Thread-based Optimization of Memory Access Patterns in ASAGI

Manuel Fasching
Thread-based Optimization of Memory Access Patterns in ASAGI

Author: Manuel Fasching
Supervisor: Prof. Dr. Michael Bader
Advisor: Sebastian Rettenberger, M.Sc.
Date: December 16, 2013
I assure the single handed composition of this bachelor’s thesis only supported by declared resources

Munich, on 16. Dezember 2013
Manuel Fasching
Abstract

ASAGI is a library which provides memory access operations for simulation applications. It tries to separate the access to input data from the application to simplify their development without worsening the performance. In high performance computing parallelization is a very important concept. ASAGI loads data to multiple nodes to keep the access time small, but it does not take care of the characteristics of distributed shared memory systems, when it accesses data. In this thesis ASAGI will be optimized for such systems. The block caching algorithm which is already implemented will be adapted for distributed shared memory systems.
# Contents

**Abstract** iv  

1 **Introduction** 1  
  1.1 Motivation .................................................. 1  
  1.2 Outline of the Thesis ........................................ 1  

2 **Non Unified Memory Access** 3  
  2.1 Basics .......................................................... 3  
    2.1.1 Shared Memory .............................................. 3  
    2.1.2 Distributed Memory ...................................... 3  
    2.1.3 Distributed Shared Memory - NUMA .................... 5  
  2.2 Cache Coherence ............................................... 5  
    2.2.1 Snooping Protocols ...................................... 6  
    2.2.2 Directory-Based Cache Coherence Protocols .......... 6  

3 **MPI** 9  
  3.1 Basics .......................................................... 9  
  3.2 Example program ............................................. 9  
  3.3 Multithread Support ......................................... 10  
  3.4 Remote Memory Access ....................................... 11  
    3.4.1 Active Target vs Passive Target Communication ..... 12  
    3.4.2 Synchronisation ......................................... 12  

4 **Threads** 14  
  4.1 Basics .......................................................... 14  
    4.1.1 Thread States .............................................. 15  
    4.1.2 Thread Functions ......................................... 15  
    4.1.3 Kernel Threads and User Threads ........................ 16  
  4.2 Critical Sections ............................................. 16  
    4.2.1 Mutual Exclusion ......................................... 17  
  4.3 Implementations ............................................... 17  
    4.3.1 POSIX Threads ............................................. 17  
    4.3.2 OpenMP .................................................... 19  

5 **Related Work** 21  
  5.1 ASAGI ............................................................ 21  
    5.1.1 Short Interface Description ............................. 21  
    5.1.2 Basic Example ............................................. 22  
    5.1.3 Block Transfers ........................................... 23
Contents

6 Problem Statement 24
   6.1 Hybrid Applications ........................................... 24
   6.2 A Hybrid Approach on NUMA Architecture ..................... 24
   6.3 A Short Example for a Hybrid Approach on NUMA Architecture 25

7 Implementation 26
   7.1 Basic Idea ....................................................... 26
   7.2 Thread Handling ............................................... 26
   7.3 Thread Synchronization ......................................... 28
   7.4 MPI Support .................................................. 29
      7.4.1 Multithread Issue of RMA .................................. 30
   7.5 Accessing Remote Memory on one Process ................. 30

8 Results 32
   8.1 SWE ............................................................. 34
   8.2 Darcy .......................................................... 34

9 Conclusion And Future Work 37
   9.1 Conclusion ..................................................... 37
   9.2 Future Work .................................................. 37
1 Introduction

1.1 Motivation

In today’s supercomputer multiple processors work in combination to gain performance. An important usage of this computers is to calculate simulations. Applications which are programmed for such systems tries to use the resources to full capacity. This means they are highly parallelized and run on several processors. Each processor calculates a specific part of the task. When parallelizing applications, it’s important to be aware of the memory design. One approach is, to save only those data at local nodes, which are really needed on this node. The access to local stored data can happen very fast.

One way to discretize abstract models is to use the concept of grids. For an optimal usage of resources, the grid has no uniform resolution. These grids are called adaptive grids. Dynamic adaptive grids changes their resolution during the simulation.

Due to such dynamic adaptive processes, the data have to be migrated between the nodes. For this purpose techniques like message passing can be used. It’s not an easy challenge to develop parallel applications with such dynamic adaptive processes. ASAGI is a try to seperate the access to input data from the application, to simplify their development. The programmer does not take care of the data location anymore. ASAGI takes care of it. It transfers data from a remote node to the local node, if the application requires appropriate values. To optimize the performance, it uses a block caching algorithm. When ASAGI is used on distributed shared memory systems, it could derive advantage from the memory design of such systems. The main goal of this thesis is, to adapt ASAGI, to use this advantage. It’s essential that the already implemented block caching algorithm does not only cache data from remote nodes but from remote memory inside a node, too.

1.2 Outline of the Thesis

In this thesis the question will be answered how much performance can be gained by optimizing ASAGI for NUMA systems.
Chapter 2 describes NUMA architecture with its charactaristics and figures out differences to other architectures.
In chapter 3 the MPI standard is shortly explained to acquaint the reader with the basic concept of mesage passing. ASAGI uses this standard to transfer data between multiple processes.
Chapter 4 describes the basic concept of threads and gives an short insight in the challenges for programmer when they want to parallelize a programm. At the end the most common libraries are shortly introduced.
Chapter 5 and 6 attends on the ASAGI library and its problems on distributed shared memory systems.
After the basic concepts the library itself and the problem was explained, the implemented
solution is introduced in chapter 7.

Chapter 8 answers the question how good the optimized library scales in contrary to its
non-optimized version. For the answer a simulation application called Sam(oo)*2 was used
for the tests.
2 Non Unified Memory Access

2.1 Basics

The bottleneck of todays computers is, to access data in the main memory. Modern processors are able to operate much faster, than the memory controller is able to provide data. Figure 2.1 shows the increase of processor speed versus memory access time since 1980. The memory access time cannot keep up with the processor speed growth. In the future this will become even worse. In the following subchapters the three basic memory designs are explained.

