Chunk List – Concurrent Data Structures

Daniel J. Szelogowski
University of Wisconsin – Whitewater
szelogowdj19@uww.edu

ABSTRACT

Chunking data is obviously no new concept; however, I had never found any data structures that used chunking as the basis of their implementation. I figured that by using chunking alongside concurrency, I could create an extremely fast run-time in regards to particular methods as searching and/or sorting.

By using chunking and concurrency to my advantage, I came up with the chunk list — a dynamic list-based data structure that would separate large amounts of data into specifically sized chunks, each of which should be able to be searched at the exact same time by searching each chunk on a separate thread.

As a result of implementing this concept into its own class, I was able to create something that almost consistently gives around 20x-300x faster results than a regular ArrayList. However, should speed be a particular issue even after implementation, users can modify the size of the chunks and benchmark the speed of using smaller or larger chunks, depending on the amount of data being stored.

Notes:

- A full implementation can be found at https://github.com/danielathome19/Chunk-List
- All code examples given are in the C# language.
- Examples are given for each primary method the class should have implemented.

TABLE OF CONTENTS

1. DISCUSSION

1.1 What is a Chunk List?

A chunk list is an array-based list of elements in which data is stored in inner lists of a certain capacity, allowing for easily modifiable and faster runtimes based on the number of elements being stored. A simple way to conceptualize a chunk list would be an ArrayList (dynamic array) of ArrayLists.

The main list would contain the “chunks”, or ArrayLists that are not allowed to be filled past a specific capacity.

Any time a “chunk” has reached capacity, a new ArrayList is added and items are added to that chunk from thereon.

By doing this process and splitting our list into chunks, we can use parallel processing to our advantage. Using concurrency, we can run each chunk on a separate thread when doing tasks such as searching or removing.

This can be expressed visually as a table, in the case of a chunk list containing the numbers 1 – 50 where the chunk size is set to 10 elements. (See Figure 1)

Fig. 1:

1 2 3	11 12	21 22	31 32	41 42
4 5 6	13 14	23 24	33 34	43 44
7 8 9	15 16	25 26	35 36	45 46
10	17 18	27 28	37 38	47 48
	19 20	29 30	39 40	49 50

1.1.1 Defining a Chunk

Relatively speaking, “chunking” as a concept can be defined similarly to its psychological definition: Chunking is a term referring to the process of taking individual pieces of information (chunks) and grouping them into larger units [1]. Essentially, we are diving up data into multiple partitions in order to manipulate each one concurrently. One chunk may contain a large or small portion of our dataset, depending on how we want the elements to be partitioned and the amount of data being stored collectively across the entire structure.

1.1.2 Efficiency of Chunking

Chunking data for purposes of efficiency is a highly common practice. To compare, network optimization utilizes the same idea in the form of packets: On the Internet, the network breaks an [data/messages] into parts of a certain size in bytes. These are the packets [2].

1.2 Where is a Chunk List Used?

The basis of the data structure makes it useful for storing very large and very small amounts of elements. Benefits shine especially when a list is unsorted:

- Fast searching
- Fast removal
- Fast insertion

In any scenario, a chunk list can be used in place of an ArrayList especially, as well as something such as a Binary Search Tree, as searching may be faster based on processing power.

1.3 Benefits

Implementation is easy and short, and sorting is quick even with large amounts of chunks. With the ease of adjustability of chunk size, the capacity can be modified to allow for faster and more efficient speeds.

1.3.1 Area-Specific Usage

The most likely real-life scenario in which a chunk link would be preferred would be for video games and optimization – particularly in the sense that many video games today, especially sandbox-style games such as Minecraft use a process called ‘chunking’ for map data – combining and decompiling maps into ‘chunks’ in order to load only the parts of the map within a radius of the player for the purpose of increased performance by reducing the amount of entities loaded within a visible area [5].

2. IMPLEMENTATION

2.1 Construction

The basis of the chunk list is the inner list. This is best implemented using some sort of dynamic list, such as ArrayList (or List in C#). This inner list will start out with a single list on the inside.

Constructor must include an integer, the chunk size. Otherwise, revert to a default size. New lists (chunks) will only be added to the main list when the chunk at the end has reached capacity. Likely, the best implementation for a constructor would be to set the chunk size to the square root (as an integer) of the amount of data being stored, as in testing this has yielded the fastest performance. [4]

A chunk list may be implemented with generics (or templates) so long as the generic type is comparable.

