASCII_16, ASCII_32, UNICODE_16, and UNICODE_32 strings. Pertaining to the string options, we implement our own string type using a UPC list. Consequently, the string types are "primitive" in nature, necessitating significantly less space than java.lang.String and allow destruction. Section 3.1 elaborates further on string types.

3.1. UPC Strings

UPC strings are a stored collection of contiguous characters. The total number of characters for the sum of all strings is 2^{63}-1, as a single UPC list is used for its storage – a limitation not soon eclipsed. An array/list of pointers maintains the collection-to-string mappings. Fig. 1 depicts the basic string data structure. Take pointer index 0 as an example. The value at index 0 is the start position in the character list; 0 in this example. The first entry is the number of characters to read (the string length), immediately followed by said characters. Thus, 11 characters are read after index 0 (i.e., 1-11), producing "Hello world." UPC supports both ASCII (stored as bytes) and UNICODE (stored as chars) characters, defined at collection instantiation. Operations such as reverse, swap, and sort modify only the index containing the pointers. For example, swapping "Hello world" with "example" in Fig. 1, will see the pointer index at 0 set to 136 and that of 10 to 0.

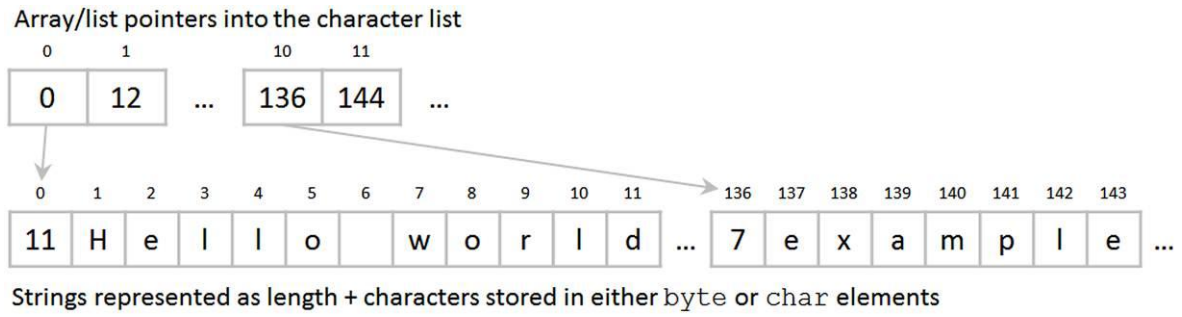

Fig. 1. Generalized diagram of a string in UPC.

The numeric designations represent the maximum string length in bits, which is a multiple of the storage type. Take UNICODE_32 for example. Since UNICODE values are stored in a UPC char list, the length variable has a default maximum of 2 bytes (16-bits). The value 32 indicates each UNICODE string can store a maximum of 2^{32}-1 characters (i.e. 2 chars for length instead of 1). Therefore, the presented options offer additional memory optimization based on known character set and string lengths. Fig. 2 presents an example where a UPC byte list (for ASCII strings) stores nucleotide characters for sequences reaching two bytes in equivalent length (i.e., a maximum length of 2^{16}-1 instead of the default 2^{8}-1 – ASCII_16).

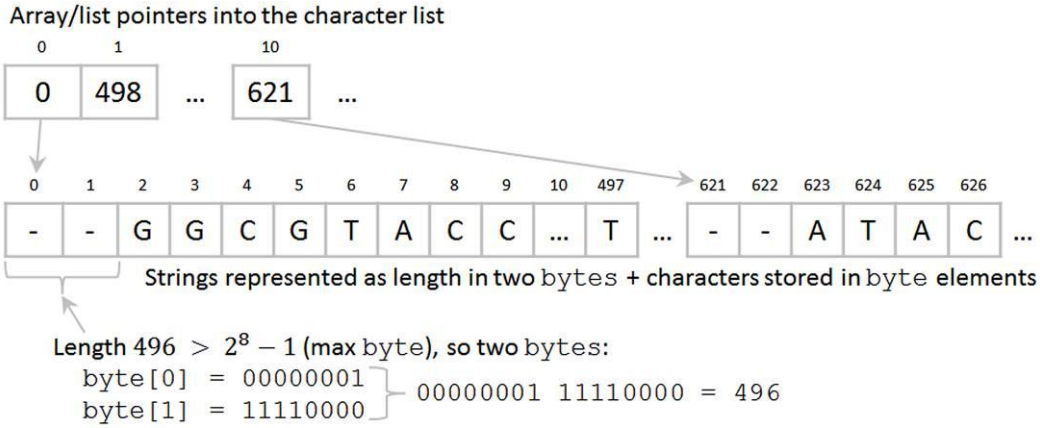

Fig. 2. ASCII_16 UPC string diagram using nucleotides in a byte character list.

Since strings are stored as contiguous blocks within a list, removing and replacing can lead to excessive memory waste. Therefore, UPC implements a reclamation process as follows. If a string is to be removed, then the position in the character list is marked as free. If a string is to be replaced, there are three options: (1) if the new length ≤ the existing length, then replace the characters and adjust the length variable, (2) if the new length > the existing length, mark the current area as free and try to reclaim another free space (by default it will look for a spot ≤ 2 times the new length), and (3) if reclamation is not possible, append the characters to the list. Strings being added will try to reclaim space before appending.

3.2. OpenJDK Expansion

Whether simple (e.g., get) or complex (e.g., rehashing), all operations in Unsafe interface with basic I/O calls such as "get" and "put." Each call is a JNI request, which, though imperceptible for a few calls, leads to performance degradation in large sets. Therefore, bulk operations performed by UPC have been ported to the development version of OpenJDK 1.9's Unsafe class.[2] Within this environment, one JDK and four JVM "share" files[3] were modified, meaning the alterations are operating system neutral, limiting the amount of work to reproduce these features in future releases and guaranteeing portability across all Java supported operating systems and CPU architectures. The features are as follows:

- A memory zeroing allocator – An Unsafe memory allocation does not zero memory, forcing the user to manually clear the range – a potentially expensive operation (e.g., if one "puts" zeros instead of setting bytes using Unsafe.setMemory) and easy to overlook. This addition employs Java's Copy::zero_to_bytes method, which does this at the OS level (very fast) and is the method utilized by Java's zeroing collections.
- Quicksort and counting sort – Basic non-recursive quicksort (for int-sized and larger data types) and counting sort (for boolean, byte, char, and short).
- Linear and binary search – Performs said searches over a memory range.
- Set and remove all – Sets or removes all values in a memory range.
- Reverse – Reverses all values in a memory range.
- Contains all – Checks to see if one memory range contains all of the values in another.
- Rehash – Rehashes primitive hash maps and hash sets when growing.

[2]OpenJDK 1.9 was updated as it represents the next iteration of Java and how this functionality could be included. However, OpenJDK 1.8u60 and 1.7u60 have both been successfully modified, with no performance degradation between versions.

[3]OpenJDK 1.9 source files ({base} = hotspot/src/share/vm/): {base}/prims/Unsafe.cpp, {base}/c1/c1_GraphBuilder.cpp, {base}/classfile/vmSymbols.hpp, {base}/opto/library_call.cpp, and jdk/src/java.base/classes/sun/misc/Unsafe.java

Only zeroing, quicksort, and reverse are currently supported UPC string JDK/JVM extensions – future modifications are planned.

One question that might arise is why extend the JDK/JVM as opposed to creating a stand-alone JNI library? Our reasoning is as follows. First, we use internal JVM methods to provide consistent and safe operations (e.g., copy, memory barriers, and atomic access). Second, compilation of the proposed modifications within the JDK/JVM results in uniform optimization and error checking. Third, as Java is portable, integration results in enhanced portability and reduction in architecture-specific implementations (again, we only modify "share" classes, relying on Java's architecture-specific rendering of methods). Lastly, as a proof of concept, illustrating how Java could natively support such collections.

3.3. UPC Sorting Algorithms

UPC implements counting sort for data types of size less than int, and a non-recursive quicksort algorithm for all others – non-recursive as very large sets quickly fill the recursive stack, causing Java heap space errors. As the JDK also relies on counting sort for the same primitive types, these times are directly comparable. However, Java uses timsort for all other requests. Timsort ranges in performance from $O(n)$ to $O(n\log n)$, whereas a two-partition quicksort (as UPC uses) from $O(n\log n)$ to $O(n^2)$ – both average $O(n\log n)$. So why the choice of quicksort over timsort? Simple, JNI calls. Quicksort requires considerably fewer JNI calls than timsort, which actually provides a performance boost when using Unsafe. The modified JVM also contains quicksort to allow stock to modified analysis.

