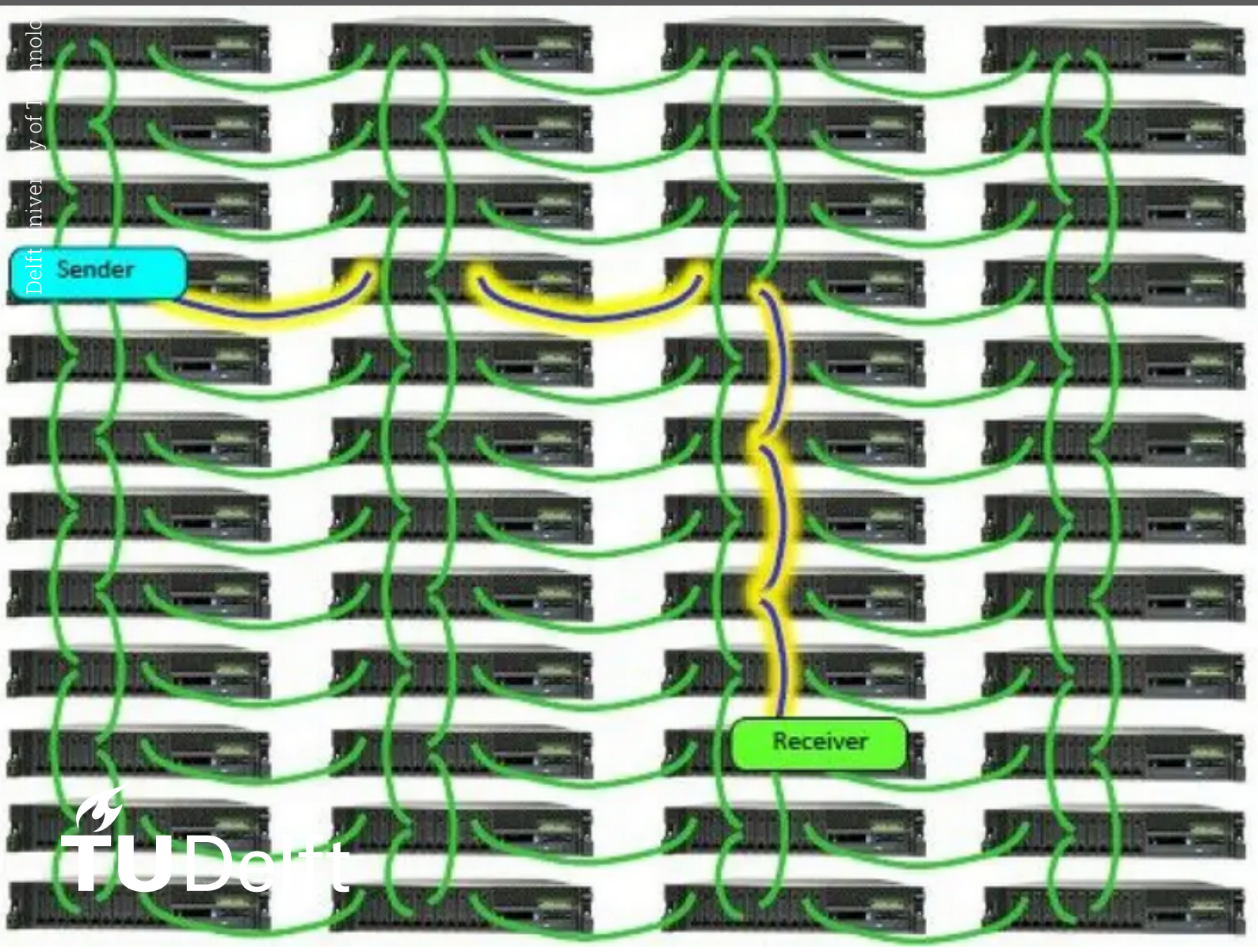
Graph processing on systems with disaggregated memory

Aiding financial crime detection in large datasets

Keyvan Khalili



Graph processing on systems with disaggregated memory

Aiding financial crime detection in large datasets

by

K. Khalili

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended publicly on Monday June 15, 2026 at 9:00 AM.

Student number:	5118190	
Thesis committee:	Prof. Dr. HP. Hofstee	TU Delft, supervisor
	Dr. K. Atasu	TU Delft
	Dr. Ir. Zaid Al-Ars	External Advisor

TU Delft

Contents

Abstract

With the rise of memory costs and the persistent under-utilization of memory in clusters, researchers have begun exploring alternative approaches to improve memory efficiency and reduce operational costs. Resource disaggregation is becoming increasingly common and sought after, driven by the emergence of new interconnect standards such as CXL and, previously, OpenCAPI. While the industry is primarily moving toward memory pooling, where memory is dynamically provisioned among applications or virtual machines, this work investigates distributed memory disaggregation and sharing. IBM's Power10 processors include hardware support that enables multiple systems to directly share memory. However, few applications have been developed to take advantage of disaggregated shared memory.

Since Memory Inception, Power10 processors' memory disaggregation hardware, is not yet fully operational, a ThymesisFlow prototype, upgraded to support a shared disaggregated memory system with the help of Apache Arrow, is used to implement a practical application. The selected application is a graph processor capable of detecting money laundering patterns in financial transaction graphs in real-time. These patterns yield transaction features that machine learning algorithms can use to identify fraudulent financial transactions.

Our proof-of-concept implementation enables the creation of a distributed graph, represented as Apache Arrow tables, that can process large datasets in real-time. The graph resides in a shared disaggregated memory region and can be accessed by multiple systems without data copying, incurring lower latency penalties than network-based data retrieval. The distributed graph processor was developed and tested using the ThymesisFlow prototype provided by the Hasso Plattner Institute.

1 Introduction

Low memory utilization has long been a major concern in clusters. The problem arises due to the runtime variability of the resources required during workloads and over-provisioning of memory in virtual machines. In a cluster, each node is usually provisioned with memory matching the maximum memory usage of the given workload to ensure that the program does not run out of memory, which can result in crashes. Moreover, users often over-estimate their workload's memory requirements to prevent crashes due to memory shortage[1]. There are multiple studies that show this pattern of memory under-utilization, with google seeing 40% memory usage in its Borg clusters[2], and Meta showing that the memory usage does not exceed 50% in 50% of VMs[3].

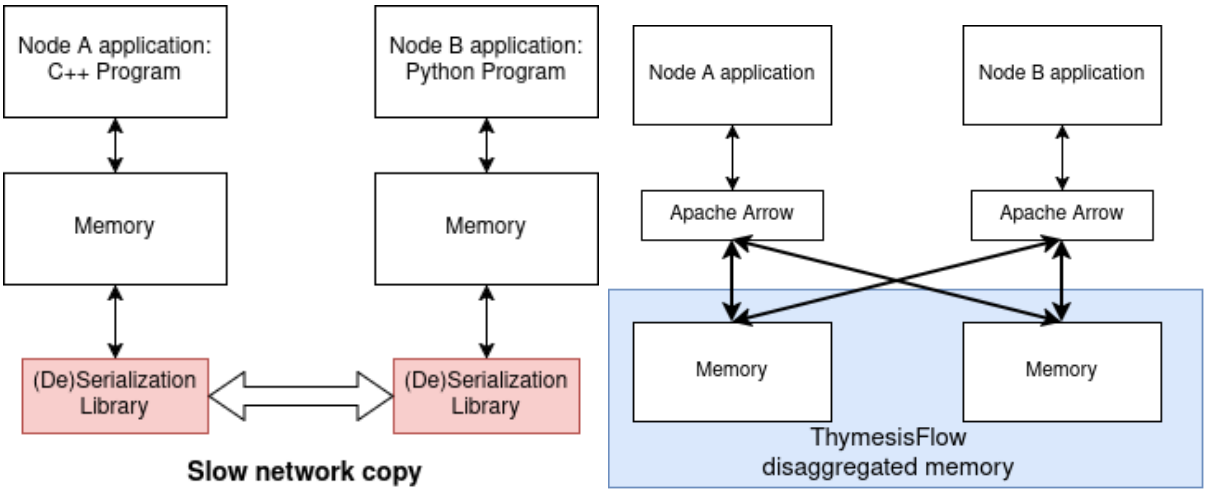
Additionally, in distributed workloads, such as big data pipelines, that span multiple nodes in a cluster, data are required to be duplicated in all nodes where they are needed, since each node can only access their own memory. This data duplication increases the overall memory usage, reducing memory utilization efficiency. The addition of more nodes to a cluster to improve distributed workloads is known as horizontal scaling (or ”scale out”). Horizontal scaling does not increase the memory that is available on each node and, due to worst case memory provisioning and data duplication, is not memory efficient.

Another way to scale the performance of the nodes in a cluster, while maintaining memory efficiency, is to add more resources to a single node, by adding more CPU, memory, storage, or network[4]. This is known as vertical scaling (or ”scale up”). While scale up systems do not suffer from memory inefficiencies of scale out systems, they provide limited scalability. In scale up systems, such as IBM's symmetric multiprocessing (SMP) systems, there is a limit to additional processors (CPUs) that can be added as data in the CPU caches have to remain consistent with coherence traffic limiting their scalability.

Disaggregated memory systems, where memory is separated from compute resources (i.e., CPUs/GPUs) and is connected with high-speed networks, can provide a solution for under-utilization of memory[1]. In these systems, based on run-time requirements of workloads and VMs, the memory can be dynamically re-provisioned[5]. Some effort has been made to improve the memory efficiency of such system architectures when using CXL pooled memory[6]. However, the disaggregated memory system used in this thesis tries to provide a solution to tackle the scaling problems of both scale up and scale out systems, that is, allowing horizontal scaling by adding nodes while increasing memory utilization when running a single workload by allowing all nodes to access all of the available memory.

The original ThymesisFlow disaggregated memory prototype, currently operational at Hasso Plattner Institute (HPI), consists of two nodes, one acting as a compute and the other as the memory node. The compute node can borrow (temporarily ”steal”) memory from the memory node[7]. The earlier ThymesisFlow prototype acted as an asymmetrical memory pool system where the memory node, which lends its memory away, could no longer use the lent memory. By implementing some wrappers, researchers at HPI, modified the way the disaggregated memory is treated in the operating system, allowing the lent memory to be seen and used by the lender (memory) node as well[8]. However, the disaggregated memory region could not be safely shared between the two nodes due to the lack of cache coherency

protocols[7]. In a later work[9, 10], software cache coherency was implemented using gRPC clients and servers that synchronize the two nodes with Apache Arrow as an immutable and language-independent in-memory data format. The addition of software cache coherency and the usage of the disaggregated memory as a mostly read-only memory region allows it to be used as shared memory. So, extra nodes can be added to the setup that can read/write from/to the shared memory using processors load/store instructions. As ThymesisFlow link has a much lower latency than network transfers, the data is not required to be copied to these additional nodes. This increases the efficiency of memory utilization while scaling the resources of the setup. Figure 1.1 shows how the ThymesisFlow prototype with Apache Arrow differs from traditional network based cluster setups.



(a) Traditional network based data sharing, where expensive (de)serialization and copy takes place

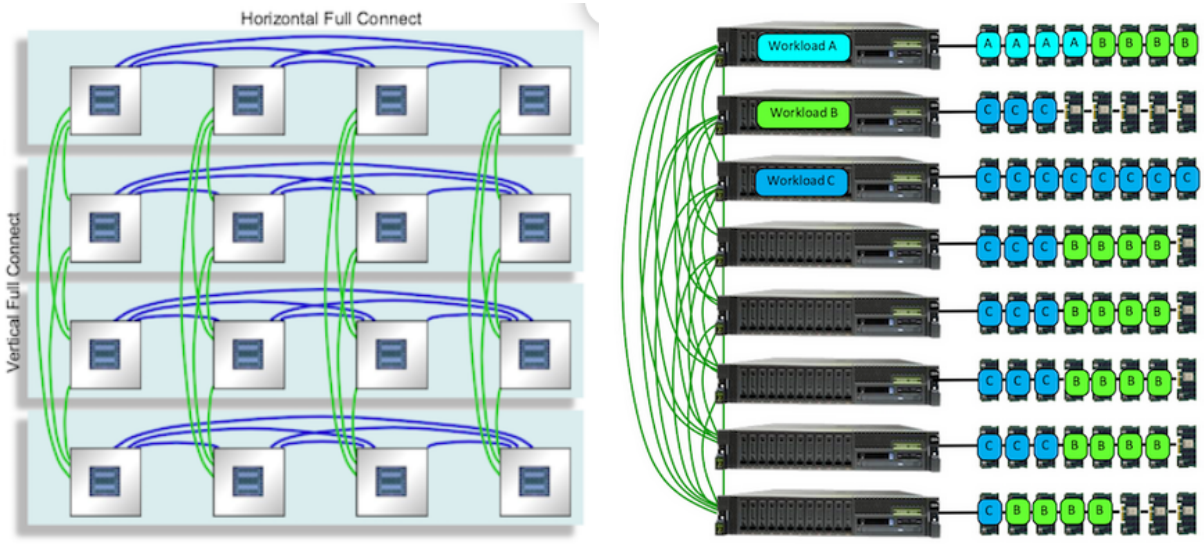
(b) ThymesisFlow based disaggregated memory with Apache Arrow in-memory format

Figure 1.1: Traditional network based vs ThymesisFlow based disaggregated memory[10]

The current ThymesisFlow setup provides asymmetrical memory sharing where the memory of one node is shared with the other, but not the other way around. Although this prototype can be upgraded to allow bi-directional memory sharing between the nodes, it is not being pursued or in development. Furthermore, this setup suffers from high latency (850-950ns[8]) and low bandwidth (6-12.2GB/s)[10]. Local memory in ThymesisFlows's Power9 server in comparison has a latency and bandwidth of 74ns (37 cycles + 64ns) and 240GB/s respectively[10]. CXL (4.0) attached memory, while performing worse than local memory, has a theoretical latency of 170-250ns[11]. A real characterization of four different CXL memory expanders, two using DDR4 and two using DDR5 DIMMs, showed a latency and bandwidth range of 214-394ns and 18-52GB/s. These performances are observed when the CXL attached memory expander is directly connected to the socket running the benchmarks. Accessing the CXL attached memory with inter-socket communication adds 94-227ns of latency[12].

IBM has basically evolved the ThymesisFlow prototype in the hardware of the Power10 processor. The feature is called "Memory Inception" where it allows rack scale memory clustering. Meaning that a distributed disaggregated memory can be shared among many processors, with up to 16 processors requiring a single hop to access the disaggregated memory. These form a massive distributed memory pool with, SMP like, near local memory latency[13]. However, this feature has not been offered as a product as it requires firmware development and operating system modifications that have not been made yet. The cache coherency guarantees of *Memory Inception* are limited to the memory within a single node, which is the same as the ThymesisFlow prototype. Figure 1.2 shows how Power10 processors can be interconnected across the rack to form a distributed disaggregated memory.

Looking at the available literature, we could not find any application that is implemented to make use of disaggregated memory. The goal of this thesis is to find a suitable application that can benefit from the increased capacity and scalability provided by disaggregated memory. Then by comparing their performance to local memory, an attempt is made to make predictions on what the performance of such an application will be should it be run on a disaggregated memory system that has better performance characteristics than ThymesisFlow setup. The most suitable application categories are



(a) Power10 Single Chip Module (SCM) inter-processor connections with each blue/green line indicating a PowerAXON connection

(b) use case of shared memory among directly connected neighbors with different NUMA latencies. Workload A and workload C have low latency requirement with data placed in closer racks. Workload B has relaxed latency requirement and is placed distant racks, i.e., NUMA regions with higher memory latency.

Figure 1.2: Inter-processor connection of Power10 processors[13]

as follows: databases, graph applications, machine learning, and virtual environments[1].

For the application to take advantage of the benefits of a disaggregated memory system, compared to the traditional network based memory distribution, the workload has to be latency bound with more random reads, which prevents the data from being effectively cached locally. Between the possible application categories, graph applications fit these metrics the most and was therefore chosen as the application class to be implemented.

Two graph applications were considered to be implemented. The first graph application considered was in genomics, namely Flye de novo genome assembler. The Flye genome assembler is used mainly for assembling large genome sequences. Genome sequencing mainly involves traversing large unknown error-riddled genome sequences and finding overlapping regions to provide an accurate repeat graph, which is a compact representation of repeated sequences in a genome[14]. This workload mainly consists of unpredictable reads with length of the reads depending on length of genome sequences or the overlapping regions. As data are re-used irregularly and some genome datasets are extremely large, requiring space efficient data representation[15] or nodes with high memory capacity, it is a workload that can benefit from disaggregated memory systems.

The other graph application considered was Graph Feature Preprocessing (GFP), which includes the processing of bank transactions in a graph to aid in the detection of fraudulent and illegal activities. This application is offered as part of the Snap Machine Learning (Snap ML) library found in the Python Package Index (PyPI). The graph processing in this application involves looking for subgraph patterns that could be an indication of criminal activity. These activities cannot be detected by only looking at transactions' information in normal tabular format. The detected subgraph patterns are then added as graph-based features to the original tabular data, providing more crucial information to the machine learning models for better detection of criminal activities[16]. As the graph can grow very large due to the large number of daily bank transactions, scaling of the application requires usage of multiple nodes that can provide the necessary memory capacity for the graph. However, accessing graph date for running sub-graph pattern recognition algorithms is unpredictable and makes efficient caching of the data in the nodes difficult. This workload, too, can benefit from disaggregated memory systems that can provide high shared memory capacity with low memory latencies.

The above leads us to the problem statements that are addressed in this thesis:

1. Is there a practical application that could leverage the advantages of disaggregated memory systems over scale up and scale out systems?
2. What are the main challenges of implementing workloads, specifically graph algorithms, on memory disaggregated systems?
3. How do the workloads perform in memory disaggregated systems compared to traditional network based distributed memory systems?

2

Background

In this chapter, the information and concepts behind ThymesisFlow, the used disaggregated memory prototype, are covered. However, only general information and relevant aspects of the prototype are mentioned. Detailed information about the design of ThymesisFlow and its characteristics can be found in previous works[7, 8, 10].

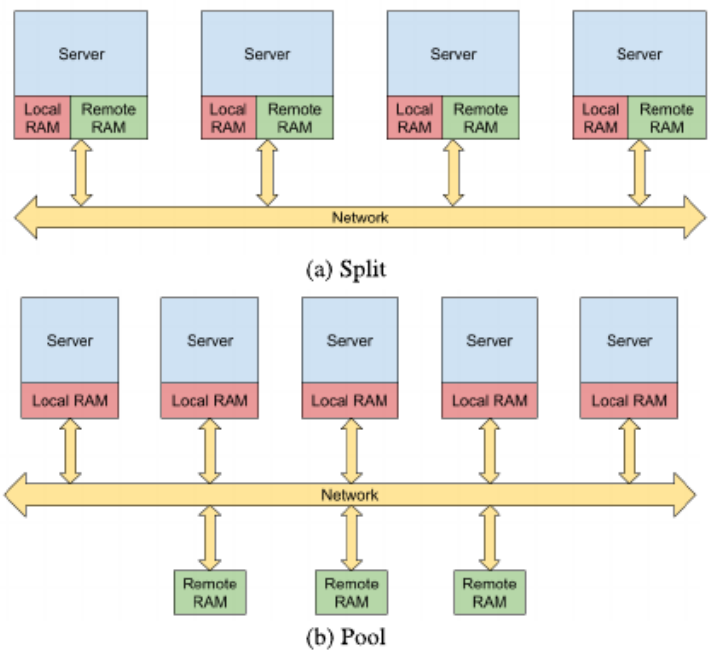
Furthermore, an introduction to Apache Arrow in-memory format is given. This includes the relevant API and memory layout that are required for understanding system behavior later on during the analysis of the benchmark results. Then, integration of Apache Arrow in the ThymesisFlow prototype for overcoming the limitations of the disaggregated memory are presented. Finally, an overview of the applications that can benefit from the ThymesisFlow prototype is given.

2.1. ThymesisFlow

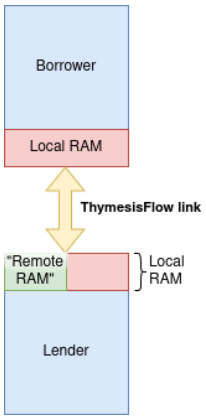
The idea of memory disaggregation has been proposed in different methods, with the goal of mainly tackling the over-allocation of memory in servers, which leads to reduced CPU utilization and increased total cost of ownership (TCO)[1]. There are two types of architecture for building remote or disaggregated memory, namely pooled or split memory, and can be seen in Figure 2.1a with ThymesisFlow falling under the pool architecture, seen in Figure 2.1b, since the lender node should not (and earlier could not) use the disaggregated memory that is lent. ThymesisFlow is a software-defined HW/SW co-designed memory disaggregation system where FPGAs, connected and interfacing with the memory bus using the OpenCAPI port, act as memory controllers and facilitate the transfer of data among nodes without the processors' involvement[7].

Although in the ThymesisFlow prototype the remote node has a processor, the remote memory is "stolen" by the borrower (compute) node and counts as used memory in the lender (memory) node. The architecture of ThymesisFlow can be seen in Figure 2.2, with FPGAs used as disaggregated memory NICs to translate effective (virtual) memory addresses to physical addresses in a few steps. The address translation done by ThymesisFlow can be seen in Figure 2.3. First, an effective address is read by the Power9 Memory Management Unit (MMU) of the compute node, with is then translated into a real address and sent to ThymesisFlow device. This real address is the internal address space of the ThymesisFlow device. Here, the memory address is translated by the ThymesisFlow Remote Memory Management Unit (RMMU) to an effective address in the memory node. This address is finally sent to the lending (remote) node.

