Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Masterarbeit

# Generalized and Optimized Memory Management with LLVM


vorgelegt von

Niclas Schroeter

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

| | |
|---|---|
| Studiengang: | Informatik |
| Matrikelnummer: | 6926553 |
| Erstgutachter: | Prof. Dr. Thomas Ludwig |
| Zweitgutachter: | Jannek Squar |
| Betreuer: | Jannek Squar |


Hamburg, 2023-12-04

# Abstract

Utilizing modern multiprocessors effectively is an important aspect of scientific applications. Thus, many such applications rely on OpenMP as a simple option for parallelization. One of the main downsides of OpenMP is its limitation to shared memory. Tools, such as CATO, aim to transform OpenMP applications to use MPI functions instead, thereby improving the scalability of applications that were formerly bound to a single compute node. The purpose of this thesis is to adapt and improve CATO in two major capacities: the first improvement is aimed at CATO's general memory management, enabling the transformation of applications that make use of arrays of arbitrary dimension. This is achieved by means of a thorough analysis of array indexing in LLVM IR and subsequent changes to the compiler pass component and the runtime library of CATO. The second improvement aims to optimize the memory management at runtime. To this end, multiple cache models are proposed and implemented, including one that conforms to the OpenMP memory model. Evaluations on multiple OpenMP kernels suggest that the optimized memory management at runtime can reduce the time to solution by more than 50% in many cases, at the cost of an increase in memory consumption.

# Contents

# 1. Introduction

Ever since it became apparent that single-core chips would not be able to sustain the growth in performance that they have shown in the previous century, due to physical limitations regarding power usage and heat generation, manufacturers started building chips with multiple cores to increase performance instead [Geer, 2005]. Alongside these multiple cores, the chips featured shared memory as well. In order to use these chips efficiently, the software running on them needs to make use of parallellization. In High Performance Computing (HPC), one of the most widely-spread parallelization techniques for shared-memory architectures is OpenMP [OpenMP, 2021]. It is a collection of compiler directives, select library routines and environment variables, available for C, C++ and Fortran. Since parallelism is introduced mainly via the compiler directives, a user is able to incrementally parallelize a given application with few small changes to the existing source code, with a low barrier to entry as well. These features, along the ease of use, make OpenMP a popular choice in HPC, particularly among scientific developers without a background in computer science.

Even though single compute nodes in current HPC systems typically feature multiple hundreds of gigabytes in main memory, the amount of memory required for current large-scale problems can extend to the range of multiple terabytes. To meet these requirements, applications need to scale across multiple nodes with distributed memory. However, as OpenMP is designed for usage with shared-memory systems and, since version 4.0, with accelerators, a different programming paradigm is required. The de-facto standard for parallellization in distributed-memory environments is MPI [MPI, 2015]. Parallelization with MPI is based on passing messages between multiple, concurrently executing processes. In order to utilize MPI in any existing application, it is necessary to fundamentally transform the underlying code, without any option for incremental parallelization. While this is doable for trained personnel, it is still a time-consuming endeavour. Since scientific developers without a background in computer science do not necessarily have these skills, performing such a transformation of their applications on their own is seldomly a viable option.

To facilitate the transformation of OpenMP codes to use MPI instead, different approaches have been investigated, for example the creation of tools for the automatic transformation of OpenMP-based code. Most of these tools take effect anywhere between compile-time and runtime, showing promising results in the process. One such tool is CATO (**C**ompiler **A**ssisted Source **T**ransformation of **O**penMP kernels) [Squar et al., 2020]. This tool aims to transform the OpenMP code at compile time via LLVM [Lattner and Adve, 2004], providing a runtime library that is subsequently

used by the transformed code as well. By using this tool, any developer could take existing OpenMP code and transform it into working MPI code, enabling a substantial increase in the potential problem sizes that can be tackled with the given application due to the ability to execute on multiple nodes, instead of being constrained to a single one.

In its current state, CATO is able to correctly transform OpenMP codes that utilize the `parallel`, `for` and `critical` directives, alongside reductions. While this is only a fraction of the available directives, an analysis of GitHub repositories suggests that these directives cover more than 50% of all OpenMP usage [Squar et al., 2022]. While an extension of CATO's ability to process more directives is desirable, the tool currently suffers from other, more pressing limitations. Presently, the dimensionality of arrays that can be used inside of kernels is limited to three due to shortcomings in the internal memory management, both during the transformation of the original kernel and within the subsequently used runtime library. This leads to the exclusion of many computational kernels. Furthermore, the transformed application suffers from a substantial increase of the time to solution. While the main goal of CATO is to enable an increase of the problem size that a given kernel can work with, a significant decrease in performance hinders the tool's applicability.

The goals of this thesis are to investigate these issues regarding memory management and performance, proposing solutions to both of them. In particular, this thesis aims to do the following:

1. Introduce generalized memory management into CATO, removing the limitations regarding the array dimensionality.

2. Analyze and improve the performance of CATO via the introduction of multiple enhancements to the source code, among which is an OpenMP memory model conforming software cache to minimize MPI communication calls.

The first goal is achieved through a thorough analysis of how array indexing functions in the context of LLVM IR and subsequent adjustments of CATO's source code. The second goal is achieved by analyzing existing bottlenecks in CATO's runtime library and tailoring specific solutions for the identified bottlenecks. All the changes are documented and extensively benchmarked to demonstrate their effect on the performance of multiple different computational kernels.

The rest of the thesis is structured as follows: Chapter 2 investigates related work regarding concepts and tools for the automatic distribution across nodes of applications written with shared-memory models in mind. Chapter 3 provides an overview of the utilized technologies, including a brief description of CATO. The generalization of CATO's memory management to enable processing of arrays of arbitrary dimensions is discussed in Chapter 4. The identification of bottlenecks at runtime and the subsequent adjustments to CATO are highlighted in Chapter 5. Chapter 6 provides an evaluation of these changes on multiple computational kernels, conducted on modern HPC hardware.

Future work and solutions for further enhancements of the changes proposed in this thesis are discussed in Chapter 7, after which the thesis is concluded in Chapter 8. The code for CATO, alongside the changes discussed in this thesis, is available on GitHub[1].

[1] `https://github.com/JSquar/cato/tree/memory-extension`

# 2. Related Work

In this section, different approaches to enable scaling of shared-memory code across multiple nodes and the corresponding research will be discussed. Presently, there are three options that have been investigated in the past or are still being investigated today:

- Rewrite with MPI or something of a similar purpose
- Distributed Shared Memory (Section 2.1)
- Tool-supported automatic transformation (Section 2.2), which includes CATO

Assuming that the shared-memory kernel is of a considerable size and that the user may not be intimately familiar with the likes of MPI, the first option is not feasible. The other two options are discussed in the following sections.

## 2.1. Distributed Shared Memory

The concept of Distributed Shared Memory (DSM) has been a topic in research since the 1980's. DSM solves the problem of scaling across multiple nodes by abstracting the distribution of memory into a virtual shared memory. This would eliminate the need to transform the existing shared-memory applications, as the entirety of the distributed memory would appear and be treated as one large shared memory. DSM approaches range from hardware, over the OS up to the software level.

Notable DSM systems at hardware level include the DASH system [Lenoski et al., 1992] and the Stanford Paradigm [Cheriton et al., 1991]. These systems were built in the 1990's, with the goal of creating a scalable multiprocessor architecture with shared memory and cache coherence. These approaches have been abandoned nowadays, as current HPC systems evidently do not feature one single address space across all nodes. Though a similar design can still be found within the compute nodes in the form of the NUMA architecture, in which each core is assigned a portion of the local memory, while still treating the entirety of the memory as shared on all levels above the hardware.

At the level of operating systems, DSM is also known under the term 'Single System Image' (SSI). There exist multiple definitions regarding SSIs, with one of the most common ones defining Single System Images as "the property of a system that hides the heterogeneous and distributed nature of the available resources and presents them to users and applications as a single unified computing resource" [Buyya et al., 2001]. This definition

covers the hardware and user levels as well, though the focus will be on the OS level for now. These operating systems have been extensively researched from the 1980's onwards, starting with Locus [Walker et al., 1983] and Plan 9 [Pike et al., 1995]. Until the early 2000's, many different SSIs were in active development, such as Kerrighed [Morin et al., 2003], MOSIX [Barak and La'adan, 1998] or Solaris MC [Khalidi et al., 1996]. Today however, development of SSIs has ceased, with very few exceptions, such as Popcorn Linux [Barbalace et al., 2014]. This development can be attributed to the lackluster adoption of kernel-level SSIs, which is based on numerous factors that are outlined more extensively in [Healy et al., 2016].

At the third level exist the software DSM systems (SDSM). While not as efficient as direct hardware support, it is the only level at which DSM is still used or researched, apart from the NUMA architecture. The most well-known example for this is the Partitioned Global Address Space (PGAS) model [Almasi, 2011]. Programming languages and libraries that support PGAS present the distributed memory of multiple nodes as one global shared memory address space. Example languages that support the PGAS model include Coarray Fortran and Unified Parallel C, example libraries include the different implementations of SHMEM.

Apart from programming languages and libraries, SDSM also exists in the form of pure runtime systems, such as cJVM [Aridor et al., 1999] from 1999, though development on it has been stopped nowadays. Similarly, Intel worked on Cluster OpenMP for a while, which itself was based on the DSM system TreadMarks [Amza et al., 1996]. The goal was to provide DSM-based OpenMP [Hoeflinger, 2006], with a resulting workflow that is similar to CATO's. The application developer would have to annotate the sequential code with the usual compiler directives, while the Cluster OpenMP compiler and runtime library extend it to run across multiple nodes. However, once again development has ceased many years ago.

Others have focused their efforts on the development of directive-based language extensions, such as XcalableMP [XcalableMP, 2018], which also incorporates certain PGAS features. The development of XcalableMP has not progressed for a few years though, which leaves PGAS as the only usable option for software-based DSM. However, since using PGAS to enable the scaling of an OpenMP application to distributed memory would require an extensive manual transformation, it is not a useful option for the previously mentioned use case, for the same reasons that the manual transformation into MPI code is not feasible either.

## 2.2. Tool-supported Transformation

The tool-supported transformation of OpenMP codes to scale to distributed memory can be regarded as a special case of SDSM. Since this topic has been the target of many research efforts in the past decades, multiple vastly different approaches and tools

have emerged. There are many examples based on source-to-source translation using different compiler frameworks. [Basumallik and Eigenmann, 2005] made use of the Cetus compiler infrastructure and introduced partial replication of data, circumventing the concept of data ownership. [Saà-Garriga et al., 2015] used the Mercurium framework to directly manipulate the Abstract Syntax Tree of the input source code, making use of a similar producer/consumer pattern as the previously mentioned work. The main difference to CATO is in the use of the compiler infrastructure. LLVM provides a very extensive, modular compiler infrastructure and a low-level IR. As has been pointed out by others [Rodríguez et al., 2013], the hardware proximity of the IR and the efficient implementation of the LLVM toolchain itself make it a good choice for production environments. Furthermore, the previously mentioned approaches rely on point-to-point communication instead of employing remote-memory access (RMA). Especially when considering modern interconnect hardware, the usage of RMA for communication, as it is done in CATO, is expected to improve the overall performance.

There have been approaches that combine source-to-source translation and the usage of runtime SDSM systems as well. One such example is the Omni OpenMP compiler [Sato et al., 1999]. Another approach relied on a modified version of TreadMarks as its SDSM runtime [Hu et al., 2000]. Both of the approaches mentioned transformed the input OpenMP application to make use of their respective SDSM runtimes by replacing the usual OpenMP calls with calls to the custom runtime library. However, once again neither of the approaches is still under active development. These two approaches are similar to CATO in respect to the general process of translation in combination with the insertion of calls to a runtime library, differing mainly in the details of the utilized compiler infrastructures and the nature of the runtime libraries, as none of them make direct use of MPI.

The feasibility of using LLVM's IR for source-to-source translation or similar tasks has been demonstrated in the past by multiple different works. [Hamidouche et al., 2011] have built a framework for the creation of hybrid programs, using a sequential input source, as well as some additional, user-provided information about the utilized algorithms. The IR of the sequential input was used to gather information about the estimated runtime of a function by having an LLVM pass count the number of necessary cycles of said function. This information is then used alongside other profiling results to estimate a good configuration of processes and threads for the final, parallel version of the application. The overarching goal of the framework is comparable to CATO's. In order to obtain the final message-passing version of an application, they make use of a modified version of BSML [Gesbert and Gava, 2009] to transform the input, as opposed to modifying the LLVM IR, as CATO does it. By only utilizing the features built-in to LLVM and not introducing another dependency to (potentially unmaintained) software, CATO comparatively lowers the barrier of entry, which is also further amplified by not requiring an XML file from the user. The authors of DiscoPoP [Norouzi et al., 2019] generate suggestions for sequential programs regarding their potential parallelization with OpenMP. To achieve this, they make use of an LLVM pass to identify dependencies between variables, combined with other profiling techniques. However, they do not act upon these suggestions

automatically and leave the final choice to the user. [Jammer and Bischof, 2021] employ LLVM in a similar manner as CATO does, albeit with different goals in mind. In the IR, they replace blocking MPI communication of buffers that are modified in OpenMP sections with the newly-introduced partitioned communication of MPI-4 [MPI, 2015]. To this end, they identify the relevant sections in the IR of the input source and adjust the communication by replacing previous communication calls and inserting the required functions for the partitioned communication scheme.

## 2.3. Optimization of Memory and Communication

In order to optimize the memory accesses and the accompanying communication within CATO, different software caches will be developed to facilitate faster accesses. The idea of utilizing software caches with DSM systems has already been partly explored in the previous century. One component of the software DSM system Munin [Bennett et al., 1990b] was an adaptive software cache coherence mechanism, which applied different techniques based on the expected access behaviour of particular data [Bennett et al., 1990a]. One of the mechanisms is a delayed update, which delays the store of modified data as long as possible, grouping them as well to reduce the number of necessary network packets. A similar mechanism will be implemented in CATO. They have also provided a classification of the different access patterns in the same work. In order to make use of these mechanisms, users had to develop shared memory programs with Presto [Bershad et al., 1988], a shared memory parallel programming environment for C++. It should be noted that neither Munin nor Presto are still used nowadays, though the aforementioned classification of the access patterns should still be relevant.

In both [Chen et al., 2009] and [Chen et al., 2010], the authors have designed some software cache implementations that are consistent with the OpenMP memory model. This consistency is a major concern for one of the caches that will be introduced to CATO. Due to the age of the aforementioned work and the ongoing refinement of the OpenMP standard, the assumptions underlying the cache implementations are partly outdated, as they are based on the now 15-year-old OpenMP version 3.0 [OpenMP, 2008]. The memory model has changed significantly until today, with the current version of the standard being 5.2 [OpenMP, 2021]. Some important changes were made to the OpenMP memory model in version 5.0 of the standard, including specializations of the flush operation into `strong`, `release` and `acquire` flushes. Furthermore, the targeted hardware, being the Cell Broadband Engine, was considerably less powerful than HPC hardware, which led to further limitations of the cache implementations, for example in their memory usage. Given that CATO targets modern-day HPC hardware, such limitations are less severe and can partly be ignored.

# 3. Utilized Technologies

CATO aims to transform OpenMP kernels to instead utilize MPI, thus enabling the transformed application to scale beyond the limitations of a single compute node. This is achieved through the utilization of LLVM, in particular by virtue of a compiler pass that replaces certain functions in the original code, for example accesses to previously shared memory, with custom functions. Said custom functions rely heavily on the one-sided communication functions of the MPI standard. The following sections of this chapter highlight these different technologies, in particular those aspects that interact directly with CATO.

## 3.1. OpenMP

OpenMP is a collection of compiler directives, select library routines and environment variables for the introduction of parallelism to C, C++ and Fortran applications. The user can choose to either parallelize a program at thread-level, offload certain portions to accelerators, like GPUs, or even combine the two. As the de-facto standard for shared-memory parallelism in HPC, OpenMP is supported by most compilers available, including gcc, clang and the Intel compilers. The execution model in use is a fork-join model. An OpenMP programs executes with a single thread at the start, running sequentially through the program until it reaches a parallel region for example. At this point, new threads can be created that will then concurrently execute the code in said region until the end of it, where an implicit barrier is encountered and the threads are joined again, only leaving the primary thread to continue. There are numerous directives to precisely control the parallelism in a given application, but for the purposes of this thesis, the main focus will be on the `parallel`, the `for` and the `critical` directives.

```
1 #pragma omp parallel
2 {
3     printf("My threadnumber is %d\n", omp_get_thread_num());
4 }
```

Listing 3.1: Example usage of the `parallel` directive.

Listing 3.1 provides a short example of how OpenMP, and the `parallel` directive in particular, can be used. In C and C++, all compiler directives start with `#pragma omp`, followed by the name of the directive. For the rest of this thesis, all code examples will feature C code, unless stated otherwise. In this example, each thread will execute the code enclosed in the braces, which in this case will print the thread ID of each thread,

as queried via one of the OpenMP library routines. The number of threads can be controlled through environment variables, in the directives themselves or via different library routines.

The `for` directive (`do` in Fortran) is one of the most used worksharing constructs. Decorating a loop with this directive leads to the distribution of the loop iterations across the team of threads. The assignment of these chunks to the threads can be controlled via the `schedule` clause. The different options for scheduling include `static`, `dynamic` and `guided`. With `static` scheduling, the iterations are divided into evenly-sized chunks and assigned to the threads in a round-robin manner. With a `dynamic` schedule, the loop iterations are also separated into evenly-sized chunks, but instead of being assigned to a thread immediately, each thread will process a chunk and then request the next one, until all chunks have been processed in this manner. The `guided` schedule behaves similarly to the `dynamic` one, but with varying chunk sizes. The chunk size is chosen proportionally to the number of unassigned iterations divided by the amount of threads. There are two further options for scheduling, namely `auto` and `runtime`, which respectively incur an automatic choice or a deferral of said choice until runtime.

```
1 int arr[50];
2 #pragma omp parallel for shared(arr) schedule(dynamic)
3 for (int i = 0; i < 50; i++)
4     arr[i] = i;
```

Listing 3.2: Example usage of the `for` directive.

Listing 3.2 provides an example of the `for` directive. As is commonly the case, this example also directly combines it with the `parallel` directive to create the threads at entry to the loop and join them again afterwards. It furthermore highlights the aspect of data sharing across threads. The visibility of variables is subject to the choice of the user. The `parallel` directive, among others, provides the clauses `shared` and `private` to control the visibility. By declaring the array `arr` as a shared variable in Listing 3.2, each thread accesses the same array in the loop. The loop index however is implicitly set to `private`, which leads to every thread creating a private version of it. Variables that are declared within parallel regions are also treated as being `private`.

The last relevant directive for this thesis is the `critical` construct. Regions enclosed by the `critical` directive inside of a parallel region are only ever executed by a single thread at a time. An example is provided in Listing 3.3. This directive facilitates synchronization of the thread team, mutual exclusion and helps with avoiding race conditions. Other directives for synchronization include the `barrier` and the `flush` directives, the latter being highlighted further during the inspection of the OpenMP memory model in Section 5.2.1.

```
#pragma omp parallel
{
    #pragma omp critical
    {
            printf("Now in critical region: %d\n",
               ↪omp_get_thread_num());
            important_var = omp_get_thread_num();
            printf("Now exiting the region\n");
    }
}
```

Listing 3.3: Example usage of the `critical` directive.

## 3.2. MPI

MPI (Message Passing Interface) is the de-facto standard for distributed memory parallelism. MPI itself is a library interface specification, addressing the message-passing parallel programming model. This model is based on having multiple processes, implying separate address spaces, compute independently of each other in a MIMD fashion, but allowing cooperation and communication through passing messages. MPI defines different operations for communication, including point-to-point communication, collectives, one-sided communication via RMA and parallel I/O. The first version of MPI was published in 1994. The standard has been adapted over time with various adjustments and additions, culminating in the most recent version, version 4.0, which was released in 2021. Since MPI is only an interface specification, many different implementations exist nowadays, for example MPICH, OpenMPI or Intel MPI.

The most common MPI operations are the ones used for point-to-point communication. The basic versions of these operations are `MPI_Send` and `MPI_Recv`, for sending and receiving messages respectively. An example of their potential use is provided in Listing 3.4.

```
if (rank == 0)
{
    MPI_Send(sendbuf, 10/*elem count*/, MPI_INT, 1/*target
       ↪rank*/, 42/*tag*/, MPI_COMM_WORLD);
}
else if (rank == 1)
{
    MPI_Recv(recvbuf, 10, MPI_INT, 0, 42, MPI_COMM_WORLD,
       ↪MPI_STATUS_IGNORE);
}
```

Listing 3.4: Example point-to-point communication with MPI.

Processes in MPI are organized in process groups called communicators. The predefined communicator `MPI_COMM_WORLD` contains all accessible processes, enabling their communication through this communicator. Within the process groups, each process has a unique identifier, namely their rank, which is a value in the interval $[0, n-1]$ with $n$ processes in the group in total. In the example, the process with `rank=0` attempts to send a buffer of 10 integers, stored in `sendbuf`, to the process with `rank=1`. The second-to-last argument in the send call is a message tag to allow the receiving process to distinguish between multiple messages. Note that `MPI_Send` is a blocking call, meaning that it will only return once the send buffer can be safely modified again. This can mean that either the message has been received by the intended target, or the message has been buffered, leading to completion without requiring a matching receive call. There are additional send calls, namely `MPI_Bsend`, `MPI_Ssend` and `MPI_Rsend`, which provide more control over the conditions that need to be met for the send call to be initiated and consequently returned, but their exact functionality is not relevant for the rest of this thesis. The receive call posted by `rank=1` is also blocking, meaning that it will only return once the message is available in the receive buffer. There is an extra argument in the receive call, when compared to the send call. The last argument, a status object, contains information on the status of the receive call, including error codes and length of the received message, among others. It is possible to pass `MPI_STATUS_IGNORE` to discard this information, as seen in the example.

There are also nonblocking variants of the point-to-point communication calls in the form of `MPI_Isend` and `MPI_Irecv`. These calls initiate the communication, but do not complete it. Instead, they require separate calls to functions such as `MPI_Wait` to complete them. In the case of the send call, completion means that the buffer can be safely reused, whereas for the receive call it means the data has been received in the corresponding buffer. Using the nonblocking variants often improves performance, as this allows for an efficient overlap of computation and communication.

Depending on the use case, one of the major downsides of point-to-point communication is the need to provide matching send and receive calls for multiple processes. This can be circumvented by utilizing the one-sided communication operations, also called RMA operations, such as `MPI_Put` or `MPI_Get`. In CATO, these are the operations that are used for inter-process communication. As the name implies, these operations only require one process to specify the necessary parameters for communication, while the actual target of the operation is not required to actively participate in the communication. The functions have been designed in such a way as to facilitate the efficient usage of modern hardware, such as DMA engines.

In order for memory to be accessible through RMA operations, a process has to expose it in the form of a window. There are different options to create such a window, the example in Listing 3.5 displays the `MPI_Win_create` function.

```
1 MPI_Win win;
2 size_t size = 50 * sizeof(int);
3 int* buffer = malloc(size);
4 MPI_Win_create(buffer, size, sizeof(int)/*disp*/,
    ↪MPI_INFO_NULL, MPI_COMM_WORLD, &win);
5 ...
6 MPI_Win_free(&win);
7 free(buffer);
```

Listing 3.5: Creating and destroying an MPI window.

When using `MPI_Win_create`, a previously allocated memory segment is exposed to other ranks through the associated window object (the last parameter). Each window has a size and a displacement unit, which is used to denote the size of a single element. Once the window is not required anymore, it needs to be destroyed via `MPI_Win_free`.

Once a memory segment is exposed via a window, it can be accessed by other processes through various functions, such as `MPI_Put` or `MPI_Get` or the more complex accumulate functions. However, these calls cannot happen at any point in time. Accessing a window via the MPI communication calls can only happen in an access epoch for the given window. These epochs are controlled through the synchronization calls. Currently, MPI provides three different mechanisms for synchronization.

The first mechanism is active target synchronization through fences. With active target synchronization, windows can be accessed only in so-called exposure epochs. The function `MPI_Win_fence` is a collective synchronization call that starts or ends an exposure epoch for the given window across all processes that are associated with the window. The fence calls are supposed to be used in pairs, with the first call starting the epoch and second one ending it. The trailing fence calls act as a barrier, returning only once all processes have entered, while simultaneously guaranteeing the completion of all RMA operations that have been issued during the exposure epoch. An example is provided in Listing 3.6. The fences are used to mark the start and end of the exposure epoch, synchronizing the processes as well. During this, each process puts one element, stored in `var`, at a displacement corresponding to their own rank into the memory of `rank=0`. While all the parameters for the actual communication are only provided by the process at which the communication call originates, all processes are still required to participate in the collective synchronization call. Note that there is no limit on the number of communication calls that can occur within one epoch.

```
1 int var = 42;
2 MPI_Win_fence(0, win);
3 MPI_Put(&var, 1, MPI_INT, 0/*target rank*/, myrank/*target
    ↪disp*/, 1, MPI_INT, win);
4 MPI_Win_fence(0, win);
```

Listing 3.6: Active target synchronization with fences.

The second mechanism is general active target synchronization. Instead of requiring a global synchronization call, as was the case with fences, general active target synchronization only requires the origin and the target processes to participate in the synchronization. This reduces the required synchronization for use cases where not every process in a communicator needs to be in an exposure and/or access epoch at the same time, which has potential benefits regarding the performance. The four functions that are used in this mechanism are `MPI_Win_post`, `MPI_Win_start`, `MPI_Win_complete` and `MPI_Win_wait`. Their usage is exemplified in Listing 3.7.

```
1  if (rank == 0)
2  {
3      MPI_Win_start(group, 0, win);
4      int var = 42;
5      MPI_Put(&var, 1, MPI_INT, 1, disp, 1, MPI_INT, win);
6      MPI_Win_complete(win);
7  }
8  else if (rank == 1)
9  {
10     MPI_Win_post(group, 0, win);
11     MPI_Win_wait(win);
12 }
```

Listing 3.7: General active target synchronization.

The target process of the operation, in this case `rank=1`, starts an exposure epoch by calling `MPI_Win_post`. This exposes the window to all processes that are specified in the first parameter `group`. The origin process, `rank=0`, starts an access epoch by calling `MPI_Win_start`, specifying all processes that the origin requires access to in the `group` argument. Note that each process specified in the `group` argument of the origin process is required to call `MPI_Win_post` and vice-versa. After that the origin process can access the window of the target. Once all access functions have been called, the function `MPI_Win_complete` is used to close the access epoch at the origin process. This will block until all outstanding RMA operations of this access epoch have completed at the origin, but not necessarily at the target. In order for that to occur, the target process has to call `MPI_Win_wait`, which will block until all matching calls to `MPI_Win_complete` have occurred and the RMA operations have completed at the target.

The third mechanism is passive target synchronization through locks. Here, the target process does not need to take part in the synchronization anymore. Only the origin process needs to lock the target window, issue the RMA calls and then unlock the window. The corresponding functions are `MPI_Win_lock` and `MPI_Win_unlock`.

```
1 if (rank == 0)
2 {
3     int var = 42;
4     MPI_Win_lock(MPI_LOCK_EXCLUSIVE, 1, 0, win);
5     MPI_Put(&var, 1, MPI_INT, 1, disp, 1, MPI_INT, win);
6     MPI_Win_unlock(1, win);
7 }
```

Listing 3.8: Passive synchronization with locks.

The example in Listing 3.8 showcases these functions. The origin process starts the access epoch by calling `MPI_Win_lock`. There are two types of locks, `MPI_LOCK_EXCLUSIVE` and `MPI_LOCK_SHARED`. The exclusive lock protects the window from concurrent accesses from other processes that also use locks. The shared lock ensures that there are no concurrent accesses from other processes that want to exclusively lock the same window. The access epoch is ended by calling `MPI_Win_unlock`, which will only return once the outstanding RMA operations have been completed at both the target and the origin. There are also flush operations that can complete outstanding RMA operations without ending the access epoch.

## 3.3. LLVM

The LLVM project is a "collection of modular and reusable compiler and toolchain technologies" [LLVM, 2023]. This infrastructure entails many different components, but for this thesis, the focus will lie on the core libraries and the C compiler clang, alongside its OpenMP runtime. The LLVM version that is used for this thesis is 13.0.0. Since LLVM is constantly changing, some of the information provided in this thesis might not apply to later LLVM versions.

### 3.3.1. The LLVM Core

While the compilation with LLVM, as displayed in Figure 3.1, is based on the popular three-phase design used in many compilers, there are some very important differences [Lattner, 2011]. Traditionally, the three phases are represented by three components: a frontend for parsing the source code, an optimizer for improving the representation from the previous component in regards to performance, and the backend, which is responsible for the translation to the target instruction set. Looking at the frontend, the usual output, which in turn would serve as the input for the optimizer, would be a language-specific Abstract Syntax Tree (AST). This is one of the notable differences with

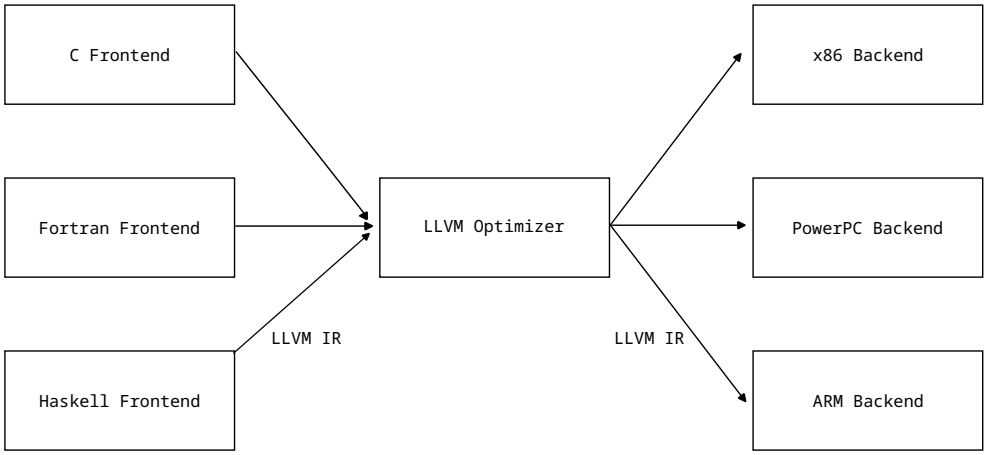


Figure 3.1.: The three-phase design in LLVM, adopted from [Lattner, 2011]

LLVM. The optimizer does not require a language-specific AST as input, but rather an intermediate representation known as the LLVM IR [Lattner and Adve, 2004]. The LLVM IR is a language-independent, typed, RISC-like instruction set, utilizing an infinite set of virtual registers in Single Static Assignment form, or SSA for short [Cytron et al., 1991]. There are three isomorphic forms: the plain-text form that is used in the examples, a binary representation and an in-memory representation that is used by the compiler. In order for the registers to attain SSA form, they must only be assigned to exactly once before they can subsequently be used. Note that memory locations in LLVM are exempt from the SSA form. LLVM is considered a load/store architecture, meaning that values are transferred exclusively through the use of the `load` and `store` instructions on typed pointers. Listing 3.9 demonstrates these properties.

```
1  entry:
2    %retval = alloca i32, align 4
3    %a = alloca i32, align 4
4    %b = alloca i32, align 4
5    store i32 0, i32* %retval, align 4
6    store i32 0, i32* %a, align 4
7    %0 = load i32, i32* %a, align 4
8    %tobool = icmp ne i32 %0, 0
9    br i1 %tobool, label %if.then, label %if.else
10
11 if.then:
12   store i32 1, i32* %b, align 4
13   br label %if.end
14
15 if.else:
16   store i32 99, i32* %b, align 4
17   br label %if.end
```

```
18
19 if.end:
20   %1 = load i32, i32* %b, align 4
21   store i32 %1, i32* %a, align 4
22   %2 = load i32, i32* %retval, align 4
23   ret i32 %2
```

Listing 3.9: Example LLVM IR, as produced by clang with Listing 3.10 as input.

```
1 int a = 0, b;
2
3 if (a) b = 1;
4 else b = 99;
5
6 a = b;
```

Listing 3.10: Reference C code.

The following description of the IR is based on [Lattner and Adve, 2004] and the LLVM language reference [LLVM, 2021]. In LLVM IR, functions consist of a set of basic blocks, which in turn consist of a list of instructions. Each basic block has a label (i.e. `entry`) and ends with a terminator instruction, such as the branch instruction `br`. Together, they form the Control Flow Graph of the function.

On the left-hand side of assignments, no register is used more than once, as is required by the SSA form. The registers, as well as all other local identifiers, can be identified via the leading `%` sign. In contrast, global identifiers (omitted in this example) carry a leading `@` instead. In LLVM IR, memory is either allocated on the heap or on the stack. Allocations on the stack are done with the `alloca` instruction that is displayed in lines 2 to 4 in Listing 3.9, whereas heap allocations would be done through the usage of language-specific memory allocation functions, such as `malloc` for C. Note that both allocations are typed.

Both the `load` and `store` instructions take at least two parameters. Since the `store` instruction writes to already allocated memory, no assignment is done. Instead, the first parameter specifies the value, alongside the type, which is to be written to the memory that is pointed to by the second parameter. In particular, the `store` at line 6 writes a 32-bit integer, its value being 0, to the address that register `%a` points to. The `load` instruction reads from memory, which implies that the result needs to be loaded into a register. As displayed in line 7, the `load` first specifies the type to load and then the address from which to load, in this case also a 32-bit integer from the address pointed to by register `%a`. There are multiple optional parameters for both of these vital instructions, such as the `align` parameter to control the alignment, however a thorough understanding of these parameters is not required for this thesis.

Another aspect of the SSA form are the join nodes, also known as the $\phi$-function. These nodes are used to join two or more previously separated branches, and to control assignments that depend on the preceding branch. In the LLVM IR, this aspect of SSA is implemented in the form of the `phi` instruction. An example is provided in Listing 3.11.

```
1 Loop:        ; Infinite loop that counts from 0 on up...
2   %indvar = phi i32 [ 0, %LoopHeader ], [ %nextindvar,
      ↪%Loop ]
3   %nextindvar = add i32 %indvar, 1
4   br label %Loop
```

Listing 3.11: Example of the `phi` instruction (example taken from [LLVM, 2021]).

The `phi` instruction is limited to two pairs of parameters. Depending on the previous branch from which the block, in this case labeled `Loop`, is entered, the value of the `%indvar` register will either be 0, if it was entered through the `%LoopHeader` block, or it will be the value in `%nextindvar`. There are many more instructions, such as `icmp` in Listing 3.9 or `add` in Listing 3.11, however most of them are trivial and do not require further introduction. Other nontrivial instructions, such as `getelementptr`, will be discussed as necessary later on.

