Improving the Balance of a Random-access Data Structure on a Migratory Thread Platform

Muktaka Joshipura, advised by Dr. Vivek Sarkar

May 2022

Introduction

During the execution of a computer program, a computer processor usually needs to access information stored in a memory chip, like a RAM chip found in standard personal computers and mobile phones. Accessing RAM is much slower than other computation on a processor, so it is generally preferable to avoid doing so if possible.

Many computer programs access information in memory locations that are close to each other at close points in time. For example, while reading a list of numbers from memory, a processor would need to read numbers from consecutive memory addresses. Such programs are said to have good memory locality. To make such algorithms run faster, standard computer architectures have evolved to have sophisticated systems that temporarily bring relevant chunks of memory into faster, smaller memory chips called caches in a process called caching. This speeds up execution because if the processor finds the information needed in cached memory, it can skip going to the RAM chip and access the information quicker.

These caching systems are often confounded by programs that do not have good memory locality. For example, a program that reads information from random memory addresses would not benefit much from expecting to find the information it needs in cached memory. Since the program cannot utilize the cache much, it cannot avoid fetching the information from the RAM chip.

A metric for determining how quickly a program can process information stored in memory is called memory bandwidth utilization, which is given by

$$\text{Memory bandwidth} = \frac{\text{Amount of memory accessed by a program}}{\text{Time taken by program}}$$

So, a caching system would increase the memory bandwidth utilization of a program with good memory locality. It would not do so for a program with poor memory locality.

There are an increasing number of relevant applications where the problems to be solved by the computer are inherently such that it is difficult for cache systems to be of much utility, because the programs written for them would have poor memory locality, also called *sparse* applications. A leading example is graph algorithms, where even simple algorithms like breadth-first search has the property that following an edge can lead outside cached memory.

In order to accelerate the execution of these algorithms, novel computers with memory-centric architectures have been developed [2]. One such architecture is used in the Emu Chick computer designed by Emu Technology (now Lucata Corporation). Characterizations by J. S. Young et al. have shown that the Emu Chick performs well at sparse applications such as pointer chasing and Sparse Matrix-Vector multiplication [6].

The Emu Chick architecture has been extended to be Lucata Pathfinder platform, which is intended to be distributed graph analytics platform [3]. The Lucata Pathfinder has a project in development to implement GraphBLAS, which is a standard API for graph algorithms to be expressed in a cross-platform, algebraic manner [4].

In this work, we will look at a data structure built for the Chick and Pathfinder platforms. We will then attempt to make a more balanced version, and we will then compare their pros and cons.

Literature Review

Prior work

On a migratory-thread platform such as the Emu Chick, a shared address space of memory is split between several CPU called nodes, and whenever a thread running on one computing element (say node 0) must read memory from another (say node 1), the thread stops running on node 0 and *migrates* to the node 1, resuming execution there. Thus designed, the Emu Chick, which does not have a cache, has shown to have memory bandwidth utilization that does not depend much on whether the program had sparse memory accesses [2].

The first independent benchmark characterization of the Emu Chick is by E. Hein et al. in [2], which involves measurements on pointer chasing, the STREAM benchmark to measure the Emu's memory bandwidth, and Sparse Matrix-Vector multiplication. A later benchmark characterization is by J. S. Young et al. in [6]. Both give pointers on programming the Emu Chick, stating that programs must be written to optimize the layout in which it stores information in order to limit the need to migrate threads of the program from having to migrate around the system.

Also of note are the implementation of the Sparse Matrix-Vector multiplication explained in detail in M. E. Belvirani et al. in [1], and a parallel bitonic sort implementation detailed in K. Veluswamy et al. in [5]. In [5], the authors also list ways to spawn threads efficiently, adjusting the number of threads spawned by an algorithm, and to prevent bottlenecks caused by migrating threads.

The Chunked Array

In any modern program it is common to use a collection of custom-sized data structures, and graph frameworks like GraphIt [7] and LucataGraphBLAS [4] use them heavily in their source code to represent node lists or edge lists. The Pathfinder and Chick platform (both developed by the Lucata Corporation), provide a method to do this, called 'chunked arrays'.