Using remote memory via ThymesisFlow does not require modifying the operating system and the user application. In the original ThymesisFlow prototype, the remote memory is attached via Linux's memory hotplug feature that allows memory to be added/removed dynamically at runtime. When remote memory is hotplugged into the compute node, it is seen as memory in one of the two unused NUMA nodes. As such, a user application can use the remote memory simply by using the `numactl -M 16 <Application>` command on the borrower node, with NUMA node 16 the location where the remote memory is mounted. An example of this is provided in *Christian Pinto's presentation*.



(a) Split and Pool memory disaggregation architectures[1]



(b) ThymesisFlow memory disaggregation

Figure 2.1: High-level overview of memory disaggregation

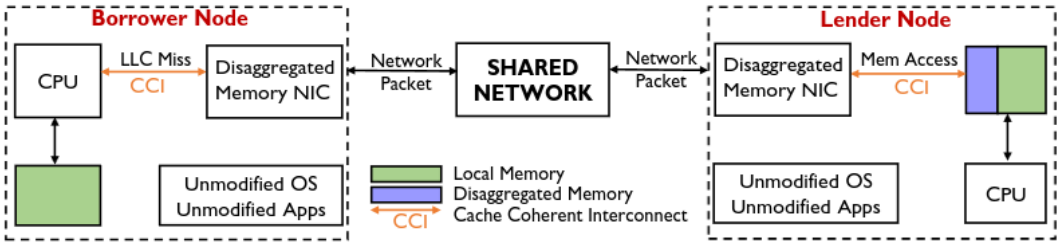


Figure 2.2: Simplified architecture of ThymesisFlow memory disaggregation system[17]

However, this thesis is run on another system at Hasso Plattner Institute (HPI), where ThymesisFlow is set up differently. The setup at HPI uses the same Power9 architecture processors and Alphadata9V3 FPGAs. In the HPI system, a library, named *libtfshmem*, maps the remote memory as a shared memory device file, which can be seen in both nodes at `/dev/mishmem-s1`. Appendix A covers how the ThymesisFlow memory is initialized in HPI's Power9 servers.

The remote memory being seen as a shared memory file further complicates how a user application can use it and requires the application to be modified, as `numactl` tool cannot be used as in previous demonstrations by *Christian Pinto*. Since it is essentially a wrapper on top of the original ThymesisFlow firmware, it still inherits the cache coherency characteristics of ThymesisFlow and OpenCAPI. However, by being accessible to both nodes, it opens the opportunity for the "stolen" memory to be used as a shared memory region instead, albeit without CPU cache line level coherency. So, the user application is responsible for maintaining coherence if the remote memory is used as shared memory.

2.2. Arrow in-memory format

Apache Arrow is an in-memory data format that aims to provide a unified memory format for columnar flat and nested data. Its data definition is language independent and is used as backend of numerous projects with support in multiple programming languages. By having a standardized memory format, different processes across different programming languages can use the same data without the need for an (de)serialization library or an interface language. Having the need to continuously (de)serialize

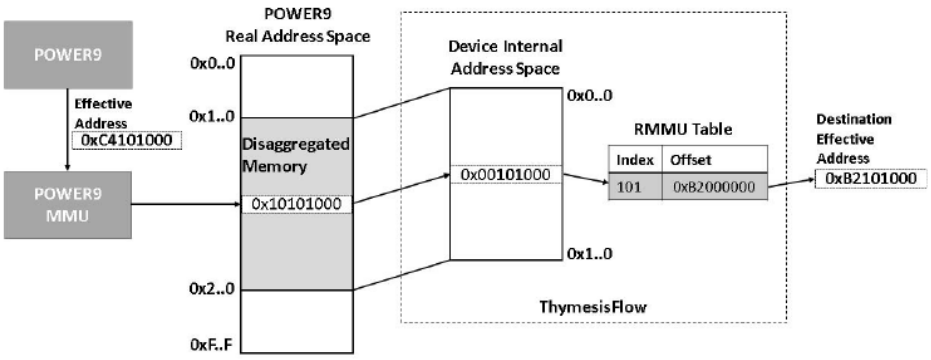


Figure 2.3: Translation of addresses across multiple layers in the compute (borrower node)[7]

the data among different processes causes inefficiencies. Apache Arrow, on the other hand, removes the need for de- and -serialization. As a result, it carries a lot less performance overhead for transferring data across processes. In this section, certain low level details of Apache Arrow are explained that are necessary in understanding how it is integrated with ThymesisFlow. Some less relevant concepts of Apache Arrow, that are well documented in previous work[10], are not repeated here. Aspects and APIs of Apache Arrow that are not deemed necessary for understanding the implementation are not covered in this paper.

Chunked Arrays

A chunked array in Apache Arrow allows the creation of a one-dimensional array in a discontiguous piece of memory. This is possible by holding zero or more Arrow Arrays, which is a sequence of values of the same type in a contiguous memory region. Chunked arrays are the main data abstraction in Apache Arrow that allows for data to be split in the disaggregated memory system.

Arrow Tables

Apache Arrow has two data structures that can be seen as a table in other data framework such as Python's Pandas. The two data structures are *Tables* and *Record Batches*. The main difference between the two is how the data within each column are located in memory. Columns in record batches are created from Arrow arrays of the same length and are always found contiguously in memory. Arrow tables, on the other hand, use chucked arrays as the building block of the columns. This allows the column data to be spread out in memory and have variable lengths. Figure 2.4 shows the difference between Arrow tables and record batches. One property of the Arrow table is that it can merge multiple Arrow record batches, with the same column data types, into a single Arrow table without copying the array data. Arrow only needs to create the meta data necessary to index the data correctly. Transformation of multiple record batches into a single table can be seen in Figure 2.6.

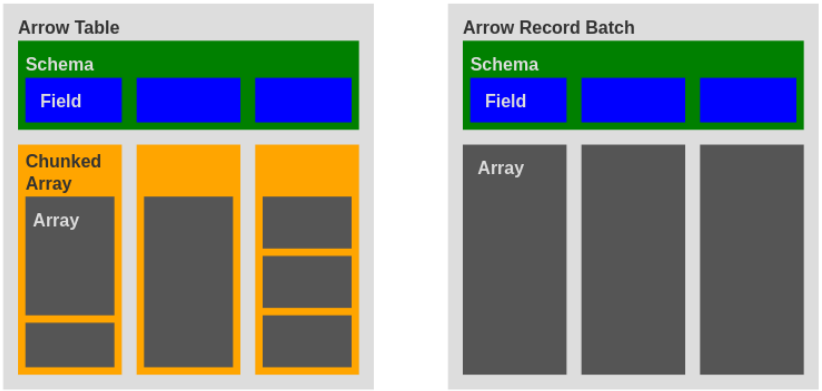


Figure 2.4: The underlying data structure of Arrow table (left) and Record Batch (right)

Memory pools

Apache Arrow provides multiple *malloc* implementations for memory allocations. The allocators are wrapped in an abstraction layer named *Memory Pool*. These are C++ classes that can call a different allocator for memory allocation. This allows users to use the appropriate allocator in accordance with their requirements. The allocators included in Apache Arrow are *mimalloc, jemalloc, and libc malloc*. Having such an abstraction layer allows new allocators to be added to the Arrow library without making any changes to the user APIs.

Arrow table initialization

In order to create an Arrow table, first all the columns have to be prepared as arrays or chucked Arrays. There are three ways to create Arrow Arrays in the C++ implementation. The first method involves using *Arrow Builder* classes to dynamically add single or multiple data, using *Append()* or *AppendValues()* APIs, to a mutable buffer. Once all the desired values are appended, by calling *Finish()* the Arrow array is created and data are marked as immutable. The second way is to manually create an Arrow buffer and set the values directly before wrapping it in an Arrow array. While the second method is faster, it is not recommended as it is unsafe and could lead to corrupt arrays[10].

Once all the Arrow arrays, analogous to columns, are ready, the Arrow table can be assembled with the help of a *Schema*, which describes the column data types and names. The code below shows how an Arrow table with three columns is initiated using the methods mentioned above.

```
// 1) Create builder which allocates using a memory pool
// 1.a) Append all values at the same time
// 1.b) Append values incrementally
arrow::UInt64Builder uint64builder(memory_pool);
uint64_t* data_local_days = (uint64_t *)malloc(SIZE*8);
for (uint64_t i = 0; i < SIZE; i++) {
    data_local_days[i] = i; // 1.b) uint64builder.Append(i);
}
ARROW_RETURN_NOT_OK(uint64builder.AppendValues(data_local_days, SIZE)); // 1.b)
    not needed with .Append()
ARROW_ASSIGN_OR_RAISE(std::shared_ptr<arrow::Array> days_array, uint64builder.
    Finish());

// 2) Allocate a buffer and write directly into it
ARROW_ASSIGN_OR_RAISE(std::shared_ptr<arrow::Buffer> months_buffer, AllocateBuffer
    (SIZE*1, memory_pool);
uint8_t* months_buffer_data = (uint8_t*)months_buffer->mutable_data();
for (uint64_t i = 0; i < SIZE; i++) {
    months_buffer_data[i] = i;
}
auto months_ad = arrow::ArrayData::Make(arrow::uint8(), SIZE, {nullptr,
    months_buffer});
std::shared_ptr<arrow::Array> remote_months = MakeArray(months_array);

// Creating column schema
std::vector<std::shared_ptr<arrow::Field>> schema_vector = {
    arrow::field("Day", arrow::uint64()), arrow::field("Months", arrow::uint8())};
auto schema = std::make_shared<arrow::Schema>(schema_vector);

// Create Table with two columns
std::shared_ptr<arrow::Table> table =
    arrow::Table::Make(schema, {days_array, months_array});
```

Physical layout

The last aspect of the Apache Arrow C++ implementation that is included is the physical layout of the column data, particularly the data layout of variable size *List Layout* and nested types as they influence the design in later stages. Complete explanation of the ”Arrow Columnar Format” and data layout of other data types, that are not covered here, can be seen in the Apache Arrow’s *specifications*.

Apache Arrow allows for the flexible creation of variable size elements within a column. This is done by using the list layout. The list layout contains two buffers, a validity bitmap and an offset buffer, along with a child array. The offset buffer is of particular interest as it is a deciding factor in the implementation proposed in this paper. The offset buffer is always one element larger than the size of the values buffer as it gives the starting and finishing index of the elements. The offsets buffer is the metadata used to access a specific index from a single contiguous values array. To further clarify the memory layout, the physical memory layout of `List<Int8> = [[12, -1, 25], null, [0, -127, 127, 50], [ ]]`, found in Arrow's documentation, can be seen below.

```
* Length: 4, Null count: 1
* Validity bitmap buffer:

  | Byte 0 (validity bitmap) | Bytes 1-63            |
  |--------------------------|-----------------------|
  | 00001101                 | 0 (padding)           |

* Offsets buffer (int32)

  | Bytes 0-3 | Bytes 4-7 | Bytes 8-11 | Bytes 12-15 | Bytes 16-19 | Bytes 20-63 |
  |-----------|-----------|------- ----|-------------|-------------|-------------|
  | 0         | 3         | 3          | 7           | 7           |   padding   |

* Values array (Int8Array):
  * Length: 7,  Null count: 0
  * Validity bitmap buffer: Not required
  * Values buffer (int8)

    | Bytes 0-6                    | Bytes 7-63            |
    |------------------------------|-----------------------|
    | 12, -7, 25, 0, -127, 127, 50 | unspecified (padding) |
```

Apache Arrow's list layout can be used for any type of data, including nested types[1], which list layout it self is a type of. In one of the designs, to be introduced in Chapter 5, nested list layout is required as well. An example of of nested list layout is included as well to show how nesting the data impacts data layout in memory. The representation of `List<List<Int8>> = [[[1, 2], [3, 4]], [[5, 6, 7], null, [8]], [[9, 10]]]`, found in the Arrow specifications as well, can be seen below.

```
* Length 3
* Nulls count: 0
* Validity bitmap buffer: Not required
* Offsets buffer (int32)

  | Bytes 0-3  | Bytes 4-7  | Bytes 8-11 | Bytes 12-15 | Bytes 16-63           |
  |------------|------------|------------|-------------|-----------------------|
  | 0          |  2         |  5         |  6          | unspecified (padding) |

* Values array (`List<Int8>`)
  * Length: 6, Null count: 1
  * Validity bitmap buffer:

    | Byte 0 (validity bitmap) | Bytes 1-63  |
    |--------------------------|-------------|
    | 00110111                 | 0 (padding) |

  * Offsets buffer (int32)

    | Bytes 0-27           | Bytes 28-63           |
    |----------------------|-----------------------|
    | 0, 2, 4, 7, 7, 8, 10 | unspecified (padding) |
```

[1]Apache Arrow has two data types, **primitive or nested**.

```
23
24   * Values array (Int8):
25     * Length: 10, Null count: 0
26     * Validity bitmap buffer: Not required
27
28       | Bytes 0-9                     | Bytes 10-63           |
29       |-------------------------------|-----------------------|
30       | 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 | unspecified (padding) |
```

Looking at the data of the nested list layout, the data is still in one contiguous piece of memory. The only difference between the nested version and normal list layout is the introduction of a second ”offsets buffer”. The first offsets buffer (”outer offsets buffer”) indicates how many lists are in the outer list and points to the second offsets buffer (”inner offsets buffer”). The inner offsets buffer then points to where the data of the inner list is located on the contiguous data buffer.

2.3. Arrow integrated with ThymesisFlow

In this section, the main limitation of ThymesisFlow, namely cache coherency, is explained. Furthermore, important design details of the previous work[10], including nodes synchronization method and Apache Arrow modifications made, to safely use the disaggregated memory as shared memory, are presented.

2.3.1. ThymesisFlow coherency

ThymesisFlow is designed as a disaggregated memory system where the borrowed memory is not meant to be used by the lender (memory) node. As such, it does not have any coherence guaranties across the nodes. This is mentioned ThymesisFlow’s documentation[7]:

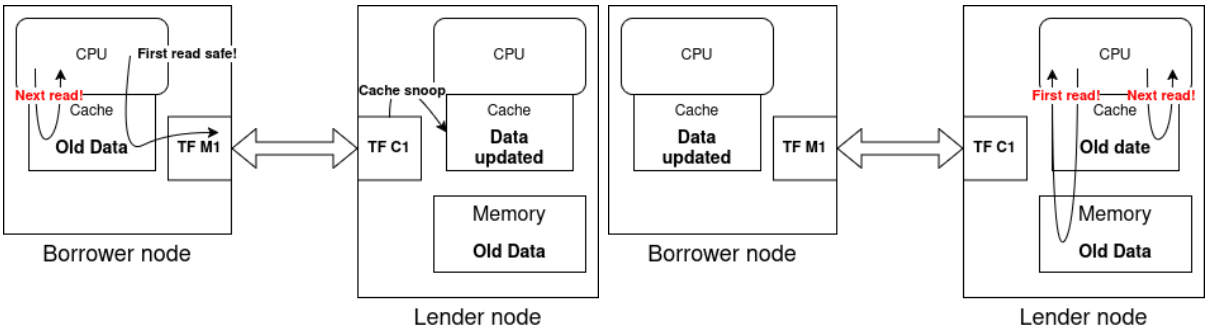
> This design bridges directly processor cache line traffic, without any further intermediate caching support, but not coherence traffic. Therefore, it enables scale-up of memory resources from the perspective of the compute node, but cannot be used to further expand its SMP domain with remote CPUs.

The problems with cache coherency arise when the data is modified by either nodes. The FPGA in the lender (memory) node is configured as a C1 device, which means the attached device is a processor with no-intend-to-cache[18]. In this mode, if the data is located in the CPU’s cache, OpenCAPI enables the FPGA to coherently read the data from the CPU’s cache line and can invalidate the cache line when it writes to memory. The FPGA on the borrower (compute) node is configure as M1 device. This is configures the FPGA as a memory controller and is only capable of forwarding load/store requests to specific memory address spaces that are assigned to it[18]. There are two main scenarios where data inconsistency happens in the nodes:

- The remote (memory) node updates the data in the shared memory region (its own memory), while the borrower (compute) node has a copy in its cache line. In this case, FPGA on the lender side is aware of the changes made in the CPU’s cache, but does not send an invalidate signal to the FPGA found in the borrower (compute) node. So, the borrower still uses the outdated data in its CPU’s cache. However, if the borrower is reading the data for the first time, the cache line is snooped by the C1 configured FGPA on the lender node and the updated data in the cache is retrieved instead of the old data found in main memory. So, first reads from the borrower are safe. The two described scenarios can be seen in Figure 2.5a
- The borrower (compute) node updates the data in the shared memory region and does not immediately push changes to memory. This allows different versions of the data to be present in CPU caches. This scenario does not happen with a write-through cache, where modifications in the cache line are immediately pushed to the main memory. In this scenario, first reads of the borrower node are not safe, since only local memory is read, and the FPGA (configured in C1 mode) is not aware of changes made by the borrower node. The two cache coherency problems from the borrower node can be seen in Figure 2.5b

The two cache data inconsistencies can be seen in Figure 2.5. If the two nodes are only reading the

data and do not have modified data in their cache lines, there is no data inconsistency among the nodes.



(a) Lender node has updated the data with while borrower node has a copy in its cache. Next read in borrower node uses the outdated data.

(b) Both borrower and lender node have copies of data in their caches. Borrower node updated the data and lender node reads the data before updated data are propagated back to main memory.

Figure 2.5: Cache coherency non-guarantees in ThymesisFlow disaggregated memory prototype

2.3.2. Guaranteeing cache coherency

As previously mentioned, the main problem arises if the data is modified. So, in order to ensure cache coherency among the nodes, the shared memory region should not be (frequently) updated. Having constant modifications to the shared memory region requires constant software organized cache line flush and invalidate instructions which are very expensive operations[10]. This is where Apache Arrow, with its immutable data structures, becomes important.

By exclusively using Apache Arrow on the shared memory region, there is zero chance of data inconsistency. However, during the initialization of the Arrow data structures the nodes have to be synchronized and the node modifying the shared region must have exclusive access to it. Previous work[10] has meticulously investigated how the data should be initialized and which node is responsible for managing the shared memory region, with design choices anticipating the design to be used in a disaggregated memory system containing multiple nodes, each offering a part of its memory as a shared memory region.

The first architectural choice that was made, was making the node hosting the disaggregated memory locally the one responsible for allocations in its shared memory region. This means that allocating data in the disaggregated memory region from the borrower (compute) node is performed by sending messages over the network to the lender (memory) node. This messaging was done with gRPC communication protocol, whose exact protocol buffers can be seen in Appendix B.2. Second, in order to ensure no active data in the cache line is corrupted, a gRPC call is sent from the lender (memory) node to the borrower (compute) node (or nodes in future setups with more than two nodes), instructing them to flush the cache lines that coincide with the allocated memory addresses, in a safe manner[10]. This is performed by the available processor instructions found in the Power Instruction Set Architecture (ISA), mainly `dcbi` (*Data Cache Block Invalidate*).

The named changes were sufficient to negate the shortcomings of ThymesisFlow's inter-node coherency limitations. By using an immutable data format and using gRPC to coordinate memory allocations, the disaggregated memory can now be used as a shared memory region.

2.3.3. Using remote memory region with Remote Memory Pool

The Arrow provided allocators were not suitable for managing the shared memory region, which is visible as a shared file system in the operating system. Therefore, a simple allocator by Embedded Aristry LLC[19] was used for the management of shared memory mapped file. The main functions, namely *malloc* and *free* were wrapped in the *Memory Pool* class as an abstraction layer. This allows existing Arrow API calls to work without the need for further modifications. As a result, initializing Arrow tables follows the same steps outlined in Section 2.2. The only modification that is required is providing the new memory pool, named `Thymesis Malloc Memory Pool`, to the builders.

However, when using it in the implementation with nested list types, it did not work and resulted in constant crashes. The main suspected cause is the fact that for initializing the nested types, values are incrementally appended in the arrays. This requires constant re-allocation. So, the allocator requires further work to be used with the same reliability and ease of Arrow provided allocators. In order to rapidly make an implementation, a more manual approach to remote memory allocation is used. It mainly involves copying the column data as an Arrow buffer to remote memory while not instantiating a new table. So, the metadata of the existing local Arrow table is used to read an Arrow buffer in remote disaggregated memory. The issue with this approach is that each Table has to be created in the local node first. This leads to data duplication and prevents creation of graphs that exceed the memory capacity of a single node. Once the allocator related issues are resolved, initializing Arrow tables in any disaggregated memory system with memory mapped files will be as simple as doing it in local memory without the aforementioned issues.

The steps in initializing the Arrow tables in the disaggregated memory are as follows:

1. Allocate a memory region of a given size. If the disaggregated memory is local to the node, the new allocator in `Thymesis Malloc Memory Pool` is used. If allocation is being initiated by a borrower node, a gRPC malloc call is made to the node that owns the disaggregated memory, where the custom allocator is used, returning the address of allocated region to the caller node.
2. A call is sent to all nodes, instructing them to flush all cache lines that correspond with the newly allocated region
3. Once all cache lines are flushed, new data are written in the allocated region
4. Arrow table creation is finalized and data are marked as immutable.

2.3.4. Synchronizing nodes

In order to safely use the shared memory region, the *Orchestrator* class is used to keep both nodes tightly in sync. This is done by using a client and server pair in each node. gRPC calls are used for both memory allocation and sharing of the program state. However, only the borrower (compute) node is allowed to send gRPC calls for memory allocation, as the borrower node does not have a local disaggregated memory region.