With the presence of the LLVM IR, the compiler can work with any input language, as long as a frontend is provided that translates the source code to LLVM IR. Furthermore, the IR serves not only as input to the optimizer, but also as its output. Due to this, each optimization pass also produces LLVM IR, which is then used by subsequent optimization passes, until the final version of the IR is passed as input to the corresponding backend for the final translation to the target architecture's instruction set.

### 3.3.2. Transforming the IR

Since the optimizer uses LLVM IR as both input and output, every optimization inbetween also operates on this IR. The transformation of the IR is one of the core aspects of the compiler infrastructure and is facilitated by the core libraries of LLVM. These C++ libraries enable a user to write their own compiler passes for subsequent use with the optimizer, which is called opt. The libraries not only allow for traversal of the IR, but also for changes to it, for example by inserting new instructions, deleting or replacing old ones. This in turn allows for the insertion of new function calls that were not in the original source code, which can then be linked against a previously unused runtime library, to name a more specific example of a transformation. As the core libraries are very extensive, further discussion of how this looks in code, as well as the design and the functionality of the libraries are going to be limited to small excerpts wherever it is deemed necessary. For a more detailed overview, consult either the documentation or the LLVM source code.

The passes that can be created as such are either analysis passes to collect information about the application or transformation passes. Once a pass has been written, it can

be registered with the optimizer and then used alongside the other passes that opt has access to and integrated into the compiling process. Examples of existing analysis passes include the creation of various Call Graphs at different granularities, while examples for existing transformation passes include various levels of Dead Code Elimination, among others. The user is also able to control the interaction between their pass and existing ones, for example by influencing when the pass is run during the optimization process.

## 3.4. CATO

CATO is a tool that transforms OpenMP kernels, replacing the OpenMP functionality with MPI functions. In doing so, the tool enables such kernels to run in environments with distributed memory, for example across multiple compute nodes instead of just a single one, which in turn increases the maximum problem size that can be computed with these kernels. CATO consists of two main components: the first component is a transformation pass, the second one is a runtime library. The transformation pass examines the LLVM IR of the input kernel, specifically the instructions related to OpenMP.

### 3.4.1. OpenMP in LLVM IR

```
1  define dso_local i32 @main() #0 {
2  entry:
3    call void (%struct.ident_t*, i32, void (i32*, i32*,
       ↪...)*, ...) @__kmpc_fork_call(%struct.ident_t* @1,
       ↪i32 0, void (i32*, i32*, ...)* bitcast (void (i32*,
       ↪i32*)* @.omp_outlined. to void (i32*, i32*, ...)*))
4    ret i32 0
5  }
6  define internal void @.omp_outlined.(i32* noalias
     ↪%.global_tid., i32* noalias %.bound_tid.) #1 {
7  entry:
8    %.global_tid..addr = alloca i32*, align 8
9    %.bound_tid..addr = alloca i32*, align 8
10   store i32* %.global_tid., i32** %.global_tid..addr, align
       ↪8
11   store i32* %.bound_tid., i32** %.bound_tid..addr, align 8
12   %call = call i32 @omp_get_thread_num()
13   %call1 = call i32 (i8*, ...) @printf(i8* getelementptr
       ↪inbounds ([23 x i8], [23 x i8]* @.str, i64 0, i64
       ↪0), i32 %call)
14   ret void
15 }
```

Listing 3.12: Excerpt of the LLVM IR for an OpenMP program.

Listing 3.12 provides an example of how the IR for an OpenMP parallel region looks like. The utilized source code is the earlier example of the OpenMP parallel pragma, displayed in Listing 3.1.

Upon entry to the main function, a call is issued to the function `__kmpc_fork_call`. In the OpenMP runtime that is part of LLVM, this call is used to create the threads. It takes at least three arguments: the first one is a struct that contains information on the source location, the second one is the number of arguments in the ellipsis, and the third one is a pointer to a function, known as a microtask in the context of LLVM's OpenMP runtime. In this particular example, the ellipsis contains no extra arguments, hence the second argument is 0. However, if the microtask required shared variables, for example if the `parallel` pragma contained a `shared` clause, then these shared variables would be passed as arguments in the ellipsis and the second argument would contain the respective amount of variables passed.

The microtask corresponds to the function that is outlined for the purposes of execution with OpenMP. The threads that are created during the fork call then execute the code that is pointed to by the microtask argument. In this particular example, the code in the parallel region only prints a string containing the thread number. This code is translated to the function `.omp_outlined.`. After storing some information regarding the global and local thread identity in the lines 8 through 12, a call is issued to a function from the OpenMP library, in this case `omp_get_thread_num`. The result is subsequently used in the following print call, before the microtask returns to the main function, joining the previously forked threads.

### 3.4.2. CATO's Transformation Process

CATO achieves its goal of enabling execution in distributed memory environments by replacing parallelism on thread level with parallelism on process level via the introduction of MPI. Instead of creating threads at the appropriate fork calls and then joining them after executing the microtask, the transformed application will initialize MPI at the very beginning, meaning that the entire application will execute concurrently on a certain number of processes, including the previously untouched, single-threaded sequential regions. Due to this, any calls to OpenMP-specific functions have to be replaced.

The first replacements that CATO performs are targeted at OpenMP functions with a trivial equivalent in MPI. Included in these functions are calls to `omp_get_thread_num`, which is replaced by retrieving the rank of the process, `omp_get_num_threads` is replaced by fetching the communicator size, and `__kmpc_barrier` is replaced by MPI barriers.

Afterwards, the fork calls are replaced by direct calls to the microtasks that were previously listed in their respective parameters. However, since the execution can now happen on distributed memory, the shared variables, which were also listed in the parameters, have to be distributed among the processes, each having a separate address space.

Due to this circumstance, memory allocations and deallocations are replaced by calls to the CATO runtime library. Furthermore, since the loads and stores of the previously shared memory can now span across different processes, they have to be replaced as well. The design and the current flaws in the transformation and the runtime library calls are discussed in Chapter 4.

The final steps are the pragma- and clause-specific transformations. CATO currently supports the transformation of the `parallel`, the `for` and the `critical` directives, as well as the `reduction` clause. To provide an example of these specific transformations, using the `for` pragma incurs calls to the OpenMP runtime to find the appropriate lower and upper bounds for the loop, as well as the increment, on a per-thread basis. These calls are replaced by appropriate versions for a distributed memory environment, which are present in CATO's runtime library.

As an LLVM transformation pass, CATO can then be embedded into the compilation process with clang and the MPI compiler wrapper. In particular, the transformations with CATO are performed as early as possible with very few passes running before, so that the other optimizations do not alter the form of the IR as CATO expects it, since the pass relies on patterns that are found in not yet optimized IR. Having CATO scheduled as early as possible also allows the following optimizations to refine the transformations made by CATO. Once all optimizations have been applied and the compilation process is finished, the resulting binary can be executed across multiple nodes with `mpiexec` or its alternatives.

# 4. Generalizing the Memory Management

This chapter is structured as follows: the first section describes how CATO distributes the previously shared memory among the processes. The second section provides more details regarding how the memory is accessed correctly, including information on the required transformations of the LLVM IR. The final section describes the shortcomings of the current implementations and the subsequent changes to alleviate them.

## 4.1. Distributing the Memory

Since the transformed application is not guaranteed to run on a single node, the previously shared memory needs to be distributed accordingly. To achieve this, the pass component of CATO replaces all previous memory allocations with calls to the runtime library. The following description of CATO's functionality is focused on shared memory arrays, as the handling for shared single values is both different and trivial. Instead of allocating memory directly, each call creates a so-called `MemoryAbstraction`. Depending on the dimensionality, or pointer depth of the memory, the creation of a `MemoryAbstraction` object has different effects.

```
1 int** d2 = malloc(sizeof(int*) * 4); //2-D alloc.
2 for (size_t i = 0; i < 4; i++)
3     d2[i] = malloc(sizeof(int) * 10); //1-D alloc.
```

Listing 4.1: Memory allocations with different pointer depths.

Listing 4.1 displays one of the options for allocating a two-dimensional array of four-by-ten integers. Variables that have a pointer depth of two or more, as is the case with the variable `d2`, are treated differently from variables or allocations with a pointer depth of one, which is displayed in the last line with the allocation of space for the actual integers. For variables with a pointer depth of two or more, the `MemoryAbstraction` object serves only as a wrapper for the allocation. In addition to the actual allocation of memory and returning the result of said allocation, it stores information regarding the dimensionality, the size of the allocation and the type in the `MemoryAbstraction` object. In essence, the allocation for memory with a pointer depth of two or more is performed equally on each process and uses the original size that was also present in the original OpenMP kernel. In comparison, the allocations for memory with a pointer depth of one, meaning one-dimensional arrays, can have different effects on each process. The creation

of a `MemoryAbstraction` for one-dimensional data leads to the even distribution of the underlying memory. Each process allocates the space for its own share of the data, after which an MPI window is created across these shares. This window can then be used for access to remote portions of the previously shared memory which is now managed in the `MemoryAbstraction`. An example distribution of the allocations in Listing 4.1 for four processes is demonstrated on the left side of Figure 4.1.

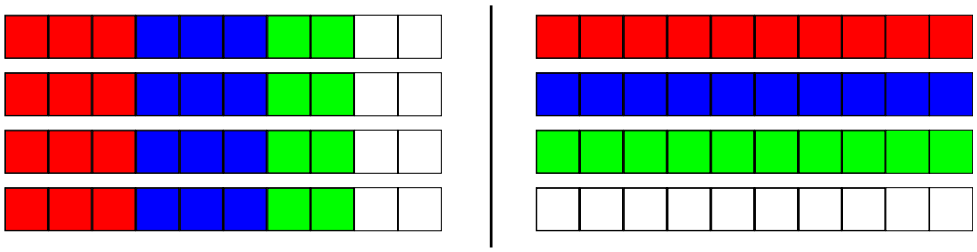

Figure 4.1.: Example distributions of a two-dimensional array after the transformation with CATO. Each colour corresponds to a different rank. For the left side, the memory was allocated as displayed in Listing 4.1. The distribution on the right is achieved if memory is allocated as shown in Listing 4.2.

Note that the original allocation of the memory has a significant impact on the distribution of the underlying data. If an array of the same dimensionality was allocated as shown in Listing 4.2, the distribution in the `MemoryAbstraction` would instead correspond to the right half of Figure 4.1.

```
1 int** d2 = malloc(sizeof(int*) * 4); //2-D alloc.
2 int* d1 = malloc(sizeof(int) * 4 * 10); //1-D alloc.
3 for (size_t i = 0; i < 4; i++)
4     d2[i] = d1 + i*10;
```

Listing 4.2: Another example for a similar allocation as in Listing 4.1. The underlying data is allocated in one contiguous array and the pointers of the two-dimensional pointer are set to the correct addresses afterwards.

## 4.2. Array Indexing

Alongside functions for allocating the memory on each process, the runtime library also provides functions to access the newly distributed memory in a store or load fashion, because the load and store instructions that are natively used in the IR cannot account for the newly distributed memory. For each of these two instructions, two versions are provided, one for accesses in the originally sequential portion of the application, and the other one for accesses in the parallel regions. The versions differ mainly in the usage of synchronization functions for their accesses, as the sequential versions utilize active target

synchronization, while the parallel versions use passive target synchronization. Finding the original loads and stores to the previously shared memory in the IR is a difficult task, however this is already implemented in CATO and therefore it is not the focus of this thesis. Accordingly, the content of this chapter is written under the assumption that all relevant load and store instructions have been identified already.

In order to correctly access the `MemoryAbstraction` objects, the runtime library functions require the indices of the original access. Therefore, these indices need to be extracted from the IR and passed on as arguments to the corresponding functions. The extraction of the indices requires an understanding of address computation and array indexing in LLVM. The most important instruction for this is the `getelementptr` instruction, or GEP for short. Listing 4.3 is an example of how an access to a three-dimensional array is implemented in LLVM IR.

```
1 %13 = load i32***, i32**** %m, align 8
2 %arrayidx13 = getelementptr inbounds i32**, i32*** %13, i64
    ↪1
3 %14 = load i32**, i32*** %arrayidx13, align 8
4 %arrayidx14 = getelementptr inbounds i32*, i32** %14, i64 0
5 %15 = load i32*, i32** %arrayidx14, align 8
6 %arrayidx15 = getelementptr inbounds i32, i32* %15, i64 1
7 %16 = load i32, i32* %arrayidx15, align 4
8 %call16 = call [...]
```

Listing 4.3: Accessing a three-dimensional array in LLVM IR.

The underlying C code of the LLVM IR excerpt is a single call to `printf` with the element of a three-dimensional array `m` at index $[1, 0, 1]$. After loading the address of `m` in the first line, the first GEP is encountered in the second line. The purpose of this instruction is to perform the address calculation for accesses to aggregate data structures, such as arrays or structs. It does not access the memory though. The first argument of the instruction is the type that is used for the calculations of the address. The second argument is a pointer that serves as the base address for the start of the calculations. The remaining argument is the index that is supposed to be accessed. Looking at the first address calculation in line 2, the type that is used for the calculation is `int**`, the start address is the address of `m` and the index for the access is 1. Since the memory is not accessed in this instruction, the result of the address calculation is instead stored in a register and then accessed in a subsequent load instruction. This pairing of index calculation followed by a load instruction is repeated two more times, once per dimension of the array, until the underlying data is accessed and the result is stored in register `%16`. Note that the GEP instruction allows for multiple indices beyond the first one as well. In doing so, the entire excerpt could be shortened to a single GEP instruction that takes the indices $[1, 0, 1]$ as its arguments. However, when looking at the unoptimized IR, such grouping does not occur. Since CATO operates on unoptimized IR, finding the

correct indices for an array access requires inspecting the entire chain of GEP and load instructions.

## 4.3. Shortcomings and Solutions

In its current state, CATO is only able to handle kernels with arrays of a dimensionality of three or less. This is due to the hardcoded index extraction during the pass, with one case per dimensionality, and the interactions of the runtime library, which displays similarly hardcoded behaviour. The following subsections go into further detail regarding one of the possible solutions for the shortcomings in the two components of CATO.

### 4.3.1. The Pass Component

CATO is already able to find all relevant loads and stores to shared memory, iterating through the def-use chain as well to find all users of the corresponding values. Furthermore, a tree structure is created for the load and store instructions by iterating recursively through the def-use chain of each user of the aforementioned instructions. Due to this, information on the pointer depth of the array and the accompanying accesses is already available as well.

Knowing the information regarding the pointer depth, extracting the correct indices from a particular access is now a matter of recognizing the patterns in the chain of instructions. With the LLVM libraries, it is possible to iterate through the instructions for address calculation and subsequent loading, starting at the bottom-most instruction, which is either a store or a load. Before going into further detail regarding the patterns, note that in LLVM, there is no difference between a variable or the instruction that produced it. For example, in Listing 4.3, the variable `%13` and the load instruction that produced it are considered to be the equal. The same is true for the variable `%arrayidx13` and the underlying GEP instruction. In the LLVM libraries, this fact is mirrored in the inheritance structure, as most classes inherit from the `Value` class. This includes the `Instruction` class, which in turn serves as the super class for the GEP and the load instructions.

To detect the correct pattern and extract the indices, the first instruction that is examined is the pointer argument of the bottom-most load or store. Both instructions have a pointer argument that points to the target memory address of the respective operation. In unoptimized LLVM IR, this argument can be one of three instructions:

1. A load instruction
2. A GEP instruction
3. A bitcast instruction

**Case 1:** If the pointer argument is a load instruction, no further address calculations via GEP instructions were necessary after the load. This is only the case if the target of the instruction is the address itself, or in other words if the index is 0. Listing 4.4 provides an example in IR, in which a one-dimensional array is accessed at index 0.

```
1 %0 = load i32*, i32** %arr, align 8, !tbaa !4
2 store i32 0, i32* %0, align 4, !tbaa !8
```

Listing 4.4: Example for a load instruction, directly followed by an access (Case 1).

**Case 2:** If the pointer argument is a GEP instruction, address calculations were necessary to access the correct memory address. Given that the GEP instruction of the unoptimized IR only has a single index in its arguments, the correct index can be extracted by examining said argument. In the example displayed in Listing 4.5, a one-dimensional array is accessed at index 1, so finding the correct index is a matter of extracting the final argument of the GEP instruction.

```
1 %1 = load i32*, i32** %arr, align 8, !tbaa !4
2 %arrayidx1 = getelementptr inbounds i32, i32* %1, i64 1
3 store i32 1, i32* %arrayidx1, align 4, !tbaa !8
```

Listing 4.5: Example for address calculation with a GEP instruction before the access (Case 2).

**Case 3:** The third case was only observed on accesses to the first dimension of an array, right after allocating memory for it. Instead of a load or GEP, the access is preceded by a bitcast instruction instead. This is illustrated in Listing 4.6. The second store in the listing writes the value 42 to the address pointed to by `%1`, which is the result of a bitcast operation on the previously allocated memory in `%call`. As with the load case, no address calculations are done, which means that the array is accessed at index 0.

```
1 %call = call noalias align 16 i8* @malloc(i64 16) #5
2 %1 = bitcast i8* %call to i32*
3 store i32* %1, i32** %arr, align 8, !tbaa !4
4 store i32 42, i32* %1, align 4, !tbaa !8
```

Listing 4.6: Example for no preceding load or GEP instruction (Case 3).

When combining these three cases, the correct indices can be extracted for an array of any dimension. Their combination results in the pseudo-code[1] listed in Listing 4.7.

[1]The actual source code is found in `src/cato/cato.cpp`

```
1  int counter = pointer_depth;
2  Instruction instr = targetInstruction.getPointerArgument();
3  List indices;
4  while (counter != 0)
5  {
6      if (isLoad(instr)) //Case 1
7      {
8          indices.append(0);
9          counter--;
10         if (counter != 0)
11             instr = instr.getPointerArgument();
12     }
13     else if (isGEP(instr)) //Case 2
14     {
15         indices.append(instr.getIndexArgument());
16         counter--;
17         if (counter != 0)
18         {
19             Instruction load = instr.getPointerArgument();
20             instr = load.getPointerArgument();
21         }
22     }
23     else if (counter == 1) //Case 3
24     {
25         indices.append(0);
26         counter--;
27     }
28 }
29 return reverse(indices);
```

Listing 4.7: Pseudo-code for the index extraction.

The cases are checked in a loop until the correct number of indices has been extracted, which corresponds to the pointer depth of the array. One aspect that has been neglected in the previous descriptions for the sake of simplicity is the need to follow the chain of users and extract the next instructions when processing multidimensional arrays. For example, in the second case (lines 13-22), the index is extracted from the GEP instruction (line 15). After checking whether there are more indices to be extracted, the next instruction is fetched that needs to be examined for this purpose. In the second case, this means that the pointer operand of the GEP instruction is inspected (line 19), which has been a load instruction in all test cases. The pointer operand of this load instruction then serves as the next instruction (line 20) that needs to be examined for index extraction purposes. This pattern can be observed in the IR in Listing 4.3 as well. For example, the pointer operand of the final load is `%arrayidx15`, which is

a GEP instruction. This instruction in turn has a pointer operand, `%15`, which is a load instruction. The instruction that would be examined next for index extraction in this case is the pointer operand of `%15`, namely `%arrayidx14`. In case 1 (lines 6-12), finding the next instruction is simpler by comparison, as it is the pointer operand of the previously examined load instruction.

While it is possible that there are other patterns that are not covered by this code, none of them emerged during any of the extensive tests. As a safety precaution, the check for case 3 (lines 23-27) catches any instruction that might occur instead of only catching the bitcast for the first index. Furthermore, this is not supposed to handle stores of pointers themselves. They need to be treated differently, as the counter for the indices does not correspond to the pointer depth of the array anymore, but rather the difference of the array pointer depth and the depth of the pointer that is to be stored. Beyond that, the same patterns apply.

### 4.3.2. The Runtime Library

After extracting the indices and inserting custom load and store instructions at compile time, the runtime component of CATO needs to map the indices to the correct memory locations across the different processes. Depending on the depth of the memory access, different transformations have been made during the pass. Accesses to shared arrays that do not target the underlying data, but rather the pointers, are not changed. For example, the store that would occur to `d2[i]` in Listing 4.1 in line 3, where the address of the allocated memory is stored, would not be changed by CATO. Due to this, any access above the lowest level on the shared memory arrays still yields the correct address. If however an access would be performed on the actual data at the lowest level, for example via `d2[0][0]`, one of the custom access functions of CATO's runtime library would be invoked instead of the normal load or store, since the accessed index is not guaranteed to be available locally.

These custom accesses have two phases: the first phase is a manual pointer chase to determine the correct `MemoryAbstraction` object for the access. Starting at the top-level `MemoryAbstraction` that represents the base address of the array, the pointers are followed to the lowest-level `MemoryAbstraction` that represents the underlying data. The second phase is the actual access of the `MemoryAbstraction` via one-sided MPI communication, utilizing `MPI_Put` for storing and `MPI_Get` for loading. Depending on whether these accesses happen in sequential or parallel regions of the original kernel, different synchronization mechanisms are employed to ensure consistency. Since a change of a value in the sequential regions needs to be visible immediately at each process, such stores or loads incur active target synchronization with fences to ensure that no process continues until the change is effective to prevent race conditions. This is necessary because the original kernel does not have the potential for race conditions in the sequential regions. Comparatively, accesses in the parallel regions use passive target synchronization via locks, which mirrors the accesses to shared memory within OpenMP.

Without using special pragmas or calling certain functions, stores and loads to shared memory do not incur any synchronization with OpenMP.

The basic functionality described in this subsection was already present in CATO. Limitations regarding the ability to process arrays of arbitrary dimensionality arose due to shortcomings in the pointer chasing phase, which was hardcoded to three dimensions. These shortcomings have been removed. Instead of describing the changes of the underlying source code, the workflow of the runtime library will be discussed in the following, starting with the general workflow of the pointer chase, as is depicted in Figure 4.2.

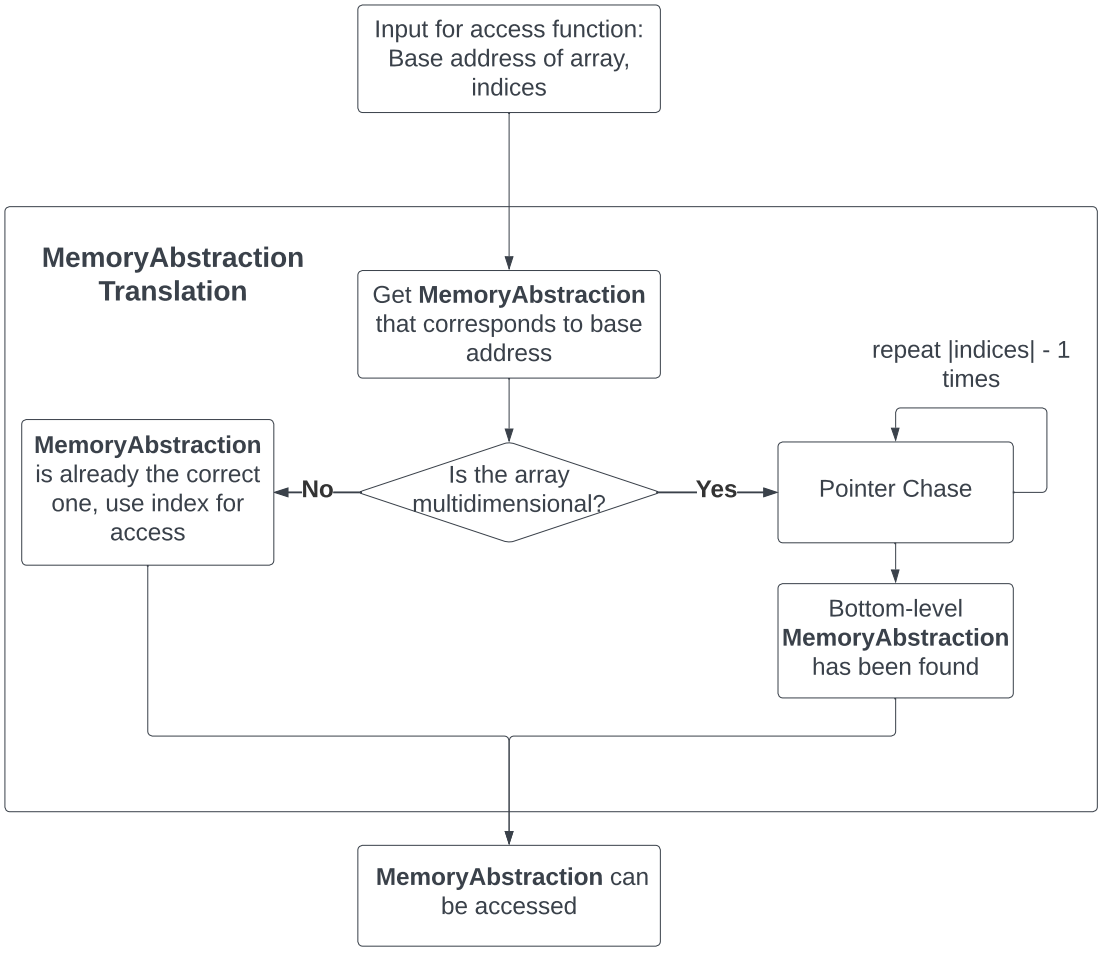


Figure 4.2.: Workflow of the pointer chase in CATO's runtime.

The relevant input to the custom access functions are the base address of the shared array and the indices of the access. In CATO, each `MemoryAbstraction` is identified and accessible via its base address. The first step in the pointer chase is fetching the

`MemoryAbstraction` corresponding to the base address of the shared array. If the array that is to be accessed is one-dimensional, in other words if the number of indices is one, no pointers need to be chased, as the base `MemoryAbstraction` is already the bottom-most one as well, therefore containing the actual data. It can then be accessed at the index passed as the other argument with one of the MPI functions.

For multidimensional arrays, pointer chasing is required. To start the chase, the first index is examined and used alongside the base address to access the element at said index. The accessed address is then compared to the base addresses of all known `MemoryAbstraction` objects to find the next one. After fetching this next `MemoryAbstraction`, the process is repeated with the next index, until the bottom-most `MemoryAbstraction`, containing the actual data, has been found. It can then be accessed with the correct MPI function at the element that is indicated by the final index, as is the case with the one-dimensional case of the workflow.

Reaching the lowest-level abstraction in the pointer chase relies on each access returning an address that corresponds to some `MemoryAbstraction` object. If such an access does not return the base address of a `MemoryAbstraction`, additional measures need to be taken. Listing 4.8 displays different types of memory allocation for multidimensional arrays, highlighting their effect on the pointer chase.

```
1  int** d2 = malloc(sizeof(int*) * 4);
2  for (size_t i = 0; i < 4; i++)
3      d2[i] = malloc(sizeof(int) * 10);
4  //Each access of d2[i] later on will yield the base address
5  //of a MemoryAbstraction
6  ------------------------------------------------------------
7  int** d2 = malloc(sizeof(int*) * 4);
8  int* d1 = malloc(sizeof(int) * 4 * 10);
9  for (size_t i = 0; i < 4; i++)
10     d2[i] = d1 + i*10;
11 //Accesses to d2[i] for i other than 0 will return an
12 //address in the middle of the d1 MemoryAbstraction
```

Listing 4.8: Influence of the memory allocations on the pointer chase at runtime.

These allocations are the same that were seen previously in Section 4.1. The first option for allocating a two-dimensional arrays leads to the creation of one `MemoryAbstraction` object per row. This is caused by the replacement of the malloc function calls with custom memory allocation functions. The custom function creates a `MemoryAbstraction` object and returns the address of it. Therefore, any later access to `d2[i]` will yield the base address of a `MemoryAbstraction`. If the option in the lower half of Listing 4.8 for allocating the memory is utilized instead, later accesses to `d2[i]` for $i \neq 0$ will yield an address that belongs to the `d1 MemoryAbstraction`, however this address would not

correspond to the stored base address, rendering the currently presented strategy for the pointer chase infeasible.

To remedy this shortcoming in the pointer chase, multiple options are available. The first option is to store the entire range of the allocated addresses per `MemoryAbstraction` alongside the base address. Then the accessed element can be compared to this range instead of only the base address. However, this would require an extensive search, checking every single `MemoryAbstraction` to find the correct one instead of only chasing through the abstractions that are relevant for any given access.

The second option is to create an extra case in the pointer chase component that covers the second memory allocation pattern in Listing 4.8. Since the datapoints are stored in a contiguous one-dimensional array, the base address of the `MemoryAbstraction` object corresponding to this particular array can always be found by accessing the first element in each dimension above the final one, i.e. `d2[0]`. This is also the case for arrays of higher dimensions. After retrieving the `MemoryAbstraction` for the one-dimensional array with the actual datapoints, the MPI functions can be called once again to access it properly. However, a new index needs to be calculated from the given parameters, since the pointer chase from the base address of the shared array did not take any indices into account[2]. After calculating this new index, the `MemoryAbstraction` can be accessed at the correct index.

All test cases produced correct results after adding this second option to the previously described pointer chase. While it is possible that there are other memory allocation patterns that are not covered by this, the first option, storing the ranges, can be used as a last resort to cover any patterns that did not emerge during testing, guaranteeing CATO's ability to process arrays of arbitrary dimensionality.

[2]The source code for the index calculation is found in `src/cato/rtlib/MemoryAbstractionHandler.cpp`

# 5. Optimizing the Memory Management

CATO's main goal is providing its users with the ability to extend the problem sizes that their OpenMP kernels can operate on. While it is acceptable for such a transformation to incur a penalty regarding the time to solution, if the magnitude of this penalty exceeds a certain threshold, CATO could be rendered unusable in a production environment. In order to prevent this and improve the run time of CATO, additional measures need to be taken.

First, the current performance of CATO needs to be assessed, while identifying the most inefficient aspects of the runtime component. To this end, CATO was used to transform an iterative stencil kernel. This kernel will be discussed in greater detail in Chapter 6. The transformed stencil kernel was then profiled with regards to the overall run time and the run time of the inserted CATO functions. This profiling was achieved through the manual addition of code for time measurements and subsequent aggregation, foregoing external tools to minimize the impact of the profiling on the run time of the measured functions. Table 5.1 provides an excerpt of the results, showcasing summed run times of the functions with the highest results.

| **CATO function** | **Time** | **Percentage of total run time** |
|---|---|---|
| Parallel Load | 116.992 s (0.641) | 74.88 % |
| Parallel Store | 28.272 s (0.177) | 18.1 % |
| Barrier | 1.306 s (1.151) | 0.84 % |
| **Sum** | 146.57 s | 93.81 % |
| **Total** | 156.238 s (0.608) | 100 % |

Table 5.1.: Profiling results (mean and standard deviation) of a stencil kernel that was transformed with CATO. The three listed functions consume the most time. The other functions are omitted because their overall contribution to the run time is negligible (below 1 second for each). The total time to solution is displayed in the last row.

The overall run time of the transformed kernel amounted to a mean value of 156.238 seconds across 3 runs. Of this time, over 90 % was spent in the inserted functions for the

| | **Index calculations** | **MPI communication** |
|---|---|---|
| Parallel Load | 6.137 s (0.017) | 90.106 s (0.646) |
| Parallel Store | 1.415 s (0.001) | 21.922 s (0.172) |

Table 5.2.: More detailed profiling results for the load and store functions of CATO, which require the most time to execute. The functions are further split into the time it takes to calculate the correct index and the subsequent MPI communication.

shared memory loads and stores in the parallel regions, followed by the custom barriers that make up less than a percent of the total run time. The access functions take by far the most time of any inserted function, the rest of which are excluded from the excerpt for the sake of readability and due to their comparatively small impact on the total time to solution. By comparison, the original OpenMP kernel requires 0.3 seconds to complete with a number of threads that equals the number of processes used for the profiled run of the transformed kernel.

The custom loads make up the vast majority of the run time for the transformed kernel. This distribution is the expected result for a 5-point stencil kernel though, since every store requires 5 loads prior. This particular kernel did not feature large regions of sequential accesses. Both the initialization of the data and the calculations were conducted in parallel regions. If, for example, the initialization would need to be done sequentially, the profiling results would contain a significant portion of sequential load and store accesses as well. However, large sequential regions likely point to limitations of the underlying kernel, as they hinder effective parallelization, be it via a tool or manually. Therefore, all future considerations are made under the assumption that the underlying kernels are dominated by parallel regions.

The shared memory accesses in the parallel regions were further decomposed into two components: the pointer chase and index calculations as the first component, and the MPI communication as the second component. As highlighted in Table 5.2, the summed run time of the second component is larger by comparison for both access functions. Note that summing the run times of the two components per access function results in a smaller number than the total time displayed in Table 5.1. This is caused by the added overhead from timing the individual components, combined with certain function calls that are necessary for CATO to function but cannot be attributed to either component.

These results were to be expected. Since every load and store to the shared arrays is replaced by CATO's own functions, the bulk of the inserted function calls consists of these access functions. Furthermore, each access to an element of the shared arrays incurs message passing via MPI. For the parallel regions, this means that an MPI window needs to be locked, a message with a single element needs to be passed via either `MPI_Put` or `MPI_Get`, and then the window needs to be unlocked again. This is especially punishing

in the case of the 5-point stencil kernel, as each element is read 5 times on average per iteration. This leads to most elements being the target of 5 individual accesses via `MPI_Get` per iteration, even though their content did not change. Note that the accesses in the sequential regions would also suffer similarly from accessing individual elements per MPI, if they were present.

Addressing the two components in the access functions for the shared memory arrays in the parallel regions has the highest potential to improve CATO's overall performance. While the accesses in the sequential regions suffer from the same problems, their effect on the overall run time is comparatively smaller due to the prevalence of the parallel regions. However, some of the proposed solutions also benefit the sequential accesses to some extent. In Section 5.1, the index calculation and the pointer chase are further discussed and a solution is provided to decrease the number of times that these calculations need to be made. Section 5.2 addresses the second component, discussing multiple avenues to lower the number of necessary messages, while also increasing their size to further improve CATO's run time performance.

## 5.1. Caching Address Calculations

In order to access the correct `MemoryAbstraction` object after calling one of CATO's access functions for shared arrays, the pointer chase and index calculations, which are displayed in the previous section's Figure 4.2, are required. As is evident from the profiling results, this address calculation can contribute a considerable amount of time to the overall run time of transformed kernels.