4. Third-Party Primitive Collections

There are many third-party primitive collections available; most suffer from the 2^{31}-1 size and destruction limitations. To the best of our knowledge, there has been only one previously released tool utilizing the Unsafe class for off-heap storage, JlargeArrays. In the following section, the employed libraries for comparative purposes are detailed. These tools include Apache Commons 1.0 [63], CarrotSearch 0.6.0 [64], Colt 1.2.0 [65], FastUtil 6.5.15 [66], Java 1.8/OpenJDK 1.9 [51], [67], JLargeArrays 1.4 [68], Koloboke 0.6.5 [69], and Trove 3.2 [70]. Table 1 presents a comparison matrix of data types supported by the aforementioned packages and the proposed UPC set. Table 2 depicts available collections, memory capabilities, and hashing mechanisms.

Table 1. Data Type Comparison Matrix

	boolean	byte	char	double	float	int	long	short	string
Apache Commons 1.0	byte[1]	✓	✓	✓	✓	✓	✓	✓	-
CarrotSearch 0.6.0	byte[1]	✓	✓	✓	✓	✓	✓	✓	✓[2]
Colt 1.2.0	✓[3]	✓[3]	✓[3]	✓[3]	✓[3]	✓[3]	✓[3]	✓[3]	✓[2,3]
FastUtil 6.5.15	✓	✓	✓	✓	✓	✓	✓	✓	✓[2]
Java 1.8/OpenJDK 1.9	✓[2]	✓[2]	✓[2]	✓[2]	✓[2]	✓[2]	✓[2]	✓[2]	✓[2]
JLargeArrays 1.4	bit/byte[1]	✓	int[1]	✓	✓	✓	✓	✓	✓
Koloboke 0.6.5	char[1,3]	char[3]	✓[3]	long[3]	int[3]	✓[3]	✓[3]	✓[3]	✓
Trove 3.2	byte[1]	✓	✓	✓	✓	✓	✓	✓	✓[3]
UPC 1.0	✓	✓	✓	✓	✓	✓	✓	✓	✓

[1] No direct support for boolean, but it can be simulated using the proposed data type
[2] Java Object support
[3] Limited collection support

Apache Commons Primitives 1.0 utilizes Java primitive arrays as its underlying storage medium. Only its list collection overlaps with UPC.

CarrotSearch High Performance Primitive Collections 0.6.0 uses Java primitive arrays for storage. Overlapping collections include lists, hash sets, and hash maps. HPPC offers support for java Objects.

Colt 1.2.0 uses Java primitive arrays for storage. Overlapping collections include lists, hash maps (int, double, and long as key and/or value, and Object as value), and matrices (double or Object only – data stored in an array, so limit is 2^{31}-1 (e.g., 46,340 for square matrices)). Colt supports java Objects.

FastUtil 6.5.15 uses multi-dimensional primitive arrays to simulate large arrays (same premise as Arrays 2.0). Overlapping collections include arrays, lists, hash sets, and hash maps. FastUtil supports for Objects.

JLargeArrays 1.4 uses Unsafe with destroy option (sun.misc.Cleaner). If the size is less-than 2^{30}, a plain Java array is used. JLargeArrays, as its name suggests, supports only arrays. The tool contains its own string type. The string character array is instantiated to a predefined maximum length, thus, each is exactly the same size in terms of memory, making random access possible without a pointer. This option, however, consumes a lot of memory and imposes an a priori string length limit.

Koloboke 0.6.5 is a highly optimized and fast hash map and hash set tool. It employs primitive arrays for all storage, utilizing Unsafe on-heap operations in certain situations to improve performance (e.g., bulk set operations). Gains are attributable to its "parallel" collections and combined storage of keys and status elements. Parallelization is achieved with one array for storing like keys and values (i.e., same key-value data types or identical storage sizes such as char-short, float-int, and double-long). Even array indices contain keys and odd (key+1) the values – thus "parallel." For those not meeting this definition, keys and values are stored in two distinct arrays ("separate"). To minimize size and lookups, the status element is kept in the keys array.

Trove 3.2 uses Java primitive arrays for storage. Overlapping collections include lists, hash sets, and hash maps, with Object support in hash maps.

Table 2. Feature Comparison Matrix

	Collections					Memory		Hashing	
	Array	List	Hash Set	Hash Map	Matrix	2^{63}-1	Delete	Addressing	Probe
Apache Commons 1.0	-	✓	-	-	-	-	-	-	-
CarrotSearch 0.6.0	-	✓	✓	✓	-	-	-	Open	Linear
Colt 1.2.0	-	✓	-	✓[1]	✓	-	-	Open	Double
FastUtil 6.5.15	-	✓	✓	✓	-	✓[2]	-	Open	Linear
Java 1.8/OpenJDK 1.9	✓	✓	✓[3]	✓	✓[4]	-	-	Chaining	
JLargeArrays 1.4	✓[5,6]	-	-	-	-	✓[6]	✓	-	-
Koloboke 0.6.5	-	-	✓	✓	-	-	-	Open	Linear
Trove 3.2	-	✓	✓	✓	-	-	-	Open	Double
UPC 1.0	✓[6]	✓[6]	✓[6]	✓[6]	✓[6]	✓[6]	✓[6]	Open	Linear

[1] Limited support: int/int, int/double, double/int, int/Object, or long/Object
[2] Multi-dimensional array simulates a large capacity – same architecture as Arrays 2.0
[3] Implemented using HashMap by setting the value to an empty Object
[4] Multi-dimensional arrays can simulate matrices
[5] For length less-than 2^{30}, regular Java arrays are employed
[6] Uses sun.misc.Unsafe

5. Evaluation

Evaluations proceed in two phases. First, libraries are compared using synthetic data, providing feature-based insights. Second, UPC and select collections are employed in realistic problems, emphasizing scale and complexity. Each method is discussed in detail in its respective section.

Testing ensued on two machines. First, a 64-bit Linux Mint 17.1 system with 8GB of memory and a

quad-core 3.4 GHz i7 3770 processor, relaying performance details on a standard desktop configuration. The second is a 64-bit Rocks 6.1 cluster node with 64GB of memory and two eight-core 2.6GHz Xeon E5-2670 processors, to provide insight into scaling. Direct library comparisons were done on the Linux Mint machine, with realistic experiments performed on both. Not conveyed in the analysis, due to space limitations, is the totality of experiments to show full cross-platform support for the UPC stock collections on Windows and Mac OS's. Suffice it to say, the UPC collections performed in like manner across all systems.

When size (a collection's memory footprint) is reported, it requires more than simply asking the JVM how much heap space it consumed, as UPC and JLargeArrays are off-heap. Therefore, the following command was executed to determine heap space utilization: ps -F -C java, where the program searches for the entry containing the correct JAR.

5.1. Effects of Destroying Objects

Destruction of objects is not explicitly explored; however, the effects are felt throughout experimental results. Despite JLargeArrays' ability to free memory, it does so only to clean up after itself; thus, no advantages are gleaned during use. UPC, on the other hand, benefits greatly from removing temporary and residual structures during, for instance, sorting (arrays and lists), growing (lists, hash sets, and hash maps), and clearing (all) operations. The obvious gains are found in final memory utilization, as other collections might still be waiting for the garbage collector to free space. In a more varied fashion, time is also affected, especially when approaching the maximum heap size. Using the construction of a large list as an example, the garbage collector is continually freeing pre-growth lists at large sizes during enlargement. The GC, however, is untargeted (meaning it tries to free space throughout the JVM, not just the list in question) and has been shown to increase runtime considerably [8], [38]-[43]. Therefore, direct memory freeing, as done by UPC, can have a profound impact on runtime.

5.2. Library Comparisons

In this section, abbreviated package comparison results are discussed. First, arrays, lists, hash sets, and hash maps are detailed using byte, double, int, and string data types, and matrices double – note, value types in hash maps mirror their key counterpart (invoking Koloboke's parallel optimization technique), with the exception of string, which has an int value type to conserve space. The justification for each data type is as follows. A byte is the smallest whole unit (i.e., it is possible to consider bits, but you must interact with the entire byte first), and should therefore provide speed and a small memory footprint. A double is both large and complex (in terms of representation); it therefore requires the most overhead of the primitive types. An int is a very common type and is between a byte and a double in terms of overhead. Finally, it is necessary to compare our string data type to existing object and character-based approaches. Unless otherwise noted, each randomly generated string has a maximum of 50 characters.

Collections considered are those in Table 2. Experiments for byte, double, and int have sizes 100,000, 1,000,000, 10,000,000, and 100,000,000. Since strings necessitate greater space, experiments terminated at 10,000,000. The matrices are square at 1,000, 10,000, and 20,000. Each test was performed 10 times with average and standard deviation recorded for various, collection-specific aspects. UPC modified results are reported for the expanded elements detailed in Section 3.2. Of note, collections lacking sort capabilities may have deflated memory footprints owing to the lack of relevant sorting data structure instantiations.

