Institut für Höchstleistungsrechnen

Bachelorarbeit

Execution of Task Parallel OpenMP Programs in Distributed Memory Environments - Preprocessor

Markus Baur
2872854

Course of Study:	Simulation Technology
Examiner:	Prof. Dr.-Ing. Dr. h.c. Dr. h.c. Prof. E.h. Michael M. Resch Prof. Dr. rer. nat. Dirk Pflüger
Supervisor:	Dr. Thomas Bönisch
Commenced:	June 1, 2018
Completed:	November 9, 2018

Abstract

There is a permanently increasing need for more computation power, for example in order to run more detailed simulations or use even more data for machine learning. If one only wants to use a shared memory system this poses a big problem because shared memory systems grow much faster in price than in computation power. This leads to the usage of clusters where a lot of shared memory systems are coupled together with a high bandwidth and low latency communication interface. Going down this road allows for several thousand times the computation power of one single shared memory system, or how it is called in a cluster, a single node. On the other hand programming fast and efficient code for a cluster is hard and difficult, one has to get the communication right and try not to loose to much performance with it. This thesis, together with another one by Johannes Erwerle[1], tries to automatically transform a program from the simpler code which can be run on a single node to one which utilizes a cluster and achieves the best scaling possible. In order to do this we extract OpenMP tasks from the program and offload them to other nodes in the cluster. The focus of this thesis is how to transform the code of the program so the runtime built in the other thesis can take over.

[1]It can be found at [Erw18].

Contents

* Chapter 2 is equivalent to the "Propaedeuticum" for "Simulation Technology" which can be handed in together with a bachelor thesis.

List of Figures

Listings

List of Abbreviations

API	application programming interface 13
cluster	distributed memory system 5, 13
HPC	high performance computing 14
runtime node	node which coordinates the execution of tasks 15
SIMD	Single Instruction Multiple Data 17
transpiler	source-to-source compiler 23
worker node	node which execute tasks 15

1 Introduction

1.1 Motivation

OpenMP[1] is an application programming interface (API) for relatively easy parallelisation of C, C++ and Fortran code. Because of this it is widely used in the scientific computing to speed up simulations. One problem when using OpenMP is that it requires a shared memory environment. This means OpenMP can only be used when the program is running on one node at a time and is limited to the resources available on this single node.

Building large systems with shared memory is expensive because one needs specialized and expensive hardware. Spending the same amount of money on a distributed memory system (cluster) almost always yields better overall performance. The problem is that programming for those clusters is usually much more complex due to differences in the way threads and processes share data. In OpenMP data common to all threads is shared implicitly because every thread can access the memory of the whole system. That is the reason why OpenMP needs a shared memory system. On the other hand there are clusters which typically consist of several mostly identical nodes with a communication system between them. There is no shared access to all of the memory and accessing memory on another node needs some form of communication. Because of this moving to a cluster would require a redesign of the whole application because changed memory on one node does not appear on another node and synchronizing memory is slow.

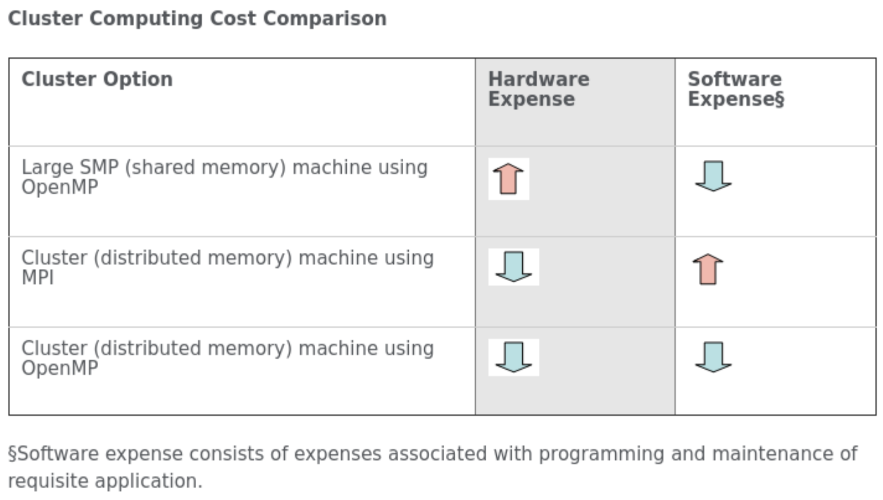
There have been several approaches to solve this problem before, some of which will be discussed later, and below you can find a still actual graphic made by Intel for their solution from 2006. This approach took OpenMP work packages and distributed it to other nodes. However, the performance gain was apparently not sufficient and the project was discontinued [Hoe06].