![Figure 2.1: Evolution of the processor-memory performance gap since 1980](image)

2.1.1 Shared Memory

The basic concept of a shared memory architecture is, that each core can access the whole physical main memory via a bussystem. Figure 2.2 shows such a design. With the expansion of multicore systems, the memory bottleneck growths. Many cores compete for accessing the memory via the bussystem, but only one core can access the memory at the same time. [HP12]

2.1.2 Distributed Memory

Figure 2.3 shows a distributed memory architecture. Every CPU has its own memory attached. The programmer is responsible for accessing remote memory. The exchange could happen by messages for example. [HP12]
Figure 2.2: The basic structure of a centralized shared-memory multiprocessor, based on a multicore chip. [HP12]
Figure 2.3: The basic architecture of a distributed-memory multiprocessor in 2011 typically consists of a multicore multiprocessor chip with memory and possibly I/O attached and an interface to an interconnection network that connects all the nodes. [HP12]

2.1.3 Distributed Shared Memory - NUMA

Distributed shared memory is a mix form of shared memory and distributed memory. A global address space exists. A core can access remote memory without any actions of the programmer. The memory bottleneck is widened, because the memory is distributed and multiple CPUs can access memory at the same time. This is possible due to multiple busses. Nevertheless there are two constraints:

1. NUMA cannot guarantee a hard time for memory operations. It depends on the memory location. Accessing local memory areas is faster, than accessing remote memory areas.

2. When multiple CPUs wants to access the physical memory of the same remote CPU at the same time, they have to compete for the access.

The big difference between distributed memory and NUMA is, that no global address space exists in distributed memory design. The CPU only knows its own memory space. In NUMA the memory addresses are mapped to the local and remote memory locations. [HP12]

2.2 Cache Coherence

A cache is a high speed buffer, which buffers data in case an application wants to access the same data later. The memory is separated into blocks, which are copied to the cache when required. The blocks are called cache lines. Of course, the cache size is not as big as the memory size. This means, that only a part of the memory can be copied into the cache. If a cache line does not longer fit any data have to be dropped out of the cache, to get space for new cache lines. Therefore a cache algorithm is needed.
In NUMA each processor has its local memory and its local cache. This causes some problems. Due to the fact, that every processor has its own copy of the data in its local cache, it’s possible that one processor holds invalid data. For example:

- Processor A and B get copies data from the main memory in their local caches
- Processor B manipulates the data, and writes the manipulated data back to the main memory.
- Processor A manipulates the data too, but calculates with old data, out of its cache.

If the caches are synchronized, it’s called coherent. The following protocols ensure cache coherence without the need of manually influence. [HP12]

### 2.2.1 Snooping Protocols

The basic concept of snooping is, that every processor listens to a snooping bus. If data at a specific address have changed by another processor, this address is broadcasted at the bus. A processor, which previously loaded data from this memory location has to update its local cache. Two different protocols can ensure this update operation.

For data changes in the main memory, the processor needs exclusive access to the data items. This method is called **write invalidate protocol**. Table 2.1 shows an example of such a write invalidate protocol for a snooping bus. When a processor changed a data item, other copies must be invalidated. This is done via the snooping bus. An alternative is the **write update protocol**. Every cached copy is updated when the original data item is changed. This method needs more bandwidth, because it has to broadcast all changes to the distributed caches.[HP12]

### 2.2.2 Directory-Based Cache Coherence Protocols

A directory tracks the state of each cache line. The following list shows example states for a simple protocol.

- **Shared** - One or more processors have the block cached and the value in memory is up to date (in all the caches).
- **Uncached** - No processor has a copy of the cache block.

<table>
<thead>
<tr>
<th>Processor activity</th>
<th>Bus activity</th>
<th>Contents of CPU A’s cache</th>
<th>Contents of CPU’s cache</th>
<th>Contents of memory location X</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU A reads X</td>
<td>Cache miss for X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CPU B reads X</td>
<td>Cache miss for X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CPU A writes a 1 to X</td>
<td>Invalidation for X</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CPU B reads X</td>
<td>Cache miss for X</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2.1: An example of an invalidation protocol working on a snooping bus for a single cache block (X) with write-back caches [HP12]
• Modified - Exactly one processor has a copy of the block and it has modified the block, so the memory copy is out of date. The processor is called the owner of the block.

Figure 2.4 shows the basic architecture, when a directory-based protocol is used. It keeps track of processors having a copy of the same data item as the local node. The communication between the memory, the processor, and the dictionary may happen over a common bus, or via a separate port to the memory, or the dictionary is part of a central node controller. Table 2.2 shows possible messages, which can be sent by a directory-based cache coherence protocol.

The advantage of this type of protocols is that it uses less bandwidth than a snooping protocol. The messages are forwarded from one point to another and not broadcasted. Because of this, a directory-based protocol is mostly used in large systems with many processors. [HP12]
Table 2.2: The possible messages sent among nodes to maintain coherence, along with the source and destination node, the contents (where P = requesting processor number, A = request address, and D = data contents), and the function of the message.[HPI2]
3 MPI

3.1 Basics

Table 4.1 lists the shared items of threads and processes. If an application is parallelized by processes, the processes must communicate with each other. In contrast to threads, processes do not share the address space. This means, a process cannot directly access the memory of another one. The Message Passing Interface (MPI\cite{For09}) specifies operations and routines for communication and synchronization between processes. MPI is only a specification. There exist several implementations of MPI. The basic concept of parallelization with MPI is, that each process executes a copy of the same program. MPI assigns a number to each process, starting at zero. It can be considered as an ID. Due to this numbering, MPI can identify an explicit process. MPI processes are called ranks. With the usage of library functions the programmer can determine on which rank the program is executed, and start the appropriate routine for this rank\cite{And00, For09}.