Given that the address calculations take as input the address of a `MemoryAbstraction` alongside a list of indices and return as output another `MemoryAbstraction` object and the index at which it needs to be subsequently accessed via the MPI functions, these address calculations can be regarded as a special case of address translations. Such translations have traditionally been an important topic in hardware design. When the concept of virtual memory emerged, providing an abstraction of the physical memory, address translations were necessary to transform the virtual addresses into the actual physical addresses. Nowadays, virtual memory is ubiquitous across the landscape of computer systems [Bhattacharjee et al., 2018]. Back in the 1970's, when hardware providers such as IBM first integrated virtual memory into their systems, for example in System/370 [Case and Padegs, 1978], the incurred degradation of performance from the numerous required address translations needed to be resolved. This degradation is the result of an increase in the overall number of memory lookups for the additional mapping information that needs to be accessed [Arpaci-Dusseau and Arpaci-Dusseau, 2018]. This led to the integration of a hardware component known as the Translation-Lookaside Buffer (TLB) [Couleur and Glaser, 1968]. In System/370, the TLB consisted of a set of registers that contained the results of the most recent address translations, thus serving as a cache to reduce the number of necessary memory lookups and translations by a

tremendous amount. Integrating this component alleviated the performance degradation that otherwise would have hindered the usage of virtual memory.

The concept of the TLB has since been used as an inspiration for some software-driven caching mechanisms. For example, to accelerate the search on in-memory index data structures, such as $B^+$ trees or hash tables, a software cache was designed to store the most frequently accessed elements, thus reducing the need to traverse the index [Wu et al., 2017]. For CATO, a similar approach will be taken. To reduce the time required to fetch the correct `MemoryAbstraction` object for a given address and indices, effectively translating them into a different `MemoryAbstraction`, an index cache is introduced.

The index cache is implemented as a key-value store. The key consists of the base address and the indices that are passed as arguments to the custom access functions of CATO. A major difference to hardware caches is the absence of cache lines. Instead, each element, or value, of the index cache represents one location within a specific `MemoryAbstraction`. Introducing this cache alters the workflow displayed in Figure 4.2. The new workflow is depicted in Figure 5.1.

Upon entry to one of the access functions, the index cache is checked for an entry with the base address and the indices that were passed as parameters. If the cache contains no such element, the previously highlighted translation mechanism involving the pointer chase is utilized. The resulting `MemoryAbstraction` object is then cached, using the input parameters of the access function as the key. Alongside the `MemoryAbstraction` and the index for the access, the elements of the index cache contain some additional information, namely regarding the process that owns the data that is supposed to be accessed. If the data resigns on a remote process, no further information is stored. If the data is local though, the buffer that underlies the `MemoryAbstraction` object is examined and the address that was accessed with the given index is stored in the cache element as well. This information can then be used to speed up processing of local data in the case of a cache hit in the future.

If the initial search of the index cache resulted in a cache hit, the right branch of the workflow diagram in Figure 5.1 is taken. Depending on whether the cache hit for the base address and the indices refers to local data or not, different actions are executed. If the data is not local, the target `MemoryAbstraction` is extracted from the cache element and the following access with the respective MPI function can be conducted. If the data is local though, the overhead that would be introduced by locking the window and calling the MPI access function can be avoided. Since the data is available locally, the memory can be accessed directly. This direct access via `memcpy` is facilitated by the extra information in the cache elements, which are only stored for local portions of a `MemoryAbstraction`. After the direct access, no further actions need to be taken and CATO's custom access function may terminate.

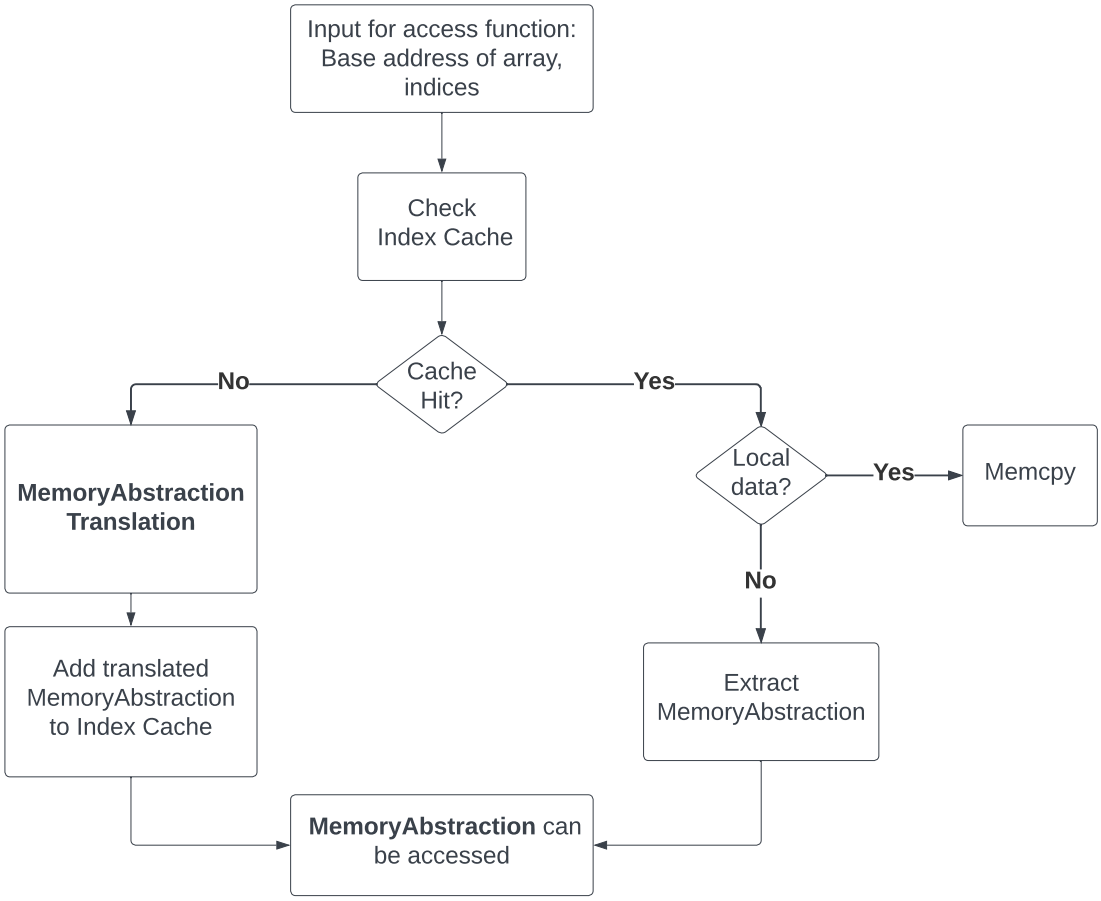


Figure 5.1.: Workflow of the address translation in CATO's runtime with the index cache. The `MemoryAbstraction Translation` procedure has been collapsed for the sake of readability.

The functionality of the index cache can be applied to both accesses in the sequential portions of the original kernel, as well as accesses in the parallel regions, but with one exception. Processing local elements via `memcpy`, thus circumventing the MPI functions including the synchronization, is only possible in the access functions for the parallel regions. This is due to the usage of passive target synchronization, which only requires the process calling the MPI communication function to partake in the synchronization. The custom access functions for the sequential regions rely on active target synchronization, for which every process needs to participate in the synchronization. Therefore, circumventing the synchronization in case of an access to local data is not feasible in the sequential areas at this time.

There are also certain cases where it is advisable to not allow the sequential accesses to create entries for the cache, for example when running initialization procedures. During

initialization, it is common to access each element of the data exactly once. Creating one cache entry per process for every single data element can quickly pollute the cache, drain memory resources and consume extra run time for no immediate benefit. While the cache pollution and the memory drain can be combatted with a proper cache eviction strategy, this eviction would only further consume run time. Therefore, sequential areas are per default only able to read cache elements and are not able to create new entries.

It is possible that entries could become invalid. For that to happen, changes to the memory mapping of the arrays would need to happen at the IR level at runtime. However, this has not happened during any of the tests or benchmarks. Therefore, the removal of invalid entries will be disregarded for the rest of this thesis.

## 5.2. Caching MPI Messages

The second component of the access functions, the MPI communication, constitutes the majority of the overall run time of the transformed kernel. Accessing singular elements via MPI functions introduces a considerable overhead to what would otherwise be a quickly executed load or store in the original kernel. Improving the performance of the second component regarding its run time will therefore require a significant reduction of the number of MPI accesses. This will be achieved through the application of multiple techniques.

**Caching instead of repeated loads:** There are certain kernels that require reading the same element of some array multiple times, without the content of said element changing inbetween these reads, as explained for the 5-point stencil. Instead of repeatedly accessing these elements via MPI and incurring the overhead of synchronization and message passing, such elements can be cached locally instead, accelerating the access past the first read.

**Grouping write operations:** Writing single elements is not efficient. Instead, all elements that are supposed to be sent to a remote process will be grouped. Grouping them enlarges the message size while simultaneously reducing the number of required messages. This reduces the synchronization and message passing overhead of the write operations.

**Reading ahead:** Many kernels display regular access patterns. In the case of the stencil kernel, the array is accessed with a unit stride. By extracting this access pattern, it is possible to enhance the custom read operations. Instead of accessing only a single element, the next $n$ elements are read and cached locally for fast access in the following iterations.

All of these techniques will be integrated via the introduction of another software cache that will operate alongside the index cache introduced in Section 5.1. The new cache will henceforth be referred to as the message cache. This message cache will contain the

results of read accesses, alongside an aggregation of elements that are supposed to be written to remote processes. Furthermore, the reading access functions will be enhanced to enable the prefetching of elements for the message cache.

The usage of software caches for DSM systems is not a novel concept. Many developers of such systems came to the conclusion that these caches are a large benefit to the performance of their respective systems. Example implementations include Ivy [Li, 1986], which operated on a cluster of Apollo workstations, and the Munin system [Bennett et al., 1990b]. The latter example furthermore introduced multiple cache coherence mechanisms to address the different access patterns of shared data objects.

### 5.2.1. Designing the Message Cache

The design of the cache and its coherence mechanism needs to adhere to certain constraints that are imposed to the entirety of CATO. Given that the purpose of CATO is the transformation of OpenMP kernels to enable the computation on larger problem sizes, it is imperative that the results the original kernel produces are equal to the results of the transformed kernel. This constraint must hold for every OpenMP kernel with specified results, which is the case if the original kernel avoids data races. While CATO alters the OpenMP model of execution via the introduction of processes and the abandonment of the fork-join model, it still aims to adhere to the memory model. If the OpenMP functionality regarding its directives is ported properly to the new process-driven execution model while also adhering to the memory model, equality between the results of the original and the transformed kernel can be guaranteed in the absence of data races, at least for the directives that are supported by CATO. Therefore, the message cache and its coherence mechanism are best designed with the OpenMP memory model in mind to ensure consistency of the results. While it is possible to design a cache coherence mechanism that is agnostic of the OpenMP memory model, unwanted side effects might be introduced that alter the final result. The original kernel was created with OpenMP and thus adhering to the memory model of OpenMP. If a cache coherence mechanism is not aware of when memory is supposed to be coherent according to the OpenMP memory model, final results of the transformed kernel may differ from their original due to incoherences in the memory and subsequent accesses to said memory.

#### The OpenMP Memory Model

The OpenMP memory model is described as a “relaxed-consistency, shared-memory model”[OpenMP, 2021]. In addition to the ability to access the memory for loading and storing, alongside threadprivate memory, each thread is also allowed to maintain a temporary view of the memory. This view can be used to cache the content of variables, thereby circumventing the need to access certain parts of the memory directly. This can be useful if, for example, the memory that should be accessed is part of a different NUMA domain. Furthermore, the view does not have to be consistent with the memory

at all times, hence the relaxed consistency.

Consistency is only enforced upon the execution of a flush operation. Since 5.0, the OpenMP standard distinguishes between three different flushes. A strong flush guarantees that the latest write of the flush-executing thread to a shared variable is completed once the flush operation finishes, thus commiting it to memory and making it accessible to other threads. It also discards the temporary view of that variable, forcing the executing thread to read the value from memory upon its next access. The other two flushes are the release and acquire flushes. A release flush commits any outstanding write operations on shared variables to the memory, while an acquire flush discards the values for any shared variable to which it has not written from its temporary view. These two flushes are usually utilized to synchronize two threads, meaning that the writes of the release flush are guaranteed to complete before the acquire flush. After completing the acquire flush, the executing thread is able to read the new value of the shared variable. Flush operations do not have to be of one type exclusively. They may even be all three at once.

While flushes can be triggered manually by using the `flush` directive, they are most often encountered implicitly, since many of the synchronization operations have implied flushes attached to them. For example, upon entering a `barrier` region, both a release and an acquire flush are performed on each thread, ensuring consistency of memory after exiting the barrier. Furthermore, explicit flushes may have a list of variables attached for which the flush is to be executed. This list is not present for implicit flushes, which leads to all shared variables being flushed in those cases.

### A Conforming Cache Model

The cache model encompasses multiple operations: a read and a write operation, and multiple flushes to accommodate for the different flush types in the OpenMP memory model. The custom access functions of CATO will use the message cache only for accesses to remote elements. Local elements will be handled without involving the message cache, because the local accesses do not require the creation and passing of a message. Each process will have its own message cache in its address space.

**Read operation:** Whenever a process attempts to read a remote element, it will first attempt to find the element in its message cache. If the element is contained therein, the value will be transferred via a `memcpy` operation and no further actions are required. If the targeted element is not present in the message cache, an MPI access is started. The value is retrieved via `MPI_Get` and written to the target memory. Furthermore, the value is stored in the message cache for subsequent accesses, replicating the data locally.

**Write operation:** When attempting to write a value to a remote process, the message will instead be stored in the message cache. Per rank, the message cache aggregates the elements that were supposed to be written to the respective rank, storing information regarding the target location alongside the actual data. The data will not be written to

the target before a flush operation is executed. Furthermore, the cached data can now be accessed in subsequent reads that target these locations, eliminating the need for a remote access. This delayed update is akin to a write-back policy, as it is seen in hardware caches.

**Flushing the read elements:** This operation mirrors the acquire flush. The contents of the message cache that were not written to are removed from the cache.

**Flushing the to-be-written elements:** This operation mirrors the release flush. In order to evict the contents of the message cache that were written to, messages need to be passed to remote processes. Per target rank, the stored elements are grouped into a single message and written to the target process via `MPI_Put`. Once the MPI operation is complete, the elements can be removed from the message cache.

For this model in particular, the synchronizing quality of the flushes in OpenMP is disregarded. Instead, CATO's synchronization mechanisms will be relied on to ensure that the completion of the writes on the process executing a release flush will happen before the process executing an acquire flush can continue accessing the data, thus relieving the cache model of this duty. This is achieved by properly placing these flushes before or after inserted barriers or locking mechanisms. For example, if a thread in an OpenMP kernel was to exit a `critical` region, it would implicitly perform a release flush, synchronizing with the thread that next enters the `critical` region, where said thread would perform an acquire flush. When transforming such kernels with CATO, entering a `critical` region requires the acquisition of a mutex. It can be ensured that the outstanding writes of the exiting process are completed before another process can enter the `critical` region by virtue of placing the flush operation before unlocking the mutex. In turn, the process next entering this region will perform its acquire flush only after acquiring the mutex.

Due to the absence of the synchronizing quality in the flushes, there is no need to model another operation for the strong flushes. Instead, a strong flush is going to be represented by a release flush, followed directly by an acquire flush.

As with the index cache before, these operations are only usable in the context of the parallel regions due to the synchronization mechanism that involves each process when using the sequential access functions.

### 5.2.2. Implementing the Message Cache into CATO

The implementation of the message cache separates the read and write portions to minimize the management overhead. If all elements were stored in a single cache, each element would need to carry additional information regarding its state (is the cached element "dirty" or not). Furthermore, clearing the cache would then require iterating over the entire data structure, only dropping the portions that are targeted by the current flush operation. Separating the read and write portions of the cache alleviates the need for additional information per element and allows for simply dropping the entire content

of the separate data structures per flush operation.

Both parts of the message cache, henceforth designated as the read message cache and the write message cache, are built on top of an in-memory key-value store. For the read message cache, the key is of the same nature as in the index cache, namely a combination of the accessed base address and the indices. The value of the read message cache only consists of the data and its size, once again foregoing the concept of entire cache lines for single elements. Upon encountering a flush operation for this cache, the key-value store is simply dropped with no need for further actions.

The write message cache uses a different key, namely the address of the bottom-most `MemoryAbstraction` object, which contains the actual data and is therefore the target of the write operation. A single element in the write message cache is an aggregation of all remote writes to that specific `MemoryAbstraction`. Figure 5.2 is a representation of such an element.

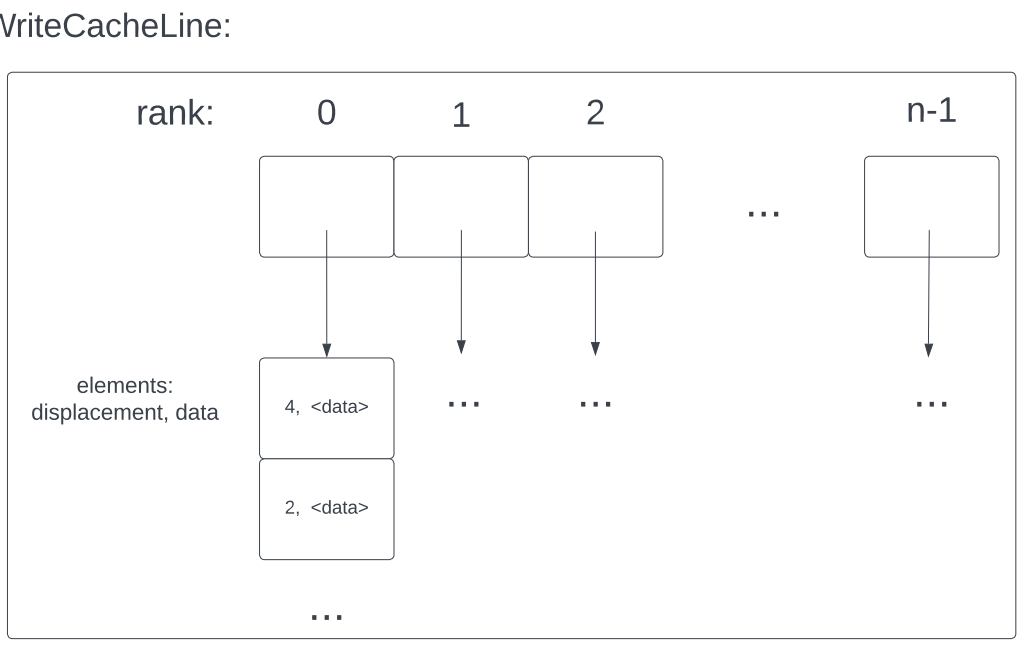


Figure 5.2.: Overview of the `WriteCacheLine` data structure. Each element of the top-level array represents a different rank. Each element therein represents a single value, stored alongside its supposed displacement in the target window.

The element of the write message cache, denominated as the `WriteCacheLine` in CATO's source code, contains an array of $n$ elements, where $n$ is the number of processes. Each element of this array is in itself an array, holding the to-be-written data in the

form of a pair: the actual data with its size and the displacement at which this value would be placed in the target MPI window. Flushing the write message cache requires multiple steps, outlined as pseudocode in Listing 5.1[1].

```
1  for writeCacheLine in writeMessageCache:
2      for rankArray in writeCacheLine:
3          sortByDisplacements(rankArray);
4          Array displacements =
               ↪extractDisplacements(rankArray);
5          Array data = extractData(rankArray);
6          MPI_Datatype dt =
               ↪createIndexedDatatype(displacements);
7
8          MPI_Win_lock(target_rank);
9          MPI_Put(data, dt);
10         MPI_Win_unlock(target_rank);
11
12         rankArray.drop();
13
14     writeCacheLine.drop();
```

Listing 5.1: Pseudocode for flushing the write message cache.

In order to drop the entire write message cache, each `WriteCacheLine` needs to be cleared. Achieving this requires clearing each rank-specific entry per `WriteCacheLine`. To this end, each rank-specific entry is first sorted by the displacements of the elements in ascending order. The sorted displacements and the accompanying data, which is also sorted now, are then extracted into separate continous arrays.

The displacements are used to create a new MPI datatype to aggregate all elements into a single message. There are varying levels of specificity regarding the derived MPI datatypes, starting with the simplest, `MPI_TYPE_CONTIGUOUS`, which is just a replication of a single datatype. Limiting the `WriteCacheLine` to only write contiguous elements could have serious performance implications for strided writes, as the result would be that each write contains just a single element again, as was the case with the uncached writes. The more general `MPI_TYPE_VECTOR` allows for the replication of datatypes into equally sized and spaced blocks. While this would suffice for strided writes with constant offsets between blocks, it would not be feasible for strides of varying size, or even random writes. Therefore, the MPI datatype that is created in the `WriteCacheLine` is `MPI_TYPE_INDEXED`, which allows for both varying block sizes and varying strides between the blocks.

After creating the datatype from the extracted displacements, the data can be sent to the target via MPI, employing passive target synchronization and a call to `MPI_Put`

[1]The actual source code is found in `src/cato/rtlib/WriteCacheLine.cpp`

to place the data in the target window. After unlocking the window, the rank-specific array entry has been successfully processed and can be dropped. Once all entries have been cleared in this manner, the `WriteCacheLine` is considered to be cleared and can be dropped as well.

The two flush operations are implemented as simple calls to the clear functions of the respectively targeted cache, meaning that the release flush would incur a clear of the write message cache, while the acquire would clear the read message cache. The flush calls are then inserted alongside the other runtime functions for the OpenMP directives. For example, before entering a parallel region, a call to the release flush is inserted and the region is only entered once all processes have completed this flush.

With this implementation, both portions of the cache are dropped at predefined points during the execution. If the caches should be dropped earlier though, due to for example constraints on the size of the caches, this can be achieved as well. For the write message cache, each `WriteCacheLine` can be informed of an upper bound for the elements per rank. Exceeding that limit triggers an early implicit flush, clearing only that rank-specific entry. This ensures that the messages stay below a certain size for each flush operation, for example the integer size limit. This is relevant because many parameters of MPI functions regarding the element count are limited to 32-bit integers, which includes the element count of `MPI_Put`. For the read message cache, a global size limit can be set to the number of cached elements. Each element that would exceed the limit then triggers the eviction of an existing element, according to some cache eviction policy. At present, the LRU policy is employed.

### 5.2.3. Readahead Capabilities

With the cache model and the implementation presented in the previous section, the message cache is operational. It is filled dynamically, as elements are requested. This means that the cache is not bound to any particular access pattern and every cached element was actually accessed at some point prior to the introduction to the cache. The initial access to read an element from a remote process however still requires the MPI functions to retrieve a message containing a single element. The overhead introduced by this is still a concern for the performance of CATO. This overhead can be ameliorated by virtue of increasing the message size and fetching more elements with every remote access.

Many kernels exhibit regular access patterns. Stencil kernels for example access consecutive elements with a certain stride, most often a unit stride. For such an access pattern, the message size of CATO's load accesses to remote processes can be increased by also loading consecutive elements and storing them in the read message cache. The next access to said consecutive element consequently does not require a remote access anymore, since the element is already present in the read message cache, leading to a more efficient local access instead. The readahead is only triggered when a load access to a remote element incurs a cache miss. Presently, the number of elements that is prefetched in this

manner is determined by a user-supplied integer, read from an environment variable. This number of elements, determining the size of the prefetched messages, will henceforth be referred to as the prefetch value. It serves as a maximum value for the prefetched messages. In cases where the number of available elements is lower than this prefetch value, all of said available elements will be read, resulting in a smaller message than specified by the prefetch value.

This readahead approach has a pendant in hardware caches as well. When discussing cache fetching algorithms, hardware caches have the choice to fetch data on demand, akin to CATO prior to the introduction of the readahead mechanism, or data can be fetched ahead of time, in which case the cache fetching algorithm is known as a prefetch algorithm.

### 5.2.4. Static Code Analysis for Improved Prefetching

For hardware caches, prefetching algorithms need to be carefully designed to actually improve the performance and not pollute the cache memory [Smith, 1982]. Not only should the prefetched lines contain information that will be required in the near future, but they should also not replace lines that are likely targets for timely accesses. While CATO's read message cache is not necessarily limited regarding its size, fetching elements that are not going to be used is still not desirable, as there is a time cost associated with processing these extra elements and storing them in the message cache.

Take as an example a convolution operation with a stride larger than one. Much like stencils, the convolution operation moves a fixed pattern across the data. However, if the stride is larger than one, the subsequent elements might not be needed for the next step of the computation, if the stride is large enough. If CATO's readahead mechanism does not account for strided access patterns, many elements that would be introduced to the read message cache might not be needed for future computations, effectively wasting the time that was needed to place said elements in the cache.

In order to alleviate this problem, CATO's readahead mechanism is extended to incorporate a stride. This stride is determined via a static code analysis. In particular, a scalar evolution analysis is performed on the indices of the shared array accesses. This code analysis is conducted with the help of the `ScalarEvolution` analysis pass that is present in LLVM.

Scalar evolution is described as the "change in the value of scalar variables over iterations of the loop" [Absar, 2018] and the LLVM pass utilizes symbolic techniques to analyze these changes. The foundation of the analysis is the concept of the so-called chains of recurrences. This concept will only be touched upon briefly, highlighting the aspects that are necessary to understand its implications for the use case at hand. For a more thorough explanation and insight into how these chains are constructed, refer to the original paper [Bachmann et al., 1994], upon which the following explanation is based as well.

The motivation for the development of these chains was to provide an efficient option to evaluate functions at regular intervals. For this purpose, functions are transformed into representations that feature recurrences, simplifying the calculation of successive steps in the target interval. The foundation of the chains of recurrences is the basic recurrence $f$, defined as the tuple $f = \{\varphi_0, \odot, f_1\}$. In this tuple, $\varphi_0$ is a constant, $f_1$ is a function defined over the natural numbers, and the operator $\odot$ is one of the operators $\cdot$ or $+$. The function $f(i)$, as represented by the tuple, is then defined as

$$f(i) = \{\varphi_0, +, f_1\}(i) = \varphi_0 + \sum_{j=0}^{i-1} f_1(j)$$

if the operator is $+$, or as

$$f(i) = \{\varphi_0, \cdot, f_1\}(i) = \varphi_0 \prod_{j=0}^{i-1} f_1(j)$$

if the operator equals $\cdot$ instead. While it is possible to represent simple functions with a basic recurrence, more complicated functions require chains of them, hence the name chains of recurrences.

In scalar evolution, this concept can be applied to induction variables. These variables progress through a series of values that form an arithmetic progression [Muchnick, 1997], as is often the case with loop control variables or array subscript values in loops. Listing 5.2 is an excerpt from a 5-point stencil, showing a portion of the main loop in which the elements for the stencil are loaded.

```
1 for(int i = 1; i < xMax-1; i++){
2     for(int j = 1; j < yMax-1; j++){
3         double star = 0.25*(matrix[ind(i-1,j)] +
              ↪matrix[ind(i+1,j)] + matrix[ind(i,j-1)] +
              ↪matrix[ind(i,j+1)]);
4         [...]
5     }
6 }
```

Listing 5.2: An excerpt of the main loop of a 5-point stencil.

The loop accesses elements of the array in a nested loop, using the loop variables as array subscripts. The `ind` macro is used to calculate a single index from the two loop variables, because the underlying array is one-dimensional. It performs the following computation: $ind(x, y) = x \cdot y_{max} + y$. Applying the scalar evolution analysis to the stencil kernel, cleaning the results and removing some additional information regarding wraparounds yields the following chain of recurrences, among others:

{{(1 + (1000 * %12)),+,1000}<%omp.inner.for.cond>,+,1}<%for.cond>

This chain, displayed in the format that is chosen as the output of the pass, is the analysis result for the IR register that holds the array subscript for accessing `matrix[ind(i-1,j)]`. Fully understanding the chain of recurrences requires some additional context: the value of `yMax` was set to 1000 for this analysis, and the initial value of register `%12` is 0 on the process with `rank=0`. The intial value of this register on other processes depends on what portion of the loop they have been assigned. The chains produced by the analysis pass are represented as nested basic recurrences. Outer recurrences have the inner recurrences as their first element in the tuple. The inner-most recurrence represents change caused by the outer-most loop. The actual recurrence is enclosed in brackets, while the appended angle brackets contain the label of the loop that this recurrence is based on. The notation of the basic recurrences matches the one that was introduced in the original paper. The first element of the tuple is a constant, the starting value. The second element is the operator, which is the + operator for both of the recurrences in the example. The final element is a function defined over the natural numbers. In many cases for loop variables, this function is a constant that is added with each iteration. In the case of the inner-most recurrence, the function is set to 1000, as an increase in the loop variable of the outer loop leads to an increase of 1000 for the array subscript. Similarly, the function for the outer recurrence, representing the inner loop, is the constant 1, as an increase of the loop variable in the inner loop increases the array subscript by 1.

The analysis results can be utilized by CATO to determine the access patterns for shared arrays in parallel regions. CATO's custom load function for parallel regions requires the base address of the shared array and the correct indices as parameters to enable the access. Thus, performing a scalar evolution analysis on these indices and processing the results can yield a stride at which an array is accessed, if the underlying kernel displays a regular access pattern.

Listing 5.3 is a pseudocode implementation[2] of how CATO utilizes the scalar evolution pass for determining array access patterns. In CATO's current state, the analysis is performed as follows: after transforming the IR of the original kernel to use the custom access functions, the scalar evolution analysis is performed. The chains of recurrences that are of interest are the ones for the final index in each custom load in the parallel regions. Therefore, a loop is executed over each parallel region, and then again over each load to shared memory in this region. The last index is extracted and the chain of recurrence is investigated. For the chain of recurrences, the outer-most basic recurrence is investigated and the function is extracted. If the access pattern of the underlying kernel is regular, this is going to be a constant. In the event that the analysis did not produce a chain for the index in question, the particular access is skipped. If the function of the outer-most basic recurrence can be extracted though, it is stored alongside the

[2]The actual source code is found in `src/cato/cato.cpp` and `SCEVHelper.cpp`

address of the targeted shared memory array in a map that uses the IR value of the array as a key and has a list of strides as its value. After extracting all access strides, a loop is executed over the map entries. The average stride for accessing a particular array is calculated and the result is subsequently used to insert a stride setting function into the IR of the parallel region. This function is inserted at the start of the parallel region and informs the readahead component of CATO that prefetches on this array need to be strided with a distance corresponding to this average value.

```
1  for each ParallelRegion:
2      map<MemAbs, List> accessStrides;
3      for each load in parallelRegion:
4          lastIndex = load.extractLastArg();
5          chain = getChainForIndex(lastIndex);
6          if (chain == null)
7              continue;
8          basicRec = chain.getOutermostBR();
9          stepSize = basicRec.function;
10
11         addressOfArray = load.extractArrayArg()
12         accessStrides[addressOfArray].append(stepSize)
13
14     for mapEntry in accessStrides:
15         int averageStride = avg(mapEntry.value);
16         if (averageStride == 0)
17             continue;
18         insert_stride_setter_into_ir(mapEntry.key,
             ↪averageStride);
```

Listing 5.3: Pseudocode for the application of the scalar evolution analysis in CATO.

## 5.3. Flow of Execution With Caches

When integrating all the previously discussed enhancements into the initial flow of execution for CATO's custom access functions, which was presented in Figure 4.2, the store and the load function display the behaviour that is shown in Figure 5.3 and Figure 5.4 respectively.

The custom store operation (Figure 5.3) utilizes the index cache in a similar manner as the load function does. Upon entry to the function, the index cache is checked for an entry corresponding to the base address and the indices. If this results in a cache hit, the next action depends on the locality of the data. For local data, a `memcpy` operation is sufficient to write the content of the given buffer to the target that was returned from the index cache. This would conclude the store operation for local data.

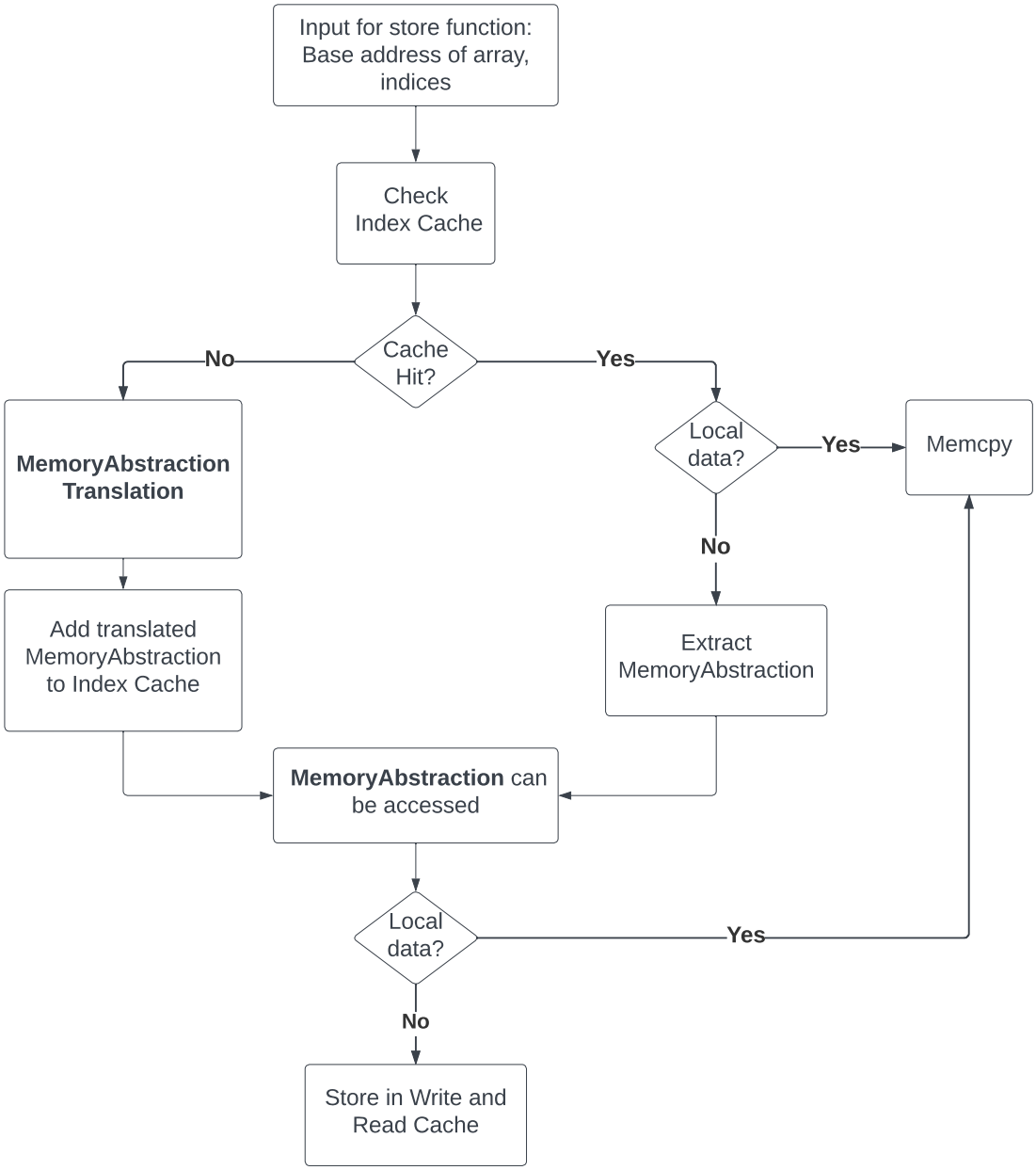


Figure 5.3.: Simplified flow of execution for CATO's custom store function in parallel regions when integrating the previously discussed mechanisms.