Other projects are still there and being actively developed, but all of them have their own drawbacks, which made them unfit for the intended purpose of this thesis.

1.2 Requirements

The proposed system has to fulfill the following requirements which originate from the day to day work of a programmer who wants to run his software on thousands of nodes but does not have the time or the resources available for a full rewrite of the software with some distributed memory API.

[1]The main source of information about OpenMP for this thesis is the OpenMP 4.5 specification from [Ope15]


Cluster Computing Cost Comparison

Cluster Option	Hardware Expense	Software Expense§
Large SMP (shared memory) machine using OpenMP	⇧	⇩
Cluster (distributed memory) machine using MPI	⇩	⇧
Cluster (distributed memory) machine using OpenMP	⇩	⇩

§Software expense consists of expenses associated with programming and maintenance of requisite application.

Figure 1.1: Cost comparison by Intel in 2006

https://software.intel.com/en-us/articles/cluster-openmp-for-intel-compilers

1.2.1 Use C++ as the base language

A large part of today's high performance code is written in C++ and rewriting it in a special high performance computing (HPC) language like Julia[2] or Regent[3] is not possible because the code was written years ago by students or employees who are no longer available. Because many of those programs already use distributed memory parallelism with OpenMP this lead to the next requirement.

1.2.2 Use OpenMP as a frontend to the developer

OpenMP is widely known by the target audience of this project and has a stable API. There was also the idea to do no changes to the OpenMP API because this would allow the program to compile for a single node with OpenMP without changes to the source. OpenMP also allows incremental parallelisation of already existing C++ source code which is extremely important for an existing codebase.

1.2.3 Scalability

There are some solutions like XMP or ClusterSS which achieve almost all of the above but which do not scale on large clusters. That is why this requirement was introduced. The solution should scale up to dozens and ideally hundreds of nodes and it must allow to use the memory on those nodes independently and thus not synchronize the memory from the main node nor use a virtual global address space. The problem here is that there is a limitation regarding the data structures which can

[2]A specialized language for HPC, see [BEKS14] for more information about the language.
[3]Another specialized language, see [SLT+15] for more information.

be transmitted, because pointers and values can not be distinguished at a low level and thus the data some pointers reference might be missing, or the available overall memory is synchronized between nodes and thus the whole program is limited to the amount of memory present on one node. There are solutions which work in between those two extremes, but each approach has drawbacks of it's own, so this is the requirement which excluded the existing solutions from the list as discussed in section 2.

1.3 Thesis splitting

In order to deal with all the work required for this project it was decided to split the work into two bachelor theses. One for the runtime and it's components like a simple scheduler, node which coordinates the execution of taskss (runtime nodes) and node which execute taskss (worker nodes) and one for transformation of the codebase and the bits needed for combining everything. In this thesis the focus is on the last part, the first part was going to be completed by another bachelor thesis, but this thesis will not be completed. It was decided that every code that runs during the compile step is covered by this thesis and every code which runs during execution of the resulting program is covered by the other thesis. There is one exception to that, namely the code to discover the size of variables is also part of this thesis even if it is ran during execution of the resulting program. Because of this most of the work presented here has not been integrated with the runtime and thus there are no performance measures included. In the end this thesis covers the part needed to transform the OpenMP program and to enable the runtime to run the tasks.