The synchronization between the two nodes is rudimentary at this stage. Each node blocks code execution and waits for the other node to reach the same state as itself. So, both the applications run on the nodes must be specifically programmed to work in intervals when using the shared memory region.

2.3.5. Disaggregated Arrow table

In addition to the ability to initialize Arrow tables in a remote node, some gRPC calls are implemented that allows only the metadata of the Record Batches, with direct pointers to the data located in disaggregated memory, to be sent to the other nodes. This way multiple Record Batches, each located in a different node, can be concatenated to form a large disaggregated table, referred to as *Huge Table* in previous work[10], spanning multiple nodes. An example of a huge table can be seen in Figure 2.6. The transformation of the Record Batches into a table is zero copy, with only new meta data created. Apache Arrow's columnar operation can then be run on the huge table just like any other local table.

2.4. Genome assembly: Flye, a de novo assembler

Genomics is a field where massive amounts of data need to be present in memory for sequencing. One of the modern genetic sequencing pipelines, namely Flye de novo assembler, was chosen as an initial application that could leverage memory disaggregation. Flye is a modern genetics sequence assembler specifically used for long read sequencing. It is able to work with both small and large datasets, reaching 100s of gigabytes.

The assembly of genome sequences in a large dataset requires a lot of memory and an attempt was made to improve the memory usage of the Flye sequencer[15]. Some effort is also put into improving the performance of the pipeline itself by making better use of the available resources. However, we could not find attempts at running long reads on a disaggregated memory system. Disaggregated

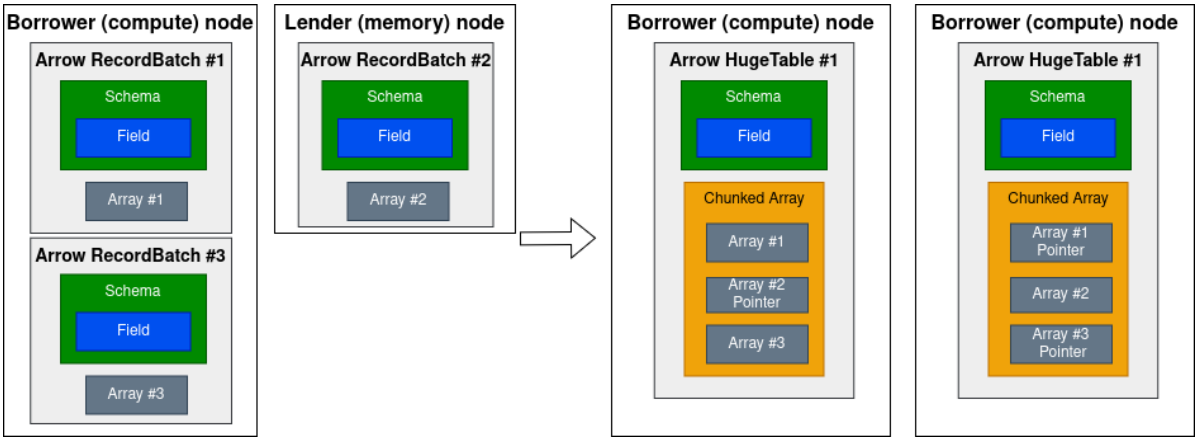


Figure 2.6: Zero copy transformation of multiple Record Batches into a single huge table

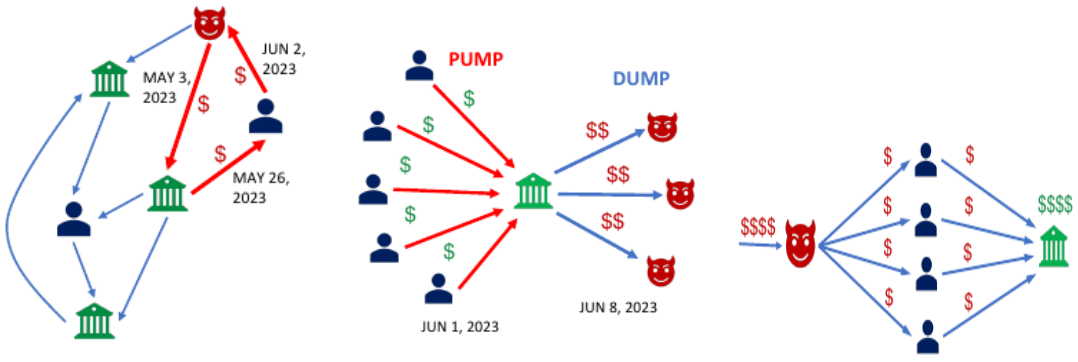
memory systems provide a much larger memory capacity, which allows for both assembling of larger datasets and keeping intermediate results in memory rather than writing them to disk.

Modifying the Flye assembler, however, is very challenging. It is a multi-staged assembly pipeline with each stage performing a unique task. Selecting the stage that could work well in a disaggregated memory, where the impact of memory latency is higher, requires deep knowledge of the algorithms used in each stage. Moreover, the Flye assembler uses third-party software for some of its calculations. Meaning that modifying the data structures in Flye, to better fit the limitations of disaggregated memory, could disrupt the data interfaces that exist, requiring changes to its underlying libraries as well.

The Flye assembler can benefit from the increased memory capacity of disaggregated memory systems and its lower memory latency compared to disk latency. However, the challenges listed above makes its adaption require significant effort. As this cannot be done with the time constraints of the thesis, this application was not selected to be implemented on the disaggregated memory system.

2.5. Financial crime detection

In financial crime detection, the goal is to find transactions and bank accounts that are connected with illegal or fraudulent activities. Financial transaction records are usually stored in tabular format with each row representing a single transaction, with columns holding basic transaction features such as timestamp, source and target account, transferred amount, currency, and payment type[20]. This tabular representation of transactions, however, is not very insightful. By presenting the transactions in a graph, with accounts and transactions as vertices and edges, respectively, more thorough and insightful analysis can be performed[16]. An example of such a graph, with few examples of financial crimes, can be seen in Figure 2.7



(a) Circular money laundering **(b)** Pump and dump **(c)** Smurfing

Figure 2.7: Crime patterns in financial transaction graphs[16]

These subgraph patterns can serve as indicators of financial crime, but are not observed in the tabular data format. A simple cycle, shown in Figure 2.7a, shows the transfer of money through multiple bank accounts to itself. This cyclic transfer of money can be an indicator of money laundering, tax avoidance [16, 21], credit card fraud [22], or stock manipulation by circular trading[16]. The second pattern shown in Figure 2.7b, named a *gather-scatter* pattern can indicate a *pump and dump* tactic to manipulate stock prices[16]. Finally the last pattern seen in Figure 2.7c, can represent a money laundering tactic, named ”smurfing”, where several intermediary bank accounts are used to introduce illicit fund into legal banking system in small sums. Discovering these suspicious patterns may enable tracing and stopping criminal activities[16].

Machine learning models can help in the detection of fraudulent transactions. Given that financial transaction are mostly in tabular format, gradient-boosting-based models provide the fastest and most accurate results. However, these models cannot discover these graph patterns and suffer from low detection rates. To improve the detection rate of the machine learning models, a Graph Feature Preprocessor (GFP), was created to add the graph-based features to the financial transaction information. As it keeps the tabular format, the same machine learning models can be re-used. Moreover, it is implemented to work in real-time fashion by augmenting a batch of transactions as they arrive while continuously running the machine learning models, as a pipeline. The overview of the graph ML pipeline and the architecture of the GFP can be seen in Figure 2.8. The GFP library is publicly available on PyPI as part of Snap ML[16]. While the GFP implementation is closed source, Snap ML is free to use with publicly available examples that demonstrate how a transaction list can be enriched with graph-based features.

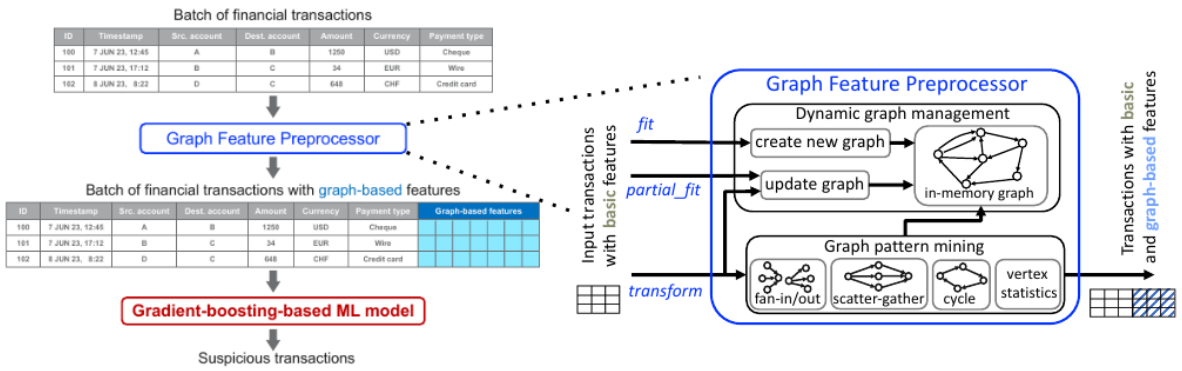


Figure 2.8: Overview of graph ML pipeline with GFP on the left, and internal workings and APIs of the GFP on the right[16]

Results in the GFP paper showed the feature enriched tabular datasets outperformed the original datasets when using the same gradient-boosting-based models (i.e., LightGBM[23] and XGBoost[24]). However, one limitation of the GFP implementation, given it was designed for real-time feature enrichment, is that it holds only a relatively small number of transactions, no older than 24 hours, in the graph. Being able to scan much larger window sizes may allow for even higher detection rates as more information is added as graph features.

A graph representing real bank transactions is much larger than the synthetic data that was used in the evaluation of the GFP. A graph consisting of tens of thousands of transactions per second can quickly become very large, requiring the usage of storage as it cannot fit in memory. Graph Algorithms with unpredictable and random reads are latency bound. One solution is to use a distributed in-memory graph to keep the extremely large graph over multiple nodes. However, as the reads are random, it is very difficult to effectively cache sections of the graph locally. Therefore, the amount of network traffic can be quite high with high latency due to the network software stack.

This is where disaggregated memory can be very beneficial. Splitting a large graph in multiple disaggregated memory systems, allows graph algorithms to perform relatively better compared to network based distributed graphs, as the latency of transferring data is much lower. This is due to the fact that in disaggregated memory systems the data does not have to travel through the network software stack and travels via direct connections that circumvent the processor. Disaggregated memory systems,

having a lower latency than network based distributed memories, are then quite suitable platforms for running such graph algorithms. Furthermore, the graph algorithm implementations being closed source, provides the opportunity to implement a custom version and modify it if required. As a result, this thesis focuses on implementing a distributed graph and graph-pattern detection algorithms on a disaggregated memory system that uses ThymesisFlow as the inter-node link.

2.6. Distributed graph framework: GraphX

Distributed data frameworks, such as Apache Spark, allow a large workload to be split among multiple nodes with communication and data transfers among the nodes done through network links. GraphX is a distributed graph processing framework that is built on top of Apache Spark. GraphX extends Spark Resilient Distributed Datasets (RDDs), the basic data structure within Apache spark for distributing data among multiple nodes, to introduce a graph abstraction. The graph is a directed multigraph and can attach user defined objects to each vertex and edge as properties.

In addition to providing a graph abstraction, it has multiple operations for extracting some, but not all, of the graph features that are of interest in financial crime detection. GraphX also includes a few basic graph algorithms that are not implemented in GFP. The graph algorithms implemented in GraphX include the following:

- *PageRank*: which measures the importance of each vertex in a graph
- *Connected components*: is an algorithm to label each connected components of the graph with ID of its lowest-numbered vertex
- *Triangle counting*: determines the number of triangles, a vertex with two adjacent vertices with an edge between, that pass through each vertex

However, it lacks algorithms for detecting multi-hop subgraph patterns, namely scatter-gather, hop-constrained simple cycles, and temporal cycles. Implementing algorithms for the detection of the previously named patterns, as in many graph analytics tasks, includes aggregating information about the neighborhood of a vertex, i.e., scanning its neighborhood. This is implemented in GraphX as the `aggregateMessages` operator where a user defined function is applied to all the edges in the graph and the results are aggregated at their destination vertex.

Although subgraph pattern detection can be implemented using GraphX, it is not a suitable framework to be used in memory disaggregated systems. The reason is that GraphX, since it uses Apache Spark, abstracts away the splitting of the data and the execution of operation among multiple nodes. So mapping data to explicit memory regions in a disaggregated memory system is not inherently possible. Therefore, in this thesis, graph algorithms are not implemented using GraphX.

3

Graph algorithms

The graph feature pre-processor (GFP) is capable of mining different graph-based features. The features fall into two types, namely graph-pattern-based and vertex-statistics-based ones. The vertex-statistics-based features are computed for all the vertices that appear in the input batch. For each vertex, a user can select any column to compute statistical properties for. For example, by selecting ”Amount” column, features such as average and total amounts of money received or sent can be calculated. The vertex-statistics-based supported features include: sum, mean, minimum, maximum, median, variance, skew, and kurtosis. How these are implemented and their time complexity are beyond the scope of this theis and can be found in GFP paper[16].

The graph-pattern-based features are computed by extracting graph patterns from the in-memory graph if the edge (i.e., the bank transaction) is part of the pattern. The list of supported graph pattern detections can be seen in Figure 3.1 and are listed below:

- Fan-in: number of unique neighboring vertices with an edge to the vertex.
- Fan-out: number of unique neighboring vertices with an edge originating from the vertex
- gather-scatter (*Pump and dump*): Combines a fan-in and fan-out pattern of the same vertex
- scatter-gather (*Smurfing*): a fan-out pattern of a vertex combined with fan-in pattern of another vertex
- simple cycle: a path from a vertex to itself, without any repeated vertices along the path
- temporal cycle: a simple cycle with edges along the path ordered in time

It can be seen that the GFP in Snap ML can produce many graph features and implementing all of the graph features listed above for a new graph is not feasible and is of little value. A selection must be made from the aforementioned graph features that are not trivial. Additionally, they should have varying levels of compute and memory operations to provide better insight into the performance of disaggregated memory systems. Among the candidate graph patterns, fan-in, fan-out, and gather-scatter are trivial and can be implemented by a simple scan of the neighborhood with little computation required.

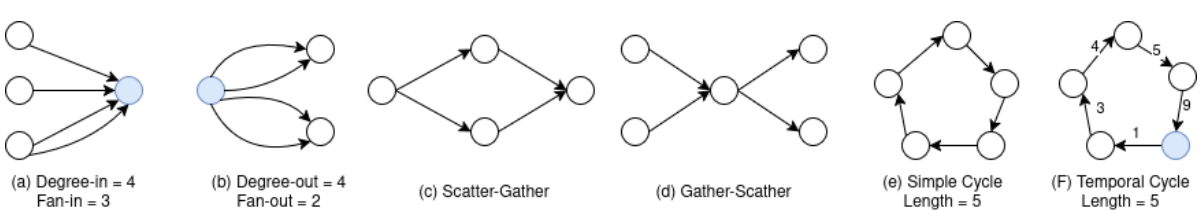


Figure 3.1: Graph patterns extracted by the graph feature preprocessor found in Snap ML library

The remaining three graph patterns, namely scatter-gather, temporal cycle, and simple cycle enumeration, are suitable graph algorithms to be implemented. Firstly, these patterns are currently not widely available in (distributed) graph frameworks. Secondly, they can highlight the weaknesses of a disaggregated memory system as they are memory intensive graph algorithms and can be impacted by the lower bandwidth and higher latency of disaggregated memory. The scatter-gather pattern detection algorithm consists of many (mostly random) memory reads and low arithmetic intensity operations for detecting common neighbors. This algorithm is chosen to be implemented since it exhibits different memory access and computation behavior than cycle detection algorithms. Due to time constraints, only one of the cycle detection algorithms could be implemented. As scatter-gather pattern detection consists of many computations, it was decided to choose the cycle detection algorithm that is more memory bound and requires fewer computations. So, simple cycle detection is chosen as the second algorithm.

This selection of graph algorithms allows the impact of disaggregated memory in graph processing to be better studied as each algorithm is impacted differently by the higher latency and the lower bandwidth of disaggregated memory.

3.1. Scatter-Gather pattern detection

The first algorithm selected, namely scatter-gather, involves a money laundering tactic (called smurfing) in which several intermediary accounts are used to integrate small sums of illicit funds into the legal banking system. The scatter-gather pattern detection consists mostly of random reads followed by a set intersection. Figure 3.2 provides the pseudo code for detecting scatter-gather patterns in a streaming manner. Additionally, it outlines how the algorithm can be run concurrently by marking independent parallel regions.

Algorithm 1: ScatterGatherStream $(\mathcal{G}(\mathcal{V}, \mathcal{E}), batch, \delta_p)$

Input: $\mathcal{G}$ - the input graph with vertices $\mathcal{V}$ and edges $\mathcal{E}$
batch - a batch of edges; δ_p - the time window

1 **parallel foreach** $(\mathrm{u} \rightarrow \mathrm{v}, t_{uv}) : batch$ **do**
2 $\quad \mathrm{TW} = [t_{uv} - \delta_p : t_{uv}]$; ▷ Time window of size δ_p
$\quad$ // The first phase
3 $\quad N_u^+ = \{ \forall x \mid (u \rightarrow x,\ t_s) \in \mathcal{E} \ \wedge\ t_s \in \mathrm{TW} \}$;
4 $\quad N_v^+ = \{ \forall x \mid (v \rightarrow x,\ t_s) \in \mathcal{E} \ \wedge\ t_s \in \mathrm{TW} \}$;
5 $\quad$ **parallel foreach** $\mathrm{w} : N_v^+$ **do**
6 $\quad\quad N_w^- = \{ \forall x \mid (x \rightarrow w,\ t_s) \in \mathcal{E} \ \wedge\ t_s \in \mathrm{TW} \}$;
7 $\quad\quad I = N_u^+ \cap N_w^-$;
8 $\quad\quad$ **if** $|I| \geq 2$ **then** report scatter-gather pattern $\{u, I, w\}$;
$\quad$ // The second phase
9 $\quad N_u^- = \{ \forall x \mid (x \rightarrow u,\ t_s) \in \mathcal{E} \ \wedge\ t_s \in \mathrm{TW} \}$;
10 $\quad N_v^- = \{ \forall x \mid (x \rightarrow v,\ t_s) \in \mathcal{E} \ \wedge\ t_s \in \mathrm{TW} \}$;
11 $\quad$ **parallel foreach** $\mathrm{w} : N_u^-$ **do**
12 $\quad\quad N_w^+ = \{ \forall x \mid (w \rightarrow x,\ t_s) \in \mathcal{E} \ \wedge\ t_s \in \mathrm{TW} \}$;
13 $\quad\quad I = N_v^- \cap N_w^+$;
14 $\quad\quad$ **if** $|I| \geq 2$ **then** report scatter-gather pattern $\{w, I, v\}$;

Figure 3.2: The Algorithm for detecting scatter-gather patterns[16]

The algorithm processes a transaction batch, with a single transaction denoted by $(u \rightarrow v, t_{uv})$ with u, the originating vertex, and v, the outgoing vertex, at timestamp=t_{uv}. It uses the timestamp and the user defined search window size (δ_p) to consider a subset of the graph with transactions (edges) that fall within $[t_{uv} - \delta_p : t_{uv}]$ time window. The algorithm then tries to find scatter-gather patterns that the transaction takes part of. It does so in two phases, depending on whether the transaction is in the scattering or the gathering part of the pattern.

The first phase of scatter-gather algorithm is visualized in Figure 3.3. During this phase, first all **outgo-**

ing vertices of v and u, denoted by N_v^+ and N_u^+, are determined. Then for each vertex $w \in N_v^+$, the **incoming** vertices are scanned in the graph. Figure 3.3b, shows the incoming vertices of a single vertex from N_v^+, denoted by N_w^-. Then, if the number of common vertices between N_u^+ and N_w^- reaches the detection threshold of two ($N_u^+ \cap N_w^- \geq 2$), a scatter-gather pattern is reported, with the number of intermediate vertices encoded as a graph feature.

Phase two of the scatter-gather pattern detection algorithm, follows the same steps as phase one with the following differences:

1. **Incoming** neighbors of v and u (N_u^-, N_v^-) are scanned instead of the outgoing ones
2. For each $w \in N_u^-$ (instead of from N_v^+), the **outgoing** neighbors (N_w^+) (instead of incoming ones) are scanned
3. the intersection of N_w^+ and N_v^- is performed

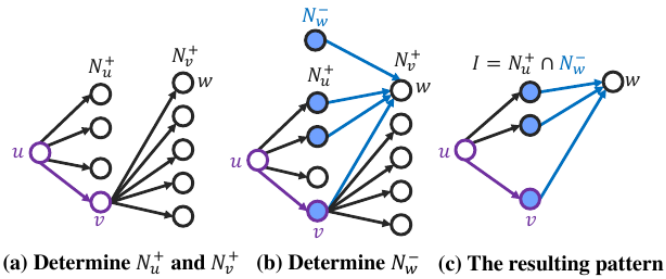


Figure 3.3: The first phase in enumeration of a scatter-gather pattern with the edge $u \rightarrow v$ in the "scatter" portion[16]

Time complexity

The scatter-gather algorithm consists of two main parts: scanning neighbors and intersecting neighboring vertices in pairs. The graph properties that impact these operations are the fan (Δ) and degree (Deg) of the vertices. In the scatter-gather algorithm the incoming/outgoing neighbors of each outgoing/incoming neighbors of the vertex are scanned. Scanning the neighbors and selecting those that fall within the search window depends on both the neighboring vertices (Δ) and the number of parallel edges (Deg). Therefore, the worst case time complexity for scanning the neighbors is $O((\Delta_{max} + Deg_{max})^2)$.