3.2 Example program

The following C program shows, how MPI can be used for data exchange between two processes.

```c
#include <mpi.h>

main(int argc, char *argv[]) {
  int myid, otherid, size;
  int length = 1, tag = 1;
  int myvalue, othervalue;
  MPI_Status status;

  /* initialize MPI and get own id (rank) */
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);

  if (myid == 0) {
    otherid = 1; myvalue = 14;
  } else {
    otherid = 0; myvalue = 25;
  }

  MPI_Send(&myvalue, length, MPI_INT, otherid, tag, MPI_COMM_WORLD);
```
MPI_Init initializes the MPI library and gets a copy of the command line arguments, which are copied to all instances. A further effect is, that the MPI_COMM_WORLD variable is instantiated, which is the collection of all started processes. MPI_Comm_size determines the number of processes. MPI_Comm_rank determines the current rank of the process. MPI_Send is used for sending a message to another process. MPI_Recv is used for receiving messages of another processes. MPI_Finalize cleans up the library and terminates the process.

To run a program in parallel, it has to be started with an appropriate command. (e.g. mpiexec). As an argument of this command, you can specify the number of processes. Let us assume, that this program was parallelized by two processes. Until line 19, only variables were set up and the MPI library was initialized. In line 19 both processes send the value of myvalue to the other process. In line 20 the value of the other process is received. This is a symmetric exchange, because one send statement is followed by one receive statement.

If this programm would be parallelized by more than 2 processes, ranks 1 to n would send their value to rank 0, but rank 0 would only receive one value from another process and send its value only to rank 1. This means, values were sent, but not received. To provide synchronisation the MPI_Send function delays until the message was buffered, or received. This can cause problems, because a MPI implementation must not buffer messages. This would lead to dead locks. Further description of MPI functions can be found in the official MPI documentation.

3.3 Multithread Support

Regarding the official MPI documentation all MPI functions are thread-safe.

"All MPI calls are thread-safe, i.e., two concurrently running threads may make MPI calls and the outcome will be as if the calls executed in some order, even if their execution is interleaved." 

This means the programmer does not need take care of serializing MPI calls. The library ensures this. Nevertheless there are three things, which a programmes has to consider.

1. Many MPI libraries must be especially compiled with multithread support.
2. For initializing the library the MPI_Init_thread function has to be used.
3. The programmer has to specify the desired the desired level of thread support. See table 3.1
### 3.4 Remote Memory Access

ASAGI does not use the “traditional” communication mechanism of MPI, which is described in the previous section. It uses remote memory access. This type of communication allows one rank to specify all communication parameters. There is no more need for sending or receiving data, like previously described. A rank can access a specified memory area of a remote rank, without the assistance of it. When using this mechanism, each rank which wants to provide access to its data, has to initialize a MPI window. The following function can be used to specify such a window.

```c
int MPI.Win_create(void *base, MPI_Aint size, int disp_unit,
                    MPI_Info info, MPI_Comm comm, MPI_Win *win);
```

<table>
<thead>
<tr>
<th>base</th>
<th>initial address of window (choice)</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>size of window in bytes (non-negative integer)</td>
</tr>
<tr>
<td>disp_unit</td>
<td>local unit size for displacements, in bytes (positive integer)</td>
</tr>
<tr>
<td>info</td>
<td>info argument (handle)</td>
</tr>
<tr>
<td>comm</td>
<td>communicator (handle)</td>
</tr>
<tr>
<td>win</td>
<td>window object returned by the call (handle)</td>
</tr>
</tbody>
</table>

When a rank has initialized a window, a remote rank can perform actions on it, by using one of these three functions:

```c
int MPI.Put(void *origin_addr,
            int origin_count, ...
```

```c
int MPI.Get(void *origin_addr,
            int origin_count, ...
```

---

Table 3.1: Possible levels of thread support [For09]
By using MPIPut it is possible to inject data in a remote window location. MPIGet reads a specified part of the remote window. MPIAccumulate accumulates the origin buffer, and the specified remote buffer in the target window. ASAGI uses MPIGet to load data from a remote rank into its local memory.[For09]

3.4.1 Active Target vs Passive Target Communication

RMA communication is divided in two categories.

- **active target communication** is similar to the standard message passing mechanism, except that only one rank is responsible for data transfer arguments. The target rank must only approve the access to its window.

- **passive target communication** means that only one process is involved in the transfer. The target provides its memory area, and each process is allowed to write or read.

ASAGI is designed to use passive target communication. [For09]

3.4.2 Synchronisation

A remote window has to be locked, when a rank performs actions on it. E.g. when executing a write operation, no other rank is allowed to read or write data into this window. Therefore many synchronisation functions are provided. We will focus on MPIWinLock, because ASAGI uses this function. It can only be used for synchronization of passive communication.
MPI

\textbf{int MPI\_Win\_lock (int lock\_type, int rank, int assert, MPI\_Win win)}

\textbf{IN} lock\_type \hspace{1em} \text{either MPI\_LOCK\_EXCLUSIVE or MPI\_LOCK\_SHARED (state)}
\textbf{IN} rank \hspace{1em} \text{rank of locked window (non-negative integer)}
\textbf{IN} assert \hspace{1em} \text{program assertion (integer)}
\textbf{IN} win \hspace{1em} \text{window object (handle)}

\textbf{int MPI\_Win\_unlock (int rank, MPI\_Win win)}

\textbf{IN} rank \hspace{1em} \text{rank of window (non-negative integer)}
\textbf{IN} win \hspace{1em} \text{window object (handle)}

\textit{MPI\_Win\_lock} starts an RMA access epoch. It ensures, that only the own rank can access the window which is specified by \textit{rank}. Other ranks would have to wait. \textit{MPI\_Win\_unlock} completes such a epoch and releases the lock. \cite{Fort09}
4 Threads

4.1 Basics

Chapter 3 describes how an application can be parallelized by using MPI. Multiple processes are started. The communication between the processes works via MPI functions. Each process has its own address space, and a single thread of control. In some situations it would be better to have multiple threads of control. These threads could run parallel and use the same address space.