5.2.1. Arrays

The metrics recorded for array evaluation are add, read all, overwrite all, sort, and size, for Java, JLargeArrays, and UPC; results shown in Fig. 3. Add simply records the amount of time required to insert the desired number of elements (excluding random value generation time). Read all indicates the speed at which the collection can be accessed (each character for strings) and overwrite all, modified. Sorting follows

the rules specified in Section 3.3. Lastly, the size of the final collection is reported. JLargeArrays does not provide a sorting mechanism, and is thus excluded from those results. For these measures, UPC modified is limited to sort for non-string types.

Universally, UPC requires the least amount of time for add and overwrite all, whereas Java and JLargeArrays overlap. At 100M, UPC is 15-17% for byte, double, and int, and 601 % for 10M string faster when adding data. For overwrite all, byte, double, and int are 19-24% and string 979% quicker. Likewise, UPC presents the minimum memory footprint by roughly 1-6% for byte, double, and int, and 15-112% for string. Read all for byte, double, and int are dominated by the 0 millisecond reads of Java, followed by UPC and JLargeArrays. For string, JLargeArrays is up to 78% and Java 62% slower than UPC. UPCM is the top collection for sorting across all data types proceeded by Java and UPC. UPCM bests Java by upwards of 17% for byte, 8% for double, and 14% for int, and UPC by 21%, 31%, and 32%; it is slightly better than Java (near 3%) and significantly faster than UPC (roughly 25%) for string sorting.

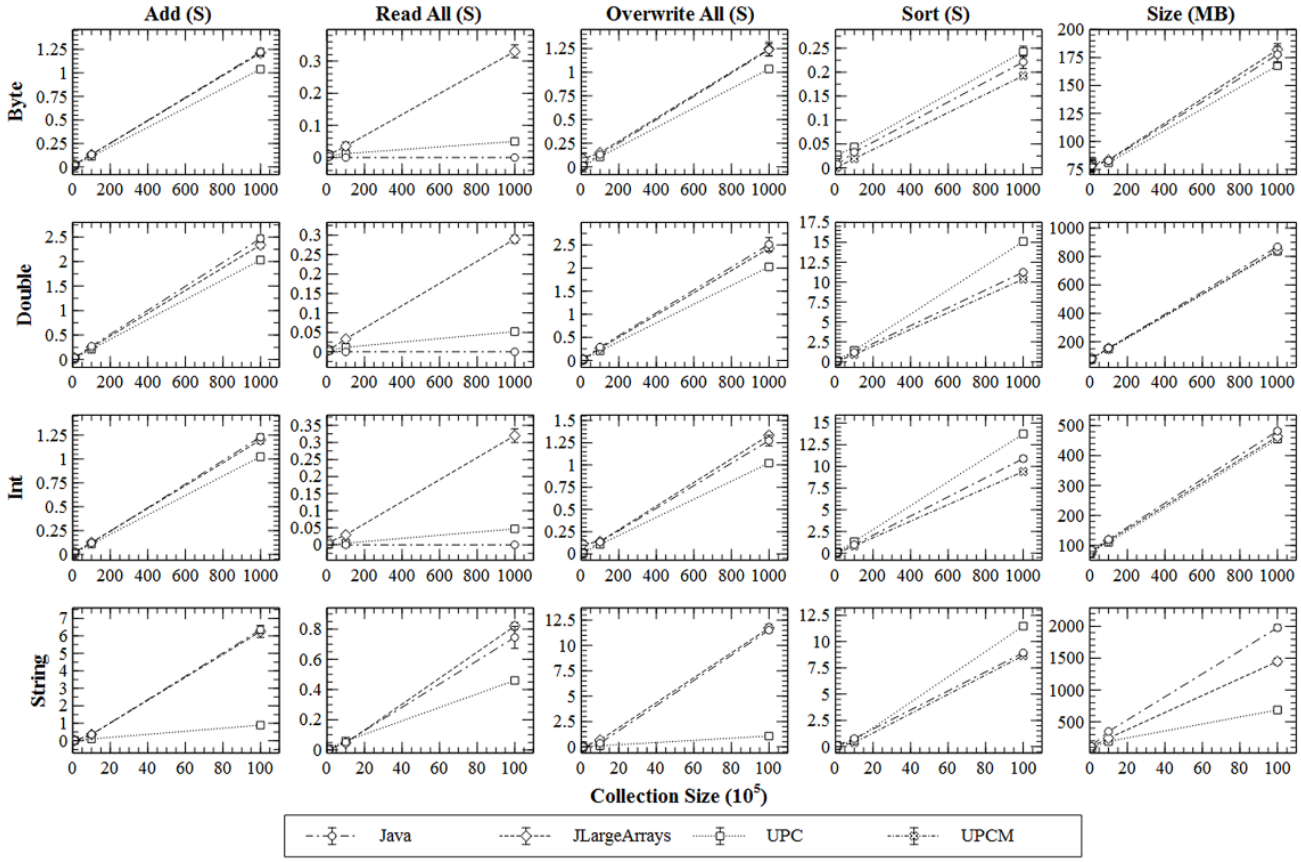

Fig. 3. Array experiment results – not shown in the plots: JLargeArrays *sort* as this feature is unsupported.

5.2.2. Lists

Six of the eight libraries compared to UPC support basic list collections. As with arrays (Section 5.2) add, read all, overwrite all, sort, and size were measured; key results noted in Fig. 4. Apache Commons, CarrotSearch for byte and string, and FastUtil,[4] do not provide internal sorting algorithms; thus, they are excluded from those results. For these measures, UPC modified is limited to sort for non-string types.

In general, the collections performed comparably across the byte tests; only UPC's size, which was anywhere from 4% to 51% smaller than the next best, stood out.

[4]The underlying multidimensional array can be sorted, however, the quicksort implementation does not discriminate between the used and unused portion (i.e., entered versus the default zero values), leading to unexpected results.

Overwrite all is omitted from Fig. 4 due to uniform results leading to difficulty in visual discernment. Java requires about 2,000-4,000% more time than the other collections for double and int. For string, UPC is 39.45% (5.08±0.05 seconds) faster than the next best Colt (8.39±0.69 seconds) at 100M, with negligible differences prior.

UPC outperforms all collections in adding content to lists by 8-23%. For reading all data, CarrotSearch is faster than UPC for double and int by roughly 30%, but measurements are in tens of milliseconds at 100M. UPC outpaces Java for string by upwards of 35%. Both Trove and Colt sort doubles more rapidly than UPCM (3-5%) and UPC (26-27%). For int and string, UPCM is slightly quicker than Trove (no string support) and Colt at around 1-2%, with UPC about 21-35% slower. As stated in Section 3.3, poor sort performance was anticipated for UPC due to the amount of JNI calls and inferior algorithm, but these results also highlight the expense of crossing the JNI barrier as identically implemented quicksort methods in UPC and UPCM produce drastically different runtimes. In terms of size, UPC dominates with a memory footprint roughly 50-60% that of the next best.