The time complexity of the set intersection depends on the number of set intersection and the size of the sets themselves. The higher the average fan-in and -out of the graph, the more set of neighboring vertices are scanned and need to be intersected. The time complexity of the set intersection depends on the intersection methods used. Below a list of most common set intersection methods can be seen:

- Naive array iteration: This involves iterating through two unsorted arrays and comparing elements in one against all elements in the other. With M and N being the size of the first and second array respectively, the worst case time complexity of this set intersection method is $O(\Delta_{max} \times M.N)$.
- look up merge (hash join intersection): This involves transforming the smaller array into a hash set (O(M)), namely unordered sets in C++, and looking up all elements from the second array in the constructed hash set (O(N)). This leads to a time complexity of $O(\Delta_{max} \times (M + N)))$

3.2. Cycle enumeration

The second algorithm that helps in detection of criminal transactions is cycle enumeration. A simple cycle in financial transactions can be seen as transferring funds from a bank account through one or multiple intermediate accounts back to the same account. There are many cycle enumeration algorithms available with varying levels of complexity and scalability, such as Tiernan, Johnson and Read-Tarjan algorithms[25]. The algorithm chosen is the Tiernan algorithm that detects elementary paths (no duplicate vertices in path) and elementary circuits (paths that lead to the starting vertex).

The Tiernan's algorithm is modified in this thesis to work in detecting simple cycles in a streaming manner. This means that after a batch of transactions are added to the graph, the algorithm only searches for cycles that start with the transaction (edge) rather than detecting all simple cycles in

the updated graph. The algorithm has two parts: **path extension**, and **cycle confirmation** during backtracking. The pseudo code of the modified algorithm, applied to each transaction in the batch, can be seen in Algorithm 2.

Each simple cycle search starts with the two vertices, involved in the transaction, forming the start of the path. Then the path is continuously extended by the outgoing neighbors of the last vertex in the path (i.e., the tail), if they are not part of the current path, until the path can no longer be extended or the maximum cycle length of 10 is reached [1]. This is a recursive and depth-first-search (DFS) algorithm. Each extended path is independent from other paths and can be concurrently processed. Figure 3.4 shows how processing of different paths can be parallelized. While each of the transactions can be done in parallel, it can leave a single thread to process large recursion trees on its own. The fine-grain parallelization that it shows prevents unbalanced trees to be done by a single thread and exploits more of the available parallelism.

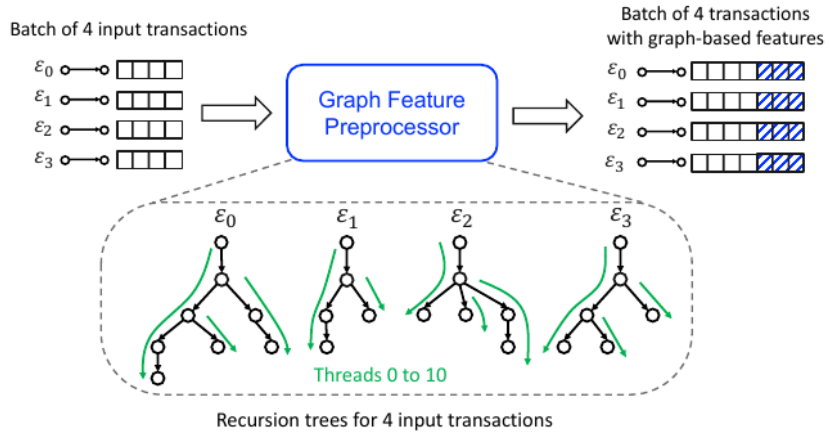


Figure 3.4: Fine-grained parallelism that GFP exploits in Snap ML library by processing independent paths concurrently, increasing the threads used from four to eleven[16].

Once path extension is not possible, the outgoing vertices of the tail are compared with the starting vertex. If the tail connects with the first vertex in the path, the cycle length is stored as an extracted feature. The process of looking for cycles while backtracking continues until all paths that start with the transaction vertices are explored. The pseudo code for detection of simple cycles, that start with transaction $(u \rightarrow v, t_{uv})$, can be seen in Algorithm 2.

[1]The Graph Feature Preprocessor in Snap ML library has a hop constraint of 10 for simple cycles.

Algorithm 2 $SimpleCycle(\mathcal{G}(\mathcal{V},\mathcal{E}),\Pi,(u\rightarrow v,t_{uv}),\delta_p)$

Input: the input graph with vertices $\mathcal{V}$ and edges $\mathcal{E}$
u - starting vertex, v - outgoing vertex, δ_p - the time window
Π - path

1: **if** Π.empty() **then**
2: Π.push_back(u);
3: Π.push_back(v);
4: **end if**

5: TW = [$t_{uv}-\delta_p : t_{uv}$];
6: $tail$ = Π.back();
7: $tail^+$ = $\{\forall x|(tail\rightarrow x,t_s)\in\mathcal{E}\wedge t_s\in TW\}$;
8: **if** Π.size() < 10 **then** ▷ Path Extension
9: **for all** $w : tail^+$ **do**
10: **if** $w\notin\Pi$ **then**
11: Π.push_back(w);
12: $SimpleCycle(\mathcal{G}(\mathcal{V},\mathcal{E}),\Pi,(u\rightarrow v,t_{uv}),\delta_p)$;
13: **end if**
14: **end for**
15: **end if**

16: found = $false$;
17: **for all** $w : tail^+$ **do** ▷ Cycle confirmation
18: **if** $w=\Pi$.front() **then**
19: found = $true$;
20: **break**;
21: **end if**
22: **end for**
23: **if** found **then** report simple cycle length $\{(u\rightarrow v,\Pi.size()\}$

24: Π.pop_back(); ▷ Back-tracking

Time complexity

Path extension, which includes scanning the outgoing neighbors and edge timestamps, depends on both the fan-out and the degree-out of the graph, and scales exponentially depending on the maximum cycle length. This can be seen as the number of edges that need to be scanned for detecting cycles. It has a worst time complexity of $O((\Delta_{max}+Deg_{max})^d)$ with $d=depth$. However, determining the worst time complexity of the complete algorithm cannot be estimated easily as it can vary significantly among batches, depending on how many paths can be explored that do not lead to a cycle. So, the worst case complexity of Tiernan's algorithm depends on the maximum number of simple paths[25] that start with the the transaction.

The looking up of the starting vertex in the outgoing vertices list, during path backtracking, is the main computation that takes place. This operation scales linearly with the fan-out of the graph if the outgoing vertices are stored in arrays. However, if the outgoing vertices are stored as hash sets in the implementation, it has a worst time complexity of O(1) regardless of the degree-out. In the implementation documented in this paper, normal arrays are used for storing outgoing vertices. Therefore, the worst time complexity of the simple cycle detection, including both circuit confirmation and path extension, is $O(\Delta_{max}\times(\Delta_{max}+Deg_{max})^d\times s_{uv})$, with s_{uv} the number of simple paths that start with $u\rightarrow v$.

3.3. Graph datasets

Although many domains such as image, speech, and natural language processing have abundant labeled data, there is no publicly available dataset for bank transactions, due to privacy and safety concerns. Therefore, effort has been put into creating realistic synthetic datasets for financial transac-

tions. In this thesis, Anti Money Laundering (AML) datasets, generated by *AML World Generator* and available online[2], are used to benchmark the algorithms[20].

There are two sets of AML datasets available, AML HI and AML LI, for high and low illicit rates, respectively. In the GFP performance benchmarks[16], both variants were used in tests, but the reported results include the performance of the GFP and ML models as a pipeline. The algorithm specific results obtained from Jovan Blanuša, one of the researchers responsible for the GFP, are run on the high illicit rate datasets. Therefore, in this thesis, only AML HI datasets are used.

The original datasets contain ten columns of data that include transaction times, bank ids, account numbers, transaction method, and the currency and amount transferred. The only information necessary for testing the algorithms include timestamps and account numbers. Therefore, in order to reduce read times when running the benchmarks, it was decided to remove unused columns. Table 3.1 shows the datasets used to evaluate the performance of the algorithms, along with the reduced datasets' size.

Table 3.1: Datasets used in benchmarking scatter-gather and simple cycle detection

Dataset	# nodes	# edges	illicit rate	time span	original size	reduced size
AML HI Small	0.5M	5M	0.102%	10 days	650.42MB	150.1MB
AML HI Medium	2.1M	32M	0.110%	16 days	3.03GB	943.0MB
AML HI Large	2.1M	180M	0.124%	97 days	17.05GB	5.2GB

[2]https://www.kaggle.com/datasets/ealtman2019/ibm-transactions-for-anti-money-laundering-aml/data

4

Methodology

In order to realize a graph capable of efficient real-time processing, two types of graphs were designed that can work in tandem, one replicating the earlier implementation found in GFP and the other utilizing Apache Arrow. The replicated graph will use the same native C++ data structures found in GFP and handles the dynamic portion of the implementation for dynamic transaction management. The new graph type based on Apache Arrow tables will store older transactions and will migrate the graph information to the remote memory. While the remote data are physically located in the remote node, the application sees it as a NUMA node within the same node (i.e., a CPU-less NUMA region in an SMP) and accesses the data by normal load/store instructions.

Additionally, a baseline was necessary to highlight the difference in performance and ease of programmability when using disaggregated memory with ThymesisFlow in comparison to a traditional network based cluster shared memory. As the available distributed graph frameworks, such as GraphX, abstract the distribution of data and do not give control for explicit data placement, a version was implemented that uses network for communication and data transfer. In order to place the main focus on the inter-node connection, the newly implemented graph algorithms were modified to retrieve graph data located in the Apache Arrow tables via gRPC calls instead of CPU load instructions that are issued to ThymesisFlow hardware/software stack.

In this chapter, the steps taken to make use of the Arrow augmented ThymesisFlow prototype for creating and running graph algorithms in a disaggregated memory system are documented. The design choices and considerations for a traditional network based implementation is included as well. Finally, an overview of the user API for the implementation and their functionalities can be found.

4.1. Dynamic graph

As mentioned in the previous chapters, while the GFP implementation found in SnapML is closed source, the data structures used are well documented in the published paper[16]. The overview of the main data structures can be seen in Figure 4.1 . Alongside the same data structures, the same optimizations that were implemented are used in our implementation. Here is the list of C++ native data structures used and their respective use case:

- Transaction log: a double-ended queue for dynamic graph management. This is used to keep track of the lifetime of transactions within the graph.
- Index: A vector of hashmaps representing the graph, with each hashmap an adjacency list of a vertex. A hashmap maps a neighboring vertex to a double-ended queue of transaction times.

  There are two index data structures in the implementation for outgoing and incoming transactions. This allows O(1) access times for scanning both the preceding and the proceeding vertices.

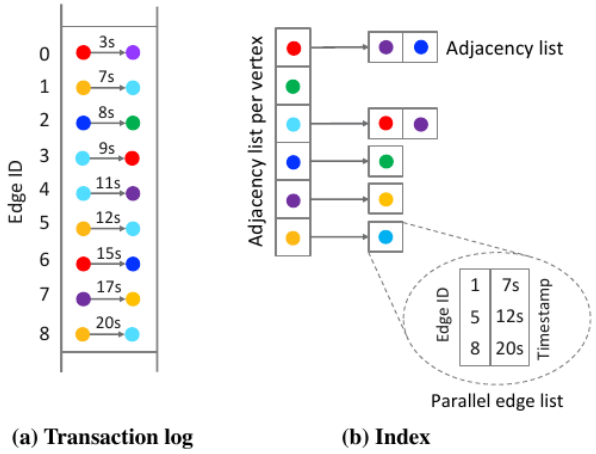


Figure 4.1: GFP in-memory dynamic multi-graph

4.2. Graph as Arrow tables

The original ThymesisFlow setup allows a portion of the memory to be lent to the neighboring nodes. Then the lent memory region cannot be used by the remote system itself without causing cache coherency problems. However, in the previous work, the ThymesisFlow setup was integrated with Apache Arrow and a synchronization method to allow for cache coherent initialization of the data. This upgraded prototype allows the memory region to be shared coherently between nodes. This is done by carefully managing the data and flushing the data present in the cache during initialization. Given that the Apache Arrow tables are immutable, there is no chance of data corruption by multiple readers. So, in order to preserve this property, Arrow tables were used.

As Arrow tables are immutable and cannot be incrementally created, a standalone real-time implementation is challenging and most likely inefficient. Therefore, a real-time implementation that exclusively uses Arrow tables was not pursued. In the final design, the Arrow portion of the graph is solely used for larger search windows in real-time and offline processing. Section 4.4 covers how the Arrow table based graph is integrated with the dynamic graph.

The simplest way to run graph algorithms using Apache Arrow could have been realized by creating an in-memory database containing all the transactions with a query engine to filter neighboring vertices within a time window. While this can be done via a combination of Arrow compute functions, it cannot be done efficiently as the number of queries per second for filtering rows and producing adjacency lists is very high. The number of relevant rows within a table is also quite small as a portion of the complete data set. Additionally, Apache Arrow itself does not have fast SQL like APIs. There are Arrow extension libraries such as Arrow Flight (SQL) to allow inquires of SQL databases, but it carries a lot of overhead when transferring small data[26]. The Acero execution engine was also considered for implementing graph algorithms as it is capable of transforming streaming data. However, Acero can be used mainly to transform incoming bank transactions and store the results. So, it can only be used to prepare the Arrow graph, not to run the graph algorithms themselves. So, it was concluded that the most feasible option was to implement table scans without the help of any Arrow processing power.

Due to the flexibility of Apache Arrow and support for nested types, the first version grouped transactions based on vertices during table initialization. This groups relevant transactions and enables algorithms to use the graph in a more efficient manner than a data base as there is no need for a full table-wide scan. The main downside of this design is that the vertices column will contain duplicate entries and the performance will drop linearly with the average degree (out/in) of the vertices.

The major operation in the graph algorithms is scanning of neighboring vertices and selecting those that have an active edge (transaction) within a user specified time window. So, the transactions can be grouped once again by the outgoing vertices as Apache Arrow allows multiple nesting layers. However, by each grouping, the extra nested layer adds another offset buffer. The performance hit for this extra random read for traversing the data buffers in the Arrow table is negligible as the offset buffers are

found in local memory in the design. The reduction in duplicate data and active memory footprint is going to offset it as well. In this improved design the number of duplicate entries in the table is reduced and now scales with the average fan (out/in), instead of degree (out/in), of a vertex. Tables 4.1 and 4.2 show how the transaction log found in Figure 4.1 is transformed into the first and improved table versions, respectively.

Table 4.1: Arrow table with grouped transactions (version 1)

Vertex	Outgoing vertices	Timestamps
Red	[Purple, Blue]	[3 , 15]
Green	[]	[]
Cyan	[Red, Purple]	[9 , 11]
Blue	[Green]	[8]
Purple	[Yellow]	[17]
Yellow	[Cyan, Cyan, Cyan]	[7, 12, 20]

Table 4.2: Arrow table with prepared adjacency lists (version 2)

Vertex	Outgoing vertices	Timestamps
Red	[Purple, Blue]	[[3], [15]]
Green	[]	[]
Cyan	[Red, Purple]	[[9] , [11]]
Blue	[Green]	[[8]]
Purple	[Yellow]	[17]
Yellow	[Cyan]	[[7, 12, 20]]

The main difference between the two arrow table versions is the removal of duplicate entries in the second column by grouping those during the table initialization at cost of introducing another random read, due to need for an additional offset buffer for indexing data of nested lists. The second version is the version that mostly resembles the C++ native implementation as the edges among the same nodes are aggregated as well, producing a final adjacency list for each (outgoing/incoming) vertex. With the design of the tables completed, other design choices that ensure the implementation performs well, considering the characteristics of the disaggregated memory system, can be considered.

4.2.1. Data placement in memory

In section 2.2, an overview of the physical memory layout of nested data types in Apache Arrow was given. The memory layouts of the proposed Arrow versions are compared here by looking to the data pertaining to a single vertex, namely the outgoing transactions originating from vertex ”n”. The two versions can be seen in Figure 4.2. The first version reduces the need for a table wide scan to filter rows, and stores the parallel edge list for each vertex in a single row. In this implementation, both columns scale with the degree (out/in) of the vertices in the graph.

In the follow-up version, by grouping the data once more, the vertices column scales with fan (out/in) of the vertex rather than degree (out/in) in the first version. This reduces the size of the table and reduces the amount of active memory. In addition, by grouping the transaction by the outgoing vertex, there is no need to merge the duplicate vertices. As a result, the second version scales better than the previous one as the graphs grow larger. As an example, if we were to scan for vertices with transactions between t = 5 and t = 15, the first table requires six reads of the timestamps column followed by another six of the vertices column to create a set of vertices. However, in the second version it requires four reads of timestamps column with two reads from the second column to create the same set of outgoing vertices. So, the second version, while also being more compact (cache friendly) and memory efficient (smaller tables), requires fewer read operations to produce the same results. Due to time constraints, while both were implemented, only the more compact Arrow table is used in this paper.

4.2.2. Arrow API avoidance

It is noteworthy that the Apache Arrow in-memory format organizes table in columnar format. Given the access patterns of the algorithms are neighborhood aggregations (scans), mostly row operations

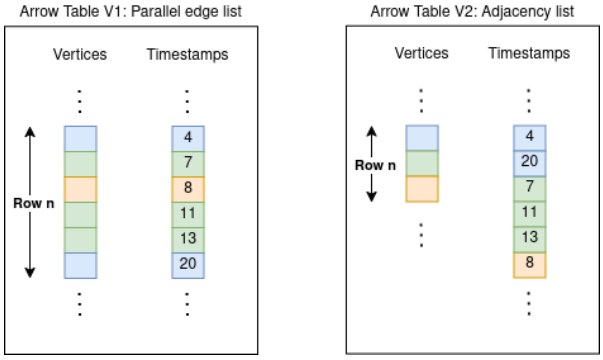


Figure 4.2: The data layout of vertex n with 6 transactions with 3 < t < 21

are performed. This makes efficient integration with Arrow compute functions more challenging. As a result, Apache Arrow was solely used as an immutable in-memory format to use the cache coherency guarantees of previous work, without leveraging Arrow's compute API.

While Apache Arrow is ideally designed to be used for columnar operations, it does have some API to read row data as well. Initially, given Arrow's zero-copy nature, the 'Slice()' methods was used to access row data by creating a new Arrow array without copying the underlying data. This resulted in extremely low performance. This indicated that the cost of continuously initializing arrow objects was far greater than the zero-copy benefit with fine grain reads.

The other API call for just reading the data, namely `Value()` and `value\_offset()`, carried a lot overhead as well. It was decided to avoid any high overhead Arrow API that could hinder performance and use raw and unsafe data reads[1].

Raw Arrow data indexing

Avoiding Arrow APIs with a high performance overhead altogether raises the question on how the data is efficiently accessed. Here the code for indexing the nested and double nested list types for reading the vertices and the timestamps, respectively, is given. Using these Arrow interfaces (, or low impact APIs,) avoid the constant shared pointers that are used in previously mentioned methods.

```
/* Storing raw pointers to Arrow table's metadata: offsets buffers, values buffers
typedef struct{
    const arrow::ListType::offset_type *vertex_offsets; // vertices offsets buffer
    const u_int64_t *vertices;  // vertices values buffer
    const arrow::ListType::offset_type *time_offsets_out;   // Outer list offsets
        buffer
    const arrow::ListType::offset_type *time_offsets_in;    // Inner list offsets
        buffer
    const int64_t *times;   // timestamps values buffer
} ArrowPointers;
*/
// Reading row number "vertex" = Bank ID
auto target_raw_offset = arrow_pt->vertex_offsets;
const u_int64_t* targets = target_base + target_raw_offset[vertex];
int64_t target_size =  target_raw_offset[vertex+1] - target_raw_offset[vertex];

const int64_t *time_base = arrow_pt->times;
auto time_raw_offset_outer = arrow_pt->time_offsets_out;
auto time_raw_offset_inner = arrow_pt->time_offsets_in;
auto time_list_offset = time_raw_offset_outer[vertex];

for(int i = 0; i < target_size; i++){   // reading outgoing vertices of row "
    vertex"
    auto effective_index = i + time_list_offset;
```

[1] Easy to use Arrow APIs have shared pointers in the backend and with frequent reads introduced massive delays in runtime

```
22      auto t_list_size = time_raw_offset_inner[effective_index+1]-
            time_raw_offset_inner[effective_index];
23      auto times = time_base + time_raw_offset_inner[effective_index];
24      for (auto j = 0; j < t_list_size; j++) {
25          // ... Reading timestamps of outgoing vertex = targets[i]
26      }
27  }
```

4.2.3. Parallelization of algorithms

There is a lot of parallelism to be exploited in detecting patterns within a graph. The implementation made use of OpenMP, OpenMP Tasks specifically, to parallelize the algorithms and increase the throughput of the implementation. However, the parameters of the OpenMP Tasks such as grain size and recursion depth were not fully optimized. Additionally, the algorithms were not heavily modified to make the best use of OpenMP's capabilities with more parallelism potential left.

In the scatter-gather detection algorithm, the parallelization levels can be seen in Figure 3.2. A fine-grain parallelism strategy found in Figure 3.4 is also implemented for simple cycle enumeration. However, the OpenMP parameters were not fully optimized to maximize load balancing and minimize the excessive spawning of OpenMP tasks in recursion trees, which leaves unrealized performance gains. The performance may even be hindered as well due to excessive task switching, but this was not investigated.