Example:

```
using System;
using Sytem.Collections.Generic;
using System.Threading.Tasks;

class ChunkList<T> where T : IComparable
{
   private List<List<T>> myList;
   private int chunkSize;

   private const int DEFAULT_SIZE = 1000;

   public ChunkList() :
this(DEFAULT_SIZE)
   {
   }

   public ChunkList(int chunkSize)
   {
      this.chunkSize = chunkSize;
      myList = new List<List<T>>();
   }
}
```

2.2 Multithreading Methods

Multithreading is an especially important part of chunk list implementation, as the basis of the list’s speed is primarily the result of concurrency. For most methods in a chunk list, a new thread can be created for each chunk to be iterated through.

A good example of this lies within C#’s Parallel.ForEach method, which will be referred to for this type of operation.

Thread synchronization is not required when iterating, however keeping track of the thread state is important in some instances.

2.3 Index-Based Methods

Accessing or modifying an element at a specified index (such as get, set, or removeAt methods) is somewhat more complex than in a regular list.

To get the chunk where the position would be located, divide the index by the chunk size and cast it to an integer: $chunk = int(index / chunkSize)$

To get the position in the chunk where the index would be, use modulo on the index by the chunk size: $chunkPosition = index \% chunkSize$

Access via list*[chunk][chunkPosition]* (Where list is the main list inside the class).

Index-Accessing Example:

```
private int convertIndexToChunk(int index)
{
   return index / chunkSize;
}
```

```
private int convertIndexToChunkPos(int
index)
{
   return index % chunkSize;
}
```

Index Example:

The following example demonstrates accessing an element at index 8 in a chunk list containing numbers 0 – 10 with chunk size 5. (See Figure 2)

Accessing the chunk: int(8 / 5) = 1

Accessing the chunk position: 8 % 5 = 3

Fig. 2:

Chunk 0

[0]	[1]	[2]	[3]	[4]
0	1	2	3	4

Chunk 1

[0]	[1]	[2]	[3]	[4]
5	6	7	***8***	9

Chunk 1

[0]	[1]	[2]	[3]	[4]
10				

2.3.1 Index Issues

One issue with using indices in a chunk list, however, is the problem where items flow left within the chunk but do not migrate left from one to another if a chunk has an open slot. To implement so may hinder performance during removal.

However, a very simple solution would be to use recursion, such as within a try-catch statement using the index + 1.

Example:

The following example demonstrates a solution to the problem by counting up the index until an open position is found:

```
public T get(int index)
{
   if (index >= size()) throw new
ArgumentOutOfRangeException();
   try
   {
      return
myList[convertIndexToChunk(index)][conv
ertIndexToChunkPos(index)];
   }
   catch (ArgumentOutOfRangeException)
   {
      return get(index + 1);
   }
}
```

2.3.2 Chunk Resizing

Should our data set grow marginally larger, we may need to resize our list. To do so however, means we'll need to rebalance our list, which is especially important if the chunk size we're changing to is smaller than the current one.

We can make a temporary list containing all of our old items, change the chunk size, clear our old list, and then reflow our data back in. While somewhat costly performance-wise, this is an operation that should not be necessary to occur often.

If the chunk size we want to adjust to is larger than the current one, however, we can simply leave the list as is and allow the elements to re-fill the chunks that are not yet at capacity.

Example:

```
public void setChunkSize(int
newChunkSize)
{
   if (newChunkSize > chunkSize)
   {
      chunkSize = newChunkSize;
   }
   else
```

```
    {
        var items = getList();
        chunkSize = newChunkSize;

        clear();

        foreach (var item in items)
        {
            add(items);
        }
    }
}

public List<T> getList()
{
    var items = new List<T>();

    foreach (var currentList in myList)
    {
        foreach (T currentItem in currentList)
        {
            items.Add(currentItem);
        }
    }

    return items;
}
```

2.4 Element Operations

We should include the standard operations equivalent to an array list. Manipulating the data will work very similarly but will focus more on the data chunks and the usage of concurrency for each method.