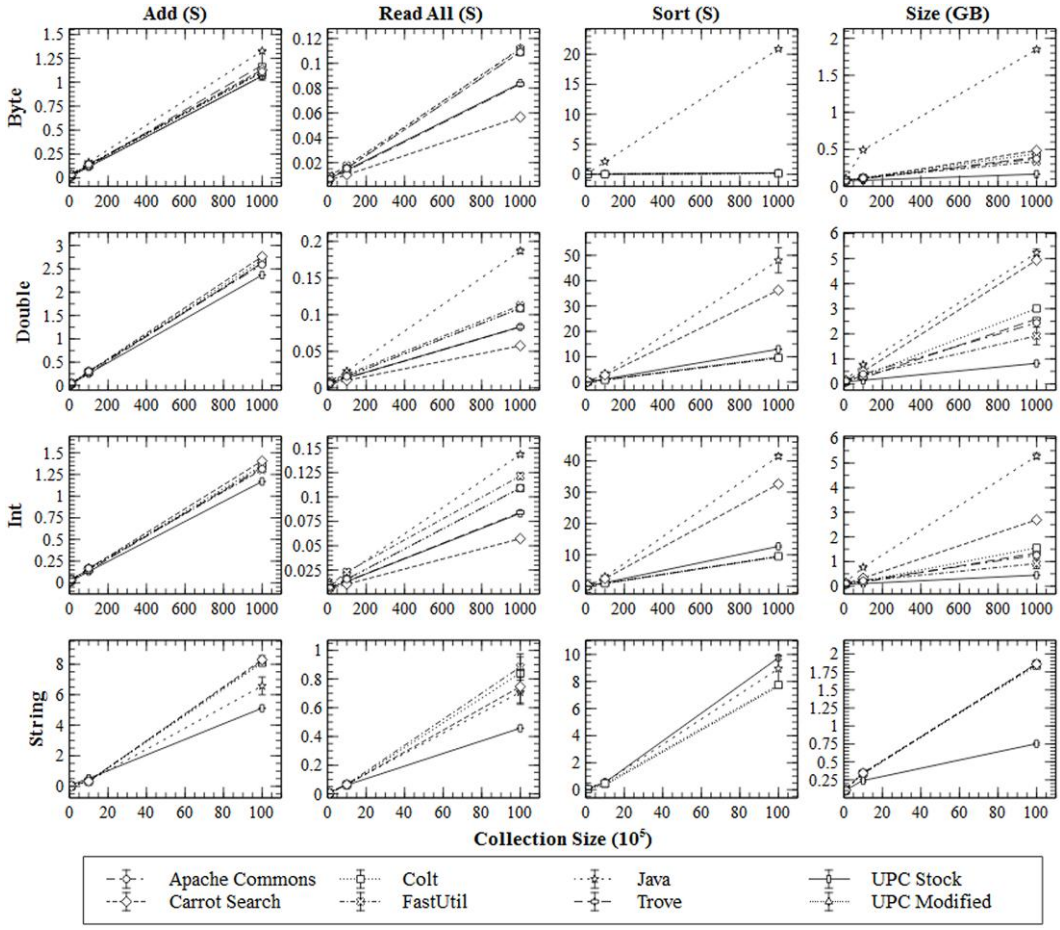

Fig. 4. List experiment results – not shown in the plots: Apache Commons, CarrotSearch byte and string, and FastUtil *sort* as this feature is unsupported, and Java double and int *add* due to poor performance.

5.2.3. Hash sets

Five hash set collections are compared to UPC for add, contains, iterator, and resulting size; key findings shown in Fig. 5. Java is reported from 100K-10M as it exceeds the allocated 8GB of memory at 100M. UPC stock and modified differ in implementation for add non-string types.

The byte data type is not included in Fig. 5 as little separation exists between collections. There are no

notable differences in times (tens to hundreds of milliseconds) and sizes (tens of megabytes) from 100K to 10M. At 100M, UPC (1.05±0.00 seconds) and UPCM (1.04±0.00 seconds) separate from the next best Koloboke (1.25±0.04 seconds) for add at 19.05% and 20.19% less time respectively. For contains, UPC (1.04±0.02 seconds) requires 11.54% less time than the second best Java (1.16±0.05 seconds). Second best, Java (80.43±7.98 MB) consumes 9.03% more space than UPC (73.17±2.20 MB).

Koloboke adds data fastest for double and int, followed by UPCM (6-7%) and UPC (9-13%), and UPC tops Trove for strings by 154%. UPC follows Koloboke for double and int contains by 14-17%, with UPC necessitating around 46% less time than CarrotSearch for string. Koloboke iterates over the double and int hash sets quickest up to 10M, and then relinquishes its number one spot to CarrotSearch followed by UPC (6-23% and 15-33% slower respectively). UPC, however, is consistently faster than the next best Java for string iteration by upwards of 43%. In terms of size, UPC, FastUtil, and CarrotSearch are the smallest with less than 3% difference between them for double and int. For string, UPC's footprint is 56% smaller than the next best Koloboke.

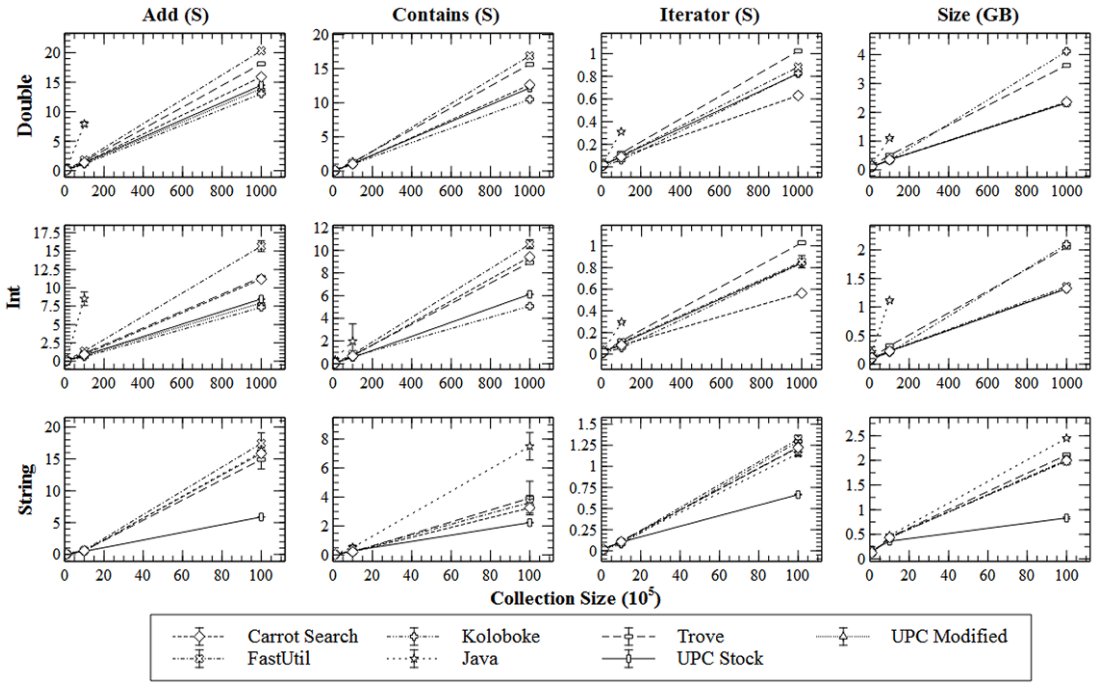

Fig. 5. Hash set experiment results – not shown in the plots: Koloboke string *add* and *contains* due to poor performance.

5.2.4. Hash maps

Six hash map collections (Colt supporting int-int only) are compared to UPC for put, contains key (all key values are checked) and value (100 random – unsupported by CarrotSearch), iterator, and resulting size – key findings shown in Fig. 6. Java is reported from 100K-10M as it exceeds the allocated 8GB of memory at 100M. UPC stock and modified differ in implementation for put on non-string types and contains value (as the string experiment uses an int value).

The byte data type provided little separation between the collections – thus its lack of inclusion in Fig. 6. For 100K to 10M, there were no notable differences as times and sizes were fairly static measuring in the tens to hundreds of milliseconds and tens of megabytes. At 100M, UPC and UPCM began to diverge from the next best Java for put and contains key. For put, UPC (2.19±0.01 seconds) is 1.79% and UPCM (2.07±0.01 seconds) 7.17% faster than Java (2.23±0.01 seconds). UPC (1.01±0.00 seconds) also performs the contains

key operation 12.93% faster than Java (1.16±0.01 seconds).

Koloboke is the top performer on put tests for double and int, followed by UPCM (up to 6% and 13% slower respectively) and UPC (upwards of 10% and 20% slower respectively). For string, UPC outperforms the grouping of Trove, FastUtil, and CarrotSearch by over 100% across the collection sizes. Koloboke is again the number one option for contains key over UPC for double and int for sets larger than 10M (at which point UPC is 23-30% faster) by about 6-12% respectively. UPC requires 62-66% of the time for string as the nearest library CarrotSearch. UPCM is second to FastUtil for double and int contains value by roughly 30% and 98% respectively. Koloboke and FastUtil best UPCM in string by 52% and 31% respectively. Although the methods are coded identically, the use of primitive arrays by Koloboke and FastUtil, and Koloboke's "parallelization" explain the time differentials. Koloboke iterates over the double and int hash maps quickest up to 10M; beyond that point, CarrotSearch is fastest. UPC is second for double and third for int by 7-25% (Koloboke is 3-4% faster than UPC for int). For string, UPC claims the top spot over the group of Trove, CarrotSearch, and Java (all within 5 milliseconds of one another) by 5-39%. Regarding size, UPC and CarrotSearch provide the best, virtually identical results for double and int, with UPC 15-20% smaller than the next best string Koloboke.