4.2.4. Ordered transactions

An algorithmic improvement in the implementation was the assumption that in a real-time scenario, the transactions would arrive mostly in the order of their timestamps. To replicate this, all datasets were first ordered. This implies that in the adjacency list, the transactions are ordered by date as well. This allows earlier termination and reduction of compute for finding neighbors within a time window.

4.3. Hybrid implementation

With the design of the dynamic graph and the Arrow table based graph completed, now it was time to create an hybrid implementation that is capable of making use of the disaggregated memory system in both real-time and offline scenarios. This required the designing of a dynamic graph manager that is able to maintain the (small) dynamic graph for real-time processing and an arrow graph, that holds older transactions in remote memory, for longer search windows for both real-time and offline processing.

The hybrid implementation consists of a graph manager that facilitates the distribution of transactions among the two different types of graphs. The graph manager utilizes the dynamic graph with native C++ data structures for recent transactions that arrive in real-time. The arrow graph is used for outdated transactions that need to be ejected from the dynamic graph and stored in the remote node. Given the immutable nature of arrow tables, new tables are periodically created without modifying the existing ones. The Arrow graph consists of an array of arrow tables, each containing transactions within a certain time window. This design allows the algorithms to not slow down, the larger the graph gets, as the graph data are split evenly among multiple arrow tables. The performance then only depends on the search window given the same average fan/degree. As most of the extracted graph features are reported to use a time window smaller than one day, it was inferred that the dynamic graph should hold at most 24 hours worth of transactions. So, the size of the dynamic and Arrow tables was limited to 24 hours as well. Figure 4.3 shows how the hybrid graph is incrementally updated. The elements with dashed lines are empty and will be created as more transactions are inserted into the hybrid graph.

Initially, both the dynamic and Arrow portion of the graph are empty at t = 0. As new transactions arrive, they are added to the dynamic graph as seen at t = 16 hours in Figure 4.3. When a new transaction batch is received that crosses a 24 hour mark, an Arrow table is created as part of the Arrow graph. Here, the question arises whether to empty the native graph an Arrow table is created to avoid data duplicates in memory. As we want to keep the hybrid implementation responsive for real-time processing, specially when using smaller search windows, the last 24 hours of transactions are kept in the dynamic graph regardless of data duplication. Thus, at t = 24, all transactions from the

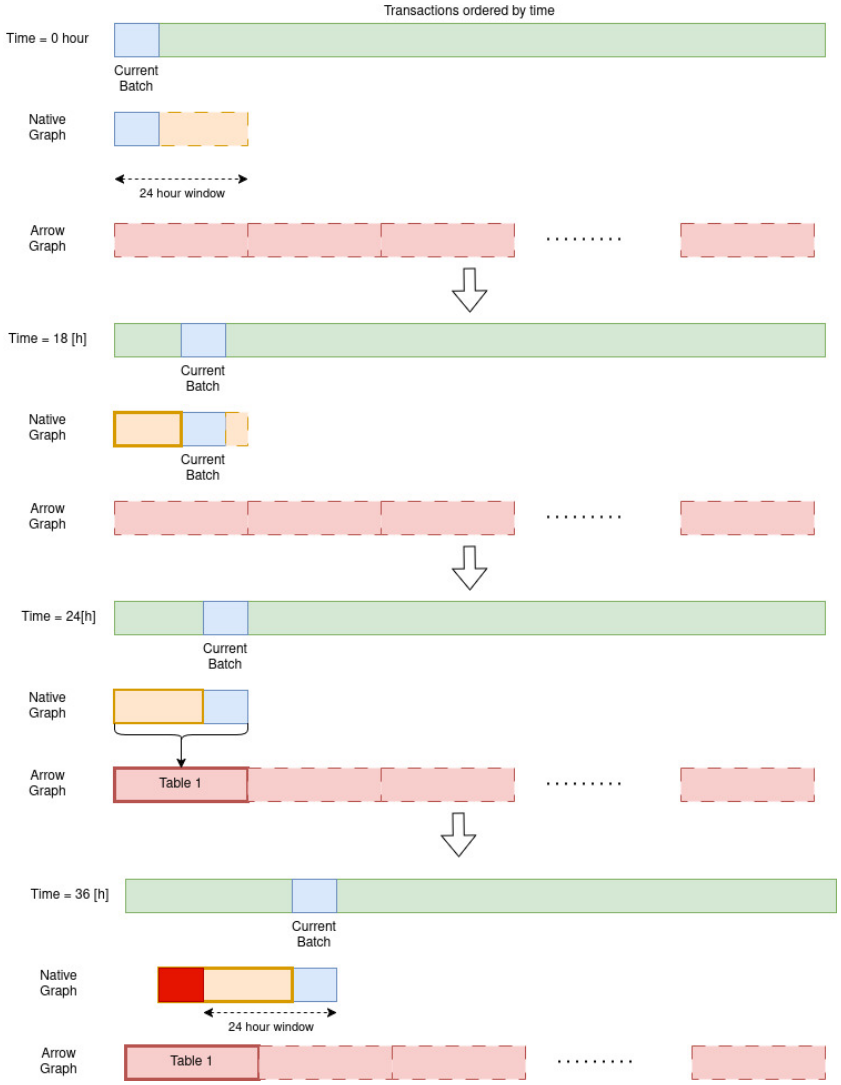


Figure 4.3: The structure of the hybrid graph consisting of a native graph and multiple arrow tables. The dashed figures signify non-saturated and future parts of the graph.

dynamic graph are copied into an Arrow table while no transaction is ejected. When a new transaction batch that exceeds the 24 hour mark arrives, the transaction log is used to eject outdated transactions from the dynamic graph. This can be seen in Figure 4.3 at t = 36 hours where the transactions marked in red are ejected.

In the hybrid graph, the data are supposed to be disaggregated with table data placed in remote memory. The way that this is implemented is that the Arrow tables' data are placed in the remote node with the local node containing only pointers to the data. An overview of this architecture can be seen in Figure 4.4. The local node has the information to access the data directly. First, it has the data indicating the minimum and maximum transaction times found in a certain arrow table to index specific Arrow tables. Secondly, it contains all the arrow specific metadata, specifically offset buffers, for correctly accessing the remote row data. Finally, it has raw pointers to the remote data table and can read them directly with the help of ThymesisFlow hardware/software stack. In the benchmarks, to show the impact of the increased memory latency of the ThymesisFlow prototype, the hybrid graph is also tested with Arrow table data located in local memory.

The hybrid graph is capable of processing transactions in real-time and offline with varying search window size. An example of how the hybrid graph scans the dynamic and Arrow graph can be seen in Figure 4.5. The figure also demonstrates how the graph is scanned in different operation modes by

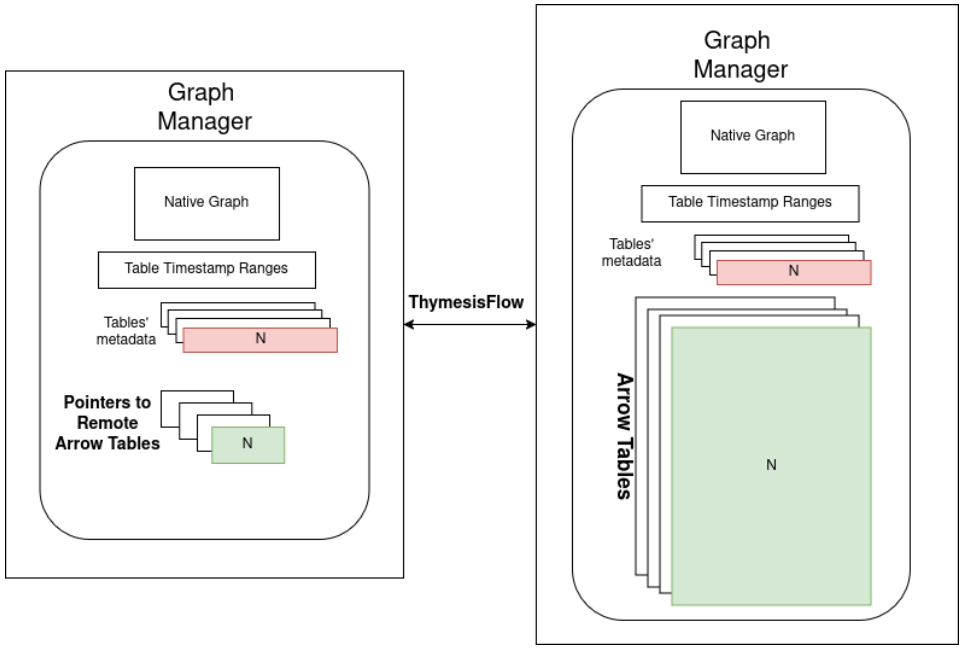
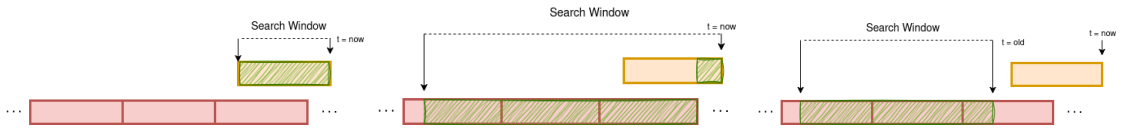


Figure 4.4: Main data components of the graph manager for TF



(a) Real-time Small window **(b)** Real-time large window **(c)** Offline

Figure 4.5: Scanning of the hybrid graph for neighboring vertices within a desired time window in native (orange) and arrow part (red).

marking the active regions. When the search window completely falls within the dynamic graph, the Arrow graph is not used. However, if the search window exceeds the dynamic graph or overlaps with the Arrow graph, the overlapping region is only scanned from the Arrow graph.

4.4. Traditional Baseline

In order to show the advantage of disaggregated memory for processing distributed graphs, one had to look for a suitable baseline for comparison. The most promising framework that was considered is GraphX. However, due to the reasons mentioned in section 2.6 it is not used as a baseline. The baseline modifies the graph algorithms by changing only the way the data is retrieved from the remote (memory) node, while keeping the underlying hybrid graph data structures the same. The architecture of the baseline can be seen in Figure 4.6.

4.4.1. Communication protocol

Arrow Flight is the first option considered for transferring data between nodes. It uses gRPC as its communication protocol and is optimized for parallel and vectorized data transfers. An earlier study[26] compared Arrow flight with other inter-node communication protocols, such Remote Direct Memory Access (RDMA) and TCP protocol, all using the same link. Arrow flight, while performing better than TCP protocol and almost matching RDMA for large data transfers over 2.56GBs in size, showed extremely poor bandwidth for small data transfers up to few kilobytes[26]. Given that vertices have limited number of neighboring vertices, fetching a complete arrow table would retrieve many unused data, leading to large data duplication and excessive memory usage as a consequence. Therefore, Arrow flight cannot be used in our implementation as the data transfers are projected to be very small and fetching complete tables goes against the goal of ensuring memory efficiency.

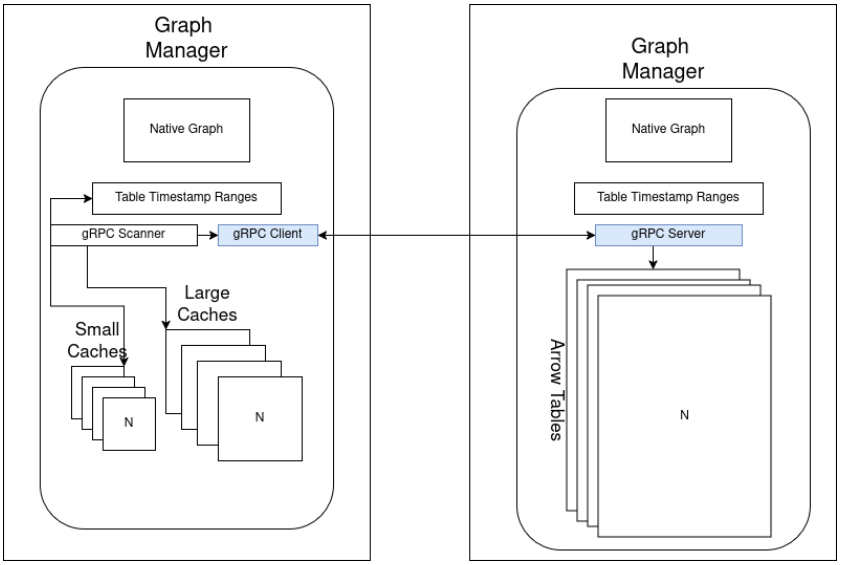


Figure 4.6: Architecture of gRPC based implementation

Therefore, an implementation with network based communication had to be created from scratch. Additionally, a software cache had to be designed for this implementation to provide a sound baseline as a comparison point. It was decided to use gRPC as the communication protocol, due to its reportedly low latency and high throughput. gRPC provides both synchronous and asynchronous APIs to allow the programmer to make better use of concurrency. The asynchronous callback API was selected and used in this implementation as it provides a balanced mix of performance and ease of programmability. If the goal was to produce the most efficient gRPC implementation, the older asynchronous completion-queue API had to be used. The Protocol Buffers that were defined in gRPC communication can be seen in Appendix B.1.

4.4.2. Software cache

The granularity of the cache becomes the next design choice. Coarse- and fine-grain caching includes caching a portion (or an entire) table, and a single row, respectively. The coarse grain caching strategy will likely lead to higher data transfers while providing no substantial benefit to performance. This is mainly due to the sparsity of the graph (low connectivity between vertices). This caching strategy leads to fetching mostly unused data with higher unnecessary memory usage as a result. The fine-grain caching strategy is a more suitable approach where a single row, containing the complete adjacency list of a vertex, is fetched at a time.

The software cache is created to reduce the number of gRPC calls, as gRPC calls were expected to carry high overhead for small data transfers, even without the presence of Arrow Flight's software stack. Using gRPC includes implementing a client/server application that handles the data transfer among nodes. It is possible to make the server do some of the processing to reduce the compute required on the local node and further reduce gRPC calls, but this makes the comparison between the inter-node communication methods less clear. The main goal of the thesis is to outline the advantages of a disaggregated memory within a cluster that allows the memory to be borrowed and fetched without introducing additional load on the remote node's CPU as well. Thus, the server in the implementation does not process the data and only transfers the row data of a given Arrow table to the compute (borrower) node, with each row representing the full adjacency list of a vertex.

The next consideration for the cache design is the structure of the cache line and the ejection strategy. Looking at Figure 4.5, it can be seen that sometimes scanned Arrow tables fall completely within the search window. By inferring the timestamp range within an Arrow table, the vertices column can be scanned directly without reading the timestamps. On the other hand, tables that fall partly within the search window need to scan the timestamps to select a subset of vertices. Therefore, two types of caches were needed to avoid fetching timestamps before they are absolutely needed. The difference

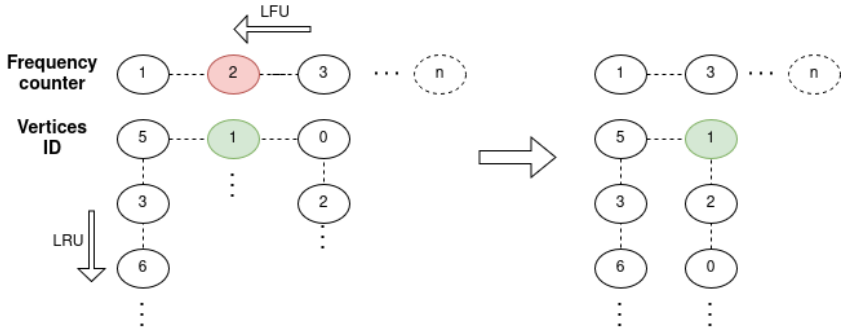


Figure 4.7: State of frequency counter when cache line of vertex id 1 is read.

of caches are:

1. Small cache: stores only vertices as its cache line. This reduces memory usage and gRPC's payload size.
2. Full cache: stores vertices alongside all timestamps as its cache line. This allows exact time windows to be applied in the selection of neighboring vertices.

In order to facilitate read times of O(1), the cache was implemented as a hashmap of cache lines, mapping a vertex to its neighboring vertices or an adjacency list. Scanning for neighboring vertices may lead to empty cache lines as they vertex may not be connected to any other nodes in certain time windows. In order to avoid wasting cache lines, each cache contains a first-in-first-out (FIFO) buffer for queries with empty lists. This maximized the useful information that is cached.

Regarding the cache line ejection strategy, multiple options were considered and implemented. These eviction strategies had no impact on the read complexity and just introduced extra data structures to manage elements in the existing cache line hashmaps. The ejection strategies considered are listed below with last one skipped due to time constraints:

- FIFO: simplest implementation with lowest software overhead. Implemented by a double ended queue
- Random: simple ejection method with minor software overhead. Implemented by a double linked list, but higher performance hit due to non-local data as elements within the queue could be removed.
- Least frequently used (LFU) & least recently used (LRU): The most complex implementation with high software overhead. A 2D double ended queue is used for managing read counts. The LFU cache lines are ejected and in case of a tie in read frequency, the LRU one is ejected.
- Least recently used (LRU): A moderately complex with implementation with moderate software overhead. This can be implemented with a double linked list to efficiently move elements from the middle to the end of the ejection queue.

Figure 4.7 shows how the read frequency of the cache lines is managed and how reading a cache line updates the counter, when using the LFU & LRU cache ejection policy. As every cache line read requires updating of the frequency counter, more measures need to be taken to allow for multi-threading scalability. To ensure fewer write locks, it was decided to use multiple caches with each cache split into multiple sections, each with its own frequency counter. Most of the data sets are quite small and lead to tens of Arrow tables at most, allowing each table to be directly mapped to a specific cache. Secondly, each cache is split into eight sub-caches (clusters or banks) that can hold a subset of vertices. By having such a split design, the chances of cache line read contentions are reduced. Alternatively, to avoid splitting the cache, a micro-service could have been implemented that receives increment requests and periodically processes a buffer of requests on an independent working thread. However, this was not pursued in this design. The multiple split cache is the approach that was selected and implemented.

Figure 4.8 shows the integration of the software cache with gRPC in scanning a remote Arrow table to

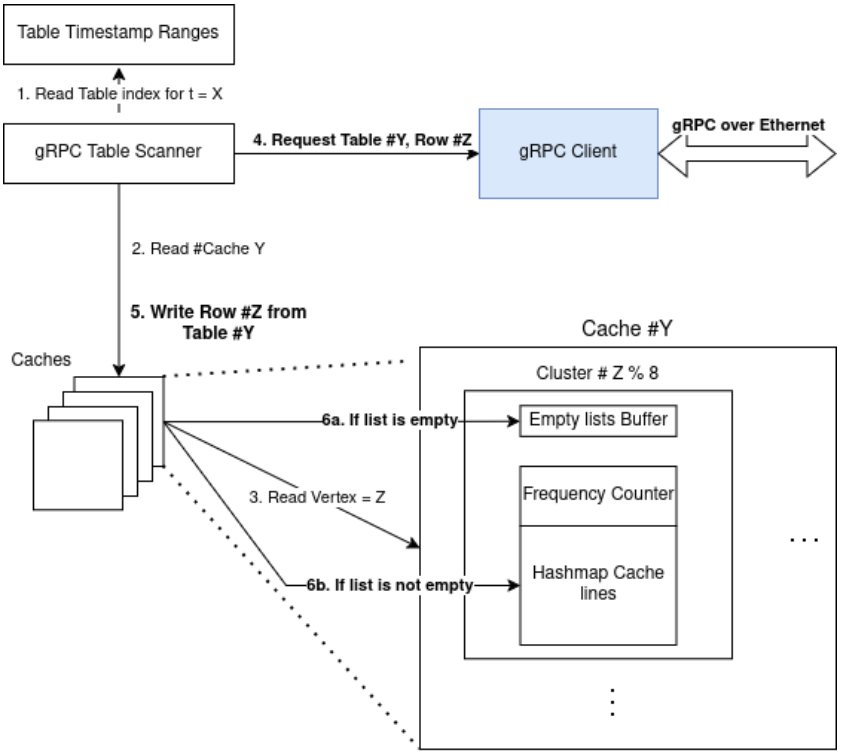


Figure 4.8: Reading the data of vertex Z within time range X from Arrow graph using gRPC communication protocol

find the neighbors of a vertex in the Arrow graph. It shows the additional steps required for reading and caching remote data using gRPC as a communication and data transfer protocol. The steps for both small- and full caches are the same, with the only difference being the type of gRPC payload that is being transferred. ThymesisFlow prototype, described in the previous section, performs all these steps via normal CPU instructions and leverages the power of the hardware to cache the data.

4.5. Dataset enlargement

One of the dimensions that was of interest is the scalability of the proposed implementation. By increasing the size of the graph, the memory footprint of the implementation and the ease of caching come into question. Although the AML large dataset is 17GB and incurs a memory usage of 10s of gigabytes, it was decided to create larger datasets to study the impact on the performance of algorithms.

In order to only increase the graph size without impacting the compute required, one has to keep the graph properties the same. In AML datasets, the timestamps are incremented in minutes. So, the dataset is enlarged by duplicating the transactions and creating new unique vertices for each involved bank account. This process can be done 59 times to create a dataset that is 60 times larger. Enlarging the graph in this manner keeps the number of fraudulent transactions, and consequently the required compute for detecting patterns, the same as before. This increases the graph's active memory footprint and decreases the locality of data. This allows the impact of remote memory with increased latency and lower bandwidth to become more apparent. This also increases the number of transactions per second to more closely resemble those of real banks. However, the large dataset is only enlarged by a factor of two in this thesis to show the trend in performance given a larger graph.

