

# Relocatable Data Structures

Ver. 0.3

Yordan Gyurchev  
[yordan@gyurchev.com](mailto:yordan@gyurchev.com)

This document is a draft. If you have any suggestions, comments, links to implementations or other feedback send them to the email address above. You can download the latest version (revision) of this document at <http://www.gyurchev.com/docs/relocate.pdf>

# Introduction

For some time now memory has been the bottleneck in performance critical applications. Current tendency is to gain more performance in hardware by exploiting concurrency – multiple processors/cores (processing unit - PU) in the same machine. Having multiple PUs fighting for the same memory resource (shared memory) quickly results in bus contention that we as programmers have to aggressively tackle if we are to take full advantage of the available computational power.

In concurrent environment every PU has access to the shared memory through the main memory bus but also has at its disposal local memory that is dedicated to that PU and is extremely fast to access. The key to greater performance is to maximize usage of local PU memory and minimize usage of shared memory. In order to do so the programmer has to constantly move data to and from this local PU memory sometime duplicating data in multiple places depending on the data decomposition.

This document sets out to enumerate and explore data structures that can be easily moved from and to local PU memory and used without additional work.

This document doesn't delve into concurrent programming topics such as task and data decomposition (see [1] for more information on the topic).

## 1 Relocatability of data structures

### 1.1 *Serial terms in parallel environment*

In a parallel environment, where multiple processing units work together to achieve better performance, it is important to measure the time spent in the serial terms of the parallel algorithm. A serial term of a parallel algorithm is one that sets up the concurrent execution of tasks and then puts together the result of the work.

$$T_{\text{total}} = T_{\text{setup}} + T_{\text{compute}} + T_{\text{finalization}}$$

The setup and finalization steps cannot be parallelized therefore we are limited on performance and we need to minimize the time spend there.

In preparation (setup) step for a PU task we usually:

- Prepare data for the task according the chosen data decomposition pattern
- Send performance critical data to the PU local memory
- Manage synchronization on shared data (if any) to avoid racing conditions
- Satisfy task execution constraints (order in which tasks should be executed)
- Other work required by the specifics of the algorithm

In the finalization step of a PU task:

- Transfer back updated data (if any)
- Manage synchronization on shared data (if any)
- Merge results with other tasks (if required)
- Other work required by the specifics of the algorithm

## ***1.2 Need for relocation and methods***

When it comes to parallel algorithms there might be multiple reasons we want to move the data structure around in memory:

- Performance – PU local memory is much faster than shared memory and reduces memory bus contention
- Having a stable snapshot of data in local memory to avoid racing conditions

In theory every data structure can be relocated. Programmers have been doing things like that for years when it comes to saving data structures into files.

The most common method is to convert any address pointers into indices and when the structure is loaded back in memory to patch the pointers. So basically moving data structure from one place in memory to another requires a run through the data patching pointers to reflect the memory address change.

Although this method is quite acceptable when it comes to file operations (file system will always be magnitudes of time slower than a patching step) it has an unacceptable overhead for us. Not only that - the patching steps happen during our most expensive serial parts of the task: setting up and finalization.

Alternatively we can use, instead of a normal data structure, one that can be moved freely in memory without the overhead of a patching step.

## ***1.3 Relocatable data structure***

A relocatable data structure for us is one that can be easily moved from one place in memory to another without further processing. It is important that in achieving that it doesn't impair performance (using that data structure) and ease of use.

It is important that a data structure knows its size so it can be moved from one place to another without additional knowledge.

## ***1.4 Local memory constraints***

An important constraint of local memory is size. Local PU memory is usually very limited: from 16K on current generation consoles to 256K on next gen. Algorithms and structures that intend to use local memory have to take account of available local memory.

Another constraint of local memory is speed of data transfer. Usually when it comes to transferring data to and from local PU memory there are channels that operate with greater priority.

### ***1.5 Buffering techniques***

On some occasions the data that needs to be processed is far larger than the available local PU memory. In these cases we process data in chunks in local memory. Effectively streaming small chunks into memory thus keeping the PU busy.