2 Related work

2.1 Beginnings of parallel computation

Parallel computing has a long history, especially within numerics and scientific computing[1]. Since then there were numerous new achievements in hardware as well as software, but Single Instruction Multiple Data (SIMD) hardware still provides the major boost to computation performance[2]. Approximately at the same time when the first SIMD hardware was introduced there was also a leap in software parallelism. The first multitasking operating system surfaced in 1961, it was now possible to run multiple programs within one computer[3].

2.2 Processes and Threads

A process provides the resources needed to execute a program e.g. address space, code, handles, environment variables and at least one thread of execution. More threads can be created in any thread, but each process starts with one thread. Threads share all process resources and can be scheduled for execution. Every thread also contains the current state of a the cpu when it was last executed and maybe some storage private to it. [4]

This definition covers much more aspects than the one about the Atlas Supervisor, but in the many years which passed since the creation of the latter, many different approaches to parallelism within an operating system where tried out. What we see here is a system in which a thread of execution only exists as a concept in software without any execution resources tied to it. On the other hand, thread creation is still something which has to be done in cooperation with the operating system. This requires context switches and allocation of system resources for each thread, furthermore the system wide scheduler has to keep track of every thread which is not yet finished. So another step which could have been gone is to create tasks within each process and then have threads working on these tasks. This would save a context switch to the operating system and would also allow to tailor the scheduler to the needs of the program.

[1]The first Single Instruction Multiple Data (SIMD) Processor was built in 1962, according to [Bur62].

[2]According to the top 500 list of supercomputers at [18] for june 2018, most of the systems use some kind of accelerator cards. Mostly they deliver the majority of the computation power and are SIMD hardware.

[3]This system was called Atlas Supervisor, it is described in detail at [KPH61].

[4]This definition was taken from Microsoft for Windows, see [Mic18]. It is assumed that it is similar for other operating systems.

2.3 Task based parallelism

In task based parallelism the work is divided into tasks which allow for fine grained parallelism with in the application because the overhead for creating a task is usually much lower than when creating a thread. Those tasks can depend upon each other and can be scheduled for execution on some backend, mostly a fixed set of threads called a thread pool. In order to access all the resources necessary for the correct execution of a task it usually needs access to the data generated by other tasks before and thus the easiest way for task execution is within the shared memory of a process. There are currently many different application programming interfaces, libraries and even languages which support task based parallelism. One of those application programming interfaces, OpenMP, has built in support in many C, C++ or Fortran compilers and is thus widely used in HPC. OpenMP and many other of these application programming interfaces, libraries and languages share the problem that they only produce an executable which only creates one process on one machine because it takes much less effort to stay in such a shared memory system.

2.4 Distributed memory and OpenMP

In order to scale an application and it's performance beyond the capabilities of a shared memory system it has to use a cluster[5]. Usually this is done by using MPI alongside OpenMP in order to add communications between shared memory systems (nodes) within a cluster. This approach raises the complexity of a program because there are now to parallelism application programming interfaces and MPI is also much more complex than OpenMP[6]. There is also another approach to this problem, replacing MPI with built in OpenMP directives and thus running applications which use OpenMP directly on a cluster.

2.5 Running OpenMP programs on a distributed memory system (cluster)

Running OpenMP programs on a cluster is not a new idea. It has been tried with various success several times in the past. On the following pages the focus shall be on Cluster OpenMP [Hoe06], ClusterSs [TFG+11], OmpSs [Hed15] and XcalableMP [Xca17]. All of those examples try to extend OpenMP with the capability to use more than one node of a cluster. The execution and memory models are taken from the sources given in the introduction chapter of each example.

[5]See chapter 1.2 for more information about this problem.

[6]A study found that students needed twice the time and significantly more code for a the implementation of a simple algorithm when parallelisation was to be done using MPI as opposed to OpenMP. Further information about the study can be found at [HB06].

2.5.1 Cluster OpenMP

This was an approach to solve the exactly same problem as this thesis by expanding the intel compiler [7]. There is one major difference, in 2006, when Cluster OpenMP was designed, tasks were not yet a part of OpenMP. Because of this Cluster OpenMP can only be used to distribute parallelized for loops.