5

Results & Discussions

In previous work, the memory characteristics of the system with the ThymesisFlow hardware/software stack were sufficiently documented and are not further expanded on in this report. There are many benchmarks that can be performed to extract useful information, to not only show the performance of graph algorithms, but to predict the performance in a different system, with different memory latency as an example, as well. However, to evaluate the implementation, mostly benchmarks that measure the overall throughput of the graph algorithms are performed. In addition, some tests are performed to get an indication of the ability of the data to be cached for scatter-gather pattern and simple cycle enumeration algorithms.

5.1. Test setup

The main benchmarks are run and the ThymesisFlow test setup at Hasso Plattner Institute (HPI). The setup consists of two IBM Power9 IC922 servers. The servers are connected together with a network fabric through two OpenCAPI enabled Alpha Data 9V3 FPGAs that are attached to host systems via OpenCAPI ports. The FGPA in the lender node is configured in C1 mode, which is a processing element with the ability to access the system memory coherently. As such, it can snoop CPU cache lines, and invalidate them when it writes to the memory region. The FPGA on the borrower node is configured in M1 mode. This means that it is seen as a memory controller with an address space mapped to it. In this mode, it allows the borrower node to access the disaggregated memory region using store/load semantics. The disaggregated memory in the original prototype was configured to appear as an extra NUMA node without any CPU. However, in the HPI's setup, a library is created that allows the disaggregated memory to be exposed as memory-mapped file[8]. The second option allows the disaggregated memory region to be visible on the lender node as well. The shared memory-mapped file can then be used in any program by mapping the file into the program's address space, using *mmap* system call in Linux. The specifications of the Power9 servers are listed below[10, 27].

- ppc64le instruction set architecture
- 12 cores in SMT-4 configuration, 48 threads per processor
- 2 out of 4 server sockets populated [1]
- 768KiB L1d cache, 768KiB L1i cache, 6MiB L2 cache, 120 MiB L3 cache
- 16 x ECC DDR4 2666 MHz 32GiB memory, 8 banks per socket
- MT27710 (connectX-4 Lx) 25Gbit/s network card

Two other platforms are used in the evaluation as well. The first system is an Cascade Lake Intel Xeon based server with 32 cores and 64 thread configuration running on IBM cloud service. This is the system used to benchmark the GFP. However, the exact processor used and other system specifications are not mentioned. Therefore, it is not possible to list the cache information or system memory that was

[1] Only one processor is used in benchmarks to ensure consistent memory latency across runs.

used in the tests. The other system is the personal computer (PC) used for the analysis of hardware independent and deterministic benchmarks. This includes the software cache hit/miss rate and its impact on throughput during processing of a complete dataset. The specifications of the local PC are listed below.

- Intel 12700H Alder Lake processor
- 14 physical cores, 20 threads (2x 6 P-core + 12 E-core)
- 544KiB L1d cache, 704KiB L1i cache, 11.5 MiB L2 cache, 24 MiB L3 cache
- 2 x non-ECC DDR5 5600MHz 32GiB memory

5.1.1. Power9 NUMA domains

The tests are performed on Power9 server that has two CPUs, each with its own memory banks. By allowing the Linux kernel to manage the threads itself, the program will run on both CPUs. One of the issues with allowing both CPUs to be utilized is less consistent memory latency, as some processes run on a CPU that is on another NUMA node with different memory latency. Additionally, the amount of CPU cache available to the application will also double. This hides the latency of ThymesisFlow to a greater degree. To inspect the available NUMA nodes and their respective latencies, `numactl` utility can be used. The relative latency between the server's NUMA nodes can be seen with command `numactl -H`. The output of the command, in the compute node, can be seen below:

```
$ numactl -H
available: 4 nodes (0,8,48,64)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
    26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
node 0 size: 261683 MB
node 0 free: 245579 MB
node 8 cpus: 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70
    71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
node 8 size: 260735 MB
node 8 free: 253952 MB
node 48 cpus:
node 48 size: 0 MB
node 48 free: 0 MB
node 64 cpus:
node 64 size: 0 MB
node 64 free: 0 MB
node distances:
node   0   8  48  64
  0:  10  40  80  80
  8:  40  10  80  80
 48:  80  80  10  80
 64:  80  80  80  10
```

To avoid inconsistent memory latencies, `numactl` is used to run the experiments on NUMA node 0, as the FGPA OpenCAPI cards are attached to this NUMA domain. This is achieved by using the following command:

```
$ numactl -N 0 -m 0 <COMMAND>
```

5.2. Datasets

The evaluation of the implementations are performed using publicly available synthetic anti-money laundering (AML) datasets produced by the AML world generator[20]. The datasets are available in different sizes and have two variants, a low (AML LI) and a high (AML HI) illicit rate dataset.

The GFP paper[16] reported the throughput of the complete pipeline that includes both graph feature extraction and ML pipeline for all six variants of the AML datasets. However, the algorithm specific performance of only AML HI datasets is documented and obtained from the researcher. Therefore, the

benchmarks are limited to AML HI datasets only. In order to test the scalability of our implementation given larger datasets, we decided to duplicate existing AML HI large dataset. The strategy for data duplication was to reinsert the already existing bank transaction into the graph with completely new unique nodes and new timestamps. In fact, this creates multiple disconnected graphs in the dataset. Adding random edges (i.e., transactions) in the graph increases the size of the dataset and the density of transactions per unit time. However, the advantage of the first approach is that the memory footprint of the dataset and transaction density will increase without changing the graph statistics, such as illicit rate, fan in/out, degree in/out, etc. Therefore, the ratio of compute and read operations remains the same while the pressure on the memory system increases. For this reason, this data enlargement strategy is used to create larger datasets.

Table 5.1 summarizes the datasets that are used and the size of the resulting graph sizes, specifically the Arrow graph. The documented size includes only the data in the vertices and timestamp columns, with the originating vertex being inferred from the row number. The size also includes the second graph with inverse edges as well. The size of the native graph, the Arrow tables' metadata and additional auxiliary data structures for graph management are excluded.

Table 5.1: Evaluation datasets and resulting Arrow graph size

Dataset	# nodes	# edges	time span	# Arrow tables	Average Table size [MB]	Total Arrow graph data size [MB]
AML HI Small	0.5 M	5 M	10 days	9	5.5MB (+/- 3MB)	100 MB
AML HI Medium	2.1 M	32 M	16 days	19	21MB (+/- 10MB)	800 MB
AML HI Large	2.1 M	180 M	97 days	98	21MB (+/- 12MB)	4100 MB
AML HI Large 2x	4.2 M	360 M	97 days	98	42MB (+/- 24MB)	8200 MB
AML HI Large 60x	126M	10.8G	97 days	98	2.52GB (+/- 720MB)	246GB

5.3. Hybrid graph vs graph feature preprocessor performance

The first benchmarks are designed to compare the efficiency of the new implementations with previous work to ensure that they are of sufficient quality. They include measuring the throughput of the new implementation for detecting scatter-gather patterns and simple cycles when processing transactions in batches of 2048 transactions. The throughput of batch processing in previous work is performed in real-time and includes only the performance of algorithms. Running the hybrid implementation in real-time mode with search windows smaller than 24 hours utilizes only the dynamic portion of the hybrid graph. This test is the most direct comparison to the GFP as the dynamic portion of the graph is its replica. In order to measure the impact of the Arrow graph, and remote memory specifically, it was necessary to run the offline version of the hybrid graph as well. The offline version includes inserting the complete dataset into the hybrid graph before processing the transactions in batches. This forces the hybrid implementation to utilize the Arrow graph and produces the same results as real-time processing.

For the scatter-gather pattern and simple cycle enumeration, a search window of six and 24 hours is chosen, respectively. These benchmarks are only performed on Power9 systems using the small and medium datasets. Figure 5.1 shows the throughput of the scatter-gather pattern detection algorithm for both datasets. The first observation is that the throughput of the hybrid graph, both the dynamic and the Arrow graph, matches and exceeds the performance of GFP in the small dataset. However, a numerical comparison with previous work is not the goal, as the platforms differ greatly from each other. Looking at results of the hybrid version, it can be seen that the real-time implementation, which used the dynamic graph, is significantly slower than the offline version that uses the Arrow graph. This is because the Arrow tables contain the data in a compact contiguous piece of memory. This provides high data locality and more efficient caching. The dynamic graph, by using hashmaps to represent the graph, provides flexibility for data insertion and removal at the cost of lower data locality, and thus performs worse.

The dashed line in the graph shows the throughput when the remote memory is used instead of the local memory via the ThymesisFlow link. Given that remote memory has significantly less bandwidth and higher latency, the perceived performance drop is not that significant. This stems from the small size of the Arrow tables when using the small dataset. The low performance drop indicates that the data is small enough for it to populate the local cache. The larger the dataset, the more remote memory fetches that take place since all data cannot fit inside the cache. This is observed in the results of the

medium dataset. The performance drop for detecting scatter-gather patterns in the medium data is 33% compared to only 10% in the small dataset.

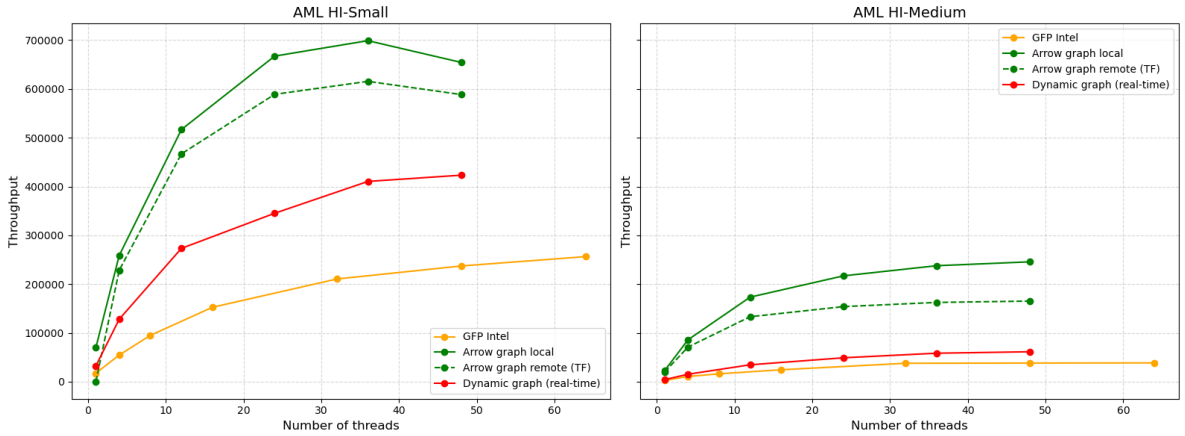


Figure 5.1: Scatter-gather detection throughput on Power9 powered ThymesisFlow prototype and Intel Xeon server from GFP

The performance of the simple cycle enumeration can be seen in Figure 5.2. Like the previous algorithm, the performance of the hybrid implementation, using both dynamic and Arrow graph, is comparable to the previous work. For simple cycle detection, it can be seen that the performance drop when using remote memory is much greater than scatter-gather pattern detection. In simple cycle detection, when using 48 threads, the performance drop for using remote memory is 37% and 61% for small and medium datasets, respectively. Simple cycle detection involves more data scans and scales exponentially compared to linearly in scatter-gather pattern detection.

Another observation that can be made is the impact of the remote memory with increased number of threads. In both scatter-gather pattern and simple cycle enumeration, the performance gap between local and remote Arrow graph increases as more threads are used. This is attributed to increased memory demand and the latency of remote disaggregated memory, which together form a bottleneck, leading to lower performance.

The results show that by designing a graph with high data locality, remote disaggregated memory can be used for real-time application of graph algorithms despite having a much higher latency and lower bandwidth than local memory.

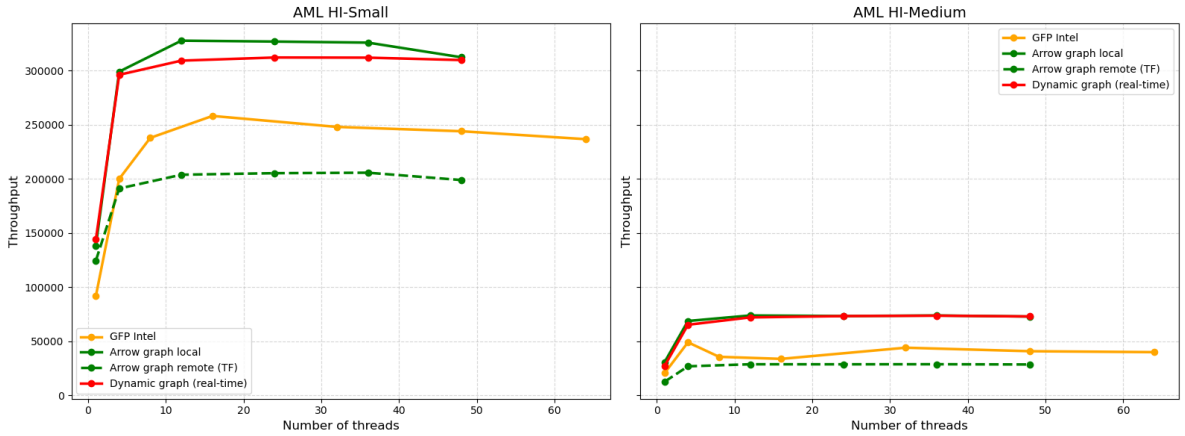


Figure 5.2: Simple cycle enumeration throughput on Power9 powered ThymesisFlow prototype and Intel Xeon server from GFP

5.4. gRPC based baseline

The baseline implementation performance is not included in the previous section as it performs poorly against both GFP and hybrid graph configurations, tested in the previous section, and requires separate benchmarks to cover the impact of different design dimensions, namely the software cache overhead, the cache parameters, and the communication link. The baseline implementation could have been optimized by modifying the algorithms and fully exploring the cache design space (size and cache line ejection strategy), but only a few cache configurations are tested using the same datasets and batch size from the previous section. The initial tests are performed with gRPC server on the same node, to first measure the impact of using the software cache and gRPC communication protocol for transferring data.

5.4.1. Cache

Cache line ejection policy

The first dimension of the baseline implementation that was explored, albeit briefly, is the cache size and the cache ejection strategy. For determining a suitable cache line ejection strategy, a brief benchmark was run, but the results are not compiled for this report as only a qualitative inference suffices. The cache line ejection strategies that are tested include LFU combined with LRU, random cache line ejection policy and FIFO buffer. The FIFO and random ejection policy, while less computationally intensive, performed noticeably worse with reasonable cache sizes. The higher performance of the LFU & LRU cache line ejection strategy is due to lower cache miss rates. It performs worse than random ejection policy and FIFO buffer only when an extremely larger cache is used. The reason for this flip is that the management overhead of the cache line counters will start to dominate and reduce the overall performance despite achieving better cache hit rates. This is because updating the nodes in the counter data structure, seen in Figure 4.7, scales linearly with the size of the cache. The worst time complexity for updating counters is $O(N + M)$, with N the number of unique read counts and M the maximum number of nodes with the same read count. FIFO and random ejection, on the other hand, do not have to keep track of cache line read counts, and can eject/insert cache lines with $O(1)$ worst case complexity regardless of the cache size.

The last cache rejection policy that is not implemented is LRU only cache ejection policy. Depending on the read patterns of the graph, it could have performed better than the current LFU & LRU policy. That would be the case if there were cache lines that are read frequently for a few batches and rarely used afterward. As the current LFU & LRU implementation has no mechanism to eject cache lines that have not been used for a substantial amount of time, it allows irrelevant cache lines, that were once frequently used, to stay in cache indefinitely. All the benchmarks that follow use LFU & LRU as their cache line ejection policy.

Cache size

Table 5.2 shows the cache sizes that are tested. Each software cache, which maps directly to the data of a single Arrow table, has a fixed number of cache lines. It stores either the full adjacency list (i.e., full cache) or the neighboring vertices only (i.e., small cache). As the number of neighboring vertices and the adjacency lists is dynamic and depends on the graph connectivity, the cache size is also dynamic and scales in the tests with the size of the datasets. The estimated cache size reported in the table is based on the small dataset.

Table 5.2: Explored cache cluster size with n = [2:4] for larger caches, with each cache having 8 clusters.

	Base			Enlarged		
	# Data lines	# Empty FIFO	Size	# Data lines	# Empty FIFO	Size
Small Cache	8192	8192	2MB	n x 8192	n x 8192	n$\times$ 2MB
Full Cache	4096	8192	5MB	n x 4096	n x 8192	n$\times$5MB

To document the performance uplift of the cache, the same search time window of six and 24 hours were selected for scatter-gather pattern and simple cycle enumeration, respectively, while maintaining a batch size of 2048 transactions. These tests are performed on a Power9 server, running both the client and the server application, and they benchmark the cache configurations presented in Table 5.2.

AML HI small scatter-gather detection results

The resulting cache hit rate and the performance relative to a disabled cache implementation for the small dataset can be seen in Figure 5.3. The x-axis represents the number of batches and ranges from zero to 2479. The y-axes present the measured throughput and cache hit rate in each batch and are represented with solid and dotted lines, respectively. Given that the results were extremely noisy, a moving average window size of 10 is applied to both data, leading to a smoothed and less noisy plot.

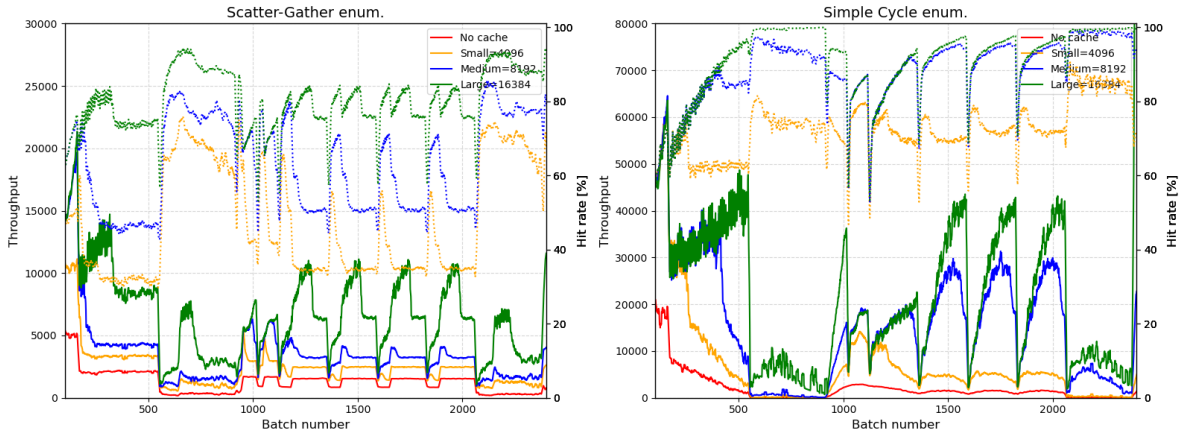


Figure 5.3: The throughput and software cache hit rate in AML HI small dataset per 10 batches, running on a Power9 server, with the server run locally. The average effective throughput of the complete small dataset for the scatter-gather pattern and simple cycle enumeration, with increasing software cache sizes, are $[676, 1776, 2567, 4700]$ and $[79, 300, 1390, 6022]$ transactions per second, respectively.

As scatter-gather pattern detection searches within a time window of 6 hours, the first 60 batches show inaccurately high performance as the real searched window is lower. By correlating the performance with the cache hit rate, one can draw conclusions as to whether the memory or the compute is the limiting factor. For example, in batches $600 - 900$ and $2100 - 2400$ the throughput is the *lowest* while the cache hit rate is the *highest*. This indicates that the transactions processed in those batches are located in denser parts of the graph where *more processing is required* per fetched data. This pattern is observed numerous times throughout the dataset in smaller batch windows. These patterns can be seen around batch numbers $1050, 1200, 1400, 1600$ and 1800. It can be seen that using the larger cache moderately improved the hit rate in these batches from 60% to 85%.

There are certain batches in this dataset, such as $1200 - 1250$ and $1450 - 1550$, where the throughput is relatively *higher* despite having *lower* cache hit rates. In these instances, the transactions are found in a portion of the graph where the vertices have lower connectivity, resulting in *reduced computations*. In these instances, it can be seen that increasing the cache size had a greater impact on hit rate than the compute intensive sections. In sections with lower available compute, where the data are reused less frequently, the cache hit rate increased from 35% to 80%, when the largest cache is used.

As expected, the performance gained when using the software cache is significant. In scatter-gather enumeration, the cache hit rate improves with increased cache size, resulting in higher performance. As the cache hit rate lies mostly around 80%, it may benefit from an even larger cache, but this was not further investigated. Looking at the effective throughput of all transactions, using 48 threads while running the server on the same node, an average throughput of 676 transactions per second is observed when no software cache is available. Performance increases by a factor of $2.63, 3.80,$ and 6.95, respectively, when using the small, medium, and large caches.

AML HI small simple cycle detection results

As simple cycle detection uses a larger search window of 24 hours, it requires more batches to be processed than the scatter-gather algorithm before reaching its real performance. Assuming a linear distribution of 2500 batches over 10 days, which is the time span of AML HI small dataset, it is expected that after 250 batches the performance will reach a steady state, since only then there are 24 hours worth of transactions in the graph. However, the throughput slowly settles after 600 batches. Since the

performance results of these batches are too noisy, they are not included in the analysis and serve only to saturate the cache.

There are two sections, in ranges $550-900$ and $2100-2450$, where performance is lower despite having the same cache hit rate. The largest cache in these two sections results in a $\sim 99\%$ cache hit rate and leads to a maximum performance uplift of 200 times compared to the version without the software cache. In these two batch windows, the largest cache also outperforms the small and medium caches by a factor of ~ 3 and ~ 3.5, respectively.