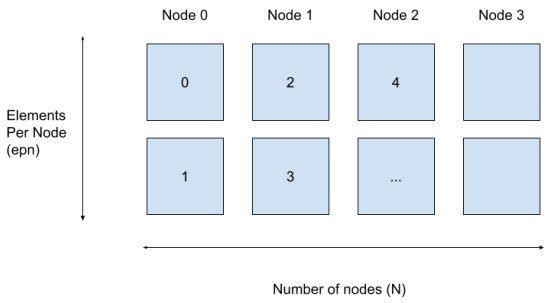

Figure 1: 5 elements on 4 nodes in a standard chunked array, showing the location of the next element

On the Chick and Pathfinder platforms, the memory of a standard chunked array of length k on N nodes is laid out by following the below rules:

1. A stripe of ceil(N/k) cells is allocated on each node.
2. When elements are added in sequence, the stripes of lower numbered nodes are filled before those of higher numbered nodes.

An illustration of the above is available in Figure 1.

In this scheme, the node number for a given index i in an array of size k on a system with N nodes is stored on is given by

$$\text{floor}(i \,/\, \text{floor}(k \,/\, N))$$

The index on the stripe is given by

$$i \,\%\, \text{floor}(k \,/\, N)$$

Since a thread reading outside the node it is currently executing on can cause a migration, and migrations can be expensive ([6]) the advantage of such a structure is that despite having its elements distributed across nodes, reading adjacent elements by index does not usually cause the thread to have to migrate between nodes, that is, accesses have sequential locality. This reduces the number of migrations needed for algorithms like sparse matrix-vector multiplication on matrices stored in the Compressed Sparse Row format.

This structure however has the problem that there is no 'balance guarantee'. In Figure 1, node 3 is completely unutilized. This means that any threads

spawned for computation on this array will ignore Node 3 completely, and instead contribute to the overutilization of other nodes on the system. Although the relative imbalance caused every time this happens is inversely proportional to the number of elements assigned to each node, this can cause damage if the elements are few but represent a large amount of computation.

A balance guarantee may also significantly simplify other algorithms such as distributed sorting, where having an even split of elements can improve design and also improve resource utilization with an algorithm like parallel prefix mergesort. Attempting to create a distributed comparison sort for the Emu is what motivation for this design.

Methodology

Proposed Solution

The best way to achieve a balance guarantee is to use a round-robin array: is to allocate elements to nodes in a round-robin fashion, with element 0 on node 0, element 1 on node 1, and so on, wrapping around when we run out of nodes, but this does not give us any sequential locality, since consecutive elements are on different nodes. Reading elements in order would then cause lots of migrations.

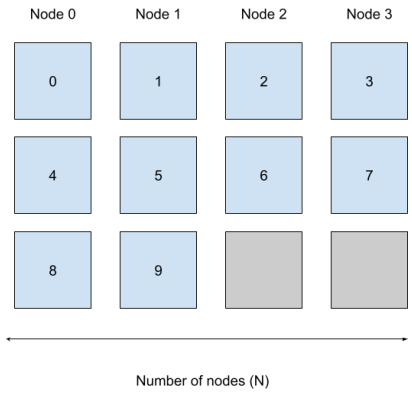

Figure 2: 10 elements on 4 nodes with round robin layout

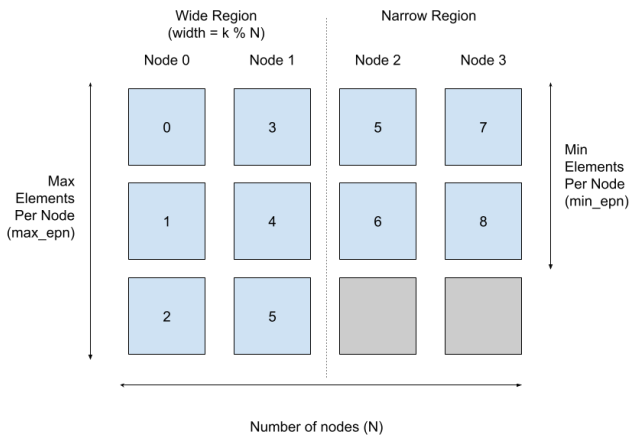

Figure 3: 10 elements on 4 nodes with proposed custom chunked layout

Instead, we can try to use the *filling pattern* of the standard chunked array (filling up lower nodes before higher nodes) on the *shape* (the set of filled elements) of the round-robin array, and we get the best of both worlds. However, we trade off the ability to add to the end of the array efficiently, which is acceptable for the purposes of fixed-size pools, or frequently sorted lists. The layout of this 'custom chunked array' is described as in Figure 3.

The principle is that the shape (the set of filled cells) a round-robin array makes can be split into two rectangles, one of which is wider by at most 1 than the other, as shown in Figure 3. These two rectangles can be filled up in a similar manner to the standard chunked array, and we would have a perfectly balanced layout of nodes which still maintains good sequential locality. In Figure 3, we still have sequential runs of 0-2, 3-5, 5-6, and 7-8.