Programming model

Cluster OpenMP extends the OpenMP syntax slightly and can be used like the standard C and C++ OpenMP. The following additions are made, first there is a new argument `pragma intel omp sharable`, which allows for the mentioned variables to be shared, secondly there is a command line option for the compiler in order to enable Cluster OpenMP features. In the end all of Cluster OpenMP is built into the intel compiler, and there is no support by any other compiler.

Execution model

Cluster OpenMP extends OpenMP by explicitly sharing memory with other nodes and then offloading chunks of OpenMP loops to those nodes. A list of nodes and the number of processes on those nodes has to be supplied during startup. The OpenMP standard is extended by the functions needed for the memory sharing and a #pragma intel omp sharable which also marks a variable for allocation in shared memory.

Memory model

Memory in Cluster OpenMP is either shared between all nodes or private to a node. Shared memory has to be created by calling a specialized malloc function provided by Cluster OpenMP. This marks all touched memory pages as shared and allows other processes to access these memory pages. The pages are mapped in all processes at the same virtual memory location and are protected against reading and writing there. If the program tries to access one of those protected pages Cluster OpenMP catches the resulting segmentation fault and copies the most recent version to that location. Writes to the page are recorded and saved in write notices. Every time the program has to synchronize the state of the pages is exchanged and when the log of write notices gets too long the processes agree on which version of the page is the most recent one. This version is kept and all others are discarded and protected again.

[7]The manual can be found at [Int07], the white paper at [Hoe06].

Discussion

In the end Cluster OpenMP is a system which provides a virtual global address space with some optimisations for OpenMP. This conflicts with the scalability requirement because all processes have to communicate with each other on barriers and if pages which are no longer in use are not reclaimed fast enough they can take up all the memory on one node. It also needs some porting effort because every malloc or free call has to be replaced and checked for issues. Furthermore the cluster has to specifically set up for Cluster OpenMP in order to run a Cluster OpenMP program on it. On the other hand this system poses almost no restrictions on the user and thus is not really difficult to adopt.

2.5.2 ClusterSs

ClusterSs is a programming model which makes use of tasks. It was designed in 2011 at the Barcelona Supercomputing Center. More details about the system can be found at [TFG+11].

Programming model

Java is the language ClusterSs was designed for and it needs annotations to those java sources, mainly all parameters are annotated as well as some methods and even classes. There are bindings for the X10 language[8] too.

Execution model

ClusterSs uses a main node where the application itself is run and a set of worker nodes which run tasks. Tasks in ClusterSs are functions with a special designation which marks them for being run on a different node. The dependency graph is built during runtime and tries to imitate a sequential run of the program.

Memory model

Memory is transferred to the worker running a task on request, which means the workers collect the needed data from the main node or other workers. In order to make more memory available on the main node, variables can be allocated on nodes. The node with the most data available in it's cache is chosen to run any given task in order to decrease the amount of memory transfers needed to run a task. Writing always takes place on copies of the data.

[8] X10 is a language specifically designed for HPC.

Discussion

At the first glance this seems to be exactly what we want, but there is one major drawback to this model. Most HPC programmers are not used to write programs in java and especially not with annotations in java. Furthermore most existing HPC codebases are not written in java and can not be rewritten within a acceptable time. Besides that this example provided some insight into how this thesis might evolve, especially when it comes to scheduling and automatic memory transfer. On the other hand writing only to copies of the data might be a problem if the copy is a large structure. But for the runtime for this project no other solution could be found.

2.5.3 OmpSs

OmpSs was created by the same group as ClusterSs, the Barcelona Supercomputing Center. It is a improvement over ClusterSs and it uses OpenMP syntax extensions to reach it's goals. Further information can be found at [Hed15].

Programming model

OmpSs is a syntax extension to OpenMP and compatible with the C, C++ and Fortran OpenMP versions. It uses a transpiler which translates the new directives into something a standard compiler understands. Annotations like `#pragma omp task` can not only apply to code blocks but can also be used to indicate that functions should always run in a task by annotation of the function declaration.

Execution model

This example uses a model similar to the one which will be presented in this thesis. A thread starts as the master thread in which the first tasks are created. Later on other tasks may itself create tasks which might be executed on different nodes.