For remote data, the correct `MemoryAbstraction` object is extracted from the cache instead, circumventing the address translation that would be necessary in case of a cache miss on the index cache. With the correct `MemoryAbstraction` object available, the actual access can continue. Writing to a remote location will not incur any MPI calls though, as the data that is supposed to be written is instead stored in the write

message cache. The contents of the cache will only be written to the intended target processes upon flushing the write cache (not depicted in the flow of execution). This would conclude the custom store operation for remote accesses. However, since the change is not visible at the target process, the new value needs to be stored in the read message cache of the origin process as well to maintain coherence, as subsequent load accesses to this particular address from the origin process must return this new value.

The custom load function (Figure 5.4) starts its flow of execution by checking the contents of the read message cache. If an entry exists for the input parameters, namely the base array address and the indices for the access, this cache entry can be extracted and the content is transferred to the target buffer via a `memcpy` operation. This would conclude the load function. If no such entry exists in the read cache, the index cache is checked next. If this leads to a cache hit, which is the case if the queried indices for the given array were already used in a previous access, one of two paths is taken. If the data resides locally, a `memcpy` operation is sufficient to load the data into the target buffer and conclude the custom load operation. For remote data, the second path is taken, which is an extraction of the correct `MemoryAbstraction` object. This circumvents the need for the `MemoryAbstraction` translation in the same manner as it was described for the store function. After finding the correct `MemoryAbstraction` object for the access, whether through means of the index cache or the address translation process, the actual access can be conducted. For remote accesses, the prefetching mechanism can be used to read multiple values from the target process. All of these prefetched remote elements can then be stored in the read message cache for faster accesses in subsequent load operations.

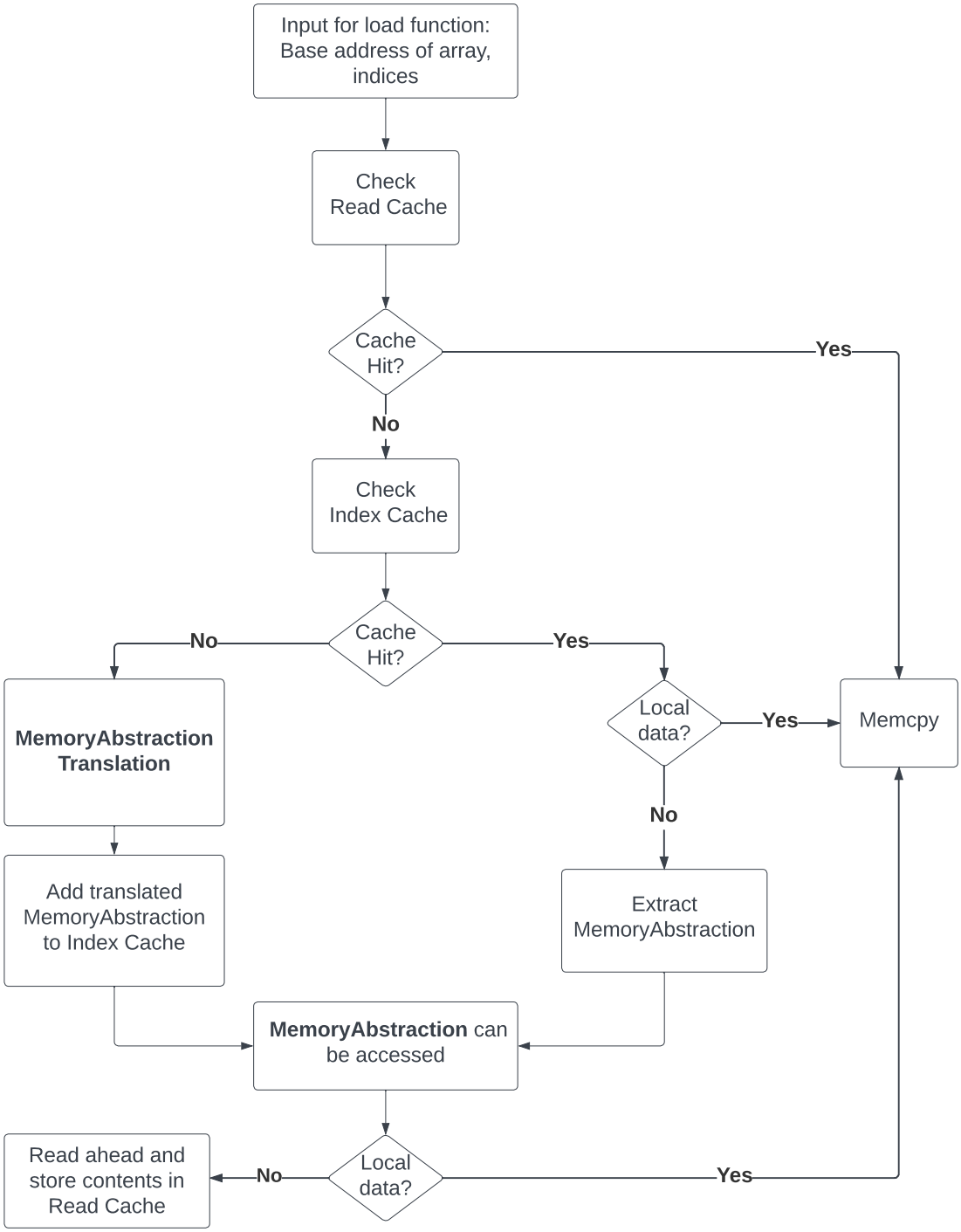


Figure 5.4.: Simplified flow of execution for CATO's custom load function in parallel regions when integrating the read and index cache, alongside the prefetching capabilities.

# 6. Evaluation

The proposed enhancements to CATO are evaluated regarding their impact on the time to solution for multiple different kernels with varying problem sizes. The experiments are conducted on the current HPC system of the DKRZ, the Levante system. The utilized nodes contain two AMD 7763 CPUs and either 256 or 512 GB of main memory. All measurements are done thrice. Accordingly, the reported values are the mean values of these measurements, which are displayed alongside their standard deviations. Due to the small magnitude of these standard deviations though, they are barely visible in many of the following figures.

All applications are compiled with clang 13.0.0, as distributed in the Debug build of LLVM 13.0.0. The utilized MPI implementation, if applicable, is Intel OneAPI MPI 2021.5.0. Further information on set environment variables, as well as the source code of the following applications, certain details and scripts for their execution, is provided in the section on reproducibility in Appendix A.

The evaluation is based on two computational kernels, each representing a different numerical method that is often encountered in HPC workloads. These specific workloads were characterized into 13 different categories in [Asanovic et al., 2006], so-called Dwarfs. The kernels utilized for this evaluation are a stencil kernel, representing the Structured Grid Dwarf, and a sparse matrix-vector multiplication (SpMV), which represents Sparse Linear Algebra. These particular choices have been made due to their differences in the access patterns on the underlying data, which will be further highlighted in their respective sections of this evaluation.

In CATO's current state, each proposed enhancement is controlled via environment variables. These variables are used to activate the individual components. In particular, the index, read message and write message cache can all be activated independently of each other. The number of elements that is prefetched is also controlled through an environment variable. Per default, no prefetching is conducted. While it is possible to limit the sizes of each cache individually, no such limitations were imposed during the following experiments.

To achieve the required coherence when the write message cache is active, the read message cache needs to be active too. This is due to the necessity to store a value in the read cache if it was previously aggregated in the write cache. Otherwise, the change would not be visible to the local process that conducted this store operation, which is not in accordance with the cache model. However, in the following experiments, activating

the write cache does not automatically incur an activation of the read cache. This allows for an isolated evaluation of the write message cache's influence on the time to solution and other metrics. Furthermore, the kernels for the evaluation do not require loads from previously written to elements prior to a flush call in parallel regions. Thus, the aggregated element of a remote store does not need to be present in the read cache for these particular kernels to produce correct results.

## 6.1. Stencil

The stencil kernel is an iterative 5-point stencil on a square matrix, as it is seen in some numerical solvers for the Poisson equation [Bottoni, 2022], for example in heat transfer kernels.

```
1  double* matrix = malloc(sizeof(double) * DIM * DIM);
2  double* matrix2 = malloc(sizeof(double) * DIM * DIM);
3  [...]
4  ------------------------------------------------
5  for(int iter = 0; iter < ITER; iter++)
6  {
7      #pragma omp parallel for
8      for(int i = 1; i < DIM-1; i++)
9      {
10         for(int j = 1; j < DIM-1; j++)
11         {
12             [...]
13         }
14     }
15     [...]
16 }
```

Listing 6.1: Code snippets from the stencil kernel. The upper half displays the memory allocation, the lower half shows the structure of the loop for the calculations.

The stencil kernel does not update the values in-place. Instead, a second matrix is used for this. This results in swapping the read matrix and the written-to matrix after each iteration. All required memory is allocated on the heap, after which the initial values of the matrices are set. Both the initialization as well as the iterative portions of the applications are enclosed in OpenMP parallel regions, using the `parallel for` directive.

The stencil kernel iterates over the matrices multiple times, with each iteration requiring multiple accesses to the same elements of the read matrix. Since the stencil is plus-shaped, most elements of the read matrix are accessed five times in any given iteration, while the elements of the written-to matrix are only accessed once. As the updates

are written to a separate matrix, the read values do not change within any given iteration.

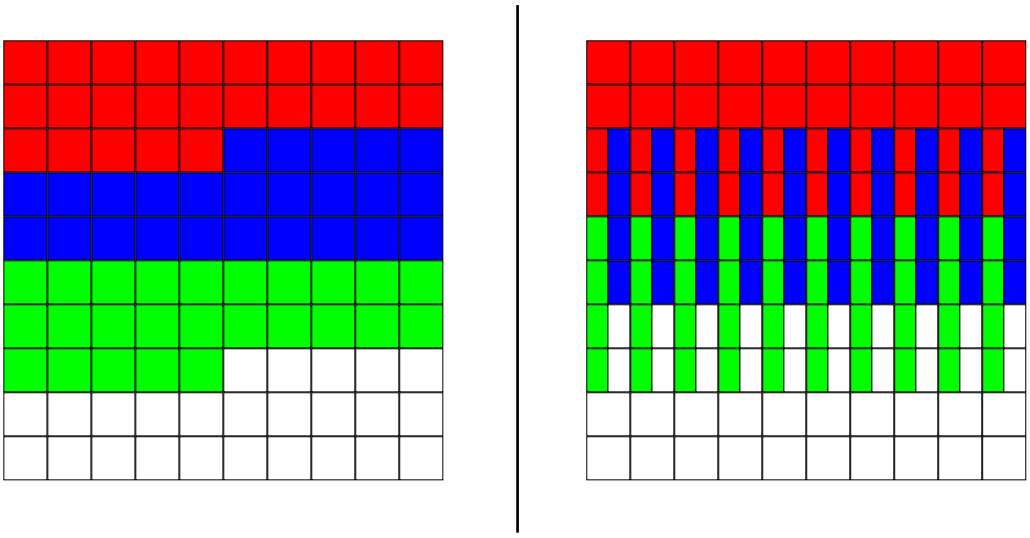

Figure 6.1.: Left: The split of a single matrix among four processes, with the memory allocation done as shown in Listing 6.1.
Right: The matrix values that are read by each given process to calculate the results of a single iteration for the same matrix. Each colour represents a different rank. Split squares are needed by two ranks.

When transforming the kernel with CATO, both matrices are split among the processes in an equal manner. As Listing 6.1 shows, the matrices are allocated as one-dimensional arrays. This allocation leads to a split of the matrices as is displayed in the left half of Figure 6.1. The contiguous nature of the underlying array allows for splits within rows. For this particular example, a matrix size of $10 \times 10$ is assumed, with four processes being used in total. The right half of the figure displays the matrix elements that a given process needs access to in order to finish the calculations of a single iteration.

CATO distributes the iterations of a parallelized for-loop in blocks of equal size. However, only the iterations of the loop immediately preceded by the `for` pragma are subject to this distribution. In Listing 6.1, the iterations that are distributed are those of the outer loop. Thus, a single distributed iteration always contains the complete inner loop, iterating over complete rows. Therefore, in the example featuring four processes on the $10 \times 10$ matrix, each process is assigned two iterations of the outer loop.

Due to the shape of the 5-point stencil, iterating across a single row requires the values of the rows above and below the target row as well. When comparing the two sides of the figure, it is apparent that most of the elements that a process requires are stored locally, with the exception of the halo lines per process, along some additional elements in certain cases. Due to the square nature of the matrix and CATO's distribution of the iterations and the memory, this observation also holds true for larger matrices. However,

if the matrix size is fixed and only the number of processes increases, the percentage of local elements among those that a process requires for its calculations will be lower, leading to an increase in the required inter-process communication.

### 6.1.1. Strong Scaling Experiments

First, CATO's baseline performance is evaluated on a smaller problem in a strong scaling experiment. The stencil kernel for this evaluation iterates 10 times over 250 MB of matrix data (500 MB in total for both matrices). As the overall amount of necessary memory is small, the experiments were conducted on the nodes featuring 256 GB main memory.

#### Time to Solution

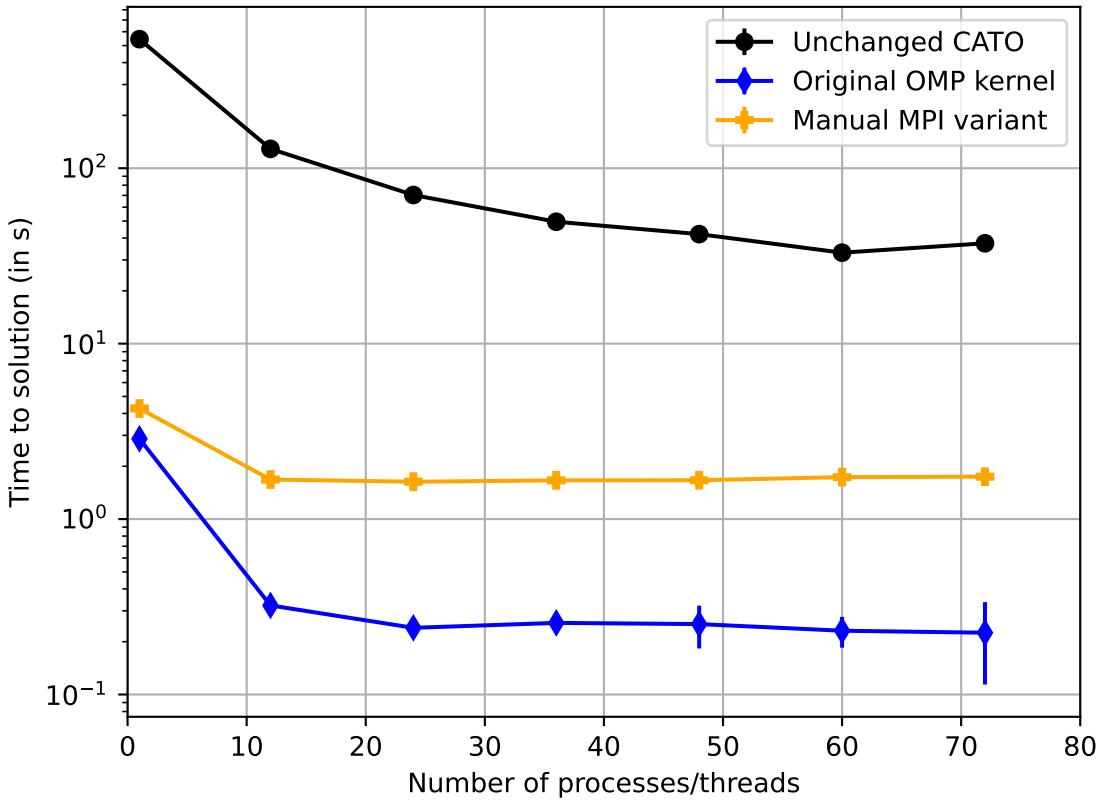


Figure 6.2.: Time to solution for different versions of the stencil kernel in a strong scaling experiment. Each colour represents a different implementation of the kernel.

Figure 6.2 displays the time to solution for three different implementations of the stencil kernel with the aforementioned parameters on a logarithmic scale. It features both the original OpenMP kernel and the transformed version, alongside a manually built

MPI version. The transformed version is referred to as "Unchanged CATO", implying that none of the proposed cache mechanisms were active for these measurements. The marks represent the process/thread numbers at which the measurements were taken. In regards to the absolute time, the difference between the OpenMP original and the manual MPI version is rather small. The OpenMP version's run time ranges from a mean value of approximately 2.87 seconds at one thread to 0.23 seconds at 72 threads. Adding more threads beyond 24 has next to no impact on the run time. The MPI version's time to solution starts at 4.27 seconds for one process and reaches a mean of 1.75 seconds for 72 processes. The MPI version reaches a minimum of 1.63 seconds at 24 processes. The added overhead from adding more processes and subsequently more communication does not benefit the time to solution for data of that size.

Both the manual MPI version and the original OpenMP version of the stencil kernel do not benefit from extending the number of processes or threads beyond 24 for such a small-scale problem. The absolute difference in their respective run times is comparatively small as well. However, the transformed version of the kernel displays run times that are higher by multiple orders of magnitude. The transformed kernel requires more than 540 seconds to finalize the calculations when using a single process and reaches a plateau of approximately 37 seconds at 72 processes.

This discrepancy in the order of magnitudes regarding the time to solution of the transformed kernel on one side, and the original OpenMP kernel and the manual MPI version on the other will persist throughout the upcoming experiments. Therefore, any figures displaying the time to solution for these three versions of a kernel will do so on a logarithmic scale, thus better visualizing the differences between all three versions in a single figure. Figures that exclusively contain the time to solution for different configurations of CATO, without either the original OpenMP kernel or the MPI version present, will display their respective results on a linear scale.

The measurements displayed in Figure 6.3 highlight the impact of the individual cache components on the time to solution for the transformed kernel. Alongside the previously seen values for unchanged CATO, the figure contains the mean times to solution for different runtime configurations on a linear scale. There are three configurations in which only a single cache component is active, as well as the final configuration with all previously described caches running. The results for the configurations in which the read message and write message cache are active start at 12 processes, as these caches only contain data from remote processes. When examining the results of the measurements with an active write cache, it is apparent that the results are almost congruent with the unmodified version. This is an expected result for this particular stencil kernel with the given parameters, as there are only very few elements belonging to remote processes that are written to in each iteration. Consequently, the number of elements that can be aggregated with this caching technique is small. Observations for the configuration that features only the read message cache are largely similar. However, as the process count increases on this fixed problem size, the number of remote elements that each process requires for

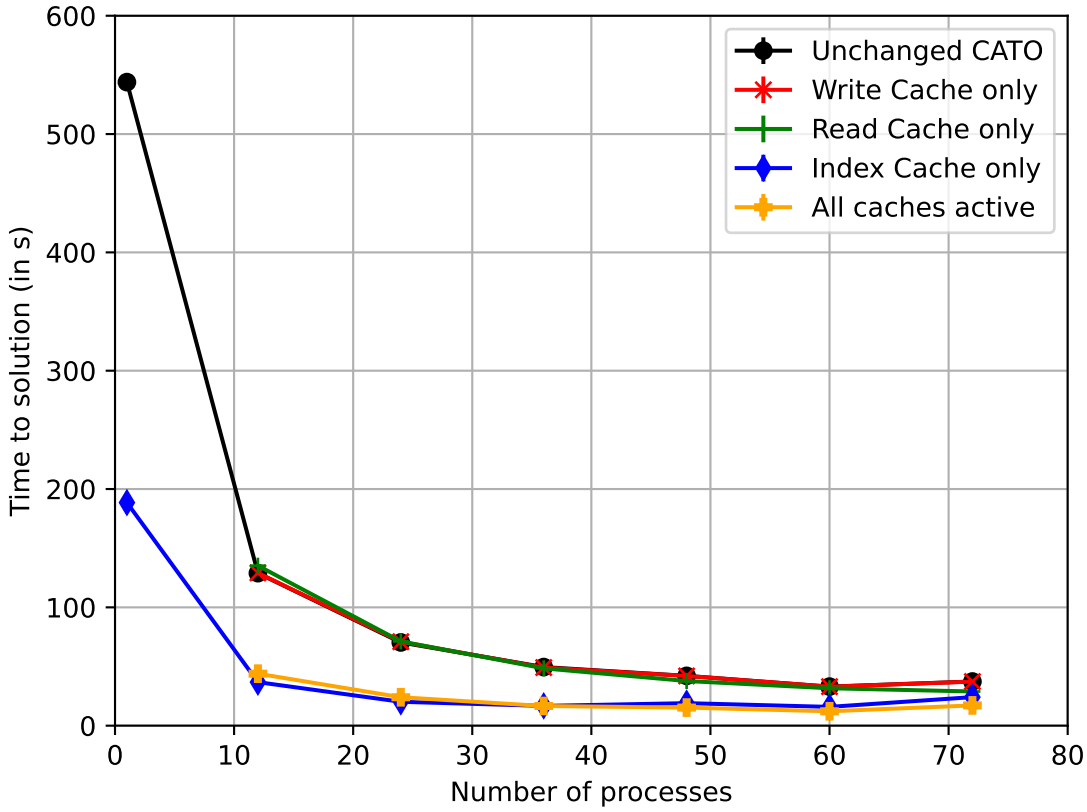


Figure 6.3.: Time to solution for different runtime configurations of the strong scaling stencil kernel after a transformation with CATO. Each colour represents a different configuration of active cache components.

its calculations rises. As such, when utilizing 60 or more processes, the time to solution for the version with an active read cache is less compared to the unchanged original. As expected, a larger number of load operations that target remote data benefit the read message cache for this particular kernel, due to the repeated loads of unchanged elements.

The index cache displays the largest impact of all cache components. The measurements for this component begin at one process, as the index cache is used to circumvent repeated address translations. In the case of local data, it even allows for the usage of a `memcpy` operation, which in turn also circumvents the need for any MPI functions. When activating only the index cache, the time to solution decreases by more than half of the original for most process counts, with the only exception encountered at 72 processes. This massive impact is also the driving factor behind the results of the measurements with all caches active, as the results for these two configurations are almost congruent as well.

During the evaluation of the experiments, the initial results for the active index cache and read cache were worse than what is currently displayed. Due to the implementa-

tion of the cache control in CATO, in which a cache is either active or inactive at all times, these two components were also active during the initialization of the matrices. The initialization procedure does not require a single load operation, meaning that the elements that were stored in the read cache were not accessed a single time during the initialization. Since the read message cache contents need to be dropped at the end of a `parallel` region in accordance to the OpenMP-conforming cache model, the stored contents were also not benefitial for the first iteration of the calculations, effectively wasting time by populating the cache during the initialization. The impact on the index cache was slightly smaller due to the similarity of the loop structure and the accompanying accesses to matrix elements in both the initialization and calculations. However, some of the entries that were touched once during the initialization were not required again later on, effectively wasting time and polluting the cache as well. If the initialization procedure and the subsequent accesses in the calculations showed a large discrepancy in the accessed elements, this negative effect would only be exacerbated. This motivated the implementation of four extra functions for CATO's cache control that can be inserted into user code. Both the read message and the index cache can now be enabled or disabled with a respective function. The measurements displayed above utilize these functions, the caches are disabled prior to reaching the main loop for the stencil calculations. Only then is the respective cache activated.

In order to understand the reason for the enormous impact of the index cache, the average times for a single access need to be investigated. The experiment with 12 processes was profiled for this purpose, Table 6.1 displays the results for the unchanged version for CATO without any software caching. The average time for fetching the correct `MemoryAbstraction` object, which consists of a pointer chase and, if necessary, index transformations, is at roughly 40 nanoseconds. The subsequent MPI calls require almost another 700 nanoseconds to complete on average, which leads to a sum of approximately 740 nanoseconds for a single access.

| **Index calculations** | **MPI functions** | **Total for one access** |
|---|---|---|
| 4.418e-08 s | 6.983e-07 s | 7.425e-07 s |

Table 6.1.: Average time to fetch the correct `MemoryAbstraction` object and for the subsequent MPI synchronization and communication in the unmodified version of CATO. The displayed results are the mean values of all accesses made by one process in an experiment with 12 processes total (evaluating more than 160 million accesses).

When activating the index cache, cache misses lead to only slightly longer access times, as displayed in the first row of Table 6.2. However, this difference is at a sub-microsecond scale. Cache hits for remote data display similar results though, even displaying larger times for the access, albeit at an even smaller scale, which harms the expressivity of these measurements. Regardless, it can be stated that a cache hit for remote data does

not have a measurable benefit over a cache miss. This is likely due to the shallow nature of the pointer chase. The memory is allocated as a one-dimensional array, which leads to the base pointer, as it is passed in the parameters to the custom access functions, already being the correct address of the `MemoryAbstraction` object. In cases where the memory is allocated differently or with higher-dimensional arrays, the observation regarding the benefit of cache hits for remote data is likely to change, as the pointer chase requires comparatively more steps.

However, when examining the results for cache hits on local elements, the reason for the index cache's impact becomes apparent. Fetching the element from the cache and conducting the `memcpy` operation requires only 157 nanoseconds, which is smaller than the other accesses by a factor of more than 5. Given the large amount of local elements that a process has to access during the calculations, reducing the required time for a single access by this factor in turn decreases the overall time to solution drastically, which matches the observations for Figure 6.3.

| **Cache hit?** | **Index calculations** | **MPI functions** | **Total for one access** |
|---|---|---|---|
| No | 9.514e-08 s | 7.236e-07 s | 8.188e-07 s |
| Yes (local) | 1.570e-07 s | - | 1.570e-07 s |
| Yes (remote) | 1.647e-07 s | 7.236e-07 s | 8.883e-07 s |

Table 6.2.: Average time to fetch the correct `MemoryAbstraction` object and for the subsequent MPI synchronization and communication when activating the index cache. Results in the *Index calculations* column for cache hits consist of both the cache search and the extraction of the content, if the search returned an element. For cache misses, the subsequent pointer chase is included in the results.

### Memory Consumption

Introducing multiple caches into CATO affects the overall memory usage of CATO. To examine this, the stencil kernel was used and the peak memory usage was tracked for the different configurations of the caches. Figure 6.4 displays the results for two different process counts. The memory usage of the original OpenMP kernels can be considered the bare minimum, as it equals the size of the two matrices at almost 500 MB. However, this should not serve as the baseline for comparisons to CATO, as the introduction of MPI in itself increases the memory consumption. Instead, when assessing the memory usage of CATO, the manual MPI variant should serve as the baseline, since going below the peak memory usage of the hand-crafted MPI stencil kernel is likely not possible for a tool with a more general purpose, such as CATO. Examining the results for 24 processes, the version of CATO that utilizes no caches displays a peak memory usage that is only 200 MB higher than the peak of the manual MPI variant, with 965 and

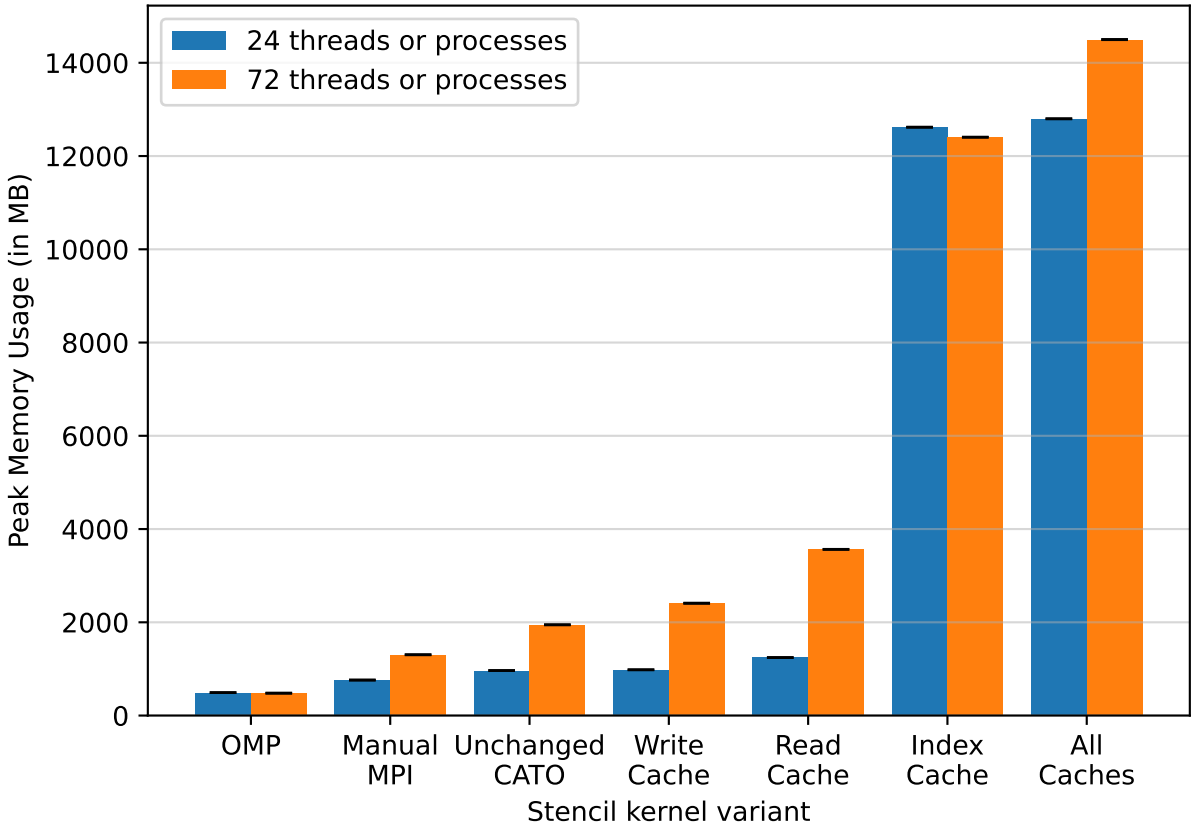


Figure 6.4.: Peak memory usage of the stencil kernel variants for 24 and 72 threads or processes respectively.

761 MB peak memory usage respectively. Similarly to the effect on the time to solution, activating the write message cache at 24 processes barely influences the results compared to the unchanged version, only increasing the peak memory consumption by 16 MB. This can be attributed to the high number of local elements that a process writes to, making the write message cache less effective. Activating the read message cache increases the memory consumption noticeably, peaking at 1.24 GB. This increase is still not drastic, which can also be attributed to the large fraction of local elements that a process loads from during the calculations. However, due to the iterations at the beginning and the end for any given process, said process has to access some remote data in load operations. These elements from the halo lines are only read from and not written to though, which explains the comparative increase in peak memory consumption between read message and write message cache. Activating the index cache increases the peak memory usage substantially, which consequently also influences the memory consumption when all caches are activated. The memory usage peaks at over 12.5 GB. This is an increase of the original problem size by a factor of 25 when looking only at the memory that is allocated in the stencil kernel itself. When comparing it to the memory usage of the manual MPI variant, it still is a substantial increase, by a factor of 16.

When examining the results for 72 processes, the baseline that the manual MPI variant sets is increased to 1300 MB. With this process count, unchanged CATO exceeds the peak of the baseline by roughly 600 MB. Due to the increase in non-local elements that need to be accessed with a higher process count at a consistent problem size, the memory consumption of the read message and the write message cache increases. For example, activating just the read message cache nigh doubles the peak memory usage instead of increasing it by roughly a quarter, as it was the case at 24 processes. The size of the index cache is almost unchanged compared to 24 processes, with only a small decrease. The memory consumption for the configuration in which all caches are active now exceeds 14 GB. This comparative increase can be attributed to the observed increase in size for both the read and the write cache.

Concluding the observations regarding the memory consumption of the caches, it is apparent that the improved time to solution comes at the cost of a substantial increase in memory usage. However, due to the abundance of main memory in an environment where the application is not constrained to a single node anymore, this tradeoff is acceptable. Furthermore, the size of the read and write cache are directly related to the number of non-local elements a process needs access to, hence the increase in size for larger process counts. The size of the index cache however seems to rely only on the problem size as a whole, regardless of data locality. It is also by far the most memory-consuming of the proposed caching mechanisms. For cases in which such an increase in memory consumption is not feasible, the cache size can be limited, in which case the LRU policy will be used to manage the index cache content.

### Other Cache Statistics

The previous runs with 24 and 72 processes respectively were also further instrumented to record certain runtime statistics regarding the different cache components.

When activating the write message cache, all stores to remote data were aggregated and only written to the target process upon encountering a flush operation. When running the transformed stencil kernel with 24 processes, a total of 2,511,506 writes to remote targets were grouped. Instead of requiring one `MPI_Put` operation, alongside synchronization calls, for each of these, the write message cache aggregated them into just 412 write calls. While this is a noticeable decrease in the total number of messages, the impact on the actual time to solution remains small. Similarly, running the application with 72 processes led to a reduction of originally 41,165,138 remote stores to 1276 instead. The effect on the time to solution remained unnoticed. Given these metrics though, it is reasonable to assume that the write message cache will have a significant impact on larger problem sizes. This will be investigated further in the weak scaling experiments that will be discussed later on during this evaluation.

Running the 24-process experiment with the read message cache on did not show any

benefits regarding the run time. A hit/miss ratio was calculated during the profiling runs for the read cache. When summing all load accesses across all processes, the total number of such accesses is 1,561,287,200. The read message cache is checked at the start of each of these accesses, regardless of the data locality, as this is only determined later on in the procedure. Of these 1.5 billion accesses, only 7,750,930 yielded a cache hit, leading to a cache hit ratio of 0.5 %. When running the experiment with 72 processes, the overall number of accesses remains unchanged for obvious reasons, but the number of cache hits increases to 133,926,080. This yields a cache hit rate of 8.58 %. These hit rates are the reason for the effect on the time to solution, or a lack thereof. The hit ratio for 24 processes is too low to impact the run time, but an increase in the number of non-local elements that need to be accessed also leads to an increase in the read cache hit ratio. At 72 processes, the hit ratio is high enough to positively influence the time to solution. These results suggest that the peak efficiency of the read cache is not captured by the stencil kernel in its current form, where there is only a small absolute number of remote read accesses.