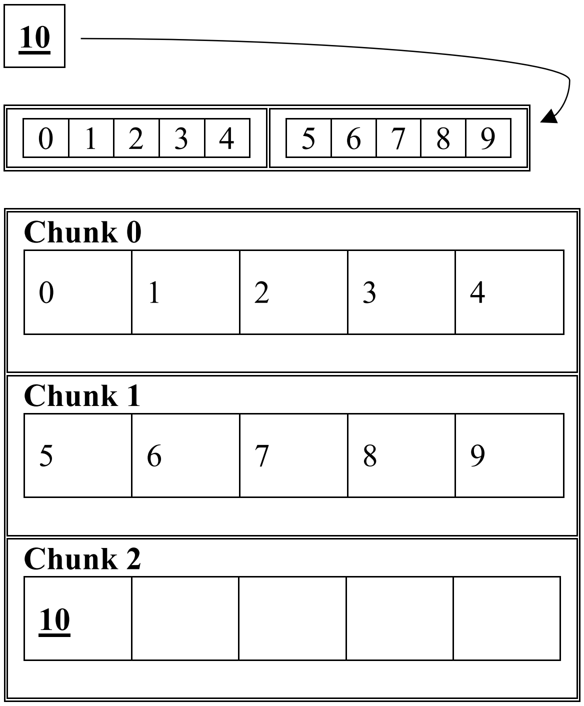
2.4.1 Adding Elements

Adding elements to a chunk list is simple; however, it does require that we check if each chunk is at capacity. Getting the size from the chunk should be Big-O (1), so this should not increase runtime marginally whatsoever.

An element will naturally fall into the first open spot, or the first chunk that is not at capacity. (See Figure 3)

If all chunks are at capacity, however, we need to add a new chunk to our list, then add the item it.

Fig. 3:



Example:

```
public void add(T t)
{
   foreach (List<T> currentList in myList)
   {
      if (currentList.Count != chunkSize)
      {
         currentList.Add(t);
         return;
      }
   }

   myList.Add(new List<T>());
   myList[myList.Count – 1].Add(t);
}
```

2.4.2 Removing Elements

Removing elements is one of the fastest computational operations in a chunk list. This is where we can start using multithreading to our advantage.

To remove an element, we can use a parallel for loop to concurrently check each chunk for the item.

We can use a binary search to get the index that we’re looking for.

This is also where we need to be able to have access to the thread’s state when we’re looping through each chunk. If we only want to remove the first found instance of an element, we need to immediately break out of the parallel for loop.

To remove all instances of an element within the list, we can still use a parallel for loop, and just call a removeAll method on each chunk.

To clear the entire list, we can simply call clear on the main list (containing the chunks).

Example:

```
public void remove(T t)
{
   Parallel.ForEach(myList, (currentList, state) =>
   {
      int indx = currentList.BinarySearch(t);

      if (indx >= 0)
      {
         currentList.RemoveAt(indx);
         state.Break();
      }
   });
}

public void removeAll(T t)
{
   Parallel.ForEach(myList, (currentList) =>
   {
       for (int i = 0; i < currentList.Count; i++)
       {
           if (currentList[i].Equals(t)) {
               currentList.RemoveAt(i);
               i--;
           }
       }
   });
}
```

2.4.3 Searching

Searching for an element is also where chunk lists shine. Once again we can use concurrency to get the shortest possible runtime, as now we can use a parallel for loop not only on the list itself, but on each chunk. Essentially, we can check most items in the list at the exact same time, meaning our runtime will be marginally smaller than using a linear search at worst case, and in the best case, a binary search.

Example:

```
public bool contains(T t)
{
   bool found = false;
   Parallel.ForEach(myList,       (currentList,
state) =>
   {
      Parallel.ForEach(currentList,
(currentItem) =>
      {
         if (currentItem,Equals(t))
         {
            found = true;
            state.Break();
         }
      });
   });

   return found;
}
```

2.4.4 Sorting

Sorting our list is a fairly complex operation, similarly to searching. To properly sort our list, we do have to make a temporary list containing all elements of our chunk list. To do otherwise would only sort the chunks, which is not ideal as we do not know which order they will be inserted in.

Using our temporary list, we can clear our main list and simply reflow all of our items back in after sorting it.

Sorting Example:
For this example, I simply used the sort method implemented within C#'s List class, which follows the following rules [3]:

- *If the partition size is fewer than 16 elements, it uses an insertion sort algorithm.*
- *If the number of partitions exceeds 2 * log N, where N is the range of the input array, it uses a heapsort algorithm.*
- *Otherwise, it uses a quicksort algorithm.*

```
public void sort() {
   var items = getList();
   items.Sort();
   clear();
   foreach (T item in items) {
      add(item);
   }
}
```

3. COMPLEXITY

3.1 Complexities – Basic Methods

We can find the computational complexities by comparing those of a standard abstract list and dividing up the data based on the equivalent methods and the chunks of data as individual lists, acting as the size of each sub-list containing its own number of elements. In instances where we see C * N for example, this would represent either the entire list, or log C * log N representing a divisional portion of the data for the sake of computation complexity.

Complexities are listed with the following variables:

- C being the number of chunks currently in the list.
- N being the number of elements per chunk.
- P being the number of processors.
- I being the index input for the operation.