### ***1.6 Local memory variants***

On some platforms access to local memory is very explicit and needs to be manually managed. In such situations the programmer has full control over what data is transferred to and from the local memory, when and how.

On other platforms local memory comes in the form of data caches that are automated but still expose some form of control (prefetch instructions etc).

In both cases performance will benefit from data structures that have better relocatability and locality properties.

## **2. Data structures**

### ***2.1 Simple structures***

Simple data structures are in general arrays. They usually have no extra information that supports the structure. They don't contain extra pointers and need no extra work when moved. They are solid blocks of data and can be easily transferred around in memory.

Important property of arrays is that if processed iteratively they can be sliced (at element boundaries) and processed in pieces should it exceed the size of available local PU memory.

There are two types of arrays: fixed arrays and dynamic arrays.

#### **2.1.1 Fixed arrays**

Fixed arrays are normal fixed sized arrays - they can't grow and they can't shrink. They always have the same size in memory. However, the size information is known only by the compiler and the array needs to be moved run-time. In order to make a fixed array relocatable we have to make known its size run-time. That needs

not to alter the array in any way. It can be achieved by using a fixed array template [?].

As we know the size of the array compile time we can always decide compile time if a fixed array will fit or not fit into the available local memory.

### **2.1.2 Dynamic arrays**

Dynamic arrays can grow and shrink in memory. That imposes problems, as the finalization step of the task needs to do additional work if the dynamic array has grown or shrunk while it was in local PU memory. If the algorithm requires such resize operation to take place you need to carefully consider the cost of the finalization step as it falls in the serial term of the algorithm.

Although we don't have the advantage of knowing upfront if a dynamic array will fit in the available local PU memory we can still take advantage of buffering methods if iterative processing is to be performed.

## **2.2 Index Pointers structures**

Index pointer structures are in general normal data structures where instead of pointers the structure elements use some kind of index or offset (local pointers) to reference other elements. This way the structure can be moved in memory but as long as we know the base address we can still operate on it without a problem.

The structure is usually a solid block in memory representing the maximum elements that can reside in the structure. It can also be more dynamic but that would come at a certain price. (see dynamic arrays)

It is more difficult to use such structures with buffering techniques. We need the whole structure in local memory, as we don't know where our index-pointers are pointing. It can be anywhere in the structure.

The small price we have to pay is calculating the absolute address from the index/offset and the base address of the structure.

Examples of structures that can be implemented this way:

- Linked lists
- Graphs
- Trees
- Hash Tables

An important quality of the local pointers method especially when it comes to local PU memory is that we can use 16 bits to represent them. Even if we use the whole of the local address space (0xffff) that would leave us with just two bytes per element before filling the 256K of local address space. Most importantly that leaves us with more space for useful data.

## 2.2.1 Internal allocation pools

It is very easy to make linked list allocation pool for a data structure where all entities are the same size and the maximum entity count is limited (by size). Allocation and disposal happen in  $O(1)$  time and no extra memory is needed for bookkeeping. Most importantly code can operate on the structure (even in local memory) allocating and disposing elements without actually touching memory routines.

Drawbacks of such allocation scheme is that:

- We need to move the bulk of data (whole block) regardless of how much is actually used
- It has limited maximum number of elements

## 2.3 Implicit data structures

An implicit data structure doesn't store any extra bookkeeping structure information. It stores the elements and nothing else. The extra information is implied – hence implicit. The implied information is the order in which the elements are stored.

Implicit data structures can be extremely fast. They don't require memory access in order to calculate the address of a structure element.

Implicit data structures can be sometimes quite rigid, as the order of the data structure can't be changed dynamically. This means that sometimes they can be quite wasteful when used inappropriately.

Implicit structures are also ideal for memory relocation purposes, as they don't contain any extra information, pointers, etc. Simply copying the data structure is fine. In that they resemble the simple data structures (arrays).

When it comes to applying buffering techniques to them they can be more cumbersome than the arrays but less so than the other structures as we have the structure knowledge compile time.