One usage of processes is to group resources. Threads focus on the execution part. They use the same resources and are executed in parallel on the CPU. This is the main thing, which threads add to a process: The possibility of multiple executions. Processes can be executed in parallel too. The main difference are the shared resources. Table 4.1 shows the shared and exclusive resources of a thread. The process items are exclusive for each process, but shared for all threads, which are operating on the process. Each thread of one process can access these resources, but one process has no direct access to these resouces of other processes. Of course, there are resources, which are shared by processes too. E.g. physical memory, disks and printers. Figure 4.1 shows possible ways to parallelize programs by using threads and processes. Each thread has its own stack. Basically a stack contains the procedures which are called, but not yet returned. Figure 4.2 illustrates the design. The following example explains, why it’s important that each thread, has its own stack. Imagine procedure X calls procedure Y, and Y calls procedure Z. The frames for X,Y, and Z will be on the stack, while Z is executing. If only one stack would exist for all threads, every thread would be forced to call the same procedures and they would have to have the same execution history. Generally this is not the purpose when using threads. Due to this, every thread must have its own stack. [Tan09]

Figure 4.1: (a) Three processes with one thread each (b) One process with three threads [Tan09]
4 Threads

<table>
<thead>
<tr>
<th>Per process items</th>
<th>Per thread items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space</td>
<td>Program counter</td>
</tr>
<tr>
<td>Global variables</td>
<td>Registers</td>
</tr>
<tr>
<td>Open files</td>
<td>Stack</td>
</tr>
<tr>
<td>Child processes</td>
<td>State</td>
</tr>
<tr>
<td>Pending alarms</td>
<td></td>
</tr>
<tr>
<td>Signals and signal handlers</td>
<td></td>
</tr>
<tr>
<td>Accounting information</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: The first column lists some items shared by all threads in a process. The second one lists some items private to each thread. [Tan09]

![Diagram showing thread states and functions](image)

Figure 4.2: Each thread has its own stack. [Tan09]

4.1.1 Thread States

Table 4.2 shows the possible states of a thread with its description. Figure 4.3 the possible transitions of the states.

4.1.2 Thread Functions

A multithreaded program normally starts with one single thread, which creates other threads by using library functions, for example `thread_create`. It’s possible to create, handle, and destroy threads with the usage of such library functions. Normally a `thread_create` function returns an identifier, for each created thread. When a thread reaches the end of its procedure, it can exit, for example by using `thread_exit`. After this call it gets no more computing time from the scheduler. Another possibility is, that a thread wants to wait for the exit of another thread. This could be realized by a function like `thread_join`. The calling thread is blocked until the specified thread has exited. As a last function, `thread_yield` is introduced. By using this function a thread can give up its computing time. Other threads can run instead of the calling thread. [Tan09]
4 Threads

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>running</td>
<td>The thread has the CPU and is active.</td>
</tr>
<tr>
<td>blocked</td>
<td>The thread waits for some event to unblock it.</td>
</tr>
<tr>
<td>ready</td>
<td>The thread is scheduled to run.</td>
</tr>
<tr>
<td>terminated</td>
<td>The thread has finished or was killed by a routine.</td>
</tr>
</tbody>
</table>

Table 4.2: The possible states of a thread

![Diagram](image-url)  
1. Process blocks for input  
2. Scheduler picks another process  
3. Scheduler picks this process  
4. Input becomes available

Figure 4.3: Possible transitions from one state to another

4.1.3 Kernel Threads and User Threads

There are two implementations of threads: kernel threads and user threads. Library functions as mentioned in 4.1.2 exist only for user threads. The structure of the implementation of these library functions is illustrated in figure 4.4(a). An application running on an operating system can parallelize its execution by using these functions. Independent from processes and threads which are started by the user, the kernel could be parallelized via threads, too. These threads perform actions like writing pages to disks, or swapping processes between main memory and disk. They can run in the background completely. The kernel can be structured totally in threads. This means, when a user thread wants to perform a system call, it does not need to change to kernel mode to perform this action. It blocks and waits for a finishing of a kernel thread, which takes the control, and performs the system call for the user thread. Figure 4.4(b) shows the concept of kernel threads.

4.2 Critical Sections

Accessing shared resources from multiple threads implies synchronization problems. When multiple threads want to access the same resource at the same time, the accesses must happen consecutive. Sections where operations happen on these resources are called critical sections. Table 4.3 shows a possible program execution of two threads. Both threads manipulate the global variable x. They’ve read the value 0 from x and stored it into their local variables y. Each thread has incremented its local variable by 1, and has written back the value to x. The right output would be 2, but both threads do not know each other. The last
4 Threads

Figure 4.4: (a) A user-level threads package. (b) A threads package managed by the Kernel [Tan09]

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 x = 0</td>
<td></td>
</tr>
<tr>
<td>2 y = x</td>
<td></td>
</tr>
<tr>
<td>3 y = x</td>
<td></td>
</tr>
<tr>
<td>4 y = y + 1</td>
<td>y = y + 1</td>
</tr>
<tr>
<td>5 x = y</td>
<td></td>
</tr>
<tr>
<td>6 x = y</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.3: Simple example of a possible program execution from two threads, and one global variable x.

operation sets x to 1, which is the wrong result. This is not the expected behaviour.