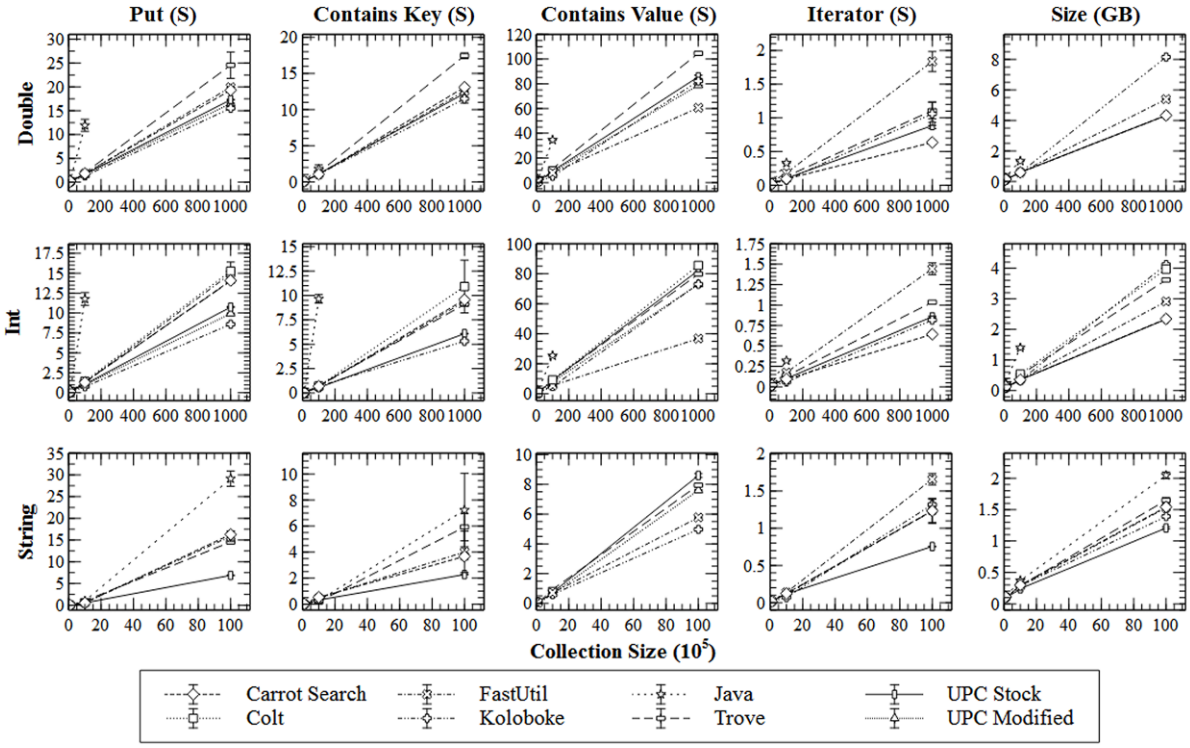

Fig. 6. Hash map experiment results – not shown in the plots: CarrotSearch *contains value* and Colt *iterator* as these feature are unsupported, and Java string *contains value* and Koloboke string *put* and *contains key* due to poor performance.

5.2.5. Matrices

Native support for matrices is found in Colt alone, with limited features, and only for doubles. Add, read all (to measure access rates), scale, transpose (deep copy), and size are the compared elements; results in Fig. 7. UPCM differs from UPC for scale and transpose.

Exploiting the efficiencies of Java's primitive double array, Colt is 3-5% faster adding data than UPC beyond 1000-by-1000. Reading the UPC matrix requires 17-18% less time compared to Colt. The functions scale and transpose are performed quickest by UPCM (444-609% and 148-1,122% faster than Colt

respectively) followed by UPC (2-515% and 11-201% faster than Colt respectively). Lastly, size is uniform across the sets, indicating UPC(M) is consistent with Java's primitive arrays in memory consumption.

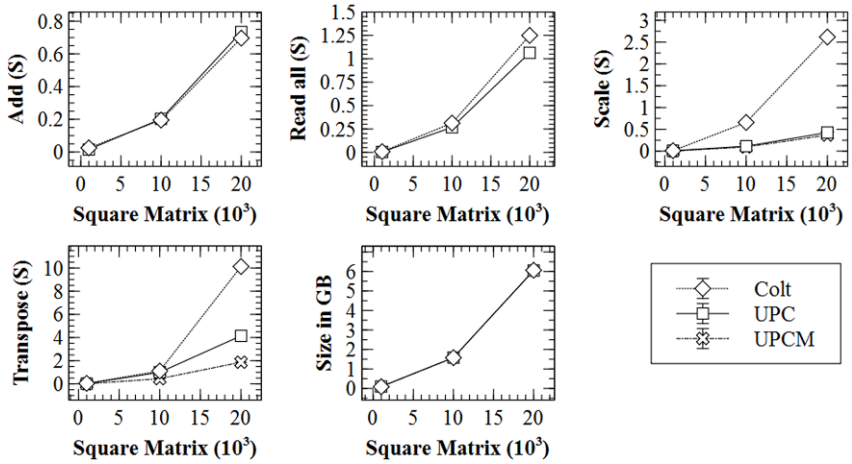

Fig. 7. Matrix experiment results.

5.3. Realistic Implementations

UPC collections are designed as backing data structures for computationally intensive situations. To illustrate the robust nature of these collections, four realistic use cases are evaluated: a binary heap implementation, matrix multiplication, binary search over a sorted unique list, and *k*-mer counting. The definition for each problem can be found in its subsection. Results are aggregated from runs of 10.

5.3.1. Binary heap

A binary heap is a basic data structure seen throughout computer science. For instance, it is used in priority queues and algorithms such as Dijkstra's shortest path, Prim's minimum spanning tree, and Huffman encoding. The underlying storage is a simple array, with in-place operations. As shown in Section 5.2, Java arrays are extremely fast and efficient. Therefore, the intent of this analysis is to show UPC arrays provide similar performance in a set, read, and swap-intensive environment. UPC is compared to Java and JLargeArrays for creating a minimum heap and its resulting size on the Linux Mint (100M, 500M, 1B, 1.5B) and Rocks (100M, 500M, 1B, 2B, 2.5B) machines – results in Fig. 8.

UPC is 4.82-5.64% on Linux Mint and 6.95-7.91% on Rocks (up to 2 billion due to the 2^{31}-1 limitation) slower than Java to create the binary heap. In terms of memory, UPC is 0.41-0.54% on Linux Mint and 0.48-1.23% on Rocks smaller than Java. Overall, JLargeArrays, which exceeds the allotted memory on the Linux Mint machine beyond 1B, is the slowest, largest, and exhibits the greatest variability amongst the tested algorithms.

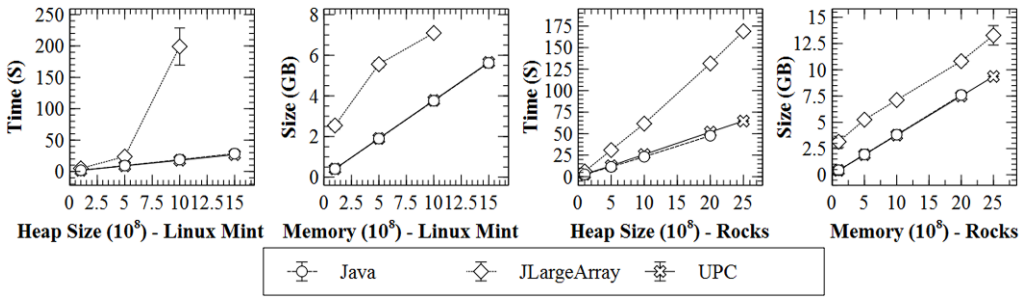

Fig. 8. Binary heap experiment results.

5.3.2. Matrix multiplication

Matrices are fundamental data structures in most fields. Physicists use them to study how particles interact and operations researchers, to solve linear equations, for instance. Colt and UPC (stock and modified versions) contain built-in matrices and are therefore compared, along with a multidimensional Java array implementation – UPCM for multiplication only. On Linux Mint, square matrices of size 1,000, 1,250, 1,500, 2,000, and 2,500 were multiplied to like-sized matrices, and on Rocks, 2,500, 5,000, 10,000, and 15,000 – Java results terminate at 5,000 due to its prohibitive runtime.

On the Linux Mint machine, both UPC implementations outperform Colt and Java for matrix multiplication. Compared to UPC, Colt, and Java, UPCM is 27.30-33.62%, 38.59-37.37%, and 96.61-96.89% faster respectively. Times on the Rocks node are slightly different, with UPCM coming in first, followed by Colt then UPC. The gap narrows from 38.16% and 44.12% at 2,500-by-2,500, to 15.06% and 18.17% at 15,000-by-15,000 respectively. In terms of size, UPC and Colt are virtually identical on both configurations with less than 1% difference. Compared to Java, they require between 14.35-22.86% on Linux Mint and 28.48-31.96% on Rocks less space.