Lets look at some examples of simple implicit binary tree data structures:

**Example 1:** Defining an implicit binary tree is to calculate the addresses of the children recursively based on the parent address.

```
ChildLeft = Parent >> 1;  
ChildRight = Parent >> 1 + 1;
```

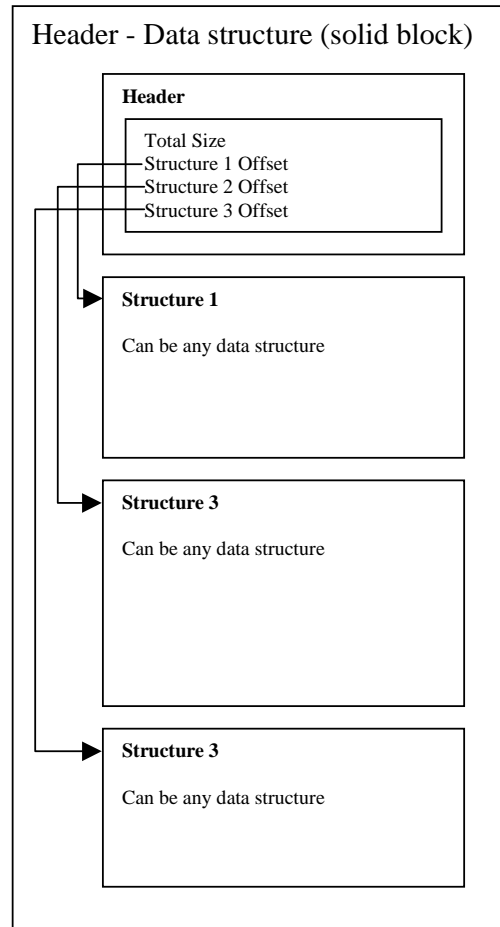
**Example 2:** Heaps. A binary heap is a priority queue data structure. It supports both insertion and extract-min in  $O(\lg N)$  time.

## 2.4 Header-Data Data structures

Too often a task is more complex than just a single algorithm and a data structure. In many cases it requires a fusion of fundamental data structures and algorithms to work together in order to achieve the final goal. In such cases we want to use multiple data structures but still have the advantage of relocatability and locality.

We can store multiple data structures in a single block of memory and then attach a header that can act as a geographical map – storing the offsets of the different sub structures. This way we can relocate the whole block but still have the advantage of not having to worry about dependencies and pointers.

This approach works on a higher level and requires careful design identification of needed data structures.



## 3. Enter the algorithm

It is important to carefully plan and think about our programs that exploit concurrency. Data structures on many occasions imply an algorithm but there is a number of ways a number of algorithms can work together.

Algorithms that maximize locality of reference and minimize amount of memory use are essential when it comes to performance.

### **3.1 In-place algorithms**

Everyone knows about the old XOR swap trick. Apart from being geeky (and very outdated) it has one important property – it doesn't need extra temp variable (register) to swap the values. It makes an in-place swap.

We are working with much larger quantities of data than a single variable these days but the principle applies. In-place algorithms have the result values occupy the same space as the source values. This doubles the space our data structures can span as we don't have to fit both source and destination in local memory.

Examples include:

- In-place sort algorithms: heapsort, insertion sort, selection sort, diminishing increment sort, shell sort, comb sort, bubble sort, etc. Quicksort can also be considered a in-place sort algorithm if we ignore the  $O(\log n)$  space used by the recursion.
- Selection algorithms – these usually have good exploitable concurrency as well

### **3.2 Other algorithms?**

## **Summary**

Data structures that can be moved easily in memory can be very useful when it comes to improving memory access performance in a multiple processor environment. However extreme care should be taken during design phase to identify the need for them as they sometimes reduce unnecessary code readability, extensibility and flexibility.

## **Links To Implementations:**

Thatcher Ulrich - relocatable templated hash table

<http://cvs.sourceforge.net/viewcvs.py/tu-testbed/tu-testbed/base/container.h?view=markup>

## **Reference**

[1] Mattson, Sanders, Massingill; “Patterns for parallel programming”