4.2.1 Mutual Exclusion

Mutual exclusion is a way to sequence the access to critical sections. Table 4.4 shows the same example with the usage of mutual exclusion. Thread A performs a lock operation. This means every shared resource which is used by thread A in the section until the unlock operation will be performed, is locked. No other thread can access this resources during the operation time of thread A. If a thread wants so, it has to wait. If a thread B does not need to access a shared resource and thread A has a lock, it does not influence the behaviour of thread B. Both threads can operate simultaneously, except if a thread wants to access a resource, which is locked by another [And00]

4.3 Implementations

4.3.1 POSIX Threads

The IEEE standard 1003.1c defines a thread package, which is called Pthreads. This standard defines over 60 functions for creating and handling user threads. The most used, and
### 4 Threads

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x=0</td>
</tr>
<tr>
<td>2</td>
<td>lock</td>
</tr>
<tr>
<td>3</td>
<td>y = x blocked (B wants to read x, but A has a lock)</td>
</tr>
<tr>
<td>4</td>
<td>y = y + 1</td>
</tr>
<tr>
<td>5</td>
<td>x = y</td>
</tr>
<tr>
<td>6</td>
<td>unlock</td>
</tr>
<tr>
<td>7</td>
<td>lock (Thread A has released the lock and informed all waiting threads)</td>
</tr>
<tr>
<td>8</td>
<td>y = x</td>
</tr>
<tr>
<td>9</td>
<td>y = y + 1</td>
</tr>
<tr>
<td>10</td>
<td>x = y</td>
</tr>
<tr>
<td>11</td>
<td>unlock</td>
</tr>
</tbody>
</table>

Table 4.4: Simple example of a possible program execution from two threads, and one global variable x by using a mutual exclusion.

The most important ones for this thesis are described here.

```c
int pthread_create(pthread_t *restrict thread, const pthread_attr_t *restrict attr, void *(*start_routine)(void*), void *restrict arg);
```

“The pthread_create() function shall create a new thread, with attributes specified by attr, within a process. If attr is NULL, the default attributes shall be used. If the attributes specified by attr are modified later, the thread's attributes shall not be affected. Upon successful completion, pthread_create() shall store the ID of the created thread in the location referenced by thread.” [Gro13]

```c
void pthread_exit(void *value_ptr);
```

“The pthread_exit() function terminates the calling thread and makes the value value_ptr available to any successful join with the terminating thread. Any cancellation cleanup handlers that have been pushed and not yet popped are popped in the reverse order that they were pushed and then executed.” [Gro13]

```c
int pthread_join(pthread_t thread, void **value_ptr);
```

“The pthread_join() function shall suspend execution of the calling thread until the target thread terminates, unless the target thread has already terminated. On return from a successful pthread_join() call with a non-NULL value_ptr argument, the value passed to pthread_exit() by the terminating thread shall be made available in the location referenced by value_ptr. When a pthread_join() returns successfully, the target thread has been terminated. The results of multiple simultaneous calls to pthread_join() specifying the same target thread are undefined. If the thread calling pthread_join() is canceled, then the target thread shall not be detached.” [Gro13]

```c
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

“The mutex object referenced by mutex shall be locked by a call to pthread_mutex_lock() that returns zero or EOWNERDEAD. If the mutex is already locked by another thread, the
calling thread shall block until the mutex becomes available.” [Gro13]

```c
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

“The pthread_mutex_unlock() function shall release the mutex object referenced by mutex. The manner in which a mutex is released is dependent upon the mutex’s type attribute. If there are threads blocked on the mutex object referenced by mutex when pthread_mutex_unlock() is called, resulting in the mutex becoming available, the scheduling policy shall determine which thread shall acquire the mutex.” [Gro13]

```c
int pthread cond wait(pthread cond t *restrict cond, pthread mutex t *restrict mutex);
```

“The pthread_cond_timedwait() and pthread_cond_wait() functions shall block on a condition variable. The application shall ensure that these functions are called with mutex locked by the calling thread” [Gro13]

```c
int pthread cond broadcast(pthread cond t *cond);
```

“The pthread_cond_broadcast() function shall unblock all threads currently blocked on the specified condition variable cond.” [Gro13]

```c
int pthread cond signal(pthread cond t *cond);
```

“The pthread_cond_signal() function shall unblock at least one of the threads that are blocked on the specified condition variable cond (if any threads are blocked on cond).” [Gro13]

### 4.3.2 OpenMP

OpenMP is in contrast to Pthread library a set of compiler instructions and library routines. The API is available for Fortran, C and C++. The advantage of OpenMP is, that it allows very easy parallelization of applications. The programmer only needs to figure out, which regions of his program should run in parallel. The following Fortran code shows an example:

```fortran
!$omp parallel do
!$omp& shared(n, grid, new), private(i, j)
  do j = 2, n-1
    do i = 2, n-1
      grid(i, j) = 0.0
      new(i, j) = 0.0
    enddo
  enddo
!$omp end parallel do
```

[And00]

The compiler directive !$omp parallel do instructs the compiler, that this region should run in parallel. The sequential environment will be splitted. The only thread which had executed the program until this directive will become the master in the parallel environment, and gets the ID 0. After ending the parallel region the master will further execute the pro-
4 Threads

gram in a sequential environment. It’s possible to specify the number of threads by using the proper API function. By default as many threads would be started, as many processors are online. OpenMP has two big disadvantages. It can only be used on shared memory machines, and the programmer cannot influence the thread handling. OpenMP realizes actions like inter thread communication implicit. [And00]

Due to the reason, that OpenMP is important for applications which are using ASAGI, and not for ASAGI itself, OpenMP is not further described in this thesis. ASAGI supports applications, which are parallelized with a library based on Pthread, like OpenMP, TBB, etc.
5 Related Work

5.1 ASAGI

ASAGI is a library, written in C++, with a C/Fortran interface, which provides given input data for high performance applications. It works on the given data in grids and needs an input file in netCDF format. ASAGI stores values of this file in main memory, and the application can access the value via its coordinates. As figure 5.1 shows you, ASAGI is started on each node and communicates via MPI. That means that each MPI rank holds an instance of ASAGI which stores a specific part of the data. A MPI query is invoked, if the application needs data, which are not stored on the own rank, but on a remote rank. To get the data via MPI is much faster, than loading it from harddisk.