Having an almost 100% cache hit rate implies that many vertices are being re-used from the cache. This happens if the same paths are traversed multiple times despite not forming a cycle. This can only be improved by changing the current simple cycle detection algorithm, namely Tiernan algorithm, to prune these cycles.

Looking at the rest of batches, namely batches $1100 - 2100$, shows a higher cache hit rate than the scatter-gather algorithm, which raises some concerns about the observed cache efficiency of cycle enumeration algorithms. It was originally hypothesized that the cycle enumeration has more random reads than the scatter-gather algorithm and would consequently require a larger cache to achieve the same cache hit rate. The higher cache hit rate could either be attributed to vertices being reused across different transactions in the batch or during the processing of a single transaction. As the Tiernan algorithm does not prune repetitive paths, it is more likely that the vertices are being reused in multiple traversals of the same path while processing a single transaction.

The inefficiency of the selected cycle enumeration algorithm fails to show how frequently the vertices' adjacent lists, and thus cache lines, are re-used among different transactions within the same or following batches. In order to make conclusions about the re-using of adjacency lists across unrelated transaction, a more advanced cycle enumeration algorithm has to be implemented. A better cycle enumeration algorithm that can be implemented is the Johnson algorithm as it improves the Tiernan algorithm by avoiding duplicate paths that do not lead to a simple cycle[25]. Due to time constraints, implementation of a new algorithm was not pursued and left for future work.

AML HI medium results

Benchmarking the complete medium sized dataset using the baseline with a software cache was not feasible due to its low throughput, relative to the GFP and the disaggregated memory. In order to include meaningful measurements about the cache and algorithm performance, only a portion of the dataset could be tested due to time constraints. The sections are selected based on the average throughput measured, using the fast Arrow graph, on the local PC. Figure 5.4 shows the throughput of the scatter-gather pattern and simple cycle enumeration algorithms for the medium dataset, using the same batch size as before. Performance fluctuations between batches are so great that they result in an extremely noisy figure, just like AML HI small dataset. Using a moving average window reduces the noise and allows for an easier analysis[2]. The performance of the scatter-gather pattern and simple cycle enumeration algorithms stabilize around the $1000th$ and $2000th$ batch, respectively.

It can be observed that the performance of both the scatter-gather pattern and simple cycle detection varies to some degree in different batches. This difference in performance across batches is due to the same reason mentioned when benchmarking the small dataset, namely difference in the processing required. There are three batch windows in the simple cycle enumeration that show a much lower throughput, namely transaction ranges 2000-3800, 8200-9500, and 14100-15500. These batches, just like the small dataset, most likely contain transactions that are located in parts of the graph where many paths are traversed multiple times. Batches $3800 - 4500$ and $14200 - 15500$ were originally selected to benchmark the performance of both graph algorithms. However, the simple cycle enumeration performance is extremely low in the second batch. Therefore, only $100(14200 - 14300)$ batches of the second batch window are used in the benchmarking of the simple cycle detection algorithm.

The benchmarks run include 50 extra batches as well to allow the caches to be saturated. The selected batches highlight the worst and best case performance for both algorithms. Figures 5.5 and 5.6 show the performance and cache hit rate for the scatter-gather pattern and simple cycle enumeration, respectively. The results follow the same patterns as those of the small dataset, with the same observed

[2]Based on the amount of noise in the results, a moving average window size of 5-25 is used.

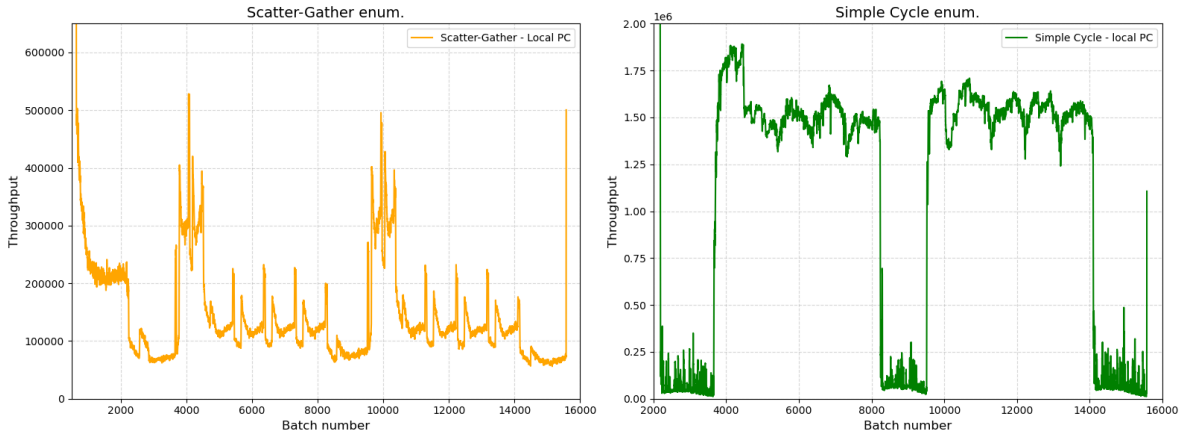


Figure 5.4: The throughput, transactions per second, of scatter-gather (Time window = 6 hours) and simple cycle (TW = 24 hours) enumeration per 10 batches, running in the local PC. The initial batches with extremely high performance that do not represent reality have been removed.

correlations between performance and cache hit rate as before. The throughput is later compared with that of ThymesisFlow in Table 5.3

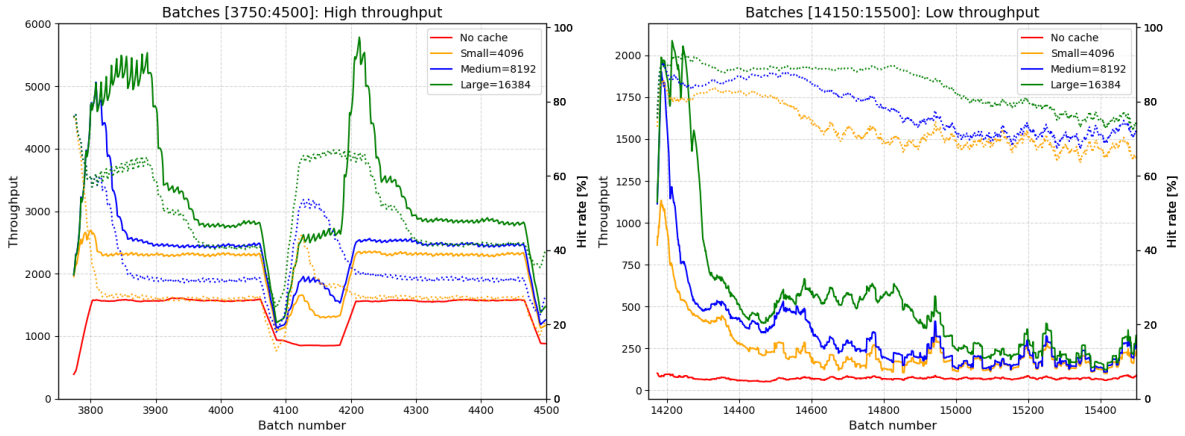


Figure 5.5: The performance of scatter-gather detection algorithm in two batch window sizes in AML HI medium dataset. The average throughput of high and low throughput batches, with increasing software cache sizes, are $[1229, 1944, 2154, 2659]$ and $[63, 141, 180, 283]$, respectively.

Performance summary

Table 5.3 summarizes the relative performance of ThymesisFlow compared to the baseline version with the largest cache. Despite the efficacy of the cache in terms of hit rate, the throughput is very low compared to ThymesisFlow. In order to match the performance of ThymesisFlow, significant effort needs to be invested in the graph algorithm implementations to reduce gRPC calls by further improving the cache and modifying the gRPC calls to carry more data per call.

Furthermore, while asynchronous gRPC calls are made concurrently, no compute is performed during the waiting. Overlapping the compute with asynchronous gRPC calls is not trivial and requires significant time and programming effort. On the other hand, ThymesisFlow benefits from the processor's out-of-order execution engine as it uses native load/store CPU instructions. Finally, the implementation of the software cache needs to be improved as well, since even the baseline version with almost 100% cache hit rate, and thus makes no gRPC calls, does not even come close to the performance of ThymesisFlow disaggregated memory system.

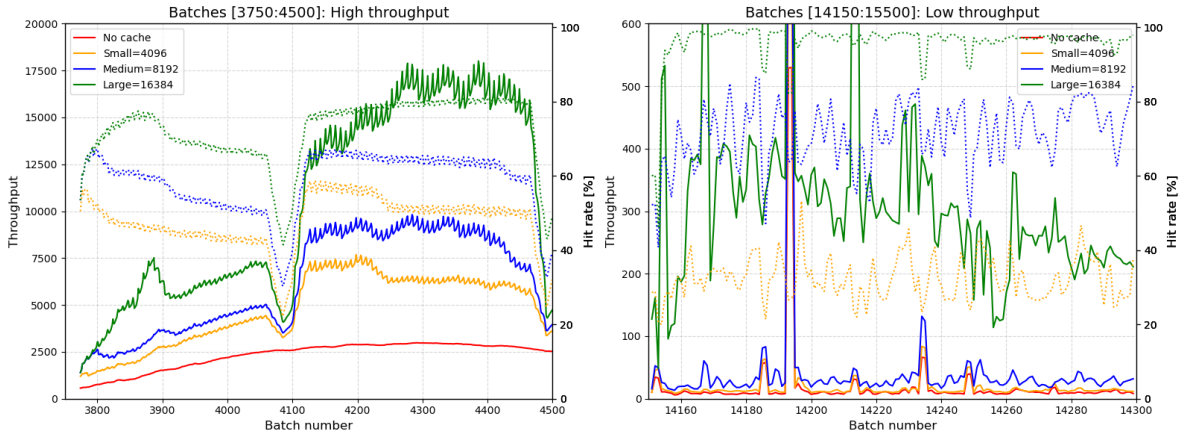


Figure 5.6: The performance of simple cycle detection algorithm in two batch window sizes in AML HI medium dataset. The average throughput of high and low throughput batches, with increasing software cache sizes, are $[1836, 3300, 4074, 5656]$ and $[9, 13, 25, 203]$, respectively.

Table 5.3: Performance of graph algorithms in AML HI small and two batches from the medium dataset from High (3800-4500) and Low (14200-15500 for SG, and 14200-14300 for CE) throughput sections, while using the hybrid graph in remote memory. Reported gRPC throughput is with the largest cache configuration.

	Scatter-gather enumeration			Simple cycle enumeration		
Datasets	gRPC	ThymesisFlow	Relative	gRPC	ThymesisFlow	Relative
Small	4699.82	588505	125x	6021.84	199029	33x
Medium High	2658.65	208085	78x	5655.77	1771500	313x
Medium Low	282.8	64572.5	228x	203.3	10828.2	53x

5.4.2. Software cache's extra overhead without gRPC calls

The gRPC and ThymesisFlow versions both use the same underlying Arrow graph data structures and apply the algorithms in the same way. The only difference between them is the way the neighboring vertices and timestamps are retrieved. ThymesisFlow uses the processor's hardware cache and retrieves data in sizes of L3 cache line, namely 128 bytes. The gRPC version, on the other hand, retrieves data in variable payloads that can range from a few bytes to kilobytes of data if the adjacency list is not found in the software cache. In the results of AML HI small dataset, specifically the simple cycle enumeration benchmark in Figure 5.3, batches $600 - 900$ showed a software cache hit rate of more than 99%. So, during these transaction batches, almost no gRPC calls are made, and data are exclusively retrieved from the software cache. Running the same batches using the ThymesisFlow can be very interesting as it shows how much the performance is impacted by introducing the software cache alone.

Figure 5.7 compares the performance of the simple cycle enumeration of the Arrow graph using ThymesisFlow and the software cache, when no gRPC calls are made. It can be seen that introducing the software cache reduces the performance by a huge margin. The ThymesisFlow disaggregated memory outperforms the baseline version, that only has a software cache added to it and makes almost no gRPC calls, by a factor of 18.6. The reason for this is the design of the software cache itself. Whenever a cache line is read, the data of the cache line is copied by the running thread as it should not block the cache line when the algorithm is being performed. The reason being is that it can keep a cache line locked for an extended amount of time even if it needs to be ejected. The impact of the software cache can be minimized by avoiding copying data. This can be realized by creating a more sophisticated cache line ejection mechanism where cache lines can be locked for concurrent reads and their ejection is delayed or skipped when they are used in algorithms. An improved software cache was not pursued because it falls outside the scope of the project.

5.4.3. Physical Ethernet link

Another component of the baseline that can be tested is the impact of the communication link between the nodes through the 25Gbit/s network cards. In order to determine this, a benchmark is repeated

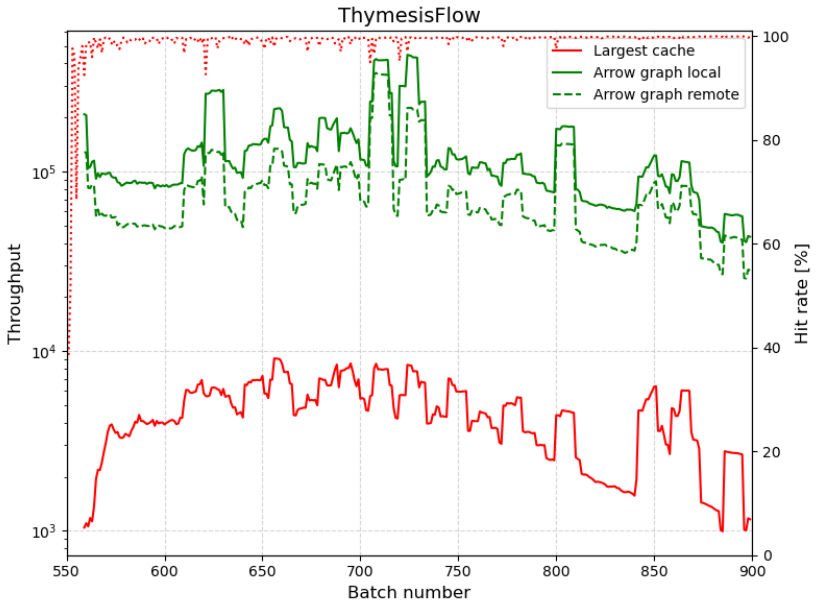


Figure 5.7: Throughput comparison between the baseline, with software cache and near zero gRPC calls, and ThymesisFlow remote memory. The average throughput of baseline version, the remote graph, and local graph for batches 600-900 are 2617, 48703.8, and 81387.2, respectively.

with the server running on either the local or remote node. When the cache is disabled, all Arrow graph scans are done via gRPC messages instead of CPU load instructions. By testing the baseline with no cache, we can show how much performance is lost due to the gRPC software stack compared to the physical link. As the cycle enumeration's performance is more sensitive to memory latency, it is the only algorithm that is included in this test, with the same 24 hour search window. The tests are run on the medium dataset, using the high throughput batches $3800 - 4500$.

Figure 5.8 shows the relative performance of the baseline when the physical network link is used. It can be seen that using the physical network link results in a performance drop of approximately 15%. However, using ThymesisFlow results in a throughput that is **965** times higher than of a locally hosted gRPC server. So, the cost of introducing the network software stack, including data (de)serialization, leads to a much larger drop in performance than including a physical link between nodes in a cluster.

5.4.4. Future optimizations

As performance is well documented in previous sections, it can be concluded that the baseline implementation using the hybrid graph with gRPC communication protocol requires significant effort and modifications, to both the algorithms and the software cache, to match the performance of the application using ThymesisFlow. The hybrid graph could easily be used with remote disaggregated memory, with some effort to initialize data in the remote region due to the limitations of the remote memory pool. However, the baseline required much more effort and is more complex without being able to match the performance of ThymesisFlow. This shows that utilizing a disaggregated memory system for graph algorithms is much more convenient for a programmer than a network based framework and matching the performance of a disaggregated memory system involves a software implementation of hardware features available in the processor, such as CPU caches and out-of-order execution engine.

To improve the baseline, the software cache design space has to be further explored to maximize the cache hit rate and minimize its overhead, which was not discussed in this report. More importantly, the defined communication protocol can be improved. The average message payload is very small, ranging from hundreds to a few kilobytes of data. With each message carrying a latency of 3ms[10], it is detrimental to have many gRPC calls for small data transfers. Aggregating multiple gRPC calls reduces the number of remote calls and leads to improved performance, but makes comparisons with disaggregated memory less direct. Moreover, it places the responsibility of managing gRPC calls on

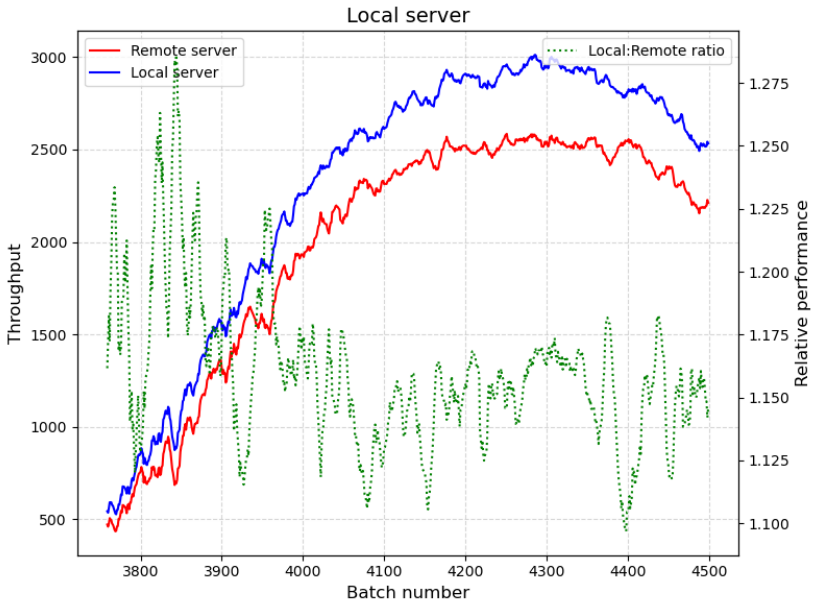


Figure 5.8: The performance of the baseline without any cache for the high throughput batches 3800-4500 of the medium dataset. The throughput, when using a local server, is on average 17% higher. The average throughput of the baseline when using local and remote servers are 1836 and 1566 transactions per second, respectively.

the programmer or the underlying library.

A major modification that has to be made to the algorithms to improve performance is to ensure that computations are being overlapped with asynchronous gRPC calls. The modifications needed for the baseline implementation using gRPC with a software cache to perform better are not implemented and are left as future work. Due to the lackluster performance of the baseline, it is not included in the remaining benchmarks.

5.5. Real-time with longer search window

A main goal of this thesis was to investigate whether disaggregated memory can be used for a realistic application, namely in financial crime detection to detect graph features in real-time. The search windows used in previous sections were chosen to mainly compare the throughput of the newly designed application with previous work to confirm the quality of the design. In this section, benchmarks are run that include much longer search windows, allowing both the dynamic and the Arrow graph, to be used simultaneously. The tests performed in this section are run on only the Power9 servers using 48 threads to document the maximum possible throughput. The extended search windows used are summarized in Table 5.4.

Table 5.4: The extended search window sizes used for graph algorithms, given in hours, for 3 original and 1 duplicated dataset.

	Datasets			
Algorithms	AML HI Small	AML HI Medium	AML HI Large	AML HI Large x2
Scatter-gather	24, 72	24, 72	72, 240, 720	240, 720
Simple Cycle	72	72	72, 240	240

Table 5.5 shows the throughput using the extended search windows, running on Power9 server using 48 threads. These tests were only able to be run for a limited time, due to time constraints. For the small and medium datasets, all transactions are processed. However, for the large dataset, each tests it run for 20 minutes. So, they have not been run long enough to fully saturate the search window, specially 240 and 740 hour (10 day and 30 day) search windows. However, the tests show that it is possible to run the larger datasets with a distributed graph running on a memory disaggregated system.

Table 5.5: Performance of real-time scatter-gather (SG) pattern and simple cycle enumeration (CE)

	ThymesisFlow				Local		
	24h	72h	240h	720h	72h	240h	720h
Small SG	277624	109966					
Small CE	309893	39290.5					
Medium SG	41469.7	19663.9					
Medium CE	72967.6	5010					
Large SG		21607.4	16535.8	13209.4	27733.4	18698.5	14729.9
Large CE		10152.3	9165.6		21238.6	9807.02	

5.6. Last level cache misses

Although ThymesisFlow is capable of running graph algorithms, the performance numbers alone are not enough to highlight the main advantages of disaggregated memory. One main advantage that was mentioned earlier is its ease of programming once the instantiation and memory allocation become more developed. Once the remote memory pool is fully developed, a developer can use remote disaggregated memory as if it were local memory on a NUMA node with higher latency.

Using the hybrid graph with ThymesisFlow performed well despite having ten times higher memory latency (850-950ns vs 74ns[8, 10]) compared to local memory. Additionally, its bandwidth, for random and sequential reads, is 35 times slower. During the execution of graph algorithms, ThymesisFlow's peak memory bandwidth reached 2.5GB/s. For instances where most of the reads were random, the remote memory's throughput averaged about 400MB/s.

In comparison to ThymesisFlow, CXL attached memory has much higher bandwidth [3] and relatively lower latency at around 170-250ns with real CXL memory expanders showing a latency of 214-394ns (+ 94-227ns)[6]. Power10's memory inception has even lower latency of around 150ns. So, it is more important to be able to relate the performance to the latency of the system and predict how high the performance could get given lower memory latency. Although, we were not able to approximate the performance due to time constraints, a simple method devised to predict the performance, once LLC cache misses are collected by `perf` tool in Linux, is presented.