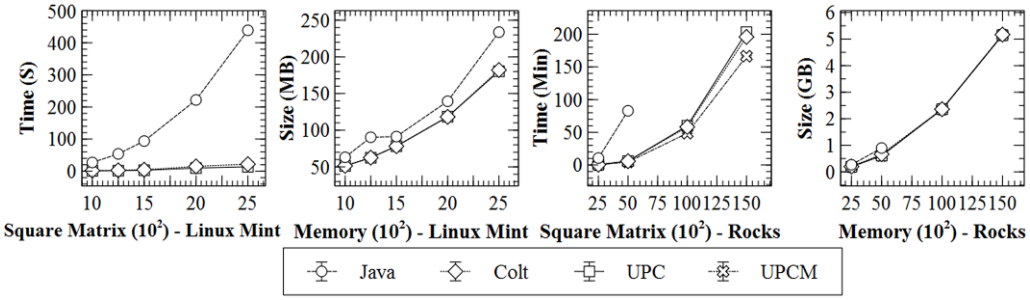

Fig. 9. Matrix multiplication experiment results.

5.3.3. Binary search over a sorted unique list

Binary search algorithms are widely used to quickly locate values within a sorted collection of items. Interesting questions might include how many distinct values exist above *x* (e.g., systolic blood pressure), or is there a reading between *x* and *y* (especially useful for decimal values when you do not know the exact number, but, e.g., a normal range)? Coupled with a scanning mechanism, a binary search can be transformed into a powerful counting algorithm. For this experiment, a list of integers will serve as the collection, and the Boolean range query the test. Both UPC versions, Java (for completeness), Colt, and Trove (based on their performance in Section 5.2) are used in this experiment. To increase the complexity, duplicate values are removed post sorting, prior to searching. The list sizes for testing are 10M, 100M, 250M, and 500M on Linux Mint, and 10M, 100M, 500M, 1B, 1.5B, 2B, and 3B on Rocks. The number of binary searches is set to 10% the size of the initial list. Due to excessive memory consumption, Java's List collection failed above 100M on Linux Mint and 500M on Rocks. Additionally, due to overall poor performance, Java results are not shown in Fig. 10. On Rocks, Trove failed for 1.5-2B as its ensureCapacity methods doubles the size of a list when it grows; thus beyond int for size greater than 1,342,177,280.[5] Colt experiences out of memory exceptions for 1-2B due to issues with its list expansion method as well, limiting the size effectively to 990,492,542.[6]

[5]Using the default constructor, Trove instantiates a list to size 10. Each invocation of grow performs a << 1 operation. Thus, the maximum size prior to exceeding 231-1 (resulting in a negative size) is 1,342,117,280.

[6]The list grows following max((oldCapacity*3)/2+1,newCapacity), where oldCapacity is the current size and newCapacity is oldCapacity+1. The problem is oldCapacity*3 surpasses int quickly, turning negative, resulting in an increase of 1 each

UPC(M) is the top performing algorithm in every category except range. For adding data, UPC is roughly 1.37-7.46% on Linux Mint and 7.20-10.51% on Rocks more efficient than the next best Trove. Likewise, UPCM sorts the list anywhere from 0.66-9.76% and 2.65-7.75% faster than the number two algorithm Trove on Linux Mint and Rocks respectively. UPC is upwards of 45% quicker for removing duplicates and 55% smaller than the second best Trove on both systems. In terms of memory, UPC consumes roughly 39-67% less than the next best on both systems.

The range queries utilize binary search, which is nothing more than a series of random access look-ups into the list. A Java primitive int array is the backing data structure for both Trove and Colt, thus they benefit from the performance gains discussed in Section 5.2. That said, on the Linux Mint machine, both Trove (5.75-4.57% in order of size) and Colt (6.13-4.91% in order of size) outperform UPC up to 500M, where UPC actually eclipses them by 6.88% and 12.34% respectively. A similar trend is seen in the Rocks, where the advantage for Trove and Colt diminish as size increases – 12.59-4.53% for Trove and 15.59-4.87% for Colt. It is worth noting UPC continues for three more tests beyond Trove and four for Colt.

Only UPC(M) successfully executed the experiment for sizes 1.5-3B, despite the fact the first two fit into an int index. These experiments indicate how UPC easily scales, and does so in a linear fashion.

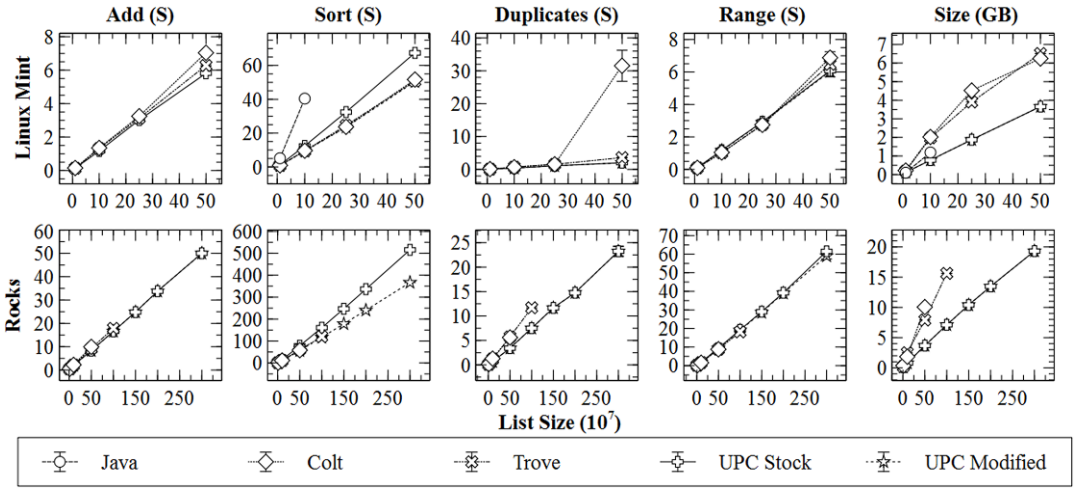

Fig. 10. Binary search over a sorted unique list experiment results.

5.3.4. Counting *k*-mers

Counting DNA subsequences of size *k* (known as a *k*-mer), is a basic operation in bioinformatics. It is essential in genome and transcriptome assembly, metagenomic sequencing, error correction of sequence reads, fast multiple sequence alignment, and repeat detection [71], [72]. This example employs a hash map with the *k*-mer as the key (a string) and count (int) as its value. A contiguous block of nucleotides are used for this experiment taken from the first human chromosome as submitted by the Genome Reference Consortium (GRCh38) [73], [74] – the sizes are as follows: 10M, 15M, 25M, and 50M on Linux Mint and 25M, 50M, 100M, 150M, and 200M on Rocks. The results are displayed in Fig. 11.

UPC dominates in both categories and configurations. As the number of nucleotides increase, so does the margin of separation in which UPC produces counts over the competing algorithms. UPC performs roughly 10-69%, 28-70%, and 38-73% faster than Koloboke, Trove, and Java across both machines respectively. UPC's memory consumption is fairly static in relation to the others at approximately 19% and 46%, 27%

insertion (using integer arithmetic and a default of 10, this occurs beyond 990,492,542). Theoretically, 231-1 elements should be allowed prior to overflowing, however, the repetitive instantiation of backing arrays quickly saturates the heap space beyond the garbage collector's ability to control.

and 52%, and 37% and 59% that of Koloboke, Trove, and Java for Linux Mint and Rocks respectively.

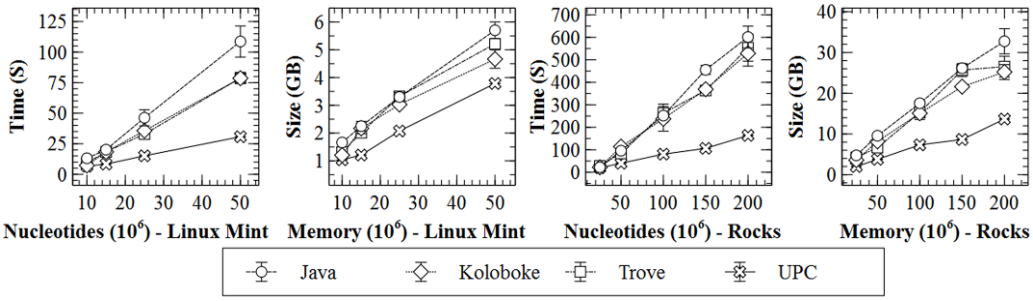

Fig. 11. Counting *k*-mers experiment results.

5.4. Discussion of Results

This section discusses the combined results from Sections 5.2 and 5.3, presenting an aggregate view of UPC(M) by collection, string implementation, and JDK/JVM.