ASAGI is more than just a wrapper for I/O operations. It is designed for simulation application, and optimizes the time of memory access operations. Let us assume, the application simulates a seismic sea wave. The wave starts in the middle and moves in all directions. This means the probability, that the application will access a value, near the coordinates of the previous one, is high. Due to the concept of dynamic adaptive mesh refinement, the application will claim values which are not stored at the local node. ASAGI uses this probability and copies not only one value from remote memory, but a whole block, which contains values in the neighborhood of the claimed one, and stores it in the local memory. When the application further claims a value of this block, the value can be directly returned without using a MPI call, which results in a decrease of return time.

For providing this function, an interface for applications is needed. A main goal of ASAGI was, to keep the interface as simple as possible.

More information about the basic idea and the design of ASAGI are furnished in the Master’s Thesis of Sebastian Rettenberger: “Ein paralleler Server für adaptive Geoinformation in Strömungssimulationen”[Ret12b]

5.1.1 Short Interface Description

Table 5.1 shows the most important functions of ASAGI.
5 Related Work

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle create(type, hint, levels)</td>
<td>Creates the handle, which is used for accessing the grid.</td>
</tr>
<tr>
<td>error open(handle, filename, level)</td>
<td>Loads data from netcdf file, and distributes it to the ranks.</td>
</tr>
<tr>
<td>get_buf_3d(handle, buf, x, y, z, level)</td>
<td>Gets the value, which is described by coordinates, and loads it into buf.</td>
</tr>
<tr>
<td>close(handle)</td>
<td>Deallocates the memory.</td>
</tr>
</tbody>
</table>

Table 5.1: ASAGI core functions

get_buf_3d(handle, buf, x, y, z, level) is one example for a “get” function. Regarding the type, and dimension many “get” functions exists, but their basic usage is the same. This short description is just for a basic understanding. A detailed description is available in the ASAGI User Manual.[Ret12a]

5.1.2 Basic Example

The following example shows, how the API is used to load and access data in C++.

Listing 5.1: Minimum Example - Load a NetCDF file and print a value

```cpp
#include <mpi.h>
#include <asagi.h>
#include <iostream>

using namespace asagi;

int main( int argc, char** argv )
{
    MPI_Init(&argc, &argv);
    Grid* grid=Grid::create();

    if( grid->open("/path/to/netcdf/file.nc" != Grid::SUCCESS) )
    {
        std::cout << "Could not load file" << std::endl;
        return 1;
    }

    std::cout << "Value at (0,0):
    " << grid->getFloat2D(0,0) << std::endl;

    Grid::close();
    MPI_Finalize();
    return 0;
}
```
5.1.3 Block Transfers

An important detail of ASAGI is the block transfer strategy. It organizes values in blocks. Depending on the blocksize, a block consists of a given amount of values. If the application requires a value, which is not located on the local process, a MPI transfer occurs. As described in section 5.1, the probability that the next “get” call requires a value near the previous one is high. Due to this, ASAGI transfers the whole block in its slave memory. The following listing describes the routine when a get function is called:

1. Calculate the block number $i$ from the coordinates
2. If block $i$ is located in the local memory area, get the value and jump to 5.
3. If block $i$ is not located in the local cache, calculate the appropriate process number, and transfer the block via MPI to the local slave memory.
4. Read the value from the local slave memory and return it.
5. Finish

Figure 5.2 and 5.3 demonstrates such block transfers.
6 Problem Statement

6.1 Hybrid Applications

A common way to parallelize processes is to use MPI. It results in a lot of overhead if you start a MPI process on every core. To keep the overhead as low as possible, applications often implement a hybrid approach. This means, one MPI process would run on every node. Each MPI process is further parallelized via threads, so that every core can be used. Threads are created with lesser overhead than MPI processes. Figure 6.1(a) shows you such a design.

This is possible, due to the fact, that modern MPI implementations are threadsafe.(Section 3.3) This design has another advantage. If multiple threads want to access the same block of the grid, it has to load the block only once, because ASAGI manages one memory area per process. Each thread can access this memory block.

6.2 A Hybrid Approach on NUMA Architecture

On NUMA architectures an approach like 6.1(a) would not be the optimal way to parallelize an application. ASAGI could perform better. The advantage of NUMA would not be used, because the data is stored on one physical memory area. Only these threads which are pinned to the local cores can access the memory in a fast way. Of a remote thread’s point of view, it is a remote memory area. As chapter 2 describes, it takes more time to access such a remote memory area, due to the bus bottleneck.

![Figure 6.1: Parallelization inside a node](Ret12b)
6 Problem Statement

6.3 A Short Example for a Hybrid Approach on NUMA Architecture

Figure 6.2 shows a simple example of NUMA architecture. Let us assume that each CPU has four cores. Our example application would be parallelized by 16 threads. We skip the inter node communication, because it does not matter for the main problem. When the programmer parallelize the application by MPI, there is no need for optimizing the communication between two MPI ranks, because MPI is a standard, which is already designed for high performance applications. Supposed Thread 0 works at CPU 0, allocates the memory and loads the data from a netCDF file.

Let us assume a “get” call from thread 15 (working on CPU 3) occurs. It wants to access the data item at the coordinates (0,0) which are located in the memory of CPU 0. The value is transmitted via the high speed bus over CPU 0, because for the application and ASAGI every memory address at the same NUMA node seems to be local. ASAGI is mainly used for simulation applications. (See section 5.1.) Thus the probability, that the next “get” call will claim a value near to the previous one, is high. It would be better ASAGI copies the whole memory block in its local cache, and does not load each individual value via the high speed bus. But therefore it has to recognize that the memory address is located at a remote area. Of course, it would be possible to start one MPI process per core or CPU, but as described MPI produces a lot of overhead. It would be better if ASAGI accesses different memory areas within one MPI process.
7 Implementation

7.1 Basic Idea

ASAGI stores values in different types of grids. Table 7.1 shows an overview of this types. When a MPI process is parallelized by threads, only one memory location is provided for all threads. Only the access is parallelized. This is a good design for shared memory machines because the access time is independent from the memory location, but on NUMA Architecture it does not take care about it. Figure 7.1(a) shows this design.