The index cache has the potential to influence every single access in an experiment. Each load and store in the parallel regions first checks the index cache for an entry and will, in case of a cache miss, add a corresponding element to the cache afterwards. Due to this, the cache hit ratio is calculated over the total number of loads and stores to the matrices in parallel regions for this application, which amounts to 1,873,544,640 accesses. For the experiment with 24 processes, the index cache displays a hit ratio of 96.64 %. The misses can be attributed to the first iteration, in which the cache is populated. The initialization of the matrices did not have any existing cache entries to work with either, and also did not create any cache entries due to the previously mentioned implications on the performance that led to the deactivation of the caches prior to the actual calculations. Furthermore, 95.95 % of the cache hits are connected to local elements, which explains the large impact on the time to solution. Running the experiment with 72 processes yields a similar overall cache hit ratio of 96.58 %. However, only 85.95 % of the cache accesses yielded a local element. This follows the observation of the shrinking number of local accesses for larger numbers of processes on fixed problem sizes.

When activating all caches at once, the overall number of cache hits for the index cache is reduced slightly due to the read message cache hits. If an entry is found in the read cache, there is no need for any further operations, as the content is simply copied into the target buffer. This suggests that for kernels with many accesses to remote data, the read message cache could lessen the load of the index cache. The corresponding index cache entries could even be evicted early to lower the overall memory consumption of the index cache component.

### Prefetching

In its current iteration, the number of elements that CATO prefetches into the read cache is controlled via an environment variable. As such, it is the user's responsibility to

set it to the appropriate value. The optimal value for this variable depends on many different factors, such as the problem size, the distribution of memory and the underlying hardware. Therefore, a good value is difficult to predict beforehand and likely requires experiments with different values. Choosing a value that is too small will not lessen the overall number of necessary message passing calls significantly enough to cause an improvement in the run time, while a value that is too large may cause cache pollution for cases in which not all elements that were prefetched are actually used by the given process.

For testing the prefetching capabilities on the stencil kernel with the previously used parameters, five different values were chosen. Upon accessing a remote element, the message retrieved via `MPI_Get` was extended by including the next 0, 128, 256, 512 or 1024 elements respectively. However, none of the values had a positive impact on the time to solution, but subsequent profiling revealed an impact on the cache hit ratio. The cache hit ratio for no prefetching is situated at 0.496 % in the experiment with 24 processes. All of the investigated prefetch values increased the hit ratio to approximately 0.72 %, with the larger prefetch values increasing this percentage only by miniscule amounts. While this is a large relative increase, the overall hit ratio is still too small to affect the time to solution. Similar results are observed on the experiment with 72 processes. No prefetching led to a cache hit ratio of 8.578 %. The tested prefetching values raised this percentage to 11 %. Once again, this is a noticeable relative increase, but without an effect on the time to solution. As with the read message cache in general, the impact of prefetching hinges on the absolute number of remote accesses, which appears to be too small in the stencil kernel with the current problem size.

### 6.1.2. Weak Scaling Experiments

The previous experiments only utilized a small amount of memory and were also confined to a single node. The results suggested that especially the read and the write cache would benefit from a larger problem size. As such, the following weak scaling experiments were conducted to further investigate this hypothesis. The experiments in this subsection were conducted on the nodes featuring 512 GB main memory. The problem size is set to 80 MB of matrix data per process, split equally among the two matrices.

Due to compatibility problems with Intel MPI and the AMD CPUs in Levante, the configuration of the runtime environment differs from the one used for the single-node strong scaling experiments. More information is provided in Appendix A.

#### Time to Solution

The times to solution for process counts ranging from one to 256 for the version transformed by CATO as well as for the manual MPI version are featured in Figure 6.5. The graph also contains the results of the unchanged OpenMP stencil, but only until a thread count of 128 is reached. Going beyond this process count required a second node for the two other versions. As with the strong scaling experiments, both the manual MPI

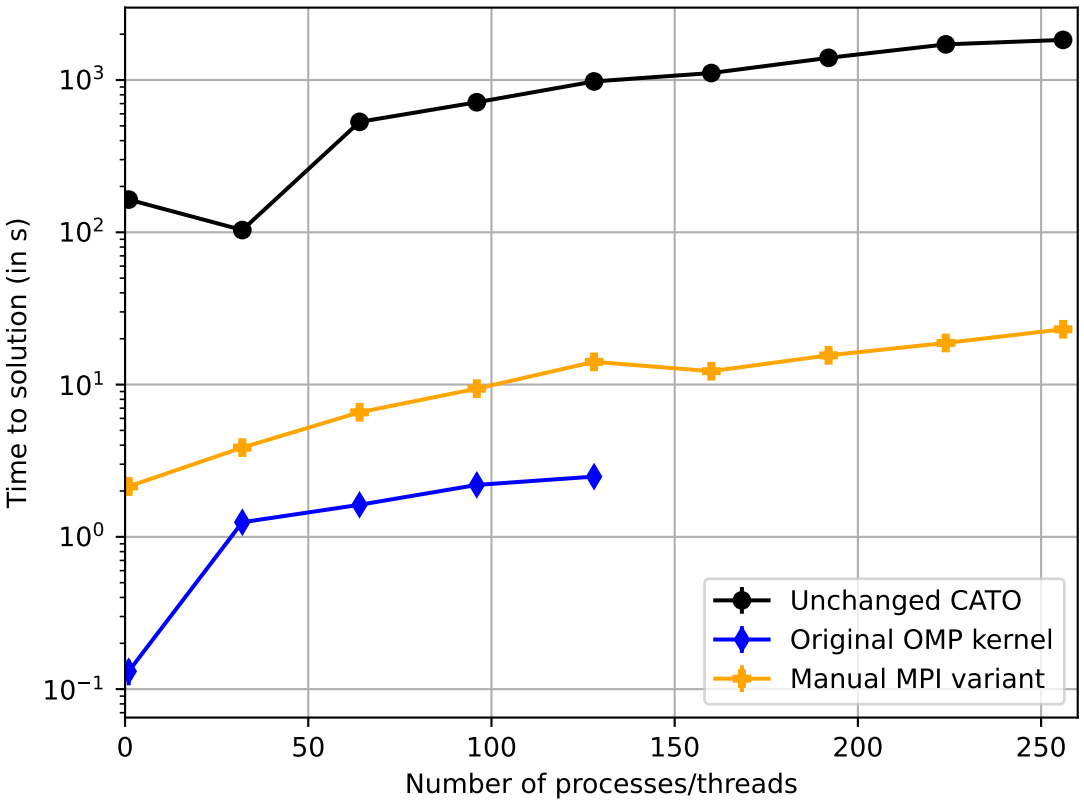


Figure 6.5.: Time to solution for different versions of the stencil kernel in a weak scaling experiment. Each colour represents a different implementation of the kernel.

version and the OpenMP version complete in a few seconds. The MPI version peaks at 23 seconds for the largest problem size at 256 processes and displays its minimum at 2.1 seconds for one process. The OpenMP version peaks at 2.5 seconds for the problem size at 128 threads, while its minimum is located at one thread, requiring only 0.1 seconds to finalize the calculations. The unchanged CATO version displays a rising tendency in the time to solution when applying weak scaling as well. Transforming the stencil and using only a single process yields a time to solution of 163 seconds. A peak is reached at 1834 seconds at 256 processes with the largest problem size. As with the strong scaling experiments, all of the run times of the transformed version are larger by multiple orders of magnitude when compared to the other two versions. A conversion to a linear scale reveals that the curve corresponding to the time to solution of the transformed version is considerably steeper than the curves of its two counterparts.

Figure 6.6 contains the times to solution for the same problem sizes with active cache components. As the problem size increases, an active write message cache reduces the time to solution visibly when compared to the baseline of unchanged CATO, while the impact is less noticeable at smaller problem sizes. The read message cache displays a similar behaviour, but it decreases the run time by a larger absolute amount. This is

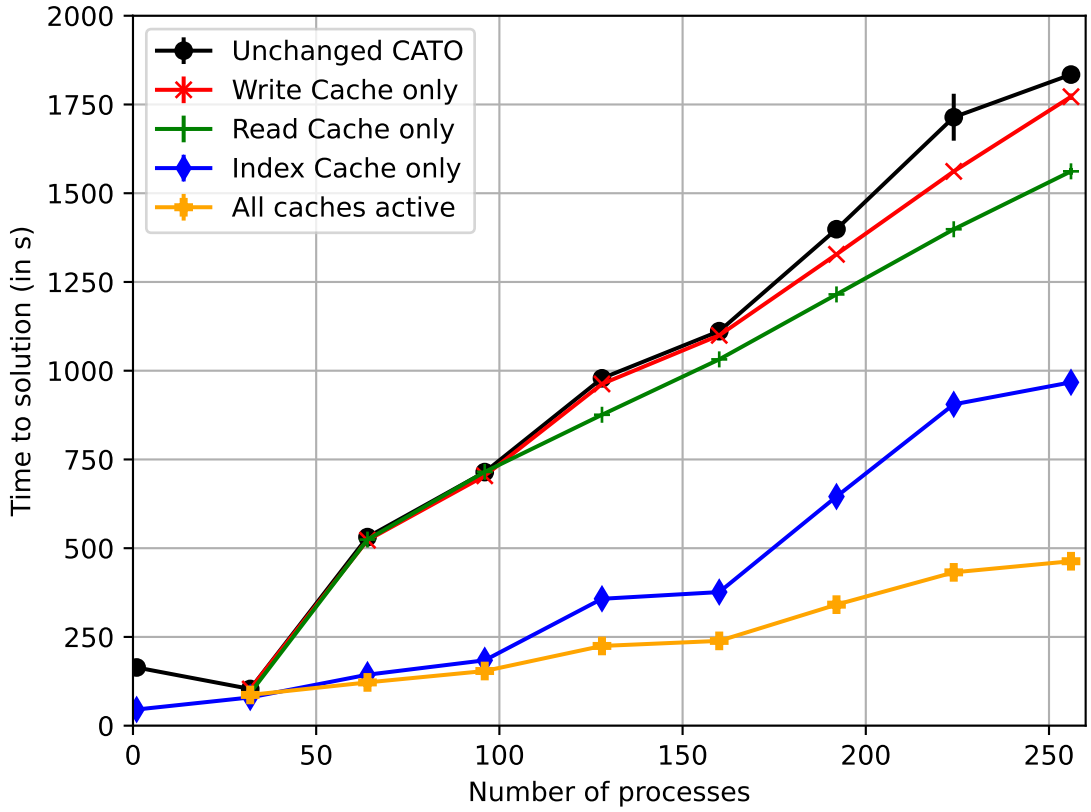


Figure 6.6.: Time to solution for different runtime configurations of the stencil kernel after a transformation with CATO, using a weakly scaling problem size. Each colour represents a different configuration of active cache components.

expected due to the inherent imbalance of loading and storing data in the stencil kernel. As hypothesized earlier, an increase in problem size benefits these two caches. The index cache reduces the run time the most once again, peaking at a time to solution of 967 seconds. When activating all caches, the time to solution is decreased even further. The times to solution for smaller process counts are still very close to the index cache results, but as the problem sizes increase and with them the impact of the read and write message caches, the run times for the experiments with all cache components active drop noticeably below the index cache results. Due to the large amount of local data, the index cache is still the highest-impact component, but as the absolute number of remote elements increases, the other cache components start to lower the time to solution more perceivably.

### Memory Consumption

For the weak scaling experiments, the memory utilization was tracked on the experiments with 128 and 256 processes, as shown in Figure 6.7. For 128 processes or threads, the raw matrix data occupies roughly 10 GB. This is almost mirrored by the memory

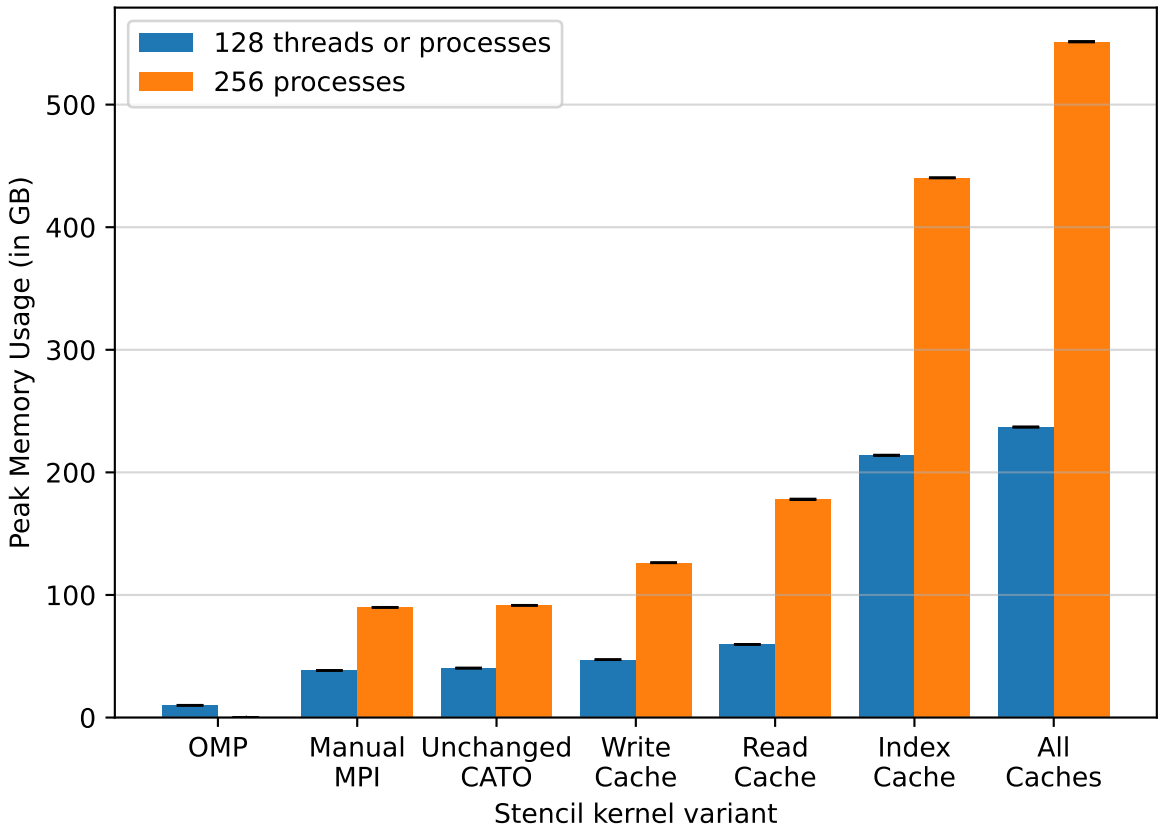


Figure 6.7.: Peak memory usage of the weak scaling stencil kernel variants for 128 and 256 processes or threads, where applicable.

requirements of the OpenMP version with this problem size. The manual MPI version exceeds this amount almost by a factor of 4, using 38.4 GB of memory. Activating no caches, the transformed kernel is very close to this memory utilization with only 40.3 GB. Compared to the strong scaling results, the memory consumption is, relatively speaking, closer to the achievable minimum that the manual MPI version poses. Activating the write cache increases the required memory to 47.2 GB. At this process count, the write cache reduced the time to solution by 16 seconds, at the cost of 7 GB. The read message cache required more memory at 59.7 GB. The difference in memory consumption between the two cache components is caused by the same fact that causes the difference in time to solution, namely the imbalance between loads and stores. The index cache bloats the memory requirements, as was the case in the strong scaling experiments. Close to 213 GB are occupied. While this is still a large absolute increase, the relative increase is smaller when compared to the strong scaling experiments. The index cache raises memory utilization only by a factor of 5.3, compared to it not being active. In the strong scaling experiments, a factor of 13 was observed instead. Activating all caches leads to the maximum memory consumption for the 128-process experiment. The peak utilization was reached at 237 GB, which roughly equates to the sum of the differences in peak

memory utilization for individual cache components.

For 256 processes, the OpenMP version is not usable due to its limitation to a single node. The problem size is set to 20.48 GB. In these experiments, the observation regarding the small difference in memory consumption between the manual MPI version and CATO without caches applies as well. Activating the write cache leads to a more severe increase in memory utilization for this problem size, with 126 GB of occupied memory. Similarly, the memory consumption of the read cache is also higher in both absolute and relative terms. Both of these increases are attributed to the larger problem size, which caused an increase in the absolute number of remote elements a process requires for its calculations. The memory consumption of the index cache peaks at the maximum in this configuration for the individial cache components once again, reaching 440 GB. In relation to uncached CATO's memory requirements, this is an increase by a factor of 4.8 for this experiment. Activating all caches displays the largest memory requirements as well, with an absolute value of 551 GB, which also is approximately equal to the sum of differences for the peaks of the individual components.

The observations regarding the memory consumption of the cache components display a similar trend to those made in the strong scaling experiments. The index cache increases the memory consumption the most. However, in relative terms, the increase is smaller than it was in the strong scaling experiments. The other two cache components also incurred a larger peak memory consumption, but with the larger problem sizes, this increase in memory consumption led to a noticeable decrease in the time to solution, for both components individually as well as when all components are active simultaneously.

### Other Cache Statistics

The previous experiments that tracked peak memory consumption were once again used to gather runtime data regarding the caches, from which certain insights can be derived.

The experiment with 128 processes features a problem size of 10 GB. Compared to the strong scaling problem size, this is an increase by a factor of 20. The larger problem size also leads to an increase in remote elements that need to be written to during the calculations. As such, the write message cache aggregated 581,372,550 individual stores into 2284 larger MPI communication operations. Compared to the strong scaling experiments, the total number of aggregated writes is increased by a factor of more than 230. The total number of the instead executed `MPI_Put` operations with larger messages only increased by a total of 1872. Similarly, the larger problem size in the 256-process experiment aggregated even more elements. 2,655,990,318 single remote writes were aggregated into only 4588 messages. The noticeable impact of the write message cache in the weak scaling experiments is attributed to the observed increase in the message sizes by large margins while simultaneously only increasing the total number of sent messages by a small absolute value.

As with the write cache, the read message cache also improved the time to solution in the weak scaling experiments, even more so than the write cache did. In the 128-process experiments, the cache hit ratio is still very low, with only 5.97 % of searches returning an entry, however the absolute number of hits is increased considerably when compared to the weak scaling experiments. The cache hits amounted to a total of 1,911,577,840. Compared to the strong scaling experiments, this absolute value is larger by multiple orders of magnitude. The experiment with 256 processes features an even larger absolute value of 8,792,404,540, while also showing an improvement in the cache hit ratio. 13.74 % of all custom loads found the corresponding element in the read message cache. This improvement is caused by the necessity to read more remote elements during the calculation for the larger problem size. Increasing the problem size further alongside the process count is likely to improve the hit ratio further, but getting close to 100 % is not possible due to the large amount of local accesses within the calculations of a process.

The index cache reduced the time to solution by the largest amount. As was the case in the strong scaling experiments, the hit ratio is very high for both instrumented weak scaling experiments, hitting 96.6 %. In the 128-process experiment, 92 % of the index cache hits were for local elements, while this number dropped to 82 % for the larger experiment with 256 processes, further showing that the number of necessary remote accesses rose alongside the problem size. The index cache accumulated 37,100,428,992 hits total in the smaller experiment with 128 processes, the larger experiment almost doubled that count. As with the strong scaling experiments, the large hit ratio along the large amount of local elements that were accessed through the index cache led to the sizeable decrease in the time to solution.

Activating all caches at once shows the same behaviour that was observed during the strong scaling experiments. The presence of the read cache reduces the total number of index cache accesses, as load accesses to previously fetched remote elements are served via a read cache element.

As suspected, the larger problem sizes especially benefitted the read and write message cache. This is caused by the larger absolute number of remote accesses, leading to more entries for the read cache and more potential stores to aggregate for the write cache. The index cache performed very well once again, mainly due to the large amount of local elements, albeit at a high memory cost.

### Prefetching

Applying the prefetching mechanism to some of the problem sizes of the weak scaling experiments led to unexpected results. In particular, the experiment configuration of 128 processes on 10 GB of matrix data was repeated with varying element counts for prefetching. During these multiple experiments, the element counts ranged from a minimum value of 8 up to a maximum value of 16,384. A single line of the underlying matrix contained 25,298 elements. Without prefetching, the read cache achieved a hit

ratio of 5.97 %. Activating the prefetch mechanism improved the hit percentage to at least 7.39 % for the smaller element counts, and up to 7.59 % for the larger values. In absolute terms, the number of read cache hits improved by more than 450,000,000 hits. While this is a notable increase in the hit count, the time to solution did not improve. This was tested for both the strong scaling and the weak scaling environment configurations (see Appendix A). Instead, the run time rose by 25-30 seconds, regardless of the utilized prefetch value. Given the unexpected nature of these results, further investigations are warranted.

#### Examination of the Prefetching Performance

The investigation of the lackluster performance of the prefetching mechanism is conducted on the 128-process experiment, comparing run time statistics of different cache and prefetch configurations on the given problem size. These statistics are gathered by adding manual profiling code to CATO for this specific purpose, capturing the run times of individual iterations of the stencil. Each execution of the stencil kernel features 10 iterations in total. The results displayed in the following figures are collected from three individual experiment runs for the given cache configurations. The error bars visualizing the standard deviation in the following figures are purposefully faint to not impede with the visual clarity.

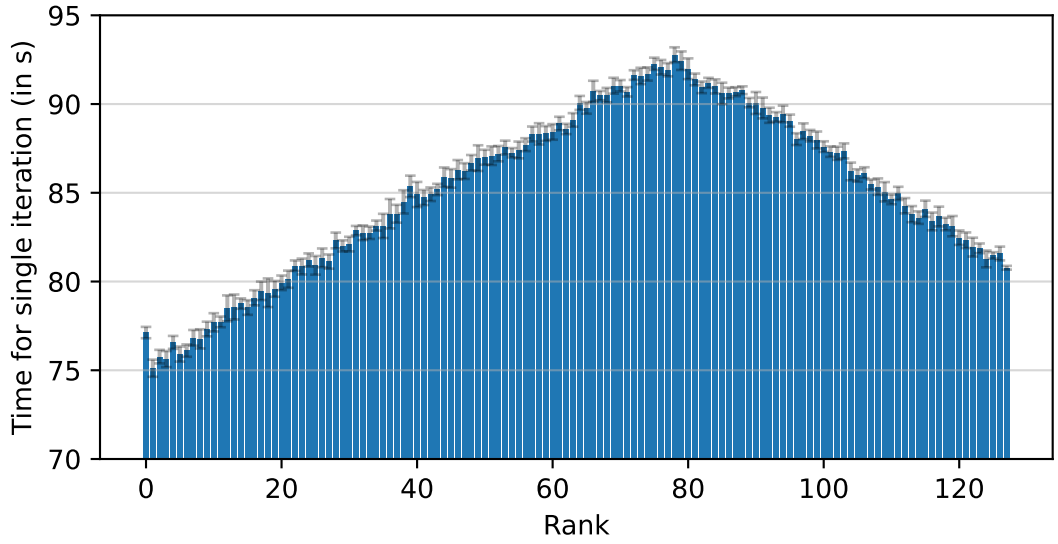


Figure 6.8.: Average per-rank run time for a single iteration of the stencil kernel without any active cache components.

Figure 6.8 contains the average run time for a single iteration of the stencil kernel for the given problem size and process count. No caches were active while capturing these statistics. Note that each iteration must be succeeded by a barrier synchronization.

Thus, the total run time for a given iteration is dictated by the slowest process, with the faster processes waiting idly at the barrier. On average, the first ranks require less time to complete an iteration than the processes with higher ranks. The mean run time steadily increases and reaches a maximum of 92 seconds at rank 78. Beyond these ranks, the average run time steadily declines. Such a pattern points towards a load imbalance. However, this seems contradictory to the distribution of iterations as it is conducted by CATO. Each rank is assigned the same amount of matrix rows to iterate over. If the total number of matrix rows is not divisible by the number of processes, the remainder is distributed by assigning an additional matrix row to the lower ranks. Thus, the number of assigned matrix rows differs by one at most, which is equivalent to a difference in workload of approximately 0.5 % between processes for this problem size.

The cause of this load imbalance cannot be attributed solely to the distribution of the iterations over the matrix, but is rather a result of the different nature of the accesses to individual elements. Due to the even distribution of the iterations, each process accesses the same amount of elements, barring the previously established 0.5 % difference. However, the number of accesses to local and remote elements differs per process.

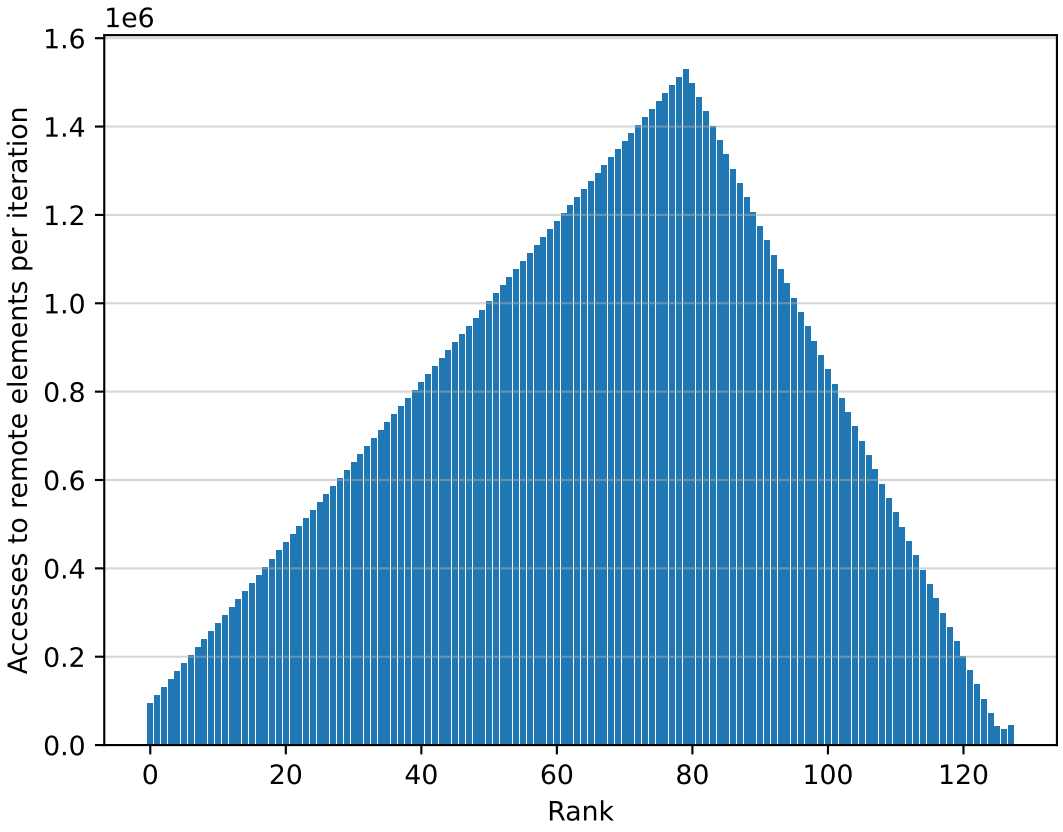


Figure 6.9.: The cumulative number of remote accesses during a single iteration of the stencil calculations, displayed per rank.

Figure 6.9 displays the number of remote accesses that a given rank has to conduct within a single iteration. The captured accesses form a pyramid shape, albeit with a tilt towards the right side. The rank with the largest amount of remote accesses, thus forming the tip of this pyramid, is rank 79. CATO distributes the memory for the matrices in blocks of equal size, applying the same logic that underlies the distribution of loop iterations. If the number of elements cannot be split without a remainder, the first $n$ ranks are assigned one additional element, where $n$ is the remainder of the division. If the number of elements and iterations used in this experiment configuration was divisible by the number of processes with no remainder, the nature of the stencil access pattern would have resulted in a symmetrical shape for the remote accesses. However, the utilized experiment configuration does not operate on such a number of elements or iterations[1]. Thus, the first ranks store more data than the latter ones, while also being assigned an additional matrix row to iterate over. This combination causes an uneven distribution of remote accesses, ultimately resulting in the visible tilt of the pyramid shape in Figure 6.9, and the subsequent load imbalance. This irregularity is especially pronounced within the last four ranks. Each of the mentioned ranks conducts considerably fewer remote accesses than their counterparts at the other end of the matrix in the ranks 0 through 3, differing by almost 100,000 remote accesses per iteration.

Compared to the previously discussed remote accesses, the number of local accesses is larger by multiple factors for each of the 128 processes. Thus, a figure containing the local accesses would display a shallow V-like shape, as the total number of accesses is approximately equal among the different processes. Comparing the pyramid-like shape to the average run time per iteration, especially regarding the location of the maximum value and the sign of the gradient between consecutive ranks, an overlap is found. A large number of remote accesses appears to negatively impact the run time of a stencil iteration. This is a sensible explanation of the perceived load imbalance, as the distribution of remote and local accesses is one of the only few differences between the processes in this transformed stencil application. However, there is one noticeable irregularity. The processes with the largest ranks display the lowest number of required remote accesses, but their respective average run time per iteration is comparable to ranks featuring a number of remote accesses that is larger by a factor of approximately 20. Several tests were conducted to ensure that this irregularity is not caused by either chance or faulty hardware. These tests included repeated executions of the experiment on different nodes, with multiple different CPU bindings and also different process counts. The irregularity regarding the average run time of the last processes persisted throughout, pointing towards a different interaction between this stencil kernel and the CATO-induced transformation with an adverse effect on the time to solution, apart from the previously discussed amount of remote accesses. At present, this interaction remains subject to future investigations and will not be discussed further in this thesis, instead

[1]Since this particular stencil kernel operates on square matrices, and the total number of iterations is equal to `DIM-2`, a perfectly even distribution of both the data and the iterations at the same time is impossible for 128 processes.

focusing on the effects of the read cache in the following.

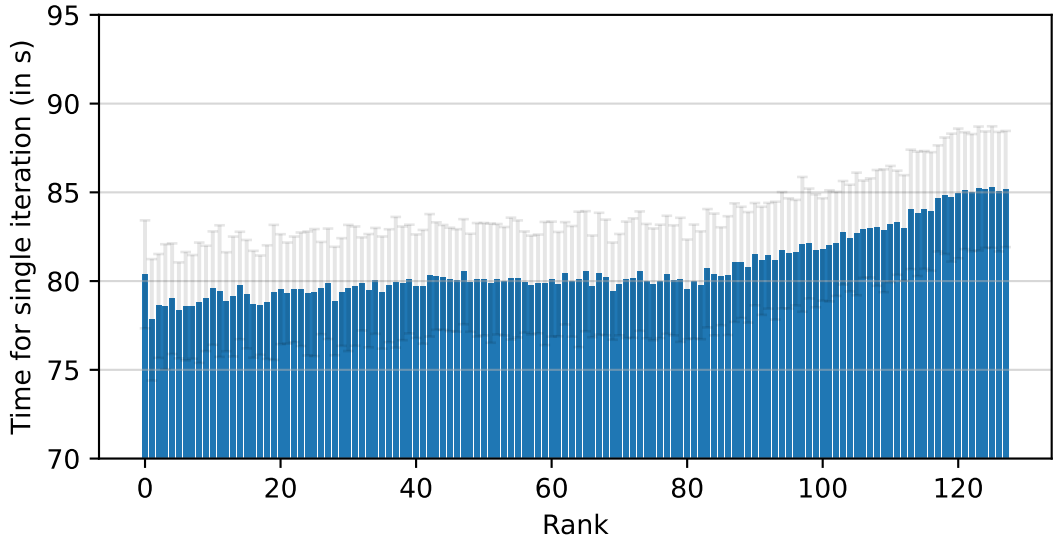


Figure 6.10.: Average per-rank run time for a single iteration of the stencil kernel with only the read cache active.

Activating only the read cache at this problem size led to a reduction of the time to solution by approximately 100 seconds. Figure 6.10 presents the average run time per iteration for this cache configuration. The first processes did not benefit from the presence of the read cache. They required roughly the same time to complete an iteration, barring some exceptions, such as rank 0. The ranks that formed the peak in the configuration without the read cache, which were located around rank 80, require significantly less time to complete an iteration on average. Instead of ranging from 75 to 92 seconds, all processes in the range of ranks 1 to 82 complete their average iteration in less than 80 seconds. Going beyond these ranks though, the mean run time steadily increases. Thus, the peak shifts from rank 78 to rank 125. The new maximum mean value is smaller though, shrinking from 92 seconds to 85. This decrease in the maximum value leads to the observed reduction in the total time to solution when activating the read cache.

Activating the prefetching mechanism did not improve the overall time to solution for any tested value of prefetched elements. Figure 6.11 displays the average run time of an iteration when activating the read cache and setting the prefetch value to 16,384 elements. Most ranks require less time to complete the average iteration when the prefetching mechanism is activated. All ranks up to 107 display mean run times of less than 80 seconds per iteration, presenting decreases of up to 5 seconds on multiple occasions. However, examining the ranks beyond 90, a steady increase can be observed, until the peak is reached at rank 125. As in the experiment with prefetching disabled, rank 125 shows the highest average time to complete an iteration at 85 seconds.

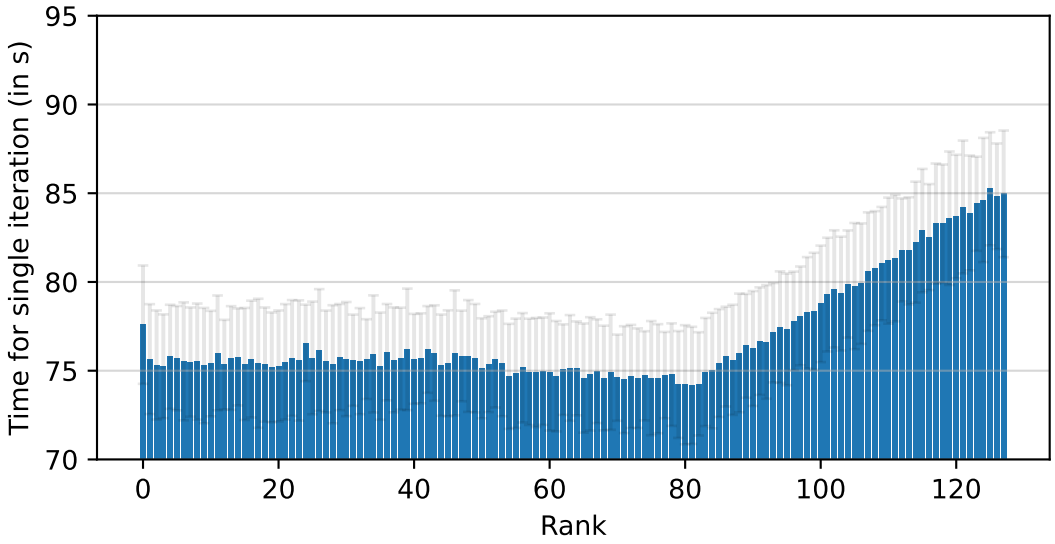


Figure 6.11.: Average per-rank run time for a single iteration of the stencil kernel with the read cache and the prefetching mechanism active. The prefetch value was set to 16,384 elements.