We can determine the number of elements per node in the narrower rectangle, min_epn by the formula

$$min\_epn = \text{floor}(k/n)$$

The number of elements in the wider rectangle, if it exists, is then given by max_epn, calculated as

$$max\_epn = \text{ceil}(k/n)$$

The number of nodes in the wider rectangle can be given by

$$w = k \,\%\, n$$

Then, we can determine the size of the wider rectangle by

$$b = w * max\_epn$$

So, we get the tuple that represents the (node number, stripe index) by the following formula

$$\text{index}(i) = \begin{cases} (\text{floor}(i/max\_epn), i \,\%\, max\_epn) & \text{if } i < b \\ w + (\text{floor}((i-b)/min\_epn), (i-b) \,\%\, min\_epn) & \text{otherwise} \end{cases}$$

Measurements

Uniformity of element allocation

For each total number of elements k, we compared the uniformity of element allocation on 8 nodes between the two strategies for both elements. This was done by measuring the statistical variance of the element distribution among the nodes.

So, if two elements were allocated among two nodes as two on the first and none on the second, the distribution would have a variance of 1, while an even distribution would have a variance of 0.

However, this variance is not the best measure of the impact the imbalance can have. Having 8 less elements in the stripe of the last node may affect variance but would not be consequential if all stripes contained at least 100 elements. To correct for this, we measured a 'relative variance' by dividing the variance by the stripe length.

The results of the absolute and relative variance calculations are plotted in Figures 4 and 5 respectively. We see that the variance caused by using the standard chunked array is much higher than when using the custom variation, but this variances has a maximum that it does not cross even with increasing nodes. We can see that the custom chunked array always has smoother allocations, but the improvement it brings over the standard chunked declines as the number of elements increase.

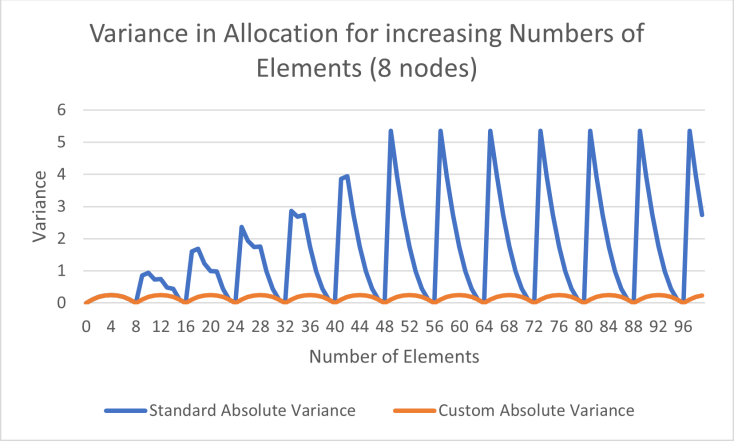

Figure 4: Absolute variance for increasing array size for each strategy

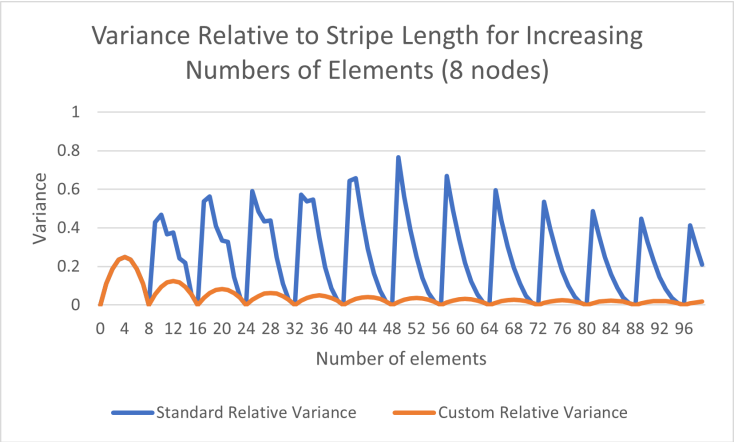

Figure 5: Relative variance for increasing array size for each strategy

Load Balancing Properties

We wrote an implementation of the proposed array and it was used to perform sparse matrix vector multiplication, where the array was used to represent the sparse matrix by using tuples of matrix coordinates and values. There were 67 such values in a matrix of dimension 27×27 with a density of 0.1.

These values were pseudorandomly assigned to various coordinates, with retries on collisions using Python's `randint` and `randrange`. The dense vector was striped round-robin.