When each thread holds its own part of the grid, it is possible to differentiate between remote and local memory. This is possible due to the fact, that each thread is pinned to one core, and one memory location. With the thread ID, it is possible to determine the location. The basic model of block copy will be extended with a check, if the value is located in remote memory of another thread. Figure 7.1(b) illustrates the idea which supports NUMA architecture.

7.2 Thread Handling

To keep the effort for the adaption of applications as low as possible, only one handle for all threads exists. It does not matter which thread calls a function. ASAGI differentiates between the threads. This means a new data structure, which encapsulates the grid class is provided. Figure 7.2 shows the basic concept of the handle.

<table>
<thead>
<tr>
<th>Class name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LocalStaticGrid</td>
<td>A basic class, which holds data. The data are written only once, at the instantiating. Once the grid is instantiated, this grid provides only read access.</td>
</tr>
<tr>
<td>LocalCacheGrid</td>
<td>This class holds the cache. At the beginning no data are stored in the cache. Only when a &quot;get&quot; function was called, and the claimed value is not available on the own process, the cache is filled with the block, where the value is located.</td>
</tr>
<tr>
<td>DistStaticGrid</td>
<td>The main purpose of this class is to get values by using RMA. It manages the MPI window and contains LocalStaticGrid and LocalCacheGrid.</td>
</tr>
</tbody>
</table>

Table 7.1: The basic grid classes of ASAGI.
Figure 7.1: The basic idea. 7.1(a) shows the actual design. 7.1(b) shows the design with NUMA support.

Figure 7.2: The application only communicates with the thread handler. ASAGI decides due to the calling thread, on which grid the call has to be forwarded.
7.3 Thread Synchronization

ASAGI has to know how many and which threads an application wants to use. For this purpose a parameter is added to the “create” function. The number of threads must be passed to ASAGI. The “create” function has to be called in a sequential environment only one time. Afterwards, every thread has to call registerThread() and open().

Listing 7.1: Example for initializing a parallel ASAGI instance with NUMA support in C++
```c++
#include <mpi.h>
#include <asagi.h>

using namespace asagi;

int main(int argc, char** argv)
{
    MPI_Init_thread(&argc, &argv);

    /* The main handle must be created in a sequential environment */
    Grid* grid=Grid::create_for_numa(FLOAT, 0, 1,
                                       omp_get_max_threads());

    #pragma omp parallel
    {
        /* Every thread has to call registerThread() and open() */
        if(grid->registerThread() != Grid::SUCCESS){
            printf("Could not register thread");
            return 1;
        }
        if(grid->open("/path/to/netcdf/file.nc") != Grid::SUCCESS){
            printf("Could not open file");
            return 1;
        }
    }
    Grid::close();
    MPI_Finalize();
    return 0;
}
```

Both functions registerThread() and open() are threadsafe. The programmer does not need to take care of synchronisation. The first thread which calls registerThread() will be the masterthread. All threads are waiting until the last one is registered. As mentioned in section 7.4, the open() function waits until the previously defined masterthread has allocated the memory. Then the other threads, are allowed to set their pointers, and load data.
7 Implementation

ASAGI uses RMA to get blocks from remote processes. When the grid is parallelized by threads and MPI we must take care of the MPI window. Each thread should be able to get values via RMA. When all threads share the same address space, the window can be placed over the whole space. But in the new design, each thread gets its own address space. It is no longer possible to lay a window over the address space, because the address rooms of the threads are not consecutive. Basically there are two methods:

1. Each thread gets its own window.
2. One window is provided, and the threads share the window.

When using option 2) it’s necessary that the memory addresses are mapped to the threads. Figure 7.3(a) shows such a mapping. For providing this, it’s necessary that the memory area is allocated by one thread, the others set only a pointer. This ensures, that the memory addresses are consecutive. The physically memory pages are chosen, when ASAGI writes data to an address. Each thread writes its part of the grid to the memory. From this it follows that the grid part of each thread is stored in its physically local memory, but the addresses are consecutive. Due to this, the communication between processes does not change.

Figure 7.3(b) shows option 1). This means that the basic addressing method between two processes has to be changed, because a process must not only take care of the remote rank, but of the remote thread on the remote rank, too. Each thread gets its own MPI window. Option 2) means more programming effort, when the grids are initialized, but during the
runtime, the existing inter process communication can be used. Due to this, option 2) was implemented.

### 7.4.1 Multithread Issue of RMA

Respective the official MPI documentation\[For09\], all MPI function should be thread-safe, but calling `MPI_Win_lock`, and `MPI_Get` from multiple threads causes an RMA synchronisation error. Unfortunately the reason for this issue couldn’t be detected during this thesis, due to time reasons. The calls were serialized by using the pthread spinlock mechanism. After the serialization the MPI communication works as expected.

```c
pthread_spin_lock(&m_threadHandle.spinlock);

MPI_Win_lock(MPI_LOCK_SHARED, remoteRank,
             MPI_MODE_NOCHECK, mpiWindow);

MPI_Get(cache,
        blockSize,
        getType().getMPIType(),
        offset * blockSize,
        blockSize,
        getType().getMPIType(),
        m_threadHandle.mpiWindow);

MPI_Win_unlock(remoteRank, mpiWindow);

pthread_spin_unlock(&m_threadHandle.spinlock);
```

For further research, I would recommend to prepare a minimum example, and get in touch with the MPI developers.