The read cache did not improve the average run time per iteration for the last ranks, neither with or without prefetching enabled. The cause of this can be discerned from the cache hit statistics of the individual ranks. Figure 6.12 superimposes the number of cache hits per rank for the two experiments in question. Both cache hit statistics form a pyramid shape that is similar to the number of total remote accesses for this problem size. Activating the prefetching mechanism raises the total number of cache hits for every rank. This effect is most pronounced for the processes around rank 80, causing differences of approximately 7.5 million for certain ranks. These additional cache hits positively affect the run time, as the average iteration at these ranks completes roughly 5 seconds earlier than without prefetching. For the last three ranks though, the initial number of cache hits without prefetching activated is very low. Neither rank exceeds 1 million total cache hits in this experiment, rank 126 in particular does not achieve any cache hits at all. The reason for this total absence of cache hits is found in the distribution of the data alongside the iterations. The only accesses to remote elements that rank 126 conducts are accesses to halo lines, which are only accessed once by a process in any given stencil iteration. The strong scaling experiments already highlighted that such cache hit amounts are not substantial enough to cause a decrease in the time to solution, and as such, the low number of cache hits is the cause for the read cache's lack of impact on the last ranks in this experiment as well. Activating the prefetching mechanism only raises the cache hits per rank by about 344,000. The time saved from these extra cache hits is not significant enough to lower the average run time for an iteration, leaving the performance of the last ranks virtually unchanged by the read cache, regardless of prefetching.

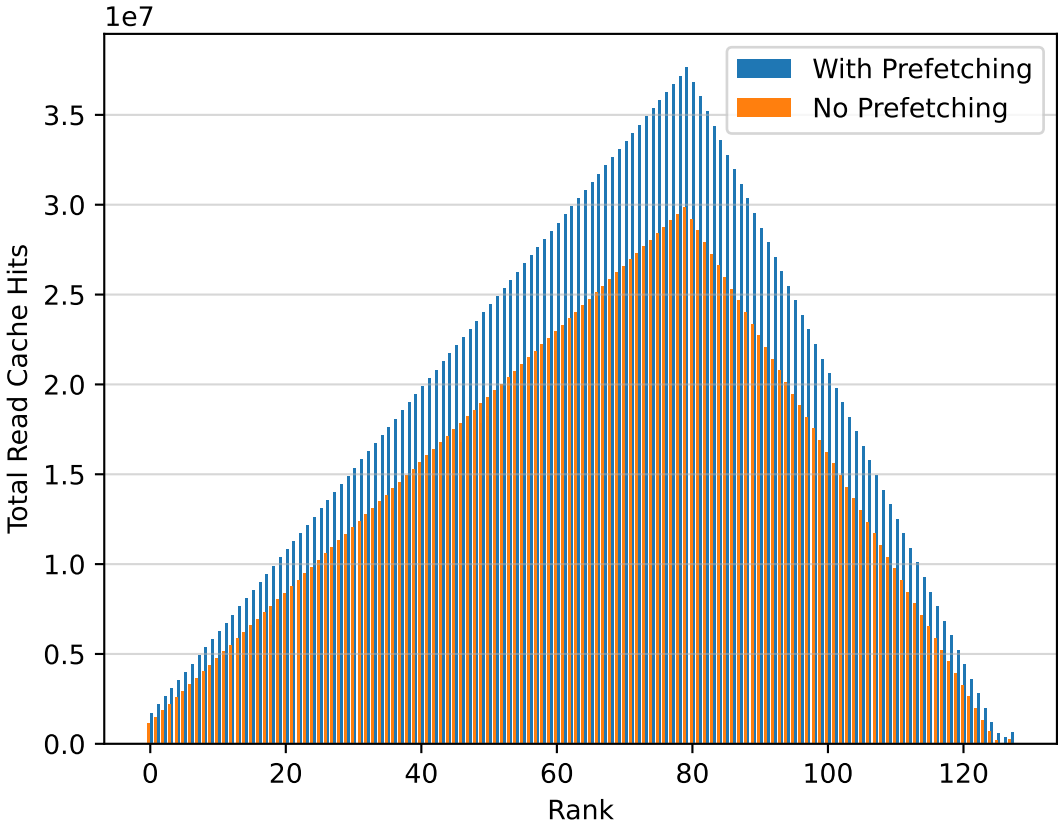


Figure 6.12.: Total read cache hits per rank on the stencil kernel, comparing no prefetching and prefetching with an element count of 16,384.

### Prefetching on an Alternative Stencil

The uneven distribution of both the matrix elements and iterations caused a noticeable load imbalance between the different processes, largely due to a vast difference in the amount of necessary remote accesses per iteration. If this load imbalance was removed by evenly distributing elements and iterations, the discrepancy between the fastest and slowest processes will likely shrink. In such a scenario, the read cache and the prefetching mechanism could be able to positively impact the average run time of each process, ultimately reducing the total time to solution.

To investigate this claim, a slightly adjusted problem size was tested. The previously used problem size was reduced by 53 KB to achieve a perfectly even distribution of the matrix data. The resulting number of per-rank remote accesses within an iteration is presented in Figure 6.13. Each rank has the same amount of remote accesses per iteration, with the exception of the last two processes. This exception is caused by the uneven distribution of the stencil iterations, as the last two processes are assigned one fewer matrix row than all other processes. The total number of remote accesses per process is very low though. By comparison, with the previously used problem size, rank 0

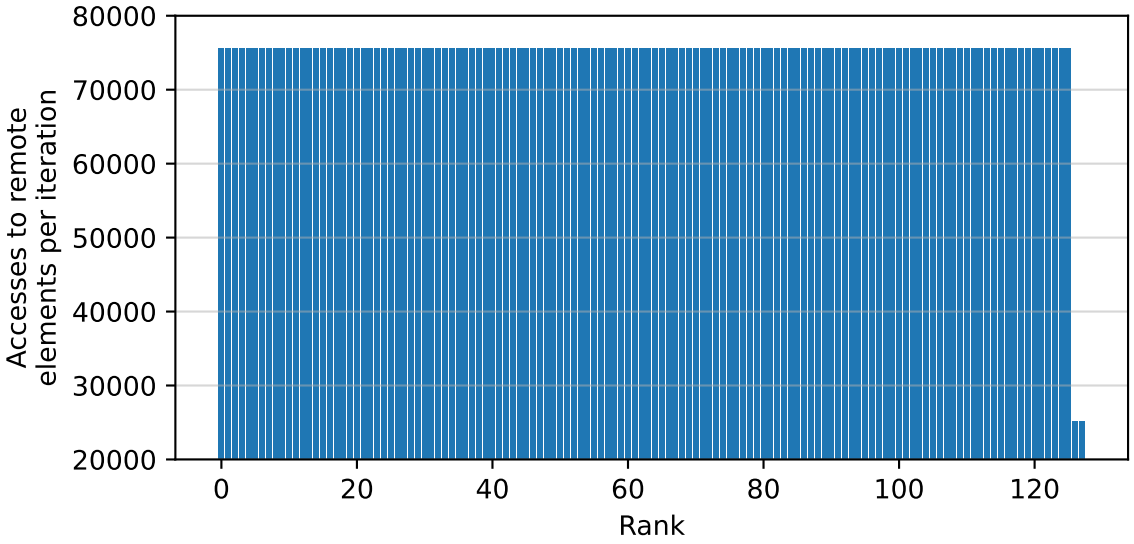


Figure 6.13.: The cumulative number of remote accesses during a single iteration of the stencil calculations, displayed per rank. The problem size for this experiment was adjusted to achieve a perfectly even distribution of the matrix data.

had one of the lowest counts of remote accesses across all processes, with 94,000 accesses. For this problem size with even distribution, every rank displays an even smaller number, as almost all accessed elements, excluding the halo lines, are local. As such, the amount of remote accesses is too small for the read cache to improve the time to solution, both with and without the active prefetching mechanism. A substantially larger problem size, causing much larger halo lines, is likely necessary for the read cache to have a positive impact in this scenario with evenly-distributed matrices, however this experiment is out of the scope of this thesis.

Instead, a different version of the stencil kernel is now considered. The main difference to the previously used stencil implementation is the memory allocation. Instead of allocating the matrix as one continuous array, it is allocated in two steps: first, space for `DIM` pointers is allocated, with `DIM` being the number of rows and columns. In the next step, memory for `DIM` elements is allocated to each of these pointers. This pattern for allocating memory was already displayed in Listing 4.1. When allocating the memory according to this pattern, the matrix is effectively distributed by columns instead of rows. Therefore, iterating through a single row requires each process to communicate with all other processes. This lowers the discrepancy in the number of remote accesses between different processes, as they can now differ at most by $\texttt{DIM} \cdot \frac{n-1}{n}$ accesses for $n$ processes in total, which corresponds to the number of remote elements in a single row. Simultaneously, this distribution also boosts the total amount of remote accesses. This communication pattern, in which communicating with all other processes is required, is sometimes encountered in n-body simulations or similar kernels, where each element is

affected by every other element.

For this experiment, 128 processes executed the calculations on 4.3 GB of matrix data with column-wise allocations. Having the read message and write message cache active, this version of the kernel requires 254 seconds to terminate. Running the previous version of the stencil kernel with the approximately row-wise distribution of the matrix data on the same problem size and process count, a time to solution of 79 seconds is achieved. This comparative performance decline is expected and can be attributed to the vast increase in necessary communication with remote processes due to the different data distribution.

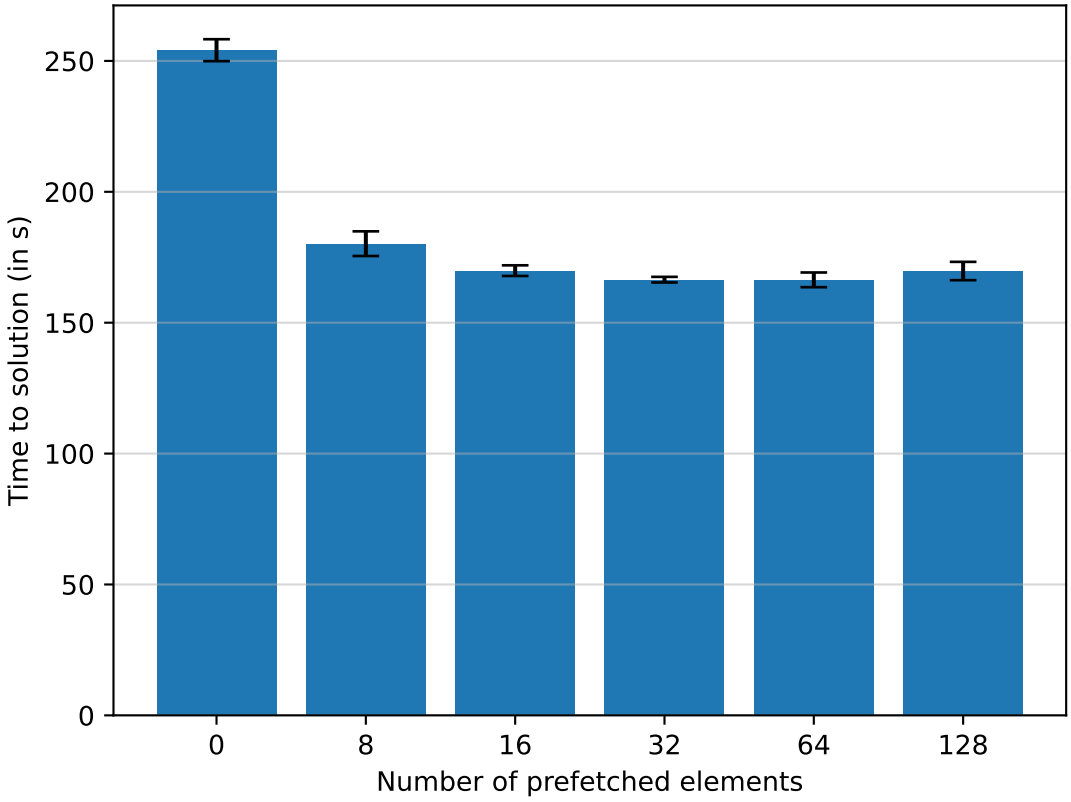


Figure 6.14.: Time to solution for the alternative version of the stencil kernel. The read and write message caches were activated, and the prefetch value was fixed to the number beneath the respective bar.

The prefetching mechanism was then evaluated on this alternative version of the stencil kernel. Both the read message and the write message cache were activated, and multiple different element counts for the prefetch mechanism were tested. The results are displayed in Figure 6.14. In this configuration, each row was split across the 128 processes, resulting in each process owning 128 elements of every row in both matrices. Therefore, the maximum number of elements that can be prefetched with a single MPI

call is 128. In contrast to the previous stencil kernel version, activating the prefetching with even a small number, such as 8 elements, improves the time to solution considerably. Going beyond 8 elements to 16 or 32 elements improves the run time even further, decreasing it from 180 seconds to 169 and 166 seconds respectively. Increasing the number of prefetched elements beyond 32 does not yield further improvements though, as a plateau is reached.

Examining the cache hit statistics, the baseline cache hit ratio without prefetching is at 79 %, with 10,608,995,950 absolute hits. By comparison, the kernel with the nigh row-wise distribution displayed a cache hit ratio of 0.46 %, with 61,921,440 absolute hits. When prefetching 8 elements per remote load access, this ratio is already increased to 96.7 %, thus adding more than 2 billion additional cache hits. Further increases to the number of prefetched elements improves this ratio by 2 %. A plateau is reached at a prefetch count of 32 elements. As with the times to solution, further increases to the prefetch count do not improve the hit ratio any further. All of the tested prefetch counts did not increase the memory utilization in a noticeable manner.

In the version of the stencil kernel with the column-wise distribution, the prefetch mechanism considerably improves the time to solution. The high number of required loads from multiple remote targets allowed for efficient prefetching. Furthermore, the load imbalance in regards to the accessed remote elements was reduced drastically. The worse performance of the later ranks was not reproducible with this adjusted version of the stencil kernel. The mean run times per iteration differed by less than 2 seconds among the 128 processes. This led to the read cache, and especially the prefetching mechanism, improving the performance of every rank, ultimately reducing the time to solution by a substantial portion.

## 6.2. SpMV

The second application for the evaluation of the proposed enhancements is a sparse matrix-vector multiplication. The sparse matrix is stored in the Compressed Sparse Row format (CSR), one of the most popular for storing sparse matrices [Saad, 2003]. In the CSR format, the nonzero elements are represented via three one-dimensional arrays. The first array, denoted as `data` for the purposes of this thesis, contains the nonzero values of the matrix, stored in order of appearance, row by row, from left to right. The second array, `indices`, contains the column indices for each element stored in `data`. Both arrays are of the same size, with their size being dictated by the number of nonzero elements in the matrix. The final array, `ptr`, contains pointers to the beginning of each row in the other two arrays. For example, the value stored in `ptr[2]` denotes the start of the original third row in the other two arrays. The length of this array is set to the number of rows plus one. The final element is the start of a fictitious row beyond the final one. It is used to mark the end of the matrix. Figure 6.15 provides an example of how a sparse matrix is represented with this format.

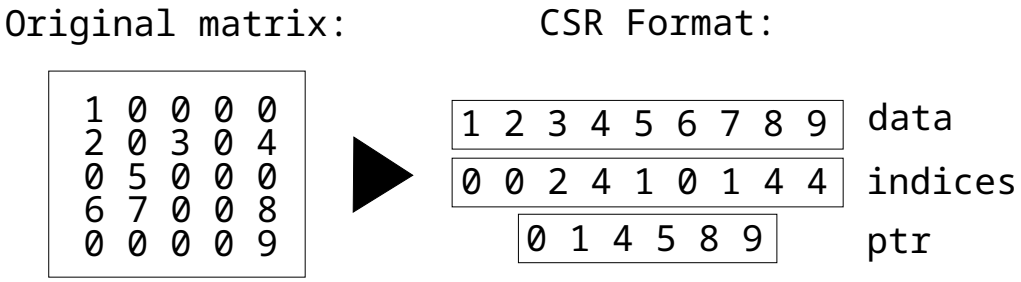


Figure 6.15.: A sparse $5 \times 5$ matrix, represented in the CSR format. `data` contains the matrix entries, `indices` contains the corresponding column indices and the `ptr` elements represent the index at which a respective row starts within the other two arrays.

For the purposes of this evaluation, a square sparse matrix, stored in the CSR format, is multiplied with a dense vector. The main loop for this multiplication is displayed in Listing 6.2. The outer loop is decorated with the `parallel for` directive. Before the loop, the matrix is generated by means of a pseudo-random number generator with a fixed seed in a sequential region. This generation is excluded from the following measurements to focus on the access pattern of the main loop and its interaction with the proposed enhancements.

```
1 for (size_t i = 0; i < DIM; i++)
2 {
3     result[i] = 0.0;
4     for (size_t k = ptr[i]; k < ptr[i+1]; k++)
5     {
6         result[i] = result[i] + data[k] * vec[indices[k]];
7     }
8 }
```

Listing 6.2: The main loop for the sparse matrix-vector multiplication. `DIM` equals the number of rows of the original matrix.

The access patterns for the individual arrays differ slightly. Both the `data` and the `indices` array are accessed sequentially, with each element only being loaded a single time. The `ptr` array is also accessed sequentially, but due to the pair-wise accesses in the inner loop, most elements of this array are accessed twice. The `vec` array contains the dense vector for this multiplication. Its elements are accessed randomly, according to the value stored in `indices[k]`. Thus, each element may be accessed multiple times, just once or not at all during the calculations. The `result` array is the only array that is written to. It is also accessed sequentially, however each element may be accessed multiple times as well, according to the difference of `ptr[i+1]` and `ptr[i]`. These access patterns differ greatly from the previously seen stencil code. In the stencil calculations,

almost each element of one matrix was loaded five times in a sequential manner, while almost each element of the second matrix was written to exactly once, also sequentially. Furthermore, the stencil iterated over these matrices multiple times. After executing the displayed main loop, the SpMV kernel terminates, thus reducing the number of repeated accesses to array elements substantially compared to the stencil.

This reduction in the repeated accesses is likely going to negatively impact the efficiency of the CATO cache components. The following subsections explore this hypothesis in the context of strong scaling and weak scaling experiments. Another reason that is likely to negatively affect the software cache perfomance is the distribution of the memory and the loop iterations. Starting with the loop iterations, each process is assigned a fixed number of iterations for the outer loop. For a total process count of $n$ processes, each individual process will be assigned a number of iterations equal to `DIM`$/n$. The first chunk is assigned to process 0, the second one to process 1 and so on.

The memory is distributed in the same manner. This leads to all accesses to the array that contains the product, denoted `result` in the above listing, being local accesses. Given that this is the only array that is written to, the write message cache will not be able to aggregate any messages, since there are none. Similarly, the read accesses are also going to be mostly local accesses. Exceptions are likely to occur at the start or end of the iterations within a given process due to the random nature of the matrix generation, but the elements inbetween are very likely to be local. The only remote elements that will be frequently read are the elements of the dense vector `vec`. Due to the seemingly random accesses to the vector during the multiplication, the required elements of the vector are likely to reside at a different process, leading to remote accesses. However, due to the same randomness, subsequent accesses to the same element are likely infrequent, which is expected to negatively impact both the read message and the index cache.

### 6.2.1. Strong Scaling Experiments

The strong scaling experiment features a square matrix of 30,000 rows. As the matrix is sparse, only 10 % of the entries are nonzero values. The vector for the multiplication also contains 30,000 entries, though with only nonzero entries. After decomposing the matrix into the CSR format, the necessary values for the calculations occupy approximately 1.4 GB of memory.

The sequential initialization, albeit disregarded for these measurements, requires considerable time to complete at the given problem size. Furthermore, it increases with the number of processes, as the synchronization that accompanies every single access in the sequential region requires more time to complete. For example, the experiment with 12 processes finishes the initialization in 15 minutes, while using 72 processes raises this duration to approximately 36 minutes. In comparison, the initialization of the original OpenMP kernel required approximately 12 seconds to complete at the given problem size. This observation led to a modification of the sequential area for the weak scaling

experiments that follow later, as the initialization would have taken too much time to reasonably complete and evaluate the experiments. This improvement is discussed at the start of Section 6.2.2.

### Time to Solution

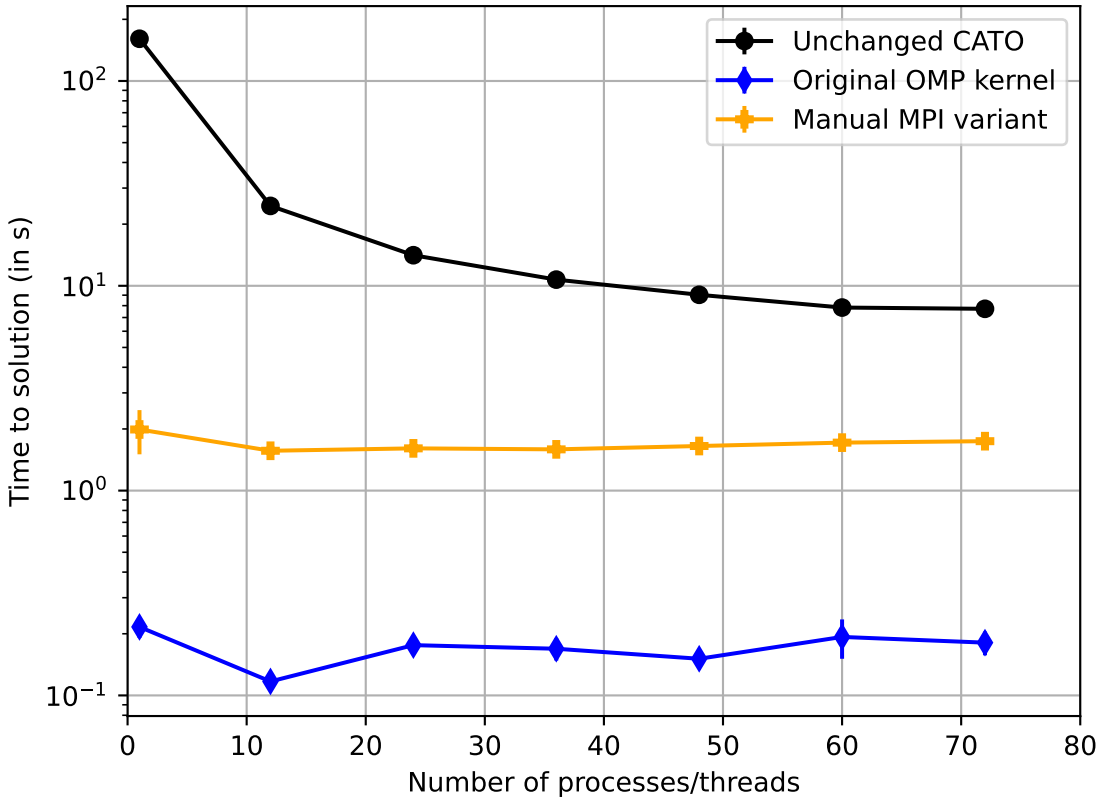


Figure 6.16.: Time to solution for different versions of the SpMV kernel in a strong scaling experiment. Each colour represents a different implementation of the kernel.

Figure 6.16 compares the time to solution of the transformed version of the SpMV kernel with its original OpenMP counterpart and a manual MPI implementation. As was the case with the stencil kernel, the transformed version is significantly slower for the given problem size. The original OpenMP kernel required less than one second to conduct the multiplication for all tested thread counts, while the MPI version required less than 2 seconds for the tested process amounts. The problem size is not large enough for these implementations to benefit from an increased count of threads or processes respectively, as their time to solution remains mostly constant.

Using only one process with the transformed kernel yields the longest time to solution once again due to the added overhead of the transformation. This overhead becomes less pronounced as the number of processes increases, leading to a decrease in the time to solution for the tested process counts. The largest process count produces a time to solution of 7.7 seconds.

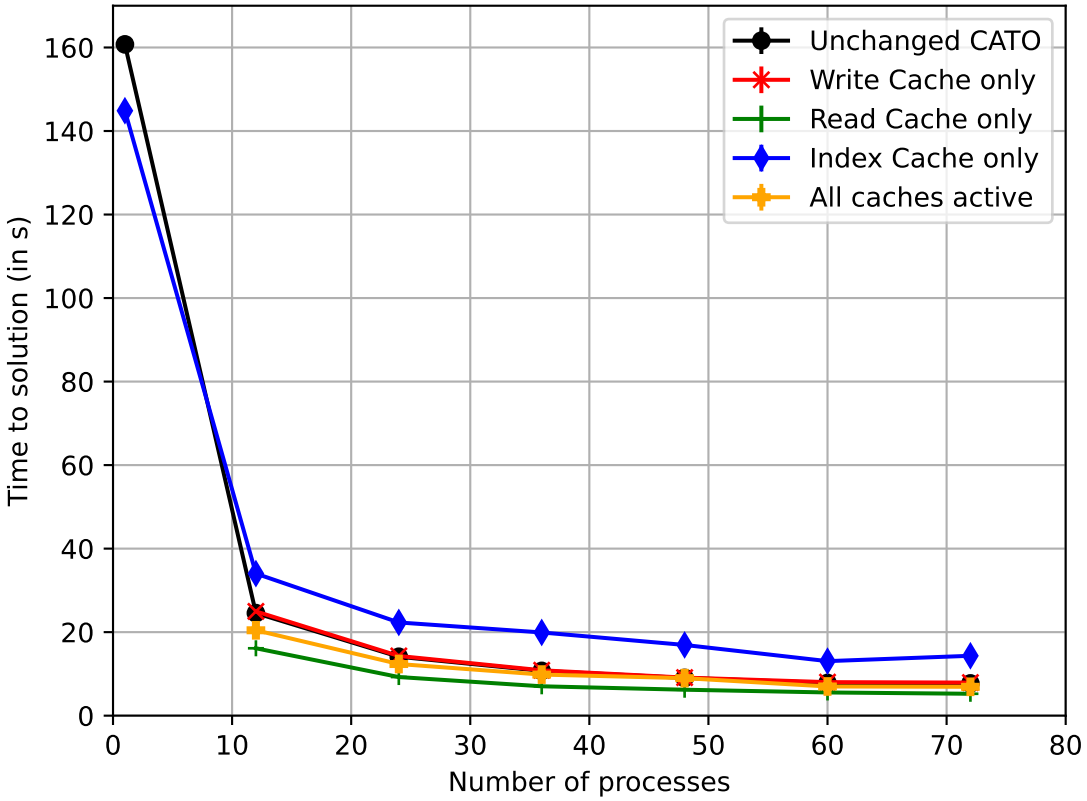


Figure 6.17.: Time to solution for different runtime configurations of the strong scaling SpMV kernel after a transformation with CATO. Each colour represents a different configuration of active cache components.

Figure 6.17 displays the measured times to solution when activating the different cache components. Contrary to the results of the stencil kernel, activating the index cache does not have a positive influence on the time to solution beyond a single process. Instead, it leads to longer run times for many of the tested process counts. This is caused by the low number of cache hits, as the elements of the matrix are only read once. However, they are still added to the cache, even though there is no future benefit in the form of cache hits. The only exception to this is the vector, but due to the nature of the seemingly random accesses, the absolute number of hits remains low. The write message cache shows no benefit either, since the write cache remains unused in this kernel. While the write message cache offers no benefit, it also does not add any measurable overhead. The measured times to solution for an active read message cache are the lowest of all

configurations, reaching a minimum of 5.2 seconds at 72 processes. While the only elements that can yield a read cache hit are the vector elements, it appears that this suffices for the read cache to improve the time to solution. Activating all caches at once leads to subpar performance, due to the negative influence of the index cache for this particular kernel. This causes the configuration with all caches active to perform worse than having only the read cache active.

### Memory Consumption

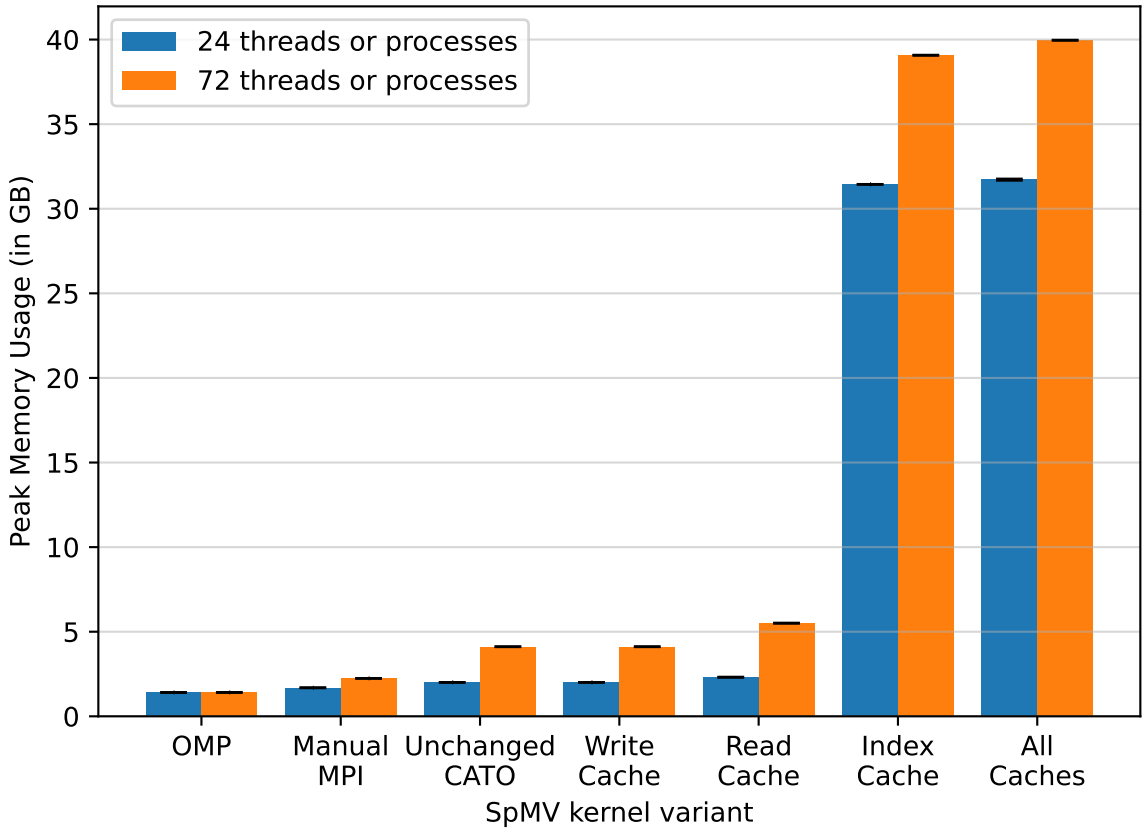


Figure 6.18.: Peak memory usage of the SpMV kernel variants for 24 and 72 threads or processes respectively.

The experiments with 24 and 72 threads or processes were subsequently instrumented to gather runtime statistics about the memory utilization once again. The results are displayed in Figure 6.18. Examining the memory consumption for the experiment with 24 threads or processes, the OpenMP variant displays minimal overhead, as was the case in the evaluation of the stencil kernel. The OpenMP version of the SpMV kernel requires only 1.4 GB of memory. This is approximately equal to the memory required to store the matrix data and the vector. The manual MPI variant, which will serve as the baseline that CATO will be compared against, displays a slight overhead, peaking at

1.7 GB of memory used. Without activating any caches, the CATO-transformed variant of the SpMV kernel adds very little overhead as well, displaying a maximum at 2 GB. The same observation is made when activating the write message cache due to the lack of remote stores. Activating the read message cache increases the peak memory utilization slightly. Reaching its peak at 2.3 GB of occupied memory, the cost for the improvement of the time to solution is negligible. The index cache however displays a very large peak in memory usage at 31.4 GB. This is caused by elements that are read exactly once, but still stored in the index cache. The cost of the index cache furthermore dominates the results of the measurements with all cache components active. In this case, memory usage peaks at 31.7 GB.

For 72 threads or processes, most of the previously made observations apply as well. The OpenMP kernel displays the same peak memory utilization, while the manual MPI variant shows a very small overhead, with a peak at 2.2 GB, which is an increase of 500 MB compared to using only 24 processes. CATO without caches displays a slightly larger growth regarding its overhead, doubling the peak memory usage to 4.1 GB. The same applies to the measurements with an active write message cache for the aforementioned reasons. The memory consumption for the read cache displays a larger relative increase for this configuration, peaking at 5.5 GB. The increase in the number of processes leads to an increase in the number of remote elements that need to be accessed by a given process. The index cache displays the largest memory consumption in this configuration as well, though the relative increase compared to the baseline performance of CATO is smaller.

### Other Cache Statistics

The experiments with 24 and 72 processes were also instrumented to capture performance metrics regarding the cache components. As highlighted in previous sections, the write message cache is not engaged at all due to the lack of remote writes. This is also reflected in the captured statistics for this component, for both instrumented experiments.

Activating the read message cache improved the time to solution noticeably, but the cache hit ratio across all processes amounts only to 19 %, with an absolute value of 85,566,014 hits in the experiment with 24 processes. The experiment with 72 processes displayed a similar hit ratio at 19.3 %. Considering the access pattern of the underlying kernel though, this is a high value. The only accesses that can generate hits are read accesses on the vector array. As every access to the vector array is paired with two accesses to the arrays for the representation of the matrix, namely the `data` and the `indices` array, the theoretically reachable maximum hit ratio is 33 %. This optimal hit ratio would require a replication of the entire vector to be present in the cache prior to the main loop. All observations on the upper bound disregard the accesses to the `ptr` array due to its comparatively small size. The local portions of the vector are also disregarded.

The read message cache increased the memory consumption only by a small value. On average, there were 31,954 read message cache entries entries present per process in the 24-process experiment, while the recorded minimum and maximum value across the processes are at 28,751 and 40,595 elements respectively. Due to the nature of the randomly-generated sparse matrix, some processes needed to access more remote elements than others. While most elements can be traced to the vector, some of these entries correspond to elements of the matrix. These entries are a waste of both processing time and memory, as they are stored in the cache only after the single access to them. Distributing the data across more processes, as was the case with the 72-process experiment, led to a larger read cache, averaging 79,931 elements instead, with a recorded minimum and maximum value of 29,585 and 129,958 respectively. This corroborates the assumption that the number of remote accesses increases with the process count and also explains the larger relative increase in memory consumption for 72 processes. Furthermore, the disparity between the largest and smallest cache grows larger as the process count increases as well.