OPERATION	**AVERAGE CASE**	**WORST CASE**
Add	Θ(C)	N/A
Remove	Θ((log C * log N) / P)	Θ(log C * log N)
RemoveAll	Θ((log C * N) / P)	Θ(log C * N)
RemoveAt	Θ(1)	Θ(C * N - I)
Set	Θ(1)	Θ(C * N - I)
Get	Θ(1)	Θ(C * N - I)

3.2 Complexities – Additional Methods

These methods have been computed based primarily off of instant computations or the computation complexity of the base method being used (e.g. the Sort method being derived from the built-in Sort method as mentioned in **2.4.4**).

OPERATION	**AVERAGE CASE**	**WORST CASE**
GetList	Θ(C ^ 2 * N)	N/A
Contains (Search)	Θ((log C * log N) / P)	Θ(C * N)
Size (Count)	Θ(C)	N/A
SetChunkSize	Θ(1)	Θ(C ^ 2 * N)
Sort	Θ(C * N * log N)	Θ(C * N ^ 2)

4. USAGE

4.1 Modern Usage

The biggest potential usage for the chunk list would definitely be for video games – any time a large amount of objects or map data would need to be contained or searched through, the data structure would provide the most efficient mean to load portions of data as well as find objects within the chunks.

4.1.1 Unit Test & Benchmarks

A working unit test can be found on the GitHub repository [4] comparing results of data computations using a chunk list, a chunk list with the chunk size set to the square root of the data quantity, and a standard array list.

4.1.2 Competition

A lot of data structures could potentially either replace or be replaced by the chunk list – in smaller amounts of data, an array list could of course be used, or even a binary tree if searching through the data is the key important part. The biggest selling point is of course the concurrency: any time speed is the biggest factor in manipulating data, the chunk list is a likely competitor compared to a graph or binary search tree.

4.2 Related Work

As mentioned previously, Minecraft's chunking system was a drawn upon concept in development of the data structure. The idea of chunking data into smaller sets in order to access the most important parts and use different threads and processor cores to manipulate data concurrently provided a baseline to the core design of the structure. Cryptocurrencies and other mathematical challenges such as Project Euler also were key to finding an importance in speed in data manipulation, as the usage in problem solving and video game optimization were too key ideas meant to be solved through the implementation of the data structure.

REFERENCES

[1] Cherry, K. “Chunking Technique for Improving Memory”, Verywell. Retrieved December 25, 2017, from: https://www.verywell.com/chunking-how-can-this-technique-improve-your-memory-2794969.

[2] HowStuffWorks.com, “What is a packet”. Retrieved December 25, 2017, from https://computer.howstuffworks.com/question525.htm.

[3] Microsoft, “List<T>.Sort Method ()”. Retrieved December 25, 2017, from Microsoft Developer Network: https://msdn.microsoft.com/en-us/library/b0zbh7b6(v=vs.110).aspx.

[4] GitHub Repository, “Chunk List Unit Test”. Retrieved July 5, 2018, from GitHub: https://github.com/danielathome19/Chunk-List/blob/master/Unit%20Test/UnitTest.cs

[5] Minecraft Wiki, “Chunk Loading”. Retrieved July 5, 2018, from Wikia: http://technical-minecraft.wikia.com/wiki/Chunk_Loading

APPENDIX

Method Headers & Complexities

OPERATION	**AVERAGE CASE**	**WORST CASE**
Constructor *() : this(DEFAULT_SIZE)*	*N/A*	*N/A*
Constructor *(int chunkSize)*	*N/A*	*N/A*
void **Add** *(T t)*	Θ(C)	*N/A*
void **Remove** *(T t)*	Θ((log C * log N) / P)	Θ(log C * log N)
void **RemoveAll** *(T t)*	Θ((log C * N) / P)	Θ(log C * N)
void **RemoveAt** *(int index)*	Θ(1)	Θ(C * N - I)
int **ConvertIndexToChunk** *(int index)*	*N/A*	*N/A*
int **ConvertIndexToChunkPos** *(int index)*	*N/A*	*N/A*
void **Set** *(int index, T t)*	Θ(1)	Θ(C * N - I)
T **Get** *(int index)*	Θ(1)	Θ(C * N - I)
List<T> **GetList** *()*	Θ(C ^ 2 * N)	N/A
bool **Contains** (Search) *(T t)*	Θ((log C * log N) / P)	Θ(C * N)
int **Size** (Count) *()*	Θ(C)	*N/A*
void **SetChunkSize** *(int size)*	Θ(1)	Θ(C ^ 2 * N)
void **Sort** *()*	Θ(C * N * log N)	Θ(C * N ^ 2)