### 7.5 Accessing Remote Memory on one Process

A simple and fast method to copy memory content from one space to another is the C function `memcpy`

```c
void *memcpy(void * restrict s1, const void * restrict s2, size_t n);
```

"The memcpy function copies n characters from the object pointed to by s2 into the object pointed to by s1. If copying takes place between objects that overlap, the behaviour is undefined."\[ISC11\]

The routine described in chapter[5] is extended:

1. Calculate the block number i from the coordinates
2. If block i is located in the local memory area, get the value and jump to 7.
3. If block i is not located in the local cache, calculate the appropriate rank, otherwise jump to 6.

4. If block i is located on the own process, calculate the thread id, copy the block via `memcpy` in the local cache, and jump to 6

5. If block i is located on a foreign process, calculate the appropriate process number, and transfer the block via MPI to the local cache.

6. Read the value from the local cache and return it.

7. Finish

This results in a three-level hierarchy for the lookup of a value.

1. Local memory area

2. Remote memory area on the own rank

3. Memory area on a remote rank.

If the claimed value is located in a remote memory area on the own rank, it is copied in the own slave memory by using the `memcpy` function. Figure 7.4 shows an example. `memcpy` is not thread-safe, but the target (the slave memory) is locked before any data are stored into it. When reading, no lock is necessary, because this is not an critical operation.

Figure 7.4: ASAGI running on one process and four threads on one local node. A grid containing 15 blocks is already initialized. The dotted lines show the possible accesses of thread 0. Any blocks would be copied with the `memcpy` function.
8 Results

This chapter describes the performance change due to the adaption of ASAGI for NUMA architecture. To measure the performance both versions of ASAGI were tested with a real application. The tests were executed on the following hardware:

- SuperMUC: Intel Xeon Westmere-EX (40 Cores/Node), 256 GB shared memory / Node, InfiniBand - network, IBM®MPI

To provide authentic test conditions, the real simulation application Sam(oa)² [MRB12] was used. Three test cases were formed:

1. Parallelization through MPI
2. An hybrid approach with NUMA support of ASAGI
3. An hybrid approach without NUMA support of ASAGI

The time which the application needs for ASAGI functions during the simulation was measured. Each test case were executed with the following scenarios:

- Darcy
- Shallow Water Equations

The graphs only show the degree of parallelization. For test case 1 it means the amount of MPI processes which are started. Table 8.1 shows the number of processes and threads for test cases 2 and 3.

<table>
<thead>
<tr>
<th>Degree of parallelization</th>
<th>Processes</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>40</td>
<td>1</td>
<td>40</td>
</tr>
<tr>
<td>80</td>
<td>2</td>
<td>40</td>
</tr>
<tr>
<td>120</td>
<td>3</td>
<td>40</td>
</tr>
<tr>
<td>160</td>
<td>4</td>
<td>40</td>
</tr>
</tbody>
</table>

Table 8.1: Number of processes and threads for the hybrid approach
Figure 8.1: SWE
8.1 SWE

Figure 8.1 illustrates the results of the tests with scenario SWE. Figure 8.1(a) shows that the difference between the ASAGI versions, increases significant from 40 threads. The reason for the consistent progress until this point could be, that the high speed bus which connects the sockets, is not hit busy before. Unfortunately testcase 1 fails over 40 processes. The reason for this couldn’t figured out within this thesis.

Especially with a low degree of parallelization, threads show their advantage. Of degree 40, the performance gain becomes noticeable. Without NUMA support, ASAGI would scale better, with a parallelization only through MPI. Due to the adaption, ASAGI scales better when it’s parallelized with threads, inside a node.

Figure 8.1(b) shows, that the performance gain of ASAGI affects the simulation time. Due to the trend, we can assume, that the application scales best with a non hybrid approach.

8.2 Darcy

Scenario darcy illustrates basically the same picture as SWE. Figure 8.2(a) shows an greater increase of the access time from 40 threads of the ASAGI version without NUMA support. For a better view figure 8.3(a) shows the same test from 40 threads. Unfortunately this time a test with an hybrid approach failed at 4 processes and 40 threads, but we can see that ASAGI with NUMA support is clearly faster than without again. The reason for the increase of time with an hybrid approach from 40 threads, can be traced back to the application, because the simulation time (Figure 8.3(b)) shows the same picture.

In both tests a parallelization without an hybrid approach was the best choice for $\text{Sam}(\text{oao})^2$. This depends on the application, and is not universal valid.
Figure 8.2: Darcy
Figure 8.3: Darcy from 40 threads
9 Conclusion And Future Work

9.1 Conclusion

The main goal of this thesis was to find out how big the performance loss is, when ASAGI is used with an hybrid approach on NUMA architecture, and to develop a version which avoid this loss. Chapter 8 shows, that ASAGI scales much better on NUMA systems with the adaption.
Due to the adoption of the existing interface, it’s easy for the programmer to migrate his application. Only the initialization has changed, the “get” calls have remained equal, due to one handle for all threads.
Unfortunatly the tests confine to one system and one application. Because of time reasons, the performance gain could not be approved with other applications. Some tests have failed because of unknown reasons, but the important one, which answer the main question have passed. Due to the failed tests it was not possible to determine, if ASAGI would scale better when it is parallelized through MPI only, than an hybrid approach, for scenario “SWE”. But the trend, and the results of the second scenario “darcy”, suggest that a hybrid approach is the best choice for ASAGI now, when it is used on NUMA architecture.

9.2 Future Work

At this time, each thread initalizes its own secondary memory. This is not necessary. It would be enough, if one secondary memory exists for each socket, because every socket has its own local memory attachted. ASAGI would have to check on which socket a thread works and create the secondary memory only once per socket. All threads which are pinned to a core on the socket get only a pointer to the memory address. To detect the socket id, likwid [THW10] could be used. One secondary memory per socket would probably increase the performance, because each thread on one socket could use the same secondary memory. This means, when 2 threads claims a value from the same remote block, it has to be transfered only once. The next thread, could read the value from the shared secondary memory.
Bibliography