The index cache also suffers from the singular accesses to the matrix elements, though the effect is much more pronounced due to the index cache storing an element for every access, regardless of its locality. The smaller experiment with only the index cache active displayed a cache hit ratio of 7 %, with an absolute number of 359,327,464 hits across all processes. While the majority of those hits was for local elements, namely for 273,761,450 hits, the added overhead from recording every accessed entry of each array in the parallel region led to the observed increase in the time to solution and also the large memory utilization. The index cache is also searched for entries during the initialization, but is not allowed to create entries during this time. Trivially, every access conducted here is a cache miss. If this initialization phase is disregarded, the cache hit ratio rises to 66 %. Of this hit ratio, approximately one third can be attributed to the remote elements of the vector and very few hits on remote elements of the `ptr` array that also resulted in cache hits for the read cache. The remaining two thirds of index cache hits, in particular all of the hits for local elements, are composed of local portions of the vector, as well as the repeated access to the `result` and `ptr` arrays in the inner loop. This comparatively large hit ratio and the repeated local accesses to the arrays are still not enough to compensate for the added overhead caused by storing all of the matrix elements. The 72-process experiment showed an even worse performance of the index cache, with a hit ratio of only 2.7 %. Not accounting for the misses in the sequential initialization, this ratio increases to 66 %, but as before, the number of absolute hits is not able to compensate for the additional overhead caused by the one-time accesses to the matrix elements.

The index cache contains mostly entries to the matrix elements, which does not lead to benefits regarding the time to solution, but still consumes a considerable amount of memory. Using 24 processes, the index cache of a process averaged 7,532,865 entries. Summing the elements that can be accessed multiple times, namely portions of the `result` and `ptr` array, as well as the entirety of the `vec` array, a total of less than 33,000 elements is accrued. Thus, millions of these cache entries are wasted.

These observations hold for the activation of all cache components at once as well, but due to the read cache enabling early termination on certain accesses, the cache hit ratio, and by extension the absolute number of cache hits of the index cache is slightly lowered. This is expected, as the same was observed for the stencil experiments.

#### Prefetching

Without prefetching, the only elements that benefit from the read message cache are the values of the dense vector and very few elements of the `ptr` array, hence the low cache hit ratio. By enabling the prefetching mechanism though, read operations targeting the other arrays can benefit as well. Instead of loading the value when it is requested and then storing it in the read cache, taking up space and time without ever being read again, the values can be prefetched instead. This can increase the cache hit ratio, though the extent of this increase is limited, as most matrix elements are local. Furthermore, the values are still only read once, even after prefetching, due to the nature of the SpMV kernel. The vector can benefit from prefetching as well. If one element is not present in the cache, the prefetching mechanism will fetch both the requested value as well as the next $n$ elements. However, for these prefetched elements, there exists the possibility of fetching the same value multiple times, leading to a potential waste of both communication and processing time. In order to evaluate the viability of the prefetching mechanism in this situation, multiple different configurations of the prefetching mechanism were tested on the 24- and 72-process experiments.

The prefetch mechanism did not change the time to solution in either experiment. In the configuration using 24 processes, the cache hit ratio of the read cache displayed a value of 19.01 %. Using prefetch values ranging from 32 to 1024 elements, this ratio only improved to 19.16 % for the smallest value, and 19.18 % for the largest. Similarly, no prefetching in the 72-process experiment already yielded a cache hit ratio of 19.28 %. Using the same prefetch values, this ratio only improved to 20.50 % for the smallest, and 20.55 % for the largest value. These improvements in the hit ratio are too insignificant to positively influence the time to solution. In terms of absolute cache hits, the largest prefetch values only increased the number by 763,451 and 5,736,018 hits for both experiments respectively. The overhead introduced with prefetching, namely the larger messages and the creation of the cache entries, appears to be too costly compared to the small improvement in the cache hits. If the absolute numbers were larger, this overhead might be amortized more easily. This hypothesis will be tested in the following weak scaling experiments.

### 6.2.2. Weak Scaling Experiments

The SpMV kernel needs to be evaluated at larger problem sizes, while extending the number of processes beyond the capabilities of a single node. The compatibility problems between the AMD hardware and Intel software persist for the following experiments as

well, which results in small differences in the runtime environment between the strong and the weak scaling experiments.

The problem size is set to 140 MB per process. Both the matrix in the CSR format and the vector are accounted for in this size. As with the stencil previously, the weak scaling experiments are conducted on the nodes with 512 GB of main memory instead of the ones with 256 GB, on which the strong scaling experiments were conducted.

### Improvements to Sequential Regions

The sequential initialization of the matrix data proved to consume a substantial amount of time in the strong scaling experiments. Due to the larger problem sizes in the following weak scaling experiments, this problem would have worsened considerably, obstructing the execution and evaluation of the experiments.

In the strong scaling experiments, it was observed that the time to initialize increased with the number of processes. This leads to the conclusion that the active target synchronization that is incurred with each individual access is causing this problem. As such, the number of necessary fence calls needs to be reduced drastically in order to solve this problem. Two different approaches were considered for this purpose.

The first option is to turn the sequential region into a parallel region by virtue of the `parallel` pragma and then enclosing the initialization code in a conditional to have only a single process execute the initialization. This changes the utilized synchronization mechanism from active target synchronization, as it is used in the sequential regions, to passive synchronization. This results in a flow of execution that mimics the original OpenMP kernel, as only a single process executes the code in the previously sequential region. This approach was tested on the problem size of the strong scaling experiment, with both 24 and 72 processes. Without this change, leaving the sequential area untouched, the time to initialize the data amounted to 15 and 36 minutes respectively. Changing the kernel in the described manner reduced this time considerably. Initializing the data required, on average, 401.8 seconds when using 24 processes, while running the experiment with 72 processes displayed a duration of 456.4 seconds. Even though a single process executes the initialization, there are still differences in the run time for differing numbers of processes, albeit less pronounced. Activating the write message cache did not benefit this approach either.

Turning the sequential region into a parallel region only changed the synchronization mechanism. The second option aims to reduce the total number of synchronization calls, while also aiming the changes mostly at CATO instead of the kernel itself. Since the values of the matrix depend on a pseudo-random number generator, the value at any given position does not depend on the values of prior entries. As such, these values do not need to be read to correctly initialize each array. One of the purposes of the synchronization mechanisms is the completion of outstanding RMA operations. However,

due to the absence of a necessity to read the prior values, it would be sufficient to start an access epoch immediately before the initialization section, and close it only after each value has been determined. This reduces the required number of necessary synchronization calls from twice the number of accesses down to just two. Currently, the functions to start and end these longer lasting epochs need to be added manually to the kernel, enclosing the region in question. Chapter 7 explores potential options to remove the need for the user to insert these calls manually, instead shifting the burden to the compiler pass. This proposed enhancement reduces the time to initialize the data even further than the previous approach. The 24-process experiment finished its initialization in 25.9 seconds on average, the 72-process experiment needed 27.6 seconds.

| | **24 processes** | **72 processes** |
|---|---|---|
| **No changes** | 1172.3 s (1.54) | 2170.0 s (3.61) |
| **Replace with parallel region** | 401.8 s (3.03) | 456.4 s (1.34) |
| **Use single access epoch** | 25.9 s (0.22) | 27.6 s (0.12) |

Table 6.3.: Mean time to finish the initialization section of the SpMV kernel in the transformed version. The problem size for the strong scaling experiments was used to determine these values.

Table 6.3 summarizes the previously discussed results. Due to the efficacy of the second approach, which reduced the number of synchronization calls, the weak scaling experiments are conducted with this modification to the SpMV kernel.

### Time to Solution

Figure 6.19 shows the times to solution for the original OpenMP kernel, the manual MPI variant and the version created by CATO for the problem sizes chosen in the weak scaling experiment. For the tested thread counts, the OpenMP version displays run times ranging from 0.06 to 1.19 seconds. The MPI version produces times to solution ranging from 2 to 13 seconds on the comparable process counts, and reaches a peak of 22 seconds at 256 processes across two nodes. The transformed version does not scale as well, exhibiting times to solution that are larger by at least one order of magnitude than the MPI version for all tested process counts. On a linear scale, the slope produced by the measurements for the transformed version is considerably steeper than its counterparts, mirroring the tendencies of the results in the weak scaling experiments on the stencil kernel.

Figure 6.20 demonstrates the effect of the different cache components on the given problem sizes of the SpMV kernel. The index cache did not have a positive influence on

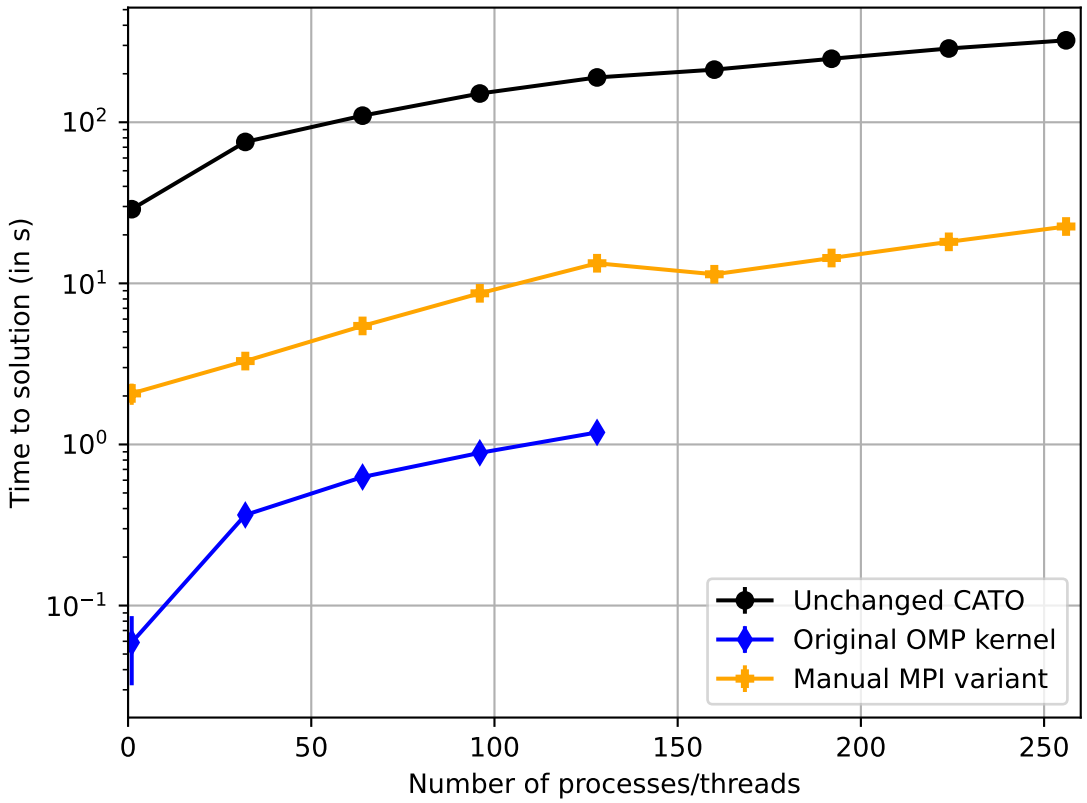


Figure 6.19.: Time to solution for different versions of the SpMV kernel in a weak scaling experiment. Each colour represents a different implementation of the kernel.

the time to solution for the smaller problem size in the strong scaling experiment. The larger problem sizes of the weak scaling experiments result in a noticeable positive impact though. With increasing problem size and process count, the time difference between the uncached version and the active index cache increases as well. The read message cache did not change the time to solution in the strong scaling experiments either. The larger problem size of the weak scaling experiment favors this cache component as well. From one to 128 processes, the read message cache reduces the time to solution the most, beyond that the impact of the index cache surpasses the read cache, presumably due to the increase in the absolute number of repeated accesses to local elements. The write message cache remains congruent with the uncached version due to the absence of remote store operations. Activating all caches produces the best times to solution, combining the impact of the index and the read cache. The absolute time values are considerably smaller than the ones produced by the uncached runs, and the curve that is generated by these results is substantially flattened as well in comparison. In conclusion, the caches improved both the absolute run times as well as the scaling behaviour of the transformed kernel.

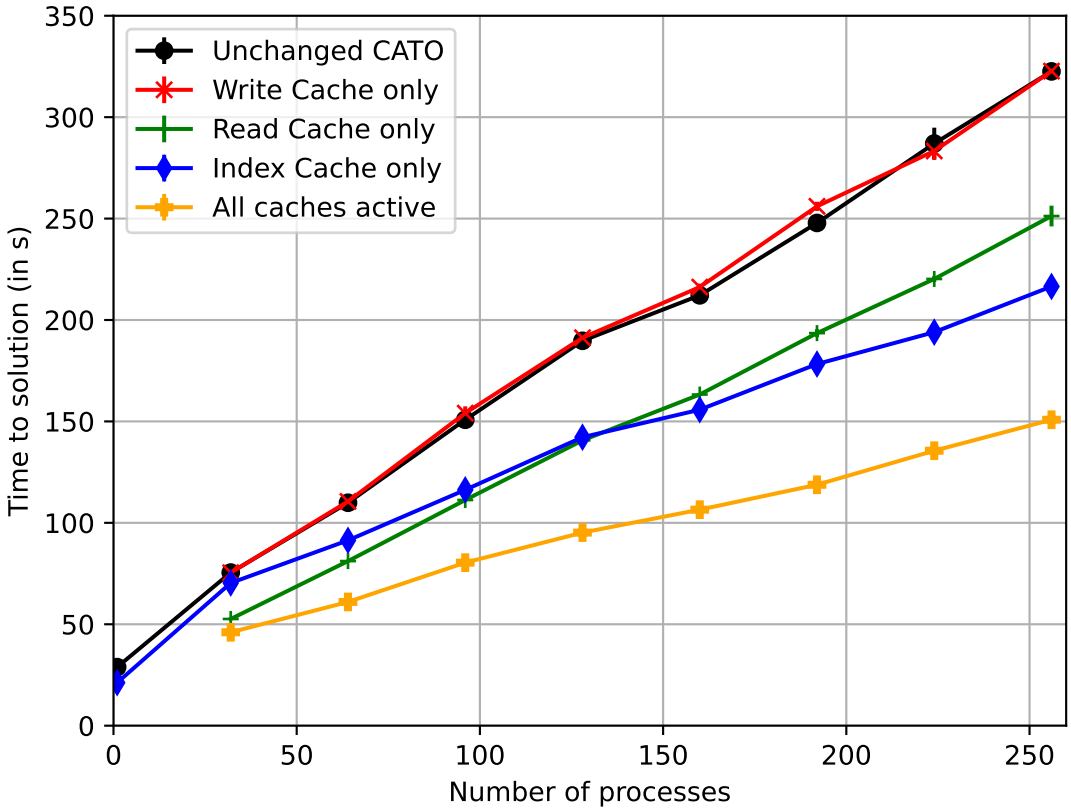


Figure 6.20.: Time to solution for different runtime configurations of the SpMV kernel after a transformation with CATO, using a weakly scaling problem size. Each colour represents a different configuration of active cache components.

### Memory Consumption

The experiments featuring 128 and 256 processes were instrumented to capture certain runtime statistics, Figure 6.21 displays the memory consumption that was recorded in the process. For a thread count of 128, the OpenMP kernel exhibits a memory consumption of nigh 18 GB. This is approximately equal to the problem size at this thread count, showing no overhead at all. The manual MPI variant requires a considerably larger amount of memory to operate. For 128 processes, it peaked at 46 GB. Without activating any caches, CATO almost matches this memory utilization, only exceeding it by 2 GB. The write message cache displays the same memory usage as the uncached version due to the absence of remote stores. Activating the read cache adds another 9 GB to the peak memory usage. This is a very small cost for the impact that it generates, as it reduces the time to solution by 50 seconds, which is over a quarter of the total. The impact on the time to solution is comparable to the index cache at this process count. The index cache however requires more than 8 times as much memory as the uncached CATO version, peaking at 403 GB. This peak also dominates the memory usage in the configuration with all caches active, leading to 412 GB of occupied memory.

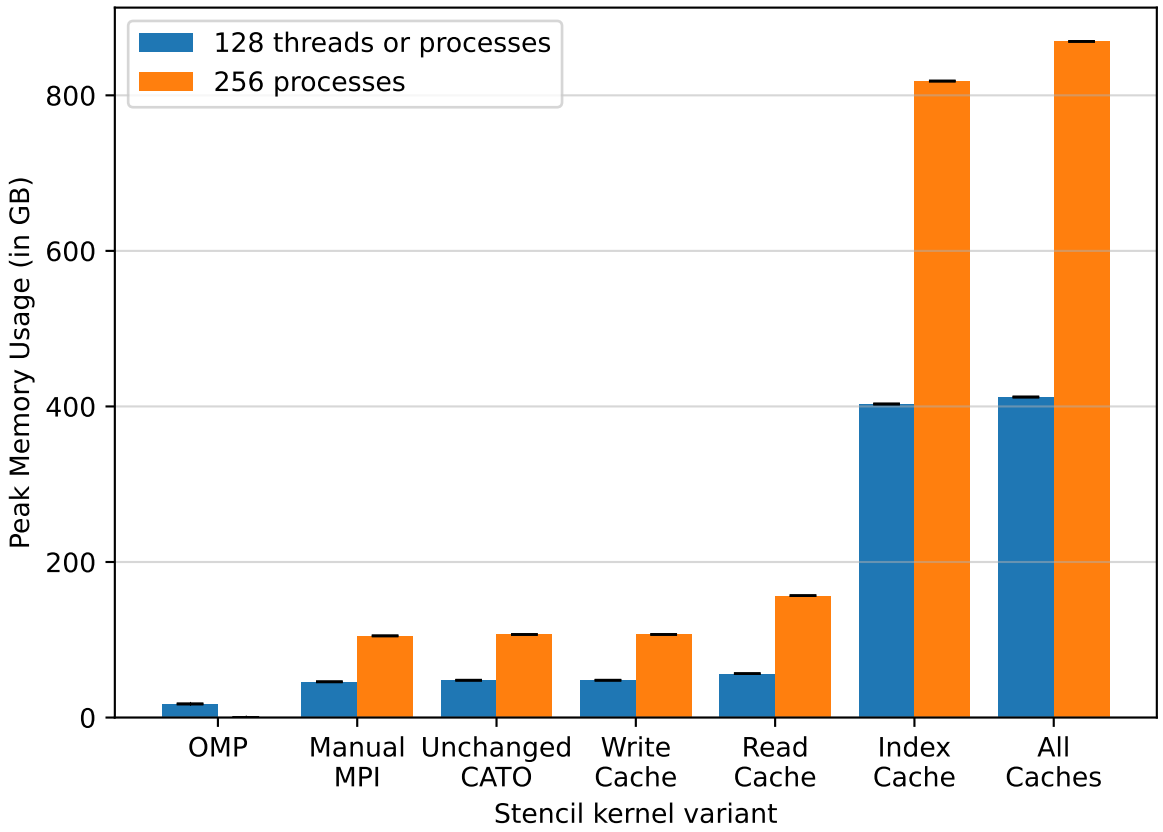


Figure 6.21.: Peak memory usage of the weak scaling SpMV kernel variants for 128 and 256 processes or threads, where applicable.

The experiment with 256 processes at the largest tested problem size mostly matches these observations. The manual MPI version peaks at 105 GB of memory usage, CATO only surpasses it by 1 GB. Activating the read cache comes at a greater cost for this experiment though. The increased number of remote read operations increased the peak memory usage by 50 GB, which is half of the total for the version without caches. In the previous experiment with 128 processes, the read cache only increased the peak by 16 %. While the read cache still has a large positive impact on the time to solution, the index cache has an even larger effect individually at this problem size. Once again this necessitates an increase in peak memory consumption, though only at a factor of 7.6 compared to the baseline version of CATO for this configuration. Activating all caches at once leads to the highest peak at 869 GB, most of which is attributed to the index cache.

Both the index cache and the read message cache had a considerable impact on the time to solution, individually decreasing it by at least 20 % at every tested problem size and process count. The index cache eventually surpassed the read message cache at larger problem sizes, reducing the time to solution the most, but the cost in terms of memory

consumption is significant. The read message cache offers a substantial reduction in the time to solution as well, albeit slightly less so than the index cache. However, the memory usage to achieve this improvement is significantly lower, making it more viable in scenarios that do not offer vast quantities of main memory.

### Other Cache Statistics

The read message cache improved the time to solution significantly for each tested problem size. Profiling the cache at 128 and 256 processes respectively revealed cache hit ratios that are similar to the strong scaling experiments. A hit ratio of 19.6 % was achieved at both of the previously mentioned configurations. In terms of absolute numbers, this translates to 1,098,738,763 hits for the 128-process experiment, while this number is almost doubled for the larger experiment at 256 processes. The hit ratios are comparable to the strong scaling experiment, but the absolute hits have increased by orders of magnitude. Though the read cache hit ratio is still subject to the boundary at 33 %, the large absolute number of cache hits leads to a considerable improvement in the time to solution. The absolute number of cache misses is very high as well, but it appears to have a neglible impact on the viability of the read message cache in terms of the time to solution.

The memory consumption of the read cache was very small compared to the index cache, especially in the case of the 128-process experiment. The cache added less than 10 GB to the captured peak. An examination of the cache sizes corroborates this observation, as the cache averaged 341,487 elements per process. The smallest and largest caches maintained 105,000 and 610,673 entries respectively. Without prefetching enabled, only elements from the vector and a few elements of the `ptr` array can yield cache hits for any given process. At this problem size, the remote elements of the vector amount to almost 105,000 elements, meaning that more than half of all entries in the read cache are part of the matrix, which cannot yield any cache hits whatsoever, pointing to a large waste of space in the read cache. A similar observation can be made when using the larger problem size and 256 processes. The average read cache across all processes contains 1,074,070 elements, with the measured minimum and maximum values being at 149,078 and 1,984,571. For the given problem size, the remote elements of the vector amount to slightly more than 149,000 elements, further supporting the observation that most entries are never accessed and thus effectively waste space. The prefetching mechanism can potentially alleviate this though, enabling a single access for most of these elements through the enhanced message sizes on remote reads.

The hit ratios of the index cache are even smaller than in the strong scaling experiments. For the problem size used at 128 processes, the index cache only achieved a hit ratio of 1.5 %. This number dropped even further with the increased problem size at 256 processes, going as low as 0.77 %. If the cache misses incurred during the initialization are factored out, the hit ratios rise to 66 % again, as was the case in the strong scaling experiments. However, the impact on the time to solution is much more significant in the weak scaling experiments. This is also the result of the large number of absolute

hits. The smaller problem size at 128 processes yielded a total number of 4,466,174,253 index cache hits. Doubling the problem size and process count consequently doubled the number of hits. In both experiments, more than 75 % of these hits were produced for local elements, namely the local portions of the `vec`, `result` and `ptr` arrays. The rest of the hits are once again attributed to the remote vector elements and a few remote `ptr` elements.

While the impact of the index cache hits on the time to solution is considerable, the amount of occupied memory is significant as well. For both problem sizes at 128 and 256 processes, the index cache averaged about 17,600,000 entries per process. The larger problem size only featured about 50,000 more entries on average. Given the weak scaling, this behaviour is expected. However, only the vector, the local portion of the `result` array and the second access to an element of the `ptr` array can yield index cache hits. For both problem sizes, summing the elements that have the potential for cache hits results in less than 200,000 elements. Consequently, the vast majority of the more than 17,000,000 entries are a waste of space. This cannot be ameliorated via the prefetching mechanism either, since prefetching currently works exclusively on remote elements.

Activating all caches at once leads to the same result as before: the read cache displays the same number of absolute hits, reducing the number of index cache hits in the process due to preemption. In conclusion, high cache hit ratios are not required for the caches to improve the time to solution. Instead, a high absolute number of hits is the driving factor for a good cache performance. This kernel also highlighted one of the weaknesses in the current cache implementation: if the read cache is active, each remote element is cached, even those that are not read a second time. Such entries should be prevented to save the space that these elements occupy. Alternatively the prefetching mechanism could be used to allow for a single cache hit, though it would still be benefitial to remove these elements with only a single access afterwards. Activating the index cache displays a similar problem, as each accessed element is introduced to the cache. The vast majority will not be accessed though due to the access pattern of the underlying kernel, leading to this large memory consumption that is dominated by elements with no accesses.

### Prefetching

Alongside improving the cache hit ratio for elements of the vector, the prefetch mechanism can potentially enable cache hits on the remote elements of the arrays that represent the sparse matrix. In the strong scaling experiments, the time to solution did not improve with prefetching enabled. This is likely due to the small problem size and the consequently low absolute number of cache hits that can be attributed to the prefetching exclusively. The overall impact of the read cache on the larger problem size in the weak scaling experiment is considerably more pronounced, by virtue of the large absolute number of cache hits. Thus, improving the cache hit ratio by a similar percentage as in the strong scaling experiments could result in vastly more absolute hits as well.

To this end, the experiment configurations with 128 and 256 processes were augmented with different values for the prefetching mechanism, ranging from 32 to 2048 elements per prefetch. In spite of the expectations, the prefetching did not improve the time to solution. For the 128-process experiment, activating the read message cache without prefetching yielded a cache hit ratio of 19.6 %. Activating the prefetch mechanism only improved this ratio to 20.4 % for the largest values. A similar change in the ratios was observed in the strong scaling experiments. However, due to the larger problem size, increasing the hit ratio by 0.8 % translates to a difference of 43,631,868 absolute cache hits. To put this value into perspective, the strong scaling experiment is considered once again. With the smaller problem size, the read message cache still reduced the time to solution by approximately 10 seconds on 24 processes. On this process count, the read message cache aggregated a total of 85,566,014 cache hits. The prefetching mechanism added almost half of that number to the cache hits in the larger weak scaling experiments, but without an according improvement of the time to solution. Though it should be noted that the process count is higher in this experiment, thus spreading the cache hits across more processes.

The larger problem size in the 256-process experiment shows similar results. Activating the prefetch mechanism does not reduce the time to solution in a statistically significant manner, even though the cache hit ratio increases to a greater extent in this experiment. Without prefetching, the read message cache displays a hit ratio of 19.6 %. At the largest prefetch value, namely 2048 elements, the cache hit ratio reaches 22.05 %. This corresponds to 274,649,694 more hits, almost tripling the cache hit count of the strong scaling experiments for reference. However, there is still no discernible effect on the time to solution. Note that the increase in the prefetch value from 1024 to 2048 elements only led to an increase of 0.01 % in the cache hit ratio, corresponding to roughly one million cache hits.

Even larger prefetch values are not likely to positively influence the time to solution either. Looking at the vector for example, at the given problem size, split across 256 processes, each individual process held less than 1000 vector elements. Thus, increasing the prefetch value beyond 1024 already does not change the size of the messages, as they are limited by the size of the target buffer. Furthermore, the prefetched messages are always extended by reading the elements that are beyond the current index. Thus, if an element with a smaller index is accessed and this element is not cached yet, the prefetch message will contain all elements with a larger index that this rank contains, assuming a prefetch value of 1024. This leads to many redundant reads, happening every time that another vector element with a smaller index at a given process is accessed, which is likely to happen due to the seemingly random access pattern. Larger prefetch values are also not likely to improve accesses to the matrix arrays either. To prove this, larger prefetch values, up to 16,384 elements, were tested as well. As predicted, the cache hit ratio increased by less than 0.001 % for such values. They did however lead to an increase in the time to solution, as even more unneeded elements are contained within the prefetched messages.

### Examination of the Prefetching Performance

The total time to solution for the SpMV kernel is also dictated by the slowest process, as the parallel region needs to be succeeded by a barrier. The lacking impact of the prefetching mechanism for this experiment cannot be attributed to a load imbalance though. This was determined on the experiment with 128 processes, using the same problem size that was used for the previous weak scaling experiments at this process count. The prefetch value for the measurements in Figure 6.22 was set to 128 elements.

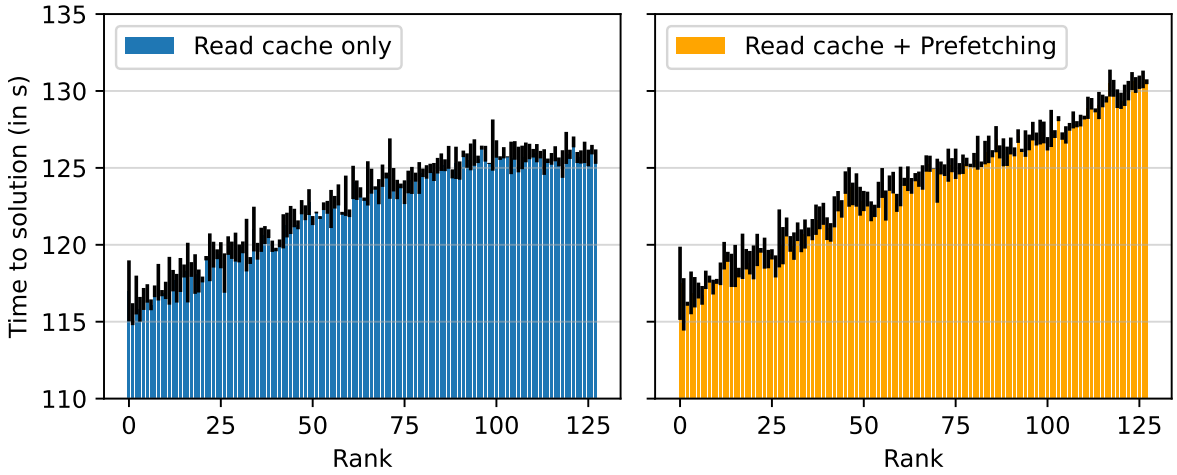


Figure 6.22.: Average per-rank run time of the SpMV kernel. The figure on the left was produced with an active read cache, but with no prefetching. The figure on the right had the prefetch mechanism set to 128 elements.

The run time of many ranks did not change after activating the prefetching mechanisms. In case of the later ranks, their time to solution rather saw an increase. This is different from the previous observations with the stencil kernel. In those experiments, the processes corresponding to the highest rank did not benefit from the read cache or the prefetching mechanism, while the other processes completed their iterations faster. This is not the case with the SpMV kernel. With no caches active, each rank requires between 160 and 170 seconds to complete the iteration. As can be seen in Figure 6.22, activating the read cache, either with or without the prefetching mechanism, reduces the time to solution for every rank by up to 45 seconds. Therefore, the load imbalance in the remote accesses that hindered the usefulness of the prefetch mechanism in the stencil kernel cannot be the cause for the lackluster impact on the SpMV kernel.

The reasons for this lack of impact are found within the times that are produced by a single access. The measurements displayed in the following were taken from three different runs of two experiment with 128 processes. The read message cache is active in both, but the prefetching mechanism is only active for one of the experiments. All runs were augmented with profiling code to measure the time to completion for individual

operations. However, instead of averaging these values over all processes, only the accesses of rank 127 were measured, due to the large discrepancy in the time to solution for these two experiments, as it was observed in Figure 6.22.

Without the prefetch mechanism active, all remote read operations load exactly one element, which is subsequently stored in the read cache. On average, this string of operations required 6 microseconds. If the element is found in the read cache instead, the operation can complete in slightly less than 300 nanoseconds. Enabling prefetching changes every remote load operation: instead of reading a single element and caching it, multiple elements are read and cached at once. When setting the prefetch value to 128, as was done for one of the two experiments, a prefetch operation finishes in less than 0.6 milliseconds. These numbers are also summarized in Table 6.4.

| **Element accessed via** | **Time to completion** |
|---|---|
| Found in cache | 0.286$\mu$s (0.012) |
| Remote Load | 6.49$\mu$s (0.049) |
| Prefetch | 585.9$\mu$s (4.515) |

Table 6.4.: Average time to access a single remote element under the listed conditions. The conditions are ordered in an ascending manner according to their respective time to completion.

Without activating the prefetch mechanism, rank 127 yielded 8,567,365 cache hits. Setting the prefetch value to 128 led to an increase in the total number of cache hits, though only by 102,528. If these additional hits would instead be serviced by normal remote loads to individual elements, the accumulated time to complete these operations amounts to 0.665 seconds, as displayed in Equation (6.1). Servicing these load operations with cache hits instead amounts to 0.029 seconds at the observed time to complete all of these operations.

$$102,528 \cdot 6.49\mu s = 0.665s \tag{6.1a}$$
$$102,528 \cdot 0.286\mu s = 0.029s \tag{6.1b}$$
$$2472 \cdot 585.9\mu s = 1.448s \tag{6.1c}$$

These additional cache hits carry a cost though, as they required multiple prefetch operations beforehand. In particular, rank 127 conducted 2472 prefetches. At the observed time to complete one of these prefetch operations, namely 0.6 milliseconds, completing all of them takes approximately 1.448 seconds, which needs to be added to the time to complete all of the additional cache-serviced loads.

The calculations and measurements can be combined to determine the reason for the prefetching mechanism's lack of impact. The prefetching itself requires too much time

compared to the time that is saved by the extra cache hits. If the cache misses, or rather the remote load operations, were more expensive, the extra time that is required by the prefetching operation could be remedied. This is also one of the differences between the stencil kernel and the SpMV kernel. The stencil kernel that worked well with the prefetching mechanism stored its elements in a two-dimensional array, thus requiring slightly longer to conduct the pointer chase, which in turn increased the time that a normal remote load operation requires to complete. The SpMV kernel stored all its data in one-dimensional arrays, comparatively shortening the time required to chase pointers, thus also shortening the time that a remote load operation requires to complete. The stencil kernel also conducted significantly more remote accesses and did not suffer from the previously mentioned redundant elements within the prefetched messages. For larger problem sizes, the prefetch mechanism might be able to improve the time to solution though. On rank 127, the number of cache hits was only increased by approximately 100,000. It is likely that significantly larger problem sizes also lead to a noticeable increase in the extra cache hits. Since previous experiments suggest that a large absolute number of cache hits is the driving factor behind good cache performance within CATO, the larger problem size may be enough to justify the extra costs of the prefetch operations. Additional experiments with such problem sizes are suggested for future evaluations of the new cache components, especially for further assessments of the prefetching mechanisms.

Note that the observed difference was slightly larger, at 5 seconds, than the calculated difference of about one second when factoring in the access times. This may be attributed to differences in the profiling overhead, as the profiling code that was added to observe the average times to solution displayed in Figure 6.22 differed compared to the code for measuring the time to completion for individual operations. The differences might also be caused by chance or other, as of yet undiscovered, interactions between CATO and the SpMV kernel. However, the final conclusion that can be drawn regarding the prefetch mechanism remains valid: the prefetch operation is too slow compared to the benefit of the added cache hits for all tested problem sizes and configurations of the SpMV kernel.

# 7. Future Work

The cache mechanisms can still be improved in multiple aspects. Static code analysis has already been used to detect strides in accesses to improve the prefetching. This analysis currently hinges on these strides being detectable in the right-most index though. This should be improved to accommodate for the other indices as well, in case that such a stride would only be detectable in the left-most index for example.