Multiple threads, one for every node on the system and element of the matrix, were spawned to perform the matrix vector multiplication on 8 nodes. The value of 67 was therefore chosen since it is slightly above a multiple of 8, which would cause an imbalance in the standard chunked array in the stripe on node 7.

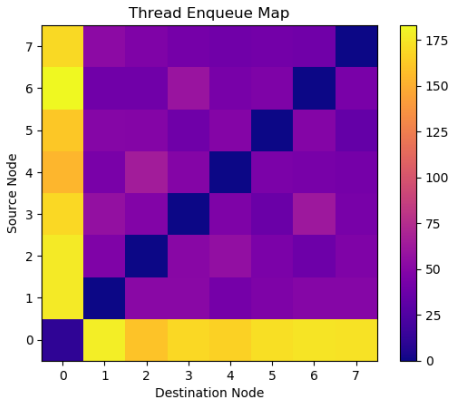

Figure 6: Migrations in Sparse Matrix-Vector product

This implementation was fed to Lucata's Pathfinder simulator on CRNCH's testbed, which tracks the origin and destination of migrations and plots it on a heatmap. The resulting map is shown in Figure 6. The image shows that the biggest source and destination for thread migrations was node 0, which is expected since it is the default location of all data not replicated across all nodes.

The most encouraging fact about the plot is that it is relatively smooth everywhere else, and it is symmetric along the secondary diagonal. This means that most of the heat difference (of which there is very little) along any transition is due to the structure of the matrix. Except node 0, there is no one node that is strongly more the source or sink than any other node, which indicates good balance.

Conclusion

Discussion

Despite the availability of chunkedar rays and newer GraphBLAS implementations, the Pathfinder and Chick systems don't currently handle more load balanced distributed data structures, likely because likely because writing data structures for migratory thread platforms is difficult, since reading the structure's own properties may cause migrations if they are not carefully implemented.

Nevertheless, we have identified a unique method to implement random-access arrays with sequential locality, and shown that it has a limited but tangible benefit over the standard chunked array. The custom chunked array can be used where balancing the number of elements in a node is a priority, and array extensibility after initialization is not a concern.

Future Work

A timing analysis of the standard and custom chunked array needs to be done to quantify the benefits of the additional balance when traded off with the additional complexity in identifying the range of each stripe.

This structure can now be used to implement other fixed size array-like structures that can take advantage of locality, like bitsets (where masking on word-sized element boundaries is possible), or algorithms like a distributed sort. An efficient bitset data structure can be used to perform faster breadth-first searches over much larger sizes of inputs.

Acknowledgements

I would like to thank all those at the Georgia Tech CRNCH center and Habanero lab who have helped me learn about fundamentals all the way from C, working with Linux, migratory thread platforms, and high performance computing. This includes my advisor Dr. Vivek Sarkar, Dr. Jason Riedy, Dr. Jeffrey Young, Dr. Prasanth Chatarasi, Prithayan Barua, and Sana Damani.

Bibliography

[1] Mehmet E Belviranli, Seyong Lee, and Jeffrey S Vetter. "Designing Algorithms for the EMU Migrating-threads-based Architecture". In: *2018 IEEE High Performance extreme Computing Conference (HPEC)*. 2018, pp. 1–7. DOI: 10.1109/HPEC.2018.8547571.

[2] Eric Hein et al. "An Initial Characterization of the Emu Chick". In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2018, pp. 579–588. DOI: 10.1109/IPDPSW.2018.00097.

[3] Lucata. *Pathfinder*. 2021. URL: https://lucata.com/solutions/pathfinder/ (visited on 05/04/2021).

[4] Jason Riedy and Shannon Kuntz. *Introducing Lucata's GraphBLAS*. 2021. URL: https://graphblas.org/GraphBLAS-Pointers/Slides/LAGraph-2021-10-13-Lucata-GraphBLAS.pdf (visited on 05/04/2021).

[5] Kaushik Velusamy et al. "Exploring Parallel Bitonic Sort on a Migratory Thread Architecture". In: *2018 IEEE High Performance extreme Computing Conference (HPEC)*. 2018, pp. 1–7. DOI: 10.1109/HPEC.2018.8547568.

[6] Jeffrey S. Young et al. "A microbenchmark characterization of the Emu chick". In: *Parallel Computing* 87 (2019), pp. 60–69. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2019.04.012. URL: https://www.sciencedirect.com/science/article/pii/S0167819118302503.

[7] Yunming Zhang et al. "GraphIt: A High-Performance Graph DSL". In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018). DOI: 10.1145/3276491. URL: https://doi.org/10.1145/3276491.