Memory model

Memory is seen as a partitioned space and only local memory can be used by a node. It is transferred when a new task is beginning execution and no deep copies are made. Because of this no pointers are allowed in the offloaded data.

Discussion

This example almost fulfills the requirements, but the vast amount of syntax extensions was deemed too much to neatly fit with the standard OpenMP requirement. For example every variable which enters into a task has to occur in a dependency clause to the task construct and other requirements which do not allow a standard OpenMP program to directly run on a cluster.

2.5.4 XcalableMP

XcalableMP (XMP) is an approach where everyhting is explicit, a user has to write where and how communication should happen but it uses annotations like OpenMP to achieve this. In order to further harness the available parallelism of modern systems OpenMP can be used for programming too. The language specification and further information can be found at [Xca17].

Execution model

With XMP this has to come first, because it is the base of all further explanations. When an XMP program starts it starts on all nodes at the same main method and then continues independently until it reaches a XMP construct, at which point communication takes place if needed and then continues with the program. This means that a single program is executed by all nodes with potentially different data.

Memory model

In XMP there is either local memory, which is the default, or global memory. Global memory is comprised of variables which are distributed across all nodes according to the annotations to the declaration of them and only the locally available part of this variable can be accessed directly. In order to access remote memory a communication construct or a explicit remote memory access, such as a coarray assignment has to be used.

Programming model

In XMP the code can be extended the same way as in OpenMP and one can also use coarray [9] statements. This annotations are called the global-view programming model, using coarray features is called local-view. The global-view model is oriented more towards existing code and can replace other synchronisation and communication methods like MPI which require much more programmer effort. On the other side there is support for the local-view method, which requires several additions to the code itself. Especially coarrays have to be allocated with a specialized allocation routine and coarray access is a syntax addition.

Discussion

Even though XMP is an interesting system it is not as easy to use as OpenMP, especially the distribution clauses are a challenge for a beginner. Also the local-view programming model seems to be not finalized, it lacks examples and some documentation. In the end it is too complex and powerful to for the task, especially it does not use OpenMP as the frontend and thus requires the programmer to learn another set of syntax rules and guidelines.

[9] A variable which remote copies can be referenced by a further index containing the node identifier, e.g. `A = 7` for local access and `A[3] = 9` in order to set the variable on node 3.

3 Solution Architecture

The goal of the whole project is to offload tasks from within a OpenMP program to other nodes in order to increase overall performance. The rest of the program is not changed in order to keep it as close as possible to the official OpenMP standard. There are several ways how to transfer or offload a task to another node in a cluster and fulfill the requirements[1]. Somehow the source code of the application has to be interpreted differently than in a standard compiling approach and there has to be some runtime to keep track of all the tasks in the system.

3.1 Extend the OpenMP task API

The straight forward approach to add something to OpenMP would be to write our own extension to the OpenMP standard and implement it. There are some problems with this approach, the larges one is that there is no canonical implementation of the standard because every compiler which supports OpenMP has a specialized implementation of it. This would limit the use of this work to only those programs which can be compiled with this compiler. Furthermore writing compiler extensions is not an easy task and would require a complicated setup of the compiling infrastructure, at least for the evaluated compilers, [2] and is thus out of scope for a bachelor thesis.

3.2 Build a simple transpiler

A source-to-source compiler (transpiler) takes the source code of a program and changes some parts. One example would be a translation between JavaScript and Python or in this case from OpenMP tasks into something the runtime can use to run the same code on another node.

The simple approach was to use a python script to transform the task constructs, but this approach soon proved to be unable to deal with some C++ specific issues. The first one was macro expansion which could add unbalanced braces and thus rendering the extraction of the associated code for a task impossible. Another problem was that dealing with comments and line continuation is a tedious task. Thus it is almost impossible to reliably determine which parts of the program should be transformed without using a full C++ parser in python. When this was clear, evaluation of clang as a parser was the way to go because it provided at least some kind of python bindings, but they proved not to be sufficient.

[1]The requirements can be found in section 1.2.

[2]The evaluated compilers were gcc and clang.