Static code analysis in general can be used extensively to improve the caches. During the evaluation of the stencil kernel, new functions were added to CATO that allow the user to manually disable and enable the caches for certain parts of an application. This in turn allowed the user to disable the caches in initialization phases that would otherwise lead to a pollution of said caches. Providing the users with this option is generally benefitial, however it places an additional burden on them to ensure that the transformed kernel performs well. This burden should be moved to the LLVM compiler pass instead. Static code analysis can be used to detect such initialization periods. In the stencil kernel for example, this phase displays certain characteristics that are detectable via such an analysis. In particular, each element is written to exactly once and never read from. If such a pattern is detected via a static code analysis, the compiler pass can insert the functions to disable the caches for these initializations, and it can also insert the functions to enable them again afterwards. This prevents the cache pollution automatically without further user involvement.

A similar strategy can be employed to improve sequential areas. At present, accessing a single value in a sequential region incurs two synchronization calls, a fence to start the access epoch, and one to end the access epoch. This leads to tremendous amounts of synchronization, also increasing the more processes are involved, as displayed during the evaluation of the SpMV kernel. The sequential initialization required a substantial amount of time, especially compared to the actual calcuations in the parallel region. However, if a static analysis can determine that the subsequent values of the array can be generated and written without reading prior values of the array, meaning that there is an independence between the current value and the prior content of the array, the synchronization calls could be drastically reduced. If such an independence is detected for a section of a sequential region, the compiler pass could insert a single fence at the start of said section. Then, all subsequent accesses of that section can be executed without closing the access epoch, removing a substantial amount of global communication. After all accesses in this section have been conducted, the compiler pass can insert the closing fence, ending the access epoch. The viability of this reduction in synchronization has been tested during the weak scaling experiments for the SpMV

kernel. However, the beginning and closing fences had been added to the kernel manually. Detecting sections with such characteristics during a static code analysis and automatically adding the appropriate function calls would remove the burden of adding these functions from the user. This removal is very important for this case in particular, as detecting these characteristics is not trivial, especially for CATO's targeted user group.

Static code analysis can also be used to improve the prefetching mechanism. Currently, the user needs to supply an integer to control the number of elements that are prefetched with each remote load. This value then applies to all arrays that are accessed in the application. However, this universal applicability is not always benefitial. In the SpMV kernel for example, the `data` array and the `ptr` array are of vastly different size and display different needs regarding the number of elements that need to be accessed. Applying the same prefetch value to both of these arrays is suboptimal. The `ptr` array likely performs better with a comparatively smaller prefetch value to prevent loading unnecessary elements. It could be possible to determine such differences in the access patterns with static code analysis. It might even be possible to employ such analyses to determine an optimal value for the number of prefetched elements. However, that requires further analysis of different prefetch values on different kernels and problem sizes, and furthermore hinges on the availability of the problem size at compile time.

Certain access patterns for arrays do not benefit from the different cache components, especially in the absence of the prefetching mechanism. For example, when considering arrays of which the individual elements are only accessed once, as was the case with the `data` and the `indices` arrays in the SpMV kernel, it might be useful to disable any sort of caching for these arrays specifically. At present, CATO's caches are either active for all arrays or for none of them. Finding a compromise, excluding only certain arrays, similar to the concept of the individual prefetch values, has the potential to improve the memory utilization significantly. Arrays with such characteristics can also be identified via static code analysis.

Another area that can be improved with static code analysis is the utilized hash function for caching. As explained in Appendix A, the conducted experiments with one-dimensional arrays utilized a custom hash function that can only be efficiently applied to kernels that exclusively feature one-dimensional arrays. Utilizing a universally applicable hash function proved to have a large negative impact on the performance. Hence, static code analysis could be applied to determine the dimensionality of the arrays within a kernel and choose from several custom hash functions that are tailored specifically towards the nature of the utilized arrays, and only falling back to the generally applicable hash function as a last resort.

All of these possible applications for static code analysis can be tackled in a different manner as well. The analysis could be bypassed by placing additional burden on the user. The functions for enabling and disabling caches are such examples. This could be extended to provide functions or pragmas that control, for instance, the applied

hash function. Another example would be the different prefetch values. A user could decorate a given array with a pragma to determine the prefetch value for this specific array. Ultimately, for most of the aforementioned features a choice needs to made regarding the analysis feasability. If the analysis requires information that may not be present at compile time, such as the problem size, or is otherwise not possible, placing additional burden on the user may be necessary to improve the overall performance of the transformed application.

The cache mechanisms offer a tradeoff: higher memory consumption for a shorter time to solution. For example, the evaluation of the weak scaling stencil experiments revealed that the peak memory consumption increased by a factor of 5 when activating the index cache on large problem sizes. A reduction in the size of the individual elements could make this tradeoff more efficient, in turn making the index cache more viable for very large problem sizes. In CATO's current state, the same index cache elements are used for both remote and local elements. Introducing polymorphic index cache elements has the potential to reduce the number of member variables by half without a negative effect on the run time. Alternatively, the index cache could be rebuilt to focus exclusively on local elements, as the largest impact was achieved on local elements. The size of the individual elements and the cache as a whole could be reduced significantly if the cache was used for local data only, as this would eliminate a large number of elements from the cache itself, while also reducing the amount of information that needs to be contained within a single cache element. The remote elements could instead be handled exclusively by the read message cache. Evaluating theses options should be one of the next step in the development of CATO's caches.

The prefetching mechanism performed below expectations. While first assumptions were made regarding the reasons, further investigations are required, including different kernels with shared arrays of different dimensions and different access patterns. Solutions to improve the efficacy of the prefetch mechanism could include an overhaul of the cache structure. The current read message cache stores individual elements in a key-value store. Thus, after prefetching, the content of the received buffer needs to be copied elementwise, resulting in as many copy operations as there are elements. This is inefficient and leads to a long time to completion, as discussed in Section 6.2.2. The prefetching mechanism could benefit from a cache structure that is more akin to the cache lines that are found in hardware caches. For such a cache structure, the prefetched buffer could optimally be stored in the cache as a whole. Such a modification would necessitate significant change though and thus requires careful consideration, and it should also be preceded by more experiments with problem sizes that are larger than any of the ones utilized during this thesis.

Many of the suggested improvements require further experiments with different kernels. Future work should therefore focus on testing with more kernels, covering other Dwarfs as well. This can lead to further insights regarding potential improvements to the caches. For example, it might uncover access patterns that invalidate cache entries in the index

cache, which has not occurred in any of the currently available tests or experiments. This would force the detection of such changes and the subsequent eviction of the previous entry. Further evaluations might also uncover different access patterns that the compiler pass does not cover yet.

Such revelations might also occur when transforming applications written in other programming languages. In CATO's current state, only C applications are known to work with the transformation. However, one of the main advantages of LLVM is the LLVM IR and its ubiquity. For example, a frontend for parsing Fortran code and emitting appropriate LLVM IR already exists, and as such, CATO could be applied to Fortran applications as well. However, the IR might display different access patterns or other differences that can stem from the differences between the programming languages, such as the column-major order that is present in Fortran. Investigating these differences and adjusting CATO accordingly could open up the tool to a new group of users, as Fortran is still a popular choice among scientific developers.

There are also other opportunities to improve CATO in general, apart from further improvements to the caches or small adjustments to the pass to accommodate for other kernels or programming languages. A worthwhile endeavour could be the reintroduction of shared memory. In its current state, CATO relies solely on processes and MPI communication, removing any notion of the originally present OpenMP threads. One possible improvement to CATO could be the reintroduction of OpenMP, enabling OpenMP-MPI hybrid parallelization. This would however require a major rework of the pass component and the runtime library. A less invasive potential improvement would be the introduction of MPI shared memory windows. While still relying solely on processes, the notion of shared memory would be reintroduced to CATO, potentially facilitating node-local accesses. While it is not guaranteed to be an improvement, it is still worth investigating.

# 8. Conclusion

In this thesis, a tool for the automatic translation of OpenMP kernels was examined. CATO's purpose is to enable OpenMP applications to scale beyond the limit of shared memory, without further user intervention by virtue of an LLVM compiler pass and an accompanying runtime library. CATO suffers from certain shortcomings though that limit the adoptability of the tool. The purpose of this thesis was to identify such shortcomings in CATO's memory management and to subsequently improve upon them. The first set of improvements was the introduction of generalized memory management. Prior to the changes, CATO was limited to kernels that employed arrays of dimensionalities up to three due to the nature of the compiler pass and the runtime library. This thesis highlighted the manner in which array indexing works in the context of the ubiquitous LLVM IR, using the insight gathered from this to adapt the pass component to be able to handle arrays of any dimensionality. The runtime library was improved in a similar manner, introducing the notion of two distinct phases, namely a pointer chase and the actual MPI calls, to subsequently improve the already existing portions of CATO's runtime memory management, which now also handles arrays of any dimensionality.

With the foundation of the generalized memory management in place, the second set of improvements to CATO could be conducted. The second set of improvements was aimed at optimizing the memory management at runtime, shortening the time to solution for transformed applications. First investigations determined that the bottleneck of CATO's performance was found in the custom load and store operations that need to be inserted into the transformed applications to enable the usage of distributed memory. These custom operations always consisted of one-sided MPI communication calls. Each call only accessed a single element though and was enclosed by two synchronization operations. The main improvements proposed in this thesis are two cache models. Their purpose is a reduction in the time to solution for transformed applications by virtue of a tradeoff in the form of larger memory consumption.

The first cache, deemed the index cache, was introduced as a TLB-esque address translation buffer, minimizing the necessity for additional pointer chases on elements that were accessed in the recent past, thus speeding up the overall access operation by allowing faster calls to the subsequent communication functions. Additionally, the index cache enabled accesses on local elements to circumvent any of the subsequent MPI operations. Evaluations of this cache component revealed a large positive impact on the time to solution, shortening it considerably, especially on large problem sizes, which are the main target of CATO. The faster processing of local elements proved to be one of the main advantages of the index cache. The reduction of the time for pointer chases was

less noticeable, which can be attributed to the focus on one- and two-dimensional arrays in the tested kernels. However, the index cache also proved to be very memory-intensive. Ongoing efforts are aimed at decreasing the memory cost of the index cache to further improve its viability.

The second cache is a message cache that conforms to the OpenMP memory model. The purpose of this cache is twofold: first, loaded remote elements are stored in the cache. This circumvents the necessity of further communication with other processes to access this previously loaded element, even if said element did not change in the mean time. The local access that the cache provides is faster than the remote access, as it eliminates the accompanying synchronization and communication. The second purpose is the aggregation of elements for remote stores. Instead of sending each element individually, as CATO would do without the cache, each element is aggregated in the message cache. The collected messages are then written to their intended targets upon meeting certain conditions, such as when encountering implicit flush operations. The aggregation reduces the number of required synchronization calls, while also enlarging the message sizes, thus making the communication more efficient. The evaluations have shown a limited impact on smaller problem sizes. The large problem sizes however benefitted considerably from the message cache, especially from the read component. The message cache was also much less memory-intensive than the index cache. It is however limited by the number of remote accesses within a kernel. Depending on the access patterns, such remote accesses might be very scarce, or even completely absent, as was the case with remote write operations in the SpMV kernel.

The thesis also proposed further improvements to the message cache in the form of a prefetch mechanism. Instead of reading only a single element upon a remote access, the following $n$ elements are also included in the MPI communication. The prefetch mechanism was further developed to be able to detect the strides at which a kernel accesses the array elements, adjusting the prefetch mechanism accordingly to account for these holes. This was enabled by a static code analysis in the form of the scalar evolution analysis pass, relying on the chains of recurrences to detect such strided accesses. The results of the prefetch mechanism were mixed. It was able to increase the cache hit ratio in each experiment, but only reduced the time to solution on one occasion. Further evaluations on even larger problem sizes should be conducted before deciding on possible improvements to the mechanism.

The thesis also proposed some smaller improvements to CATO. While the main focus lied on the improvement of the parallel regions, the sequential regions could also benefit from some of the proposed cache mechanisms. Additional improvements were made to the sequential regions by eliminating the necessity for the repeated synchronization calls on store operations in sections with certain characteristics. Such regions can instead be enclosed by manual synchronization calls, reducing the required synchronization functions drastically in such situations. The thesis also provides multiple directions for future work, focusing heavily on additional static code analysis to further improve the

utilization of the caches, for example.

One of CATO's main shortcomings is the long time to solution for the transformed applications, which hinders its usability in a production environment. The proposed changes, especially the newly introduced cache components, substantially reduced the time to solution on the tested kernels. Their effectiveness did not rely on a large cache hit ratio, but rather on a large absolute number of cache hits. Larger problem sizes facilitate this requirement, making the cache components very effective for CATO's intended purpose of enabling the transformed kernels to work on larger problem sizes.

# Bibliography

[Absar, 2018] Absar, Javed (2018). Scalar Evolution - Demystified. In *European LLVM Developers Meeting*.

[Almasi, 2011] Almasi, George (2011). *PGAS (Partitioned Global Address Space) Languages*, pages 1539–1545. Springer US, Boston, MA.

[Amza et al., 1996] Amza, Cristiana, Cox, Alan L, Dwarkadas, Sandhya, Keleher, Pete, Lu, Honghui, Rajamony, Ramakrishnan, Yu, Weimin, and Zwaenepoel, Willy (1996). Treadmarks: Shared Memory Computing on Networks of Workstations. *Computer*, 29(2):18–28.

[Aridor et al., 1999] Aridor, Yariv, Factor, Michael, and Teperman, Avi (1999). cJVM: a Single System Image of a JVM on a Cluster. In *Proceedings of the 1999 International Conference on Parallel Processing*, pages 4–11. IEEE.

[Arpaci-Dusseau and Arpaci-Dusseau, 2018] Arpaci-Dusseau, Remzi H and Arpaci-Dusseau, Andrea C (2018). *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, LLC.

[Asanovic et al., 2006] Asanovic, Krste, Bodik, Ras, Catanzaro, Bryan Christopher, Gebis, Joseph James, Husbands, Parry, Keutzer, Kurt, Patterson, David A, Plishker, William Lester, Shalf, John, Williams, Samuel Webb, et al. (2006). The Landscape of Parallel Computing Research: A View from Berkeley.

[Bachmann et al., 1994] Bachmann, Olaf, Wang, Paul S, and Zima, Eugene V (1994). Chains of Recurrences—a method to expedite the evaluation of closed-form functions. In *Proceedings of the international symposium on Symbolic and algebraic computation*, pages 242–249.

[Barak and La'adan, 1998] Barak, Amnon and La'adan, Oren (1998). The MOSIX Multicomputer Operating System for High Performance Cluster Computing. *Future Generation Computer Systems*, 13(4-5):361–372.

[Barbalace et al., 2014] Barbalace, Antonio, Ravindran, Binoy, and Katz, David (2014). Popcorn: a replicated-kernel OS based on Linux. In *Proceedings of the Linux Symposium, Ottawa, Canada*.

[Basumallik and Eigenmann, 2005] Basumallik, Ayon and Eigenmann, Rudolf (2005). Towards Automatic Translation of OpenMP to MPI. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 189–198.

[Bennett et al., 1990a] Bennett, John K, Carter, John B, and Zwaenepoel, Willy (1990a). Adaptive Software Cache Management for Distributed Shared Memory Architectures. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 125–134.

[Bennett et al., 1990b] Bennett, John K, Carter, John B, and Zwaenepoel, Willy (1990b). Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 168–176.

[Bershad et al., 1988] Bershad, Brian N, Lazowska, Edward D, and Levy, Henry M (1988). Presto: A system for object-oriented parallel programming. *Software: Practice and Experience*, 18(8):713–732.

[Bhattacharjee et al., 2018] Bhattacharjee, Abhishek, Lustig, Daniel, and Martonosi, Margaret (2018). *Architectural and Operating System Support for Virtual Memory*. Springer.

[Bottoni, 2022] Bottoni, Maurizio (2022). *Physical Modeling and Computational Techniques for Thermal and Fluid-dynamics*, chapter Numerical Solution of Poisson Equation, pages 257–290. Springer.

[Buyya et al., 2001] Buyya, Rajkumar, Cortes, Toni, and Jin, Hai (2001). Single system image. *The International Journal of High Performance Computing Applications*, 15(2):124–135.

[Case and Padegs, 1978] Case, Richard P and Padegs, Andris (1978). Architecture of the IBM System/370. *Communications of the ACM*, 21(1):73–96.

[Chen et al., 2009] Chen, Chen, Manzano, Joseph B, Gan, Ge, Gao, Guang R, and Sarkar, Vivek (2009). A Study of Different Instantiations of the OpenMP Memory Model and Their Software Cache Implementations.

[Chen et al., 2010] Chen, Chen, Manzano, Joseph B, Gan, Ge, Gao, Guang R, and Sarkar, Vivek (2010). A Study of a Software Cache Implementation of the OpenMP Memory Model for Multicore and Manycore Architectures. In *Euro-Par 2010-Parallel Processing: 16th International Euro-Par Conference, Ischia, Italy, August 31-September 3, 2010, Proceedings, Part II 16*, pages 341–352. Springer.

[Cheriton et al., 1991] Cheriton, David R, Goosen, Hendrik A., and Boyle, Patrick D. (1991). Paradigm: A highly scalable shared-memory multicomputer architecture. *Computer*, 24(2):33–46.

[Couleur and Glaser, 1968] Couleur, John F and Glaser, Edward L (1968). Shared-access data processing system. US Patent 3,412,382.

[Cytron et al., 1991] Cytron, Ron, Ferrante, Jeanne, Rosen, Barry K, Wegman, Mark N, and Zadeck, F Kenneth (1991). Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490.

[Geer, 2005] Geer, David (2005). Chip Makers Turn to Multicore Processors. *Computer*, 38(5):11–13.

[Gesbert and Gava, 2009] Gesbert, L and Gava, F (2009). New syntax of a high-level BSP language with application to parallel pattern-matching and exception handling. *LACL (Laboratory of Algorithms, Complexity and Logic), University of Paris-Est (Paris 12), Tech. Rep. TR-LACL-2009-06*.

[Hamidouche et al., 2011] Hamidouche, Khaled, Falcou, Joel, and Etiemble, Daniel (2011). A Framework for an Automatic Hybrid MPI+OpenMP code generation. In *SpringSim (hpc)*, pages 48–55. Citeseer.

[Healy et al., 2016] Healy, Philip, Lynn, Theodore, Barrett, Enda, and Morrison, John (2016). Single System Image: A Survey. *Journal of Parallel and Distributed Computing*, 90-91.

[Hoeflinger, 2006] Hoeflinger, Jay P (2006). Extending OpenMP to clusters.

[Hu et al., 2000] Hu, Y Charlie, Lu, Honghui, Cox, Alan L, and Zwaenepoel, Willy (2000). OpenMP for Networks of SMPs. *Journal of parallel and distributed computing*, 60(12):1512–1530.

[Jammer and Bischof, 2021] Jammer, Tim and Bischof, Christian (2021). Automatic Partitioning of MPI Operations in MPI + OpenMP Applications. In *High Performance Computing: ISC High Performance Digital 2021 International Workshops, Frankfurt am Main, Germany, June 24–July 2, 2021, Revised Selected Papers 36*, pages 191–198. Springer.

[Khalidi et al., 1996] Khalidi, Yousef YA, Bernabeu-Auban, Jose M, Matena, Vlada, Shirriff, Ken, and Thadani, Moti (1996). Solaris MC: A Multi Computer OS. In *USENIX Annual Technical Conference*, pages 191–204.

[Lattner, 2011] Lattner, Chris (2011). LLVM. In *The Architecture of Open Source Applications: Elegance, Evolution, and a Few Fearless Hacks*, volume 1, chapter 11.

[Lattner and Adve, 2004] Lattner, Chris and Adve, Vikram (2004). LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California.

[Lenoski et al., 1992] Lenoski, Daniel, Laudon, James, Gharachorloo, Kourosh, Weber, W-D, Gupta, Anoop, Hennessy, John, Horowitz, Mark, and Lam, Monica S. (1992). The Stanford Dash Multiprocessor. *Computer*, 25(3):63–79.

[Li, 1986] Li, Kai (1986). *Shared virtual memory on loosely coupled multiprocessors*. Yale University.

[LLVM, 2021] LLVM (2021). *LLVM Language Reference Manual*. `https://releases.llvm.org/13.0.0/docs/LangRef.html`.

[LLVM, 2023] LLVM (2023). LLVM Homepage. `https://llvm.org/`. Last access on 27.07.2023.

[Morin et al., 2003] Morin, Christine, Lottiaux, Renaud, Vallée, Geoffroy, Gallard, Pascal, Utard, Gaël, Badrinath, Ramamurthy, and Rilling, Louis (2003). Kerrighed: A Single System Image Cluster Operating System for High Performance Computing. In *European Conference on Parallel Processing*, pages 1291–1294. Springer.

[MPI, 2015] MPI (2015). *MPI: A Message-Passing Interface Standard Version 4.0*. Message Passing Interface Forum. `https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf`.

[Muchnick, 1997] Muchnick, Steven (1997). *Advanced Compiler Design Implementation*. Morgan kaufmann.

[Norouzi et al., 2019] Norouzi, Mohammad, Wolf, Felix, and Jannesari, Ali (2019). Automatic Construct Selection and Variable Classification in OpenMP. In *Proceedings of the ACM International Conference on Supercomputing*, pages 330–341.

[OpenMP, 2008] OpenMP (2008). *OpenMP Application Program Interface Version 3.0*. OpenMP Architecture Review Board. `http://www.openmp.org/mp-documents/spec30.pdf`.

[OpenMP, 2021] OpenMP (2021). *OpenMP Application Programming Interface Version 5.2*. OpenMP Architecture Review Board. `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf`.

[Pike et al., 1995] Pike, Rob, Presotto, Dave, Dorward, Sean, Flandrena, Bob, Thompson, Ken, Trickey, Howard, and Winterbottom, Phil (1995). Plan 9 from Bell Labs. *Computing systems*, 8(3):221–254.

[Rodríguez et al., 2013] Rodríguez, Gabriel, Martín, María J, González, Patricia, Touriño, Juan, and Doallo, Ramón (2013). Compiler-Assisted Checkpointing of Parallel Codes: The Cetus and LLVM Experience. *International Journal of Parallel Programming*, 41:782–805.

[Saà-Garriga et al., 2015] Saà-Garriga, Albert, Castells-Rufas, David, and Carrabina, Jordi (2015). OMP2MPI: Automatic MPI code generation from OpenMP programs. *arXiv preprint arXiv:1502.02921*.

[Saad, 2003] Saad, Yousef (2003). *Iterative Methods for Sparse Linear Systems*. SIAM.

[Sato et al., 1999] Sato, Mitsuhisa, Satoh, Shigehisa, Kusano, Kazuhiro, and Tanaka, Yoshio (1999). Design of OpenMP Compiler for an SMP Cluster. In *Proc. of the 1st European Workshop on OpenMP*, pages 32–39. Citeseer.

[Smith, 1982] Smith, Alan Jay (1982). Cache Memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530.

[Squar et al., 2020] Squar, Jannek, Jammer, Tim, Blesel, Michael, Kuhn, Michael, and Ludwig, Thomas (2020). Compiler Assisted Source Transformation of OpenMP Kernels. In *2020 19th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 44–51. IEEE.

[Squar et al., 2022] Squar, Jannek, Schroeter, Niclas, Fuchs, Anna, Kuhn, Michael, and Ludwig, Thomas (2022). Content queries and in-depth analysis on version-controlled software. *Procedia Computer Science*, 207:1261–1270.

[Walker et al., 1983] Walker, Bruce, Popek, Gerald, English, Robert, Kline, Charles, and Thiel, Greg (1983). The LOCUS Distributed Operating System. *ACM SIGOPS Operating Systems Review*, 17(5):49–70.

[Wu et al., 2017] Wu, Xingbo, Ni, Fan, and Jiang, Song (2017). Search Lookaside Buffer: Efficient Caching for Index Data Structures. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 27–39.

[XcalableMP, 2018] XcalableMP (2018). *XcalableMP Language Specification Version 1.4*. XcalableMP Specification Working Group. `https://xcalablemp.org/download/spec/xmp-spec-1.4.pdf`.

# Appendices

# A. Reproducibility

The purpose of this appendix is to provide all the necessary information to reproduce the results of the experiments that were conducted in the creation of this thesis.

## A.1. Runtime Environment

The experiments were conducted on Levante, using the nodes with either 256 GB or 512 GB of main memory. The thesis indicates which nodes are used on a per-experiment basis. The runtime environment for all experiments (and for the build process of CATO) was initialized with the script in Listing A.1.

```
1  #!/usr/bin/env bash
2  #Used a modified version of spack that still sets
     ↪LD_LIBRARY_PATH on loads
3  spack unload --all
4  spack load --first cmake
5  spack load gcc@11.2.0/bcn7
6  spack load intel-oneapi-mpi%gcc
7  spack load --first netcdf-c%gcc +mpi ^intel-oneapi-mpi
8  spack load llvm@13.0.0 build_type=Debug
9
10 source $(spack location -i intel-oneapi-mpi%gcc)/setvars.sh
11
12 export I_MPI_CC=clang
13 export I_MPI_CXX=clang++
14 #export I_MPI_FABRICS=shm
15 #export FI_PROVIDER=shm
16 export I_MPI_PMI=pmi
17 export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so
18
19 export CATO_ROOT="</path/to/cato>"
```

Listing A.1: Script to initialize the runtime environment for CATO. This was executed at the start of every experiment.

This script loads all required dependencies for CATO via spack and executes the initialization script for Intel OneAPI MPI. It also sets some additional variables to use the correct PMI library. Furthermore, it contains two lines that are currently commented

out. These lines are used to set the communication fabric. For the strong scaling experiments, which were constrained to a single node, the performance of MPI applications improved when explicitly setting the communication fabric to shared memory. Not setting this value would lead to the MPI implementation defaulting to `mlx`, which is less efficient for pure intra-node communication. The experiments that were conducted on two nodes cannot use this setting for trivial reasons, however there is an option to still use shared memory for intra-node communication and a different OFI-capable fabric for inter-node communication, by virtue of setting `I_MPI_FABRICS` to `shm:ofi`. This led to unexplainable segmentation faults though. Other HPC users with similar hardware reported similar experiences with Intel OneAPI MPI on a single occasion in the recent past[1], but without a solution. Hence, the weak scaling experiments use `mlx` as the communication fabric for both intra- and inter-node communication. This results in a worse performance compared to the strong scaling experiments.

## A.2. Measurements

In order to obtain certain metrics about the applications at runtime, the following facilities were used:

- Timing: measuring time to solution was done by prefixing the `time` command (`/usr/bin/time` in particular).
- Fine-grained timings, for example regarding the run time of individual functions, as it was needed for the manual profiling during the thesis, were obtained via the `MPI_Wtime` function
- Memory usage: the peak memory utilization was determined by calling `getrusage` on each process right before termination, accessing the `ru_maxrss` of the `rusage` struct, and then summing the values, if necessary.

Executing any MPI application thus resulted in a line that matches the following format: `{ time srun -n 128 ./modified.x 2» errs 1» out; } 2» times`.

## A.3. Hash Functions

The cache components of CATO rely on key-value stores, in particular the `unordered_map` provided in C++. The index cache and the read message cache utilize the same information as their respective keys, namely the address of the base `MemoryAbstraction` object, as it was passed to the custom access function, alongside the indices. Initially, the caches utilized a hash function that is universally applicable, capable of hashing the provided information for arrays of arbitrary dimensionality with low probability of

[1] `https://community.intel.com/t5/Intel-HPC-Toolkit/Simple-MPI-hello-world-fails-on-a-server-with-Mellanox-ConnectX/m-p/1369208/highlight/true`

collision. This resulted in longer run times though, due to the amount of operations that are necessary to provide such a hash function. The experiments conducted in the context of this thesis instead employ simplified hash functions, specifically tailored to be as fast to compute as possible for the dimensionality of the given arrays. In case of the one-dimensional arrays for example, the hash function simply added the index to the base address of the `MemoryAbstraction`. This is servicable for one-dimensional arrays, resulting in very few collisions, but not advisable for arrays of higher dimensionality. Chapter 7 provides some more information on how such custom hash functions can be utilized in the future to ensure that the hashing is done as efficiently as possible.

## A.4. Benchmarks

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <omp.h>
4  #include <sys/resource.h>
5  #include <sys/time.h>
6  #include "cato_control.h"
7
8  #define DIM <INSERT MATRIX SIZE HERE>
9  #define ITER 10
10
11 #define ind(x,y) ((x)*DIM + (y))
12
13 int main(){
14     double* matrix = malloc(sizeof(double) * DIM * DIM);
15     double* matrix2 = malloc(sizeof(double) * DIM * DIM);
16
17     disable_index_cache();
18     disable_read_cache();
19
20     #pragma omp parallel for
21     for(int i = 0; i < DIM; i++){
22         for (int j = 0; j < DIM; j++){
23             matrix[ind(i,j)] = 0.0;
24             matrix2[ind(i,j)] = 0.0;
25         }
26     }
27
28     #pragma omp parallel for
29     for (int i = 0; i < DIM; i++){
30         matrix[ind(0,i)] = 1.0;
31         matrix[ind(DIM-1,i)] = 1.0;
```

```
32          matrix[ind(i,0)] = 1.0;
33          matrix[ind(i,DIM-1)] = 1.0;
34          matrix2[ind(0,i)] = 1.0;
35          matrix2[ind(DIM-1,i)] = 1.0;
36          matrix2[ind(i,0)] = 1.0;
37          matrix2[ind(i,DIM-1)] = 1.0;
38      }
39
40      enable_index_cache();
41      enable_read_cache();
42
43      for(int iter = 0; iter < ITER; iter++){
44          #pragma omp parallel for
45          for(int i = 1; i < DIM-1; i++){
46              for(int j = 1; j < DIM-1; j++){
47                  double star = 0.25*(matrix[ind(i-1,j)] +
                        ↪matrix[ind(i+1,j)] +
                        ↪matrix[ind(i,j-1)] +
                        ↪matrix[ind(i,j+1)]);
48                  matrix2[ind(i,j)] = matrix[ind(i,j)]/2.0 +
                        ↪star/2.0;
49              }
50          }
51
52          double* tmp = matrix;
53          matrix = matrix2;
54          matrix2 = tmp;
55      }
56
57      free(matrix);
58      free(matrix2);
59
60      #ifdef MEMUSAGE
61      struct rusage usage_end;
62      getrusage(RUSAGE_SELF, &usage_end);
63      printf("MEM USAGE END: %ld KB\n", usage_end.ru_maxrss);
64      #endif
65  }
```

Listing A.2: The OpenMP stencil kernel.

Listing A.2 provides the code for the stencil kernel. The header file named `cato_control.h` contains the function declarations to manually control the caches and set the fences. The header and the functions were commented out where necessary to achieve the wanted effect (i.e. if the performance of the original OpenMP kernel was tested).

Note the memory allocations, as each matrix is stored in a contiguous array with a macro for index translations.

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/resource.h>
#include <sys/time.h>
#include <omp.h>
#include "cato_control.h"

#define DIM <INSERT MATRIX SIZE HERE>
#define ELEM <INSERT NUMBER OF NONZERO ENTRIES HERE>
#define DENSITY 0.1

int main(){
    size_t buff_size = ELEM;
    double* data = malloc(sizeof(double) * buff_size);
    size_t* indices = malloc(sizeof(size_t) * buff_size);
    size_t* ptr = malloc(sizeof(size_t) * (DIM + 1));
    double* result = malloc(sizeof(double) * DIM);
    double* vec = malloc(sizeof(double) * DIM);

    srand48(42);

    struct timeval start;
    struct timeval end;
    gettimeofday(&start, NULL);

    size_t current_index = 0;
    fence();
    for (size_t i = 0; i < DIM; i++){
        double d = drand48();
        if (i % 2 == 0){
            d *= -1.0;
        }
        vec[i] = d;
        ptr[i] = current_index;
        for (size_t j = 0; j < DIM; j++){
            if (drand48() < DENSITY){
                data[current_index] = drand48();
                indices[current_index] = j;
                current_index++;
            }
```

```
41          }
42      }
43      ptr[DIM] = current_index;
44      fence();
45
46      gettimeofday(&end, NULL);
47      if (omp_get_thread_num()==0){
48          double time = (end.tv_sec - start.tv_sec) +
              ↪(end.tv_usec - start.tv_usec) * 1e-6;
49          fprintf(stderr, "%.3lf\n", time);
50      }
51
52      #pragma omp parallel for default(none) shared(result,
          ↪ptr, data, indices, vec)
53      for (size_t i = 0; i < DIM; i++){
54          result[i] = 0.0;
55          for (size_t k = ptr[i]; k < ptr[i+1]; k++){
56              result[i] = result[i] + data[k] *
                  ↪vec[indices[k]];
57          }
58      }
59
60      free(data);
61      free(indices);
62      free(ptr);
63      free(vec);
64      free(result);
65
66      #ifdef MEMUSAGE
67      struct rusage usage_end;
68      getrusage(RUSAGE_SELF, &usage_end);
69      printf("MEM USAGE END: %ld KB\n", usage_end.ru_maxrss);
70      #endif
71  }
```

Listing A.3: The OpenMP SpMV kernel.

Listing A.3 provides the code for the SpMV kernel. Note the `fence` function calls. These were used to manually start and end access epochs for all shared arrays. The functions are all present in CATO. However, for this to work, an extra set of sequential load and store functions is required. These are currently not shipped with CATO, but they can be easily implemented by removing the synchronization calls within these functions.

The SpMV kernel also required the exact number of nonzero entries to determine the

size of the buffers for parts of the CSR matrix, due to the absence of `realloc` in CATO. Initial experiments tried to circumvent this by adding some additional space to the buffers, setting the `buff_size` to a value such as `DIM*DIM*DENSITY*1.00005`. However, if the number of nonzero entries is not exactly equal to this number, the distribution of the entire matrix data will be skewed, as the processes with the largest ranks would mainly hold memory that is not initialized and also not part of the matrix, leading to an excessive and uncharacteristic amount of remote accesses. Therefore, the buffers needed to be fixed to the exact number of nonzero elements, which was easily determined by virtue of preceding dry runs.

The length of the initialization period was determined via the `gettimeofday` functions calls. The time that was determined as such would later on be subtracted from the total time to solution, as reported by the `time` command, to obtain the run time of the main loop. The initialization itself relied on the usage of uniformly distributed pseudo-random numbers, as generated by the `drand48` function. The seed was always set to the same value across each run.

# List of Figures

# List of Listings

# List of Tables

## Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe.

| Hamburg, 04.12.23 | Niclas Schroeter |
| --- | --- |
| Ort, Datum | Unterschrift |