Regarding arrays, UPC dominated adding, overwriting, and storing data. UPCM proved the fastest for sorting, with Java besting UPC. Java required 0 milliseconds to read all byte, double, and int data compared to UPC (measured in tens of milliseconds), but necessitated nearly twice the time for string. These findings are supported by the binary heap results (read-swap intensive), in which UPC ran parallel to Java with single digit differences. As discussed throughout, Java arrays are exceedingly efficient, thus our intent was to indicate UPC's consistent performance.

Implemented as arrays, matrices also provide insight into fixed structure manipulation. Whereas Colt populated matrices slightly faster, UPC(M) provided superior read, scale, and transpose performance. These results carried over into matrix multiplication, where UPCM was significantly faster, with little distinction between Colt and UPC.

As with arrays, UPC lists outperformed all others in adding, overwriting, and storing data. UPCM was one of the top sorting collections across all data types – best for int and string. UPC placed first in reading string and second in non-string types. The binary search over a sorted unique list outcomes corroborate these conclusions.

The hash-based collections had similar findings. Regardless of data type, UPC's overall memory footprint was moderately to significantly less than the next best. For non-string data types, UPC(M) followed Koloboke for inserting data (add/put) and searching for hash entries (contains/contains key) . UPC(M) trailed FastUtil for searching hash map values (contains value) and CarrotSearch for iterating. For strings, UPC(M) was unmatched in every category except hash map contains value, which was essentially an int array experiment (i.e., the value of the hash map is an int). The counting *k*-mers experiment also illustrated the power of UPC string key-based hash maps.

Concerning the UPC string data type, the reported results indicate their power and compactness. The speed of access is generally 40%+ faster and size 15-50% smaller than the next best.

The JDK/JVM extensions reduced overall runtimes slightly to significantly. For inserting data into non-string hash-based collections, UPCM was 4-8% faster. The contains value hash map operation decreased by 9-14% from UPC to UPCM. List and array sorting also declined by 28-47%. For matrix scaling and transposing, UPCM reduced runtimes by 14-17% and 54-56% respectively. These results simultaneously indicate the expense of JNI operations and the performance gains anticipated by providing native support.

The claims of support beyond 2^{31}-1 were substantiated by the binary heap and binary search over a

sorted unique list experiments on the Rocks cluster node. The former includes an array of size 2.5 billion, clearly denoting a continued linear progression in time in line with Java's performance if projected beyond its hard limit. The latter increased a list to a size of 3 billion with the same results. Whereas the final run at 3 billion was the only one larger than 2^{31}-1, Trove, Colt, and Java all failed to reach even their theoretical limits (Colt and Java prior to 1 billion and Trove, 1.5 billion).

In summary, UPC(M) performed at the highest level across all tests; especially for string collections and total memory utilization. Coupled with the ability to destroy collections, the UPC library provides the developer with a comprehensive suite of fast, small, and powerful backing data structures necessary to support large-scale computing.

6. Conclusions and Future Work

Java is a versatile and robust programming language in domain appropriate implementations. Large-scale applications, such as scientific and high performance computing, have typically alluded Java's grasp due, in large part, to its lack of direct memory management, object-based collections, and an array/collection size limitation of 2^{31}-1. While third-party collections have sought to remedy the situation, most focus on the second element by introducing primitive collections, while failing to add scale and memory management.

The UPC(M) collections, as presented in this article, address all three aspects, thus providing a complete solution to the problem. Using the non-public sun.misc.Unsafe class, UPC(M) allows for the allocation and destruction of long-indexed, primitive arrays, lists, hash maps, hash sets, and matrices. Testing indicates the collections provide comparable to superior performance to extent approaches, with proven linear scaling beyond the Java imposed hard-limit of 2^{31}-1 elements. The JVM extension provides additional benefits to bulk operations, while demonstrating Java's ability to support such actions if included in the language.

In the immediate future, arrays and lists of pairs and triples will be finalized (currently experimental), 3-dimensional matrices crafted, and JDK/JVM string functions expanded. The long-term view is to fully support concurrent collections. This requires modifying the JVM to support compare-and-swap features beyond byte (which exists, but does not currently have an Unsafe interface), int, and long, to avoid conversion costs (e.g., double-to-long) and bloated memory (e.g., char-to-int). From the set of tools used in this article, only Java currently provides concurrent collections.

References

[1] Oracle Corporation. *Learn about Java Technology.* Retrieved October 1, 2015, from http://www.java.com/en/about

[2] Baitsch, M., Li, N., & Hartmann, D. (2010). A toolkit for efficient numerical applications in Java. *Advances in Engineering Software*, *41(1)*, 75–83.

[3] Chen, K., & Chen, J. B. (2007). Aspect-based instrumentation for locating memory leaks in Java programs. *Proceedings of the 31st Annual IEEE International Computer Software and Applications Conference* (pp. 23–28).

[4] Daly, C., Horgan, J., Power, J., & Waldron, J. (2001). Platform independent dynamic Java virtual machine analysis: The Java Grande Forum benchmark suite. *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande* (pp. 106–115).

[5] Georges, A., Buytaert, D., & Eeckhout, L. (2007). Statistically rigorous Java performance evaluation. *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications* (pp. 57–76).

[6] Georges, A., Eeckhout, L., & Buytaert, D. (2008). Java performance evaluation through rigorous replay compilation. *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems*

Languages and Applications (pp. 367–384).

[7] Hofer, P., & Mössenbck, H. (2014). Fast java profiling with scheduling-aware stack fragment sampling and asynchronous analysis. *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools* (pp. 145–156).

[8] Li, J. Y., Chen, J. Y., & Fong, A. S. (2014). Analyzing the characteristic of Java objects. *Applied Mechanics and Materials,* 303-306, 2329–2332.

[9] Mytkowicz, T., Diwan, A., Hauswirth, M., & Sweeney, P. F. (2010). Evaluating the accuracy of Java profilers. *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation* (pp. 187–197).

[10] Sartor, J. B., & Eeckhout, L. (2012). Exploring multi-threaded Java application performance on multicore hardware. *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (pp. 281–296).

[11] Shafi, A., Carpenter, B., & Baker, M. (2009, June). Nested parallelism for multi-core HPC systems using Java. *Journal of Parallel and Distributed Computing*, *69(6)*, 532–545.

[12] Shafi, A., Carpenter, B., Baker, M., & Hussain, A. (2009). A comparative study of java and C performance in two large-scale parallel applications. *Concurrency and Computation: Practice and Experience, 21(15)*, 1882–1906.

[13] Taboada, G. L., Ramos, S., Expósito, R. R., Touriño, J., & Doallo, R. (2013). Java in the high performance computing arena: Research, practice and experience. *Science of Computer Programming, 78(5)*, 425–444.

[14] Taboada, G. L., Touriño, J., & Doallo, R. (2009). Java for high performance computing: Assessment of current research and practice. *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java* (pp. 30–39).

[15] Šor, V., & Srirama, S. N. (2014). Memory leak detection in Java: Taxonomy and classification of approaches. *Journal of Systems and Software, 96*, 139–151.

[16] Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M., & Paterson, J. (2007). A survey of literature on the teaching of introductory programming. *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education* (pp. 204–223).

[17] Roberts, E. (2004). Resources to support the use of Java in introductory computer science. *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education* (pp. 233–234).

[18] Rountree, N., Rountree, J., & Robins, A. (2002). Predictors of success and failure in a CS1 course. *ACM SIGCSE Bulletin*, *34(4)*, (121–124)

[19] Smith, P. A., & Boyd, G. (2001). Introducing OO concepts from a class user perspective. *Journal of Computing Sciences in Colleges, 17(2)*, 152–158.

[20] TIOBE Software. *T* Retrieved October 1, 2015, from http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[21] Vassilios, B., & Nicholaos, P. *Trendy Skills*. Retrieved October 1, 2015, from http://trendyskills.com

[22] Bull, J. M., Smith, L. A., Westhead, M. D., Henty, D. S., & Davey, R. A. (2010). A benchmark suite for high performance Java. *Concurrency: Practice and Experience, 12(6)*, 375–388.

[23] Knoll, P., & Mirzaei, S. (2010). Scientific computing with Java. *Computer Applications in Engineering Education*, *18(3)*, 495–501.

[24] Luri, X., & Torra, J. (2009). High-performance computing in java: The data processing of Gaia. S*ci-Comp XXL*.

[25] Baker, M., & Carpenter, B. (2000). MPJ: A proposed Java message passing API and environment for high performance computing. *Parallel and Distributed Processing*, *1800*, 552–559.