In order to calculate the impact of remote memory on performance, one has to look at the difference in data handling and look at the average memory access time. Due to the out-of-order-execution engine of modern processors, it is not possible to accurately predict the performance of another system with different memory latency as the impact of the hit penalty is reduced by out of order execution which depends on the compiler and the processor. The assumption is made that the algorithms are memory bound and the performance is dependent on the memory access times only, and the simple formula used to predict performance by the amount of time it takes for the data to be loaded from memory can be seen below, with T, M, and P representing hit time, miss rate, and miss rate penalty, respectively:

$$Memory\ access\ time = T_{L1} + M_{L1} \times (T_{L2} + M_{L2} \times (T_{L3} + M_{L3} \times P_{L3}))) \tag{5.1}$$

As the workload is the same, the number of cache misses in all cache levels remains the same, with the Last Level Cache (LLC) miss penalty the only differing variable. This leads to a simpler formula, with the LLC miss penalty being either the L3 cache miss penalty or ThymesisFlow disaggregated memory latency:

$$Memory\ access\ time = Cache\ time + LLC\ miss\ rate \times LLC\ miss\ penalty \tag{5.2}$$

It is better to collect the LLC misses with the algorithms running in offline mode, as it makes the approximation more accurate by (almost exclusively) reading graph data from the Arrow graph only. The LLC cache misses can be collected with the following command: `perf stat -e LLC-loads,LLC-load-misses <COMMAND>`. Then, by filling in the runtime for the same number of batches in the formula above, the *cache time* and *LLC miss rate* can be solved. Then, approximately predicting the performance of a system with different memory latency can be done by filling in the memory latency as *LLC miss penalty* in the aforementioned formula.

[3] The bandwidth depends on the number of CXL links and can reach 100s of GB/s when using all CXL lanes.

5.7. Future work

The following list shows skipped benchmarks that could have been performed to collect valuable insight in graph algorithms and modeling them in other systems with different memory characteristics. However, these were not deemed necessary for conveying the

- Infinite batch throughput: The original GFP paper includes measurements with infinite batch size. Having an infinite batch size allows for more parallelism to be exploited and highlights the scalability of the implementation. As the main goal was not to produce the most efficient implementation, including it would not have added meaningful insight.
- gRPC latency: This was performed in precious work and is about 3ms on average. However, the average latency for a set of asynchronous can be measured as well to document the gRPC's overhead.
- Horizontal scaling: With integration of Apache Arrow with ThymesisFlow, the remote node is also capable of using the disaggregated memory. By Running a portion of the compute on the lending node, the scaling of the algorithms within a cluster of nodes, and bus contention as a consequence, could have been investigated as well.

Additionally, there are numerous limitations that were faced during the implementation of the graph algorithms, regarding the state of the custom remote memory pool that can reduce the amount of effort required by other researchers to use the disaggregated memory in their experiments. Furthermore, there are some new research questions has been unraveled that are worth pursuing regarding distributed graph algorithms. The potential projects for future work are listed below.

1. By fully developing and issues of the current memory allocator, it can enable other researchers that use Apache Arrow or want to adapt their work to use Apache Arrow to run their tests in disaggregated memory systems as well without any additional effort.
2. As the disaggregated memory system can be used as shared memory, the orchestrator class can be updated to allow both nodes to work in parallel. These new benchmarks can then show well the performance of the workload scales when the shared memory region is used by multiple nodes simultaneously.
3. It is also possible to pursue implementing graph algorithms using GraphX distributed graph framework. At the time of writing, we were not able to find distributed implementation of graph-based pattern detection algorithms. It is interesting to know how well GraphX would perform running more complex subgraph pattern detection algorithms.

6

Conclusion

Through a literature study on workloads that use disaggregated memory systems, a lack of applications developed to leverage disaggregated memory was observed. In this thesis, the feasibility of using disaggregated memory systems in a realistic application was investigated by implementing graph processing algorithms.

The graph processing application was based on a graph feature preprocessor, found in SnapML library, and is capable of detecting two subgraph patterns, namely scatter-gather pattern and simple cycles. In order to use the disaggregated memory as a shared memory region, a hybrid graph is implemented on top of the ThymesisFlow integrated Apache Arrow framework to ensure full software coherency. The hybrid graph is comprised of two graphs: a dynamic graph using native C++ data structures, and an Arrow graph made up of multiple Arrow tables. The Arrow graph contains old transactions and is located in the shared memory region, while the dynamic graph holds the latest transactions only.

Finding a traditional distributed graph implementation to use as a baseline proved to be more troublesome. Although GraphX looked promising at first, the amount of effort and time required to implement the selected graph algorithms was substantial. Furthermore, the implementation of the algorithms would have differed greatly from those created for the hybrid graph, making the comparison focused on the communication link difficult. Therefore, a baseline implementation that uses the network for transferring data, had to be designed. In order to keep the focus on the communication link, the hybrid graph is kept in tact. Minimal changes are made to the implementation of graph algorithms. In the baseline implementation, only reading of the Arrow tables is modified. Instead of reading the tables directly, required row data are retrieved by gRPC calls. In addition, a software cache is implemented to reduce the impact of increased latency of gRPC calls.

The main qualitative measurement metric that is used for the implementations is the throughput, namely the number of transactions they can process per second. The benchmarks test the performance of the algorithms on Arrow graphs using both local and disaggregated memory. The benchmarks, run on synthetically generated datasets, showed that the newly implemented algorithms performed similarly and sometimes better than the previous work, namely the GFP. Using the disaggregated memory with ThymesisFlow, reduces the performance of the benchmarks by 40-60%, relative to local benchmarks and depending on the size of the datasets. Running the benchmarks in real-time mode on much larger datasets, using the full hybrid graph, showed that the performance drop introduced by ThymesisFlow remains acceptable, despite its significantly higher memory latency. This suggests that disaggregated memory systems are well-suited for graph algorithms, provided the data access patterns are cache-friendly

The research questions outlined and covered in this work are repeated and answered in the following list:

1. **Is there a practical application that could leverage the advantages of disaggregated memory systems over scale up and scale out systems?**

In this paper, two practical graph processing-based applications in different industries are deemed suitable workloads for disaggregated memory systems. Graph-based pattern detection algorithms performed well in terms of performance, despite the ThymesisFlow prototype's high memory latency.

2. **What are the main challenges of implementing workloads, specifically graph algorithms, on memory disaggregated systems?**
   There are multiple challenges in implementing any workload on disaggregated memory systems. First, the application instances running on different nodes must be aware of the program state of others and must synchronize their write operations to the remote memory region. Second, The workload's data structures must be made immutable for cache coherency to hold. The challenges regarding data management can be completely alleviated by using the newly added remote memory pool, if the implemented remote memory pool is fully developed. The only challenge that remains is the transforming of the underlying data structure in workloads to Arrow tables while ensuring compatibility of data interfaces with third-party software that is usually used in large workloads.
3. **How do the workloads perform in memory disaggregated systems compared to traditional network based distributed memory systems?**
   Graph processing workloads, such as scatter-gather pattern and simple cycle detection, work exceptionally well when compared to the gRPC communication based data transfers. This holds especially when the same amount of effort is put into adapting the workload for cluster shared memory as getting a network based adaptation to perform as well as ThymesisFlow requires significant effort for efficient software caching.

There are still many graph algorithms across different disciplines that can use disaggregated memory systems with Apache Arrow to scale their performance by adding nodes while ensuring memory-utilization efficiency. By showcasing two graph-based pattern detection algorithms in a practical workload, we hope to have opened the door for more workloads to be adapted and tested in memory disaggregated systems.

References

[1] Mohammad Ewais and Paul Chow. “Disaggregated Memory in the Datacenter: A Survey”. In: *IEEE Access* 11 (2023), pp. 20688–20712. DOI: 10.1109/ACCESS.2023.3250407.

[2] M. Tirmazi et al. “Borg”. In: *Proceedings of the Fifteenth European Conference on Computer Systems* (2020), pp. 1–14. DOI: 10.1145/3342195.3387517.

[3] Huaicheng Li et al. “Pond: CXL-Based Memory Pooling Systems for Cloud Platforms”. In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ASPLOS 2023. Vancouver, BC, Canada: Association for Computing Machinery, 2023, pp. 574–587. ISBN: 9781450399166. DOI: 10.1145/3575693.3578835. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/3575693.3578835.

[4] Dina Henderson. *Cloud Scalability: When Should You Scale-Up vs. Scale-Out?* https://www.ibm.com/think/topics/scale-up-vs-scale-out. IBM Think, accessed 2026-06-08.

[5] Georgios Zervas et al. “Optically Disaggregated Data Centers With Minimal Remote Memory Latency: Technologies, Architectures, and Resource Allocation

Invited

”. In: *J. Opt. Commun. Netw.* 10.2 (Feb. 2018), A270–A285. DOI: 10.1364/JOCN.10.00A270. URL: https://opg.optica.org/jocn/abstract.cfm?URI=jocn-10-2-A270.

[6] Minho Ha et al. “Dynamic Capacity Service for Improving CXL Pooled Memory Efficiency”. In: *IEEE Micro* PP (Mar. 2023), pp. 1–9. DOI: 10.1109/MM.2023.3237756.

[7] Christian Pinto et al. “ThymesisFlow: A Software-Defined, HW/SW co-Designed Interconnect Stack for Rack-Scale Memory Disaggregation”. In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2020, pp. 868–880. DOI: 10.1109/MICRO50266.2020.00075.

[8] Felix Eberhardt et al. “Moses: Heap Partitioning for Semantic Data Tiering”. In: *Proceedings of the 2nd Workshop on Disruptive Memory Systems*. DIMES ’24. Austin, TX, USA: Association for Computing Machinery, 2024, pp. 25–32. ISBN: 9798400713033. DOI: 10.1145/3698783.3699386. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/3698783.3699386.

[9] Philip Groet et al. “Leveraging Apache Arrow for Zero-copy, Zero-serialization Cluster Shared Memory”. In: *arXiv preprint arXiv:2404.03030* (2024). arXiv: 2404.03030 [cs.DC].

[10] P. M. Q. Groet. “Zero-serialization, Zero-copy Memory Pooling in Compute Clusters: Disaggregated Memory Made Accessible”. MA thesis. Delft, The Netherlands: Delft University of Technology, 2023. URL: https://repository.tudelft.nl/record/uuid:e879eefc-4b8e-4a4b-b70e-f95128993aca.

[11] Tobias Mann. *Just How Bad Is CXL Memory Latency?* Dec. 2022. URL: https://www.nextplatform.com/store/2022/12/05/just-how-bad-is-cxl-memory-latency/1653612 (visited on 06/08/2026).

[12] Jinshu Liu et al. “Systematic CXL Memory Characterization and Performance Analysis at Scale”. In: *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ASPLOS ’25. Rotterdam, Netherlands: Association for Computing Machinery, 2025, pp. 1203–1217. ISBN: 9798400710797. DOI: 10.1145/3676641.3715987. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/3676641.3715987.

[13] William J. Starke and Brian W. Thompto. “IBM’s POWER10 Processor”. In: *IEEE Hot Chips 32 Symposium (HCS 2020)*. IEEE, Aug. 2020, pp. 1–43. URL: https://hc32.hotchips.org/assets/program/conference/day1/HotChips2020_Server_Processors_IBM_Starke_POWER10_v33.pdf (visited on 06/08/2026).

[14] Mikhail Kolmogorov et al. “Assembly of long, error-prone reads using repeat graphs”. In: *Nature Biotechnology* 37.5 (May 2019), pp. 540–546. ISSN: 1546-1696. DOI: 10.1038/s41587-019-0072-8. URL: https://doi.org/10.1038/s41587-019-0072-8.

[15] Borja Freire, Susana Ladra, and José R. Paramá. “Memory-Efficient Assembly Using Flye”. In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 19.6 (2022), pp. 3564–3577. DOI: 10.1109/TCBB.2021.3108843.

[16] Jovan Blanuša et al. “Graph Feature Preprocessor: Real-time Subgraph-based Feature Extraction for Financial Crime Detection”. In: *Proceedings of the 5th ACM International Conference on AI in Finance*. ICAIF ’24. Brooklyn, NY, USA: Association for Computing Machinery, 2024, pp. 222–230. ISBN: 9798400710810. DOI: 10.1145/3677052.3698674. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/3677052.3698674.

[17] Archit Patke et al. “Evaluating Hardware Memory Disaggregation under Delay and Contention”. In: *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2022, pp. 1221–1227. DOI: 10.1109/IPDPSW55747.2022.00210.

[18] OpenCAPI Consortium. *OpenCAPI 3.1 Transaction Layer Specification*. Version 1.0. Approved for distribution to non-members for learning purposes. OpenCAPI Consortium. Jan. 28, 2020. URL: https://computeexpresslink.org/wp-content/uploads/2024/02/OpenCAPI-3.1-Transaction-Layer_28Jan2020.pdf.

[19] Embedded Artistry. *embedded-resources: Embedded Artistry Templates, Documents, and Source Code*. GitHub repository. 2026. URL: https://github.com/embeddedartistry/embedded-resources (visited on 06/08/2026).

[20] Erik Altman et al. “Realistic Synthetic Financial Transactions for Anti-Money Laundering Models”. In: Dec. 2023. DOI: 10.52202/075280-1300.

[21] Toyotaro Suzumura and Hiroki Kanezashi. *Anti-Money Laundering Datasets: InPlusLab Anti-Money Laundering DataDatasets*. http://github.com/IBM/AMLSim/. 2021.

[22] Jack Nicholls, Aditya Kuppa, and Nhien-An Le-Khac. “Financial Cybercrime: A Comprehensive Survey of Deep Learning Approaches to Tackle the Evolving Financial Crime Landscape”. In: *IEEE Access* 9 (2021), pp. 163965–163986. DOI: 10.1109/ACCESS.2021.3134076.

[23] Guolin Ke et al. “LightGBM: a highly efficient gradient boosting decision tree”. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS’17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 3149–3157. ISBN: 9781510860964.

[24] Tianqi Chen and Carlos Guestrin. “XGBoost: A Scalable Tree Boosting System”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 785–794. ISBN: 9781450342322. DOI: 10.1145/2939672.2939785. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/2939672.2939785.

[25] Jovan Blanuša, Kubilay Atasu, and Paolo Ienne. “Fast Parallel Algorithms for Enumeration of Simple, Temporal, and Hop-constrained Cycles”. In: *ACM Trans. Parallel Comput.* 10.3 (Sept. 2023). ISSN: 2329-4949. DOI: 10.1145/3611642. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/3611642.

[26] Tanveer Ahmad. “Benchmarking Apache Arrow Flight - A wire-speed protocol for data transfer, querying and microservices”. In: *Benchmarking in the Data Center: Expanding to the Cloud*. BID’22. Seoul, Republic of Korea: Association for Computing Machinery, 2022. ISBN: 9781450393249. DOI: 10.1145/3527199.3527264. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/3527199.3527264.

[27] IBM Redbooks. *OpenCAPI Memory Interface Facility*. IBM Redpaper REDP-5494. International Technical Support Organization, IBM, 2019. URL: https://www.redbooks.ibm.com/redpapers/pdfs/redp5494.pdf (visited on 06/08/2026).

A

Initializing Power9 servers

The ThymesisFlow setup by the Hasso Plattner Institute consists of two Power9 IC922 servers. In the setup the borrower and the lender node are names *fp03* and *fp04*, respectively. The setup starts with initializing the disaggregated memory from node *fp04* using the following command:

```
# Create the memory mapped file with size in 1MB blocks
sudo /opt/thymesisflow/init_shmem_file.sh 32768

# Check to see if shared memory file is created correctly
ls -l /dev/mishmem-s1

# Attach the disaggregated memory to the shared memory-mapped file
/opt/thymesisflow/bin/thymesisf-cli \
    attach-memory \
    --afu IBM,RMEM \
    --cid 1 \
    --size 34359738368 \
    --port 2
```

Then node *fp03* can be configured by the following commands:

```
# Load the custom kernel module for mapping remote memory to file
sudo /opt/mishmem/insmod-mishmem-s1.sh

# Check to see if kernel is loaded
ls -l /dev/mishmem-s1

# Attach the disaggregated (remote) to the shared memory-mapped file
/opt/thymesisflow/bin/thymesisf-cli \
    attach-compute \
    --afu IBM,RMEM \
    --cid 1 \
    --size 34359738368 \
    --port 2 \
    --ea 0x100000000 \
    --no-hotplug
```

B

Proto Buffers

B.1. Graph manager cache retrieval

In the dynamic graph manager, gRPC messages with protocol buffers are implemented for retrieving data upon cache misses. The protocol buffer definitions for these gRPC calls can be found below.

```
// Copyright 2026 Keyvan Khalili.
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

syntax = "proto3";

package tf_grpc_cache;

// The greeting service definition.
service CacheServer {
  rpc GetFast (CacheRequestFast) returns (CacheReplyFast) {};
  rpc Get (CacheRequest) returns (CacheReply) {};
}

// Requests outgoing vertices of a vertex from an Arrow table
message CacheRequestFast {
  int32 table_index = 1;
  uint64 vertex = 2;
  bool forward_graph = 3;
}

// Responds with a list of outgoing vertices
message CacheReplyFast {
  repeated uint64 vertex = 1;
}

// Requests complete adjacancy list, i.e., outgoing vertices & timestamps
message CacheRequest {
```

```
  int32 table_index = 1;
  uint64 vertex = 2;
  bool forward_graph = 3;
}

// Responds with two lists (outgiong vertices & timestamps)
message CacheReply {
  message AdjacancyList {
    uint64 vertex = 1;
    repeated int64 timestamp = 2;
  }
  repeated AdjacancyList adjacancy_list = 1;
}
```

B.2. Inter-node synchronization & communication

Here the defined protocol buffers for synchronizing nodes and relaying allocator calls to the owning node can be seen.

```
// Copyright 2023 Philip Groet
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

syntax = "proto3";

package grpc_tfo;

service OrchestratorServer {
  rpc Malloc(MallocRequest) returns (MallocReply) {};
  rpc Reallocate(ReallocateRequest) returns (ReallocateReply) {};
  rpc Free(FreeRequest) returns (FreeReply) {};
  rpc FlushBlocks(FlushRequest) returns (FlushReply) {};

  rpc Unlock(UnlockRequest) returns (UnlockReply) {};
  rpc SendTable(TableChunk) returns (TableChunkReply) {};
  rpc SendRecordBatchMD_v1(RecordBatchMD_v1) returns (SendRecordBatchMDReply_v1);
  rpc SendRecordBatchMD_v2(RecordBatchMD_v2) returns (SendRecordBatchMDReply_v2);

  rpc ClearAllHugeTables(PingMessage) returns (PingReply);

  rpc SendState(ApplicationState) returns (StateReply) {};

  rpc Ping(PingMessage) returns (PingReply) {};
}

message MallocRequest {
  uint32 size = 1;
}
message MallocReply {
  uint64 address = 1;
}
```

```
43
44 message ReallocateRequest {
45   uint32 old_size = 1;
46   uint32 new_size = 2;
47   uint64 address = 3;
48 }
49 message ReallocateReply {
50   uint64 success = 1;
51   uint64 address = 2;
52 }
53
54 message FreeRequest {
55   uint64 address = 1;
56   /// @param size Allocated size located at buffer. An allocator implementation
57   ///   may use this for tracking the amount of allocated bytes as well as for
58   ///   faster deallocation if supported by its backend.
59   uint64 size = 2;
60 }
61 message FreeReply {
62   uint64 success = 1;
63 }
64
65 message FlushRequest {
66   message Block {
67     uint64 pointer = 1;
68     uint64 size = 2;
69   }
70
71   repeated Block block = 1;
72 }
73 message FlushReply {
74   bool success = 1;
75 }
76
77 message UnlockRequest {
78   uint64 address = 1;
79 }
80
81 message UnlockReply {
82   bool success = 1;
83 }
84
85 message TableChunk {
86   // This indicates the location of the Table in the bigger dataset.
87   uint32 start_index = 1;
88
89   bytes ipc_table_payload = 2;
90 }
91
92 message TableChunkReply {
93   bool success = 1;
94 }
95
96 message RecordBatchMD_v1 {
97   message Buffer {
98     uint32 size = 1;
99     uint32 capacity = 2;
100    uint64 pointer = 3;
101  }
102
103  repeated Buffer buffer = 1;
```

```
104   uint32 num_rows = 2;
105   uint32 start_index = 3;
106 }
107 message SendRecordBatchMDReply_v1 {
108   bool success = 1;
109 }
110
111 message RecordBatchMD_v2 {
112   message Field {
113     message Buffer {
114       // uint32 capacity = 1; // As we will not be writing from remote, this is
              not necessary for the other party to know
115       int64 size = 2;
116       uint64 pointer = 3;
117       uint32 level = 4;
118     }
119
120     bytes name = 1;
121     uint64 type = 2; // static_cast<Type>(data[0])
122     int64 length = 3;
123     int64 null_count = 4;
124     repeated Buffer buffer = 5;
125   }
126
127   uint64 rows = 1;
128   uint64 start_index = 2;
129   repeated Field field = 3;
130 }
131 message SendRecordBatchMDReply_v2 {
132   bool success = 1;
133 }
134
135 message ApplicationState {
136   uint32 node_id = 1;
137   uint32 flag = 2;
138 }
139
140 message StateReply {
141   bool success = 1;
142 }
143
144 message PingMessage {
145
146 }
147
148 message PingReply {
149   bool success = 1;
150 }
```