3.3 Clang tooling based transpiler

In order to circumvent all those problems it was decided to use one of the existing C++ parsers and, because it is relatively easy to build programs with it, clang and the llvm backend [3] was chosen. There one can hook into the code parsing and rewrite parts of the code on the fly. In this code there is a method which is called whenever the traversed code encounters a OpenMP task directive and then the necessary headers are pulled in and the task is rewritten. Furthermore the main method is rewritten in order to set up and tear down the runtime properly. The code associated with the task is extracted and stored in a globally accessible map in order to let the runtime find the code again later. Task clauses are either evaluated or prepared for evaluation and then attached to a task struct which is defined in the header[4].

In order to transfer a variable to another node in a cluster one has to determine the size of the memory the variable references. This should be done during the transpiler phase, because determining the size of a variable at runtime is a error prone and not intended in C++, especially if the variable is a pointer or an array, or even worse a pointer into an array. In the end this also leads to a real drawback, no struct, object or array which contains pointers can be sent using the current method. It was not possible to mirror all memory on every node, because this would violate the scalability requirement as detailed in section 1.2.3.

Currently the transpiler uses two pass evaluation, on the first pass the rewriting of the source code takes place, the second pass is used for all task extraction routines. This split is made because all constructs are traversed in order, so the associated code of a task is extracted before it can be processed itself.

[3]More information about clang and llvm can be found at [LA04].

[4]The header can be found in section 9.1.1.

4 OpenMP Tasking Constructs

In order to understand the steps which are needed to allow a task to be run on a different node one first has to understand OpenMP tasks in general. This chapter is about the tasking constructs as defined in the OpenMP manual [Ope15]. All terms used in the description of the internal mechanics od OpenMP are used the same way as in the manual.

4.1 Overview

There are several tasking constructs in OpenMP, but not all of them are processed by the transpiler because some are rarely used or too complex at this point in time and are thus postponed to a later version of this project. Most constructs can be configured with clauses, which are discussed later on for the relevant constructs.

All constructs start with `#pragma omp` and the name, clauses are added at the end. Some constructs modify the following block, others act more like a method call. Which blocks are modified and how is detailed in the respective paragraphs.

4.1.1 Task construct

This is the main OpenMP construct to start a new asynchronous computation. It is declared by a pragma which affects the following block, which can be a simple expression, a call or a code block. When the control flow of the program encounters a task directive, a task is generated and the encountering thread enters a task scheduling point where it can execute the new task or defer it for later execution by another thread in the current thread team. A Task construct can be modified by several clauses, for example one which demands immediate execution or another one for specifying task local data.

As this is the most important tasking construct it is transformed by the transpiler and has a separate chapter discussing all the clauses and the transformation in general.

4.1.2 Taskloop construct

A for loop modified by a taskloop construct spawns tasks for each loop iteration or a number of iterations combined. This restricts those loops to the canonical loop form as on page 53 of [Ope15], which essentially says that the loop has to have an initializer, an increment expression and a test expression. There are even more clauses than on the task construct and supporting it would require even more work. Furthermore it is rarely used in the example codes and thus was not considered worth the effort for this thesis.

4.1.3 Taskloop simd construct

This construct distributes the loop according to the taskloop construct and the resulting tasks are modified to use simd internally according to the simd constructs of OpenMP. Because of the tremendous amount of complexity here it is not addressed at all in this thesis, as it also inherits all clauses of taskloop and simd.

4.1.4 Taskyield construct

The taskyield construct is a standalone construct and thus works like a function call. When it is encountered during program execution the current task can be suspended and another task starts execution. In the runtime there is currently no support for task suspension, so it is of no use to transform it and it is not transformed yet. In the future it might be replaced by a function call into the runtime.

4.1.5 Taskwait construct

The taskwait construct is also a standalone construct, which waits for all direct child tasks to finish execution. Note that it does not wait for grandchildren, so it has to be added in every task which has to make sure all children terminate. The current task is suspended and another task can be picked up for execution if there is any. As it is an important way to synchronize and order tasks it is transformed into a function call.

4.1.6 Taskgroup construct