[26] Baker, M., Carpenter, B., Fox, G., Ko, S. H., & Lim, S. (1999). Mpijava: An object-oriented java interface to MPI. *Parallel and Distributed Processing*, *1586*, 748–762.

[27] Open MPI. *Open Source High Performance Computing*. Retrieved October 1, 2015, from http://www.open-mpi.org

[28] Bull, J. M., & Kambites, M. E. (2000). JOMP - An OpenMP-like Interface for Java. *Proceedings of the ACM 2000 Conference on Java Grande* (pp. 44–53).

[29] Klemm, M., Bezold, M., Veldema, R., & Philippsen, M. (2007). JaMP: An implementation of OpenMP for a Java DSM. *Concurrency and Computation: Practice and Experience*, *19(18)*, 2333–2352.

[30] Kaminsky, A. (2007). Parallel java: A unified API for shared memory and cluster parallel programming in 100% Java. *Proceedings of the 9th Workshop on Java and Components for Parallelism. Distribution and Concurrency. International Parallel and Distirbuted Processing Symposium* (pp. 1–8).

[31] Taboada, G. L., Tourio, J., & Doallo, R. (2008). Java fast sockets: Enabling high-speed Java communications on high performance clusters. *Computer Communications*, *31(17)*, 4049–4059.

[32] Yan, Y., Grossman, M., & Sarkar, V. (2009). JCUDA: A programmer-friendly interface for accelerating Java programs with CUDA. *Proceedings of the 15th International Euro-Par Conference on Parallel Processing* (pp. 887–899).

[33] Wohlgemuth, G. *JaCuda*. Retrieved October 1, 2015, from http://jacuda.sourceforge.net

[34] Calvert, P. *java-gpu - Support for offloading parallel-for loops in Java to NVIDIA CUDA compatible cards*. Retrieved October 1, 2015, from https://code.google.com/p/java-gpu

[35] Hutter, M. *jocl.org - Java bindings for OpenCL*. Retrieved October 1, 2015, from http://jocl.org

[36] Chafik, O. *JavaCL - OpenCL bindings for Java*. Retrieved October 1, 2015, from https://code.google.com/p/javacl

[37] Mallon, D., Taboada, G., Tourio, J., & Doallo, R. (2009, February). NPB-MPJ: NAS parallel benchmarks implementation for message-passing in Java. *Proceedings of the 17th Euromicro International Conference on Parallel, Distrubuted, and Network-based Processing* (pp. 181–190).

[38] Amedro, B., Bodnartchouk, V., Caromel, D., Delbe, C., Huet, F., & Taboada, G. (2008). Current state of Java for HPC. *Technical Report RT-0353, INRIA*, Retrieved, from https://hal.inria.fr/inria-00312039/document.

[39] Andersen, J. L., Todberg, M., Dalsgaard, A. E., & Hansen, R. R. (2013). Worst-case memory consumption analysis for SCJ. *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems* (pp. 2–10).

[40] Portillo-Dominguez, A., Wang, M., Magoni, D., Perry, P., & Murphy, J. (2014, June). Load balancing of Java applications by forecasting garbage collections. *Proceedings of the 13th International Symposium on Parallel and Distributed Computing* (pp. 127–134).

[41] Troiano, L., De Pasquale, D. (2009, December). A java library for genetic algorithms addressing memory and time issues. *Proceedings of the World Congress on Nature and Biologically Inspired Computing* (pp. 642-647).

[42] Xian, F., Srisa-an, W., & Jiang, H. (2008). Garbage collection: Java application servers Achilles heel. *Science of Computer Programming*, *70(23)*, 89–110.

[43] Zhao, Y., Shi, J., Zheng, K., Wang, H., Lin, H., & Shao, L. (2009). Allocation wall: A limiting factor of Java applications on emerging multi-core platforms. *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (pp. 361–376).

[44] Chase, D. R. (2014, October). Hg: Panama/panama/jdk: Summary: push of version-2 prototype of Arrays2.0. Work remains, comment is requested. Retrieved, October 1, 2015, from http://mail.openjdk.java.net/pipermail/panama-dev/2014-October/000038.html

[45] OpenJDK. *OpenJDK Mercurial Repositories*. Retrieved, October 1, 2015, from http://hg.openjdk.java.net
[46] OpenJDK. *Valhalla*. Retrieved, from http://openjdk.java.net/projects/valhalla
[47] Goetz, B. (2014, December). State of the specialization. Retrieved, from http://cr.openjdk.java.net/~briangoetz/valhalla/specialization.html
[48] Rose, J. (2012, March). Value types in the VM. Retrieved, from https://blogs.oracle.com/jrose/entry/value_types_in_the_vm
[49] Rose, J. (Sep 2014). JEP 169: Value objects. Retrieved, from http://openjdk.java.net/jeps/169
[50] Rose, J., Goetz, B., & Steele, G. (2014, April). State of the values. Retrieved, from http://cr.openjdk.java.net/~jrose/values/values-0.html
[51] OpenJDK. *OpenJDK*. Retrieved, from http://openjdk.java.net
[52] The apache software foundation. *Apache DirectMemory*. Retrieved, from http://directmemory.apache.org
[53] Heliosearch. Retrieved, from http://heliosearch.org
[54] VoltDB, Inc. *VoltDB: In-Memory Database, NewSQL & Real-Time Analytics*. Retrieved, from http://voltdb.com/memory-database-newsql-real-time-analytics-voltdb
[55] Chronicle. *Chronicle Map*. Retrieved, from http://openhft.net/products/chronicle-map
[56] Google. *Guava*. Retrieved, from https://code.google.com/p/guava-libraries
[57] Horii, H., & Thomas, R. G. (2014). *Memory allocation method, program, and system*. Patent EP2755129 A1.
[58] Sandoz, P. (2014, July). Atomic VARHANDLES. *JVM Language Summit*. Retrieved, from http://medianetwork.oracle.com/video/player/3731019200001
[59] Sandoz, P. (Sep 2014). Safety not guaranteed: Sunmisc. Unsafe and the quest for safe alternatives. *JavaOne*. Retrieved, from https://www.oracle.com/javaone/sessions/index.html
[60] Sandoz, P. (2014, January). Survey on sun.misc.Unsafe. Retrieved 2014, from http://mail.openjdk.java.net/pipermail/core-libs-dev/2014-January/024650.html
[61] OpenJDK. *Project Jigsaw.* Retrieved, from http://openjdk.java.net/projects/jigsaw
[62] Lea, D., & Sandoz, P. (2015). Retrieved, from *Enhanced Volatiles*, http://openjdk.java.net/jeps/193
[63] The apache software foundation. Retrieved, from http://commons.apache.org/proper/commons-primitives
[64] Oriński, S., & Weiss, D. HPPC: High performance primitive collections for java. Retrieved, from http://labs.carrotsearch.com/hppc.html
[65] CERN. *Colt.* Retrieved, from http://dst.lbl.gov/ACSSoftware/colt
[66] Vigna, S. Fastutil: Fast and compact type-specific collections for java. Retrieved, from http://fastutil.di.unimi.it
[67] Oracle corporation. *Java Software.* Retrieved, from https://www.oracle.com/java/index.html
[68] Wendykier, P. *JLargeArrays*. Retrieved, from https://github.com/IcmVis/JLargeArrays
[69] Chronicle. *Koloboke Collections.* Retrieved, from http://chronicle.software/products/koloboke-collections
[70] Eden, R., Parent, J., Randall, J., & Friedman, E. D. *Trove.* Retrieved, from http://trove.starlight-systems.com
[71] Marçais, G., & Kingsford, C. (2011). A fast, lock-free approach for efficient parallel counting of occurrences of *k*-mers. *Bioinformatics*, *27(6)*, 764–770.
[72] Melsted, P., & Pritchard, J. K. (2011). Efficient counting of *k*-mers in DNA sequences using a bloom filter. *BMC Bioinformatics*, *12*, 333.
[73] National center for biotechnology information. *Download the Complete Genome for An Organism*.

Retrieved, from http://www.ncbi.nlm.nih.gov/guide/howto/dwn-genome

[74] National center for biotechnology information (2013, December). *New Human Genome Assembly (GRCh38) Released!.* Retrieved, from http://www.ncbi.nlm.nih.gov/news/12-23-2013-grch38-released

Ray Hylock is an assistant professor in the Department of Health Services and Information Management at East Carolina University, USA. He received his PhD in health informatics from the University of Iowa in 2013. His research primarily focuses on computation advancements to support patient care in the areas of health care databases/data warehouses, federation, advanced data structures, optimization, and heuristics. Currently, he is laying the foundation for the HSIM Computational Lab to support data and computationally intensive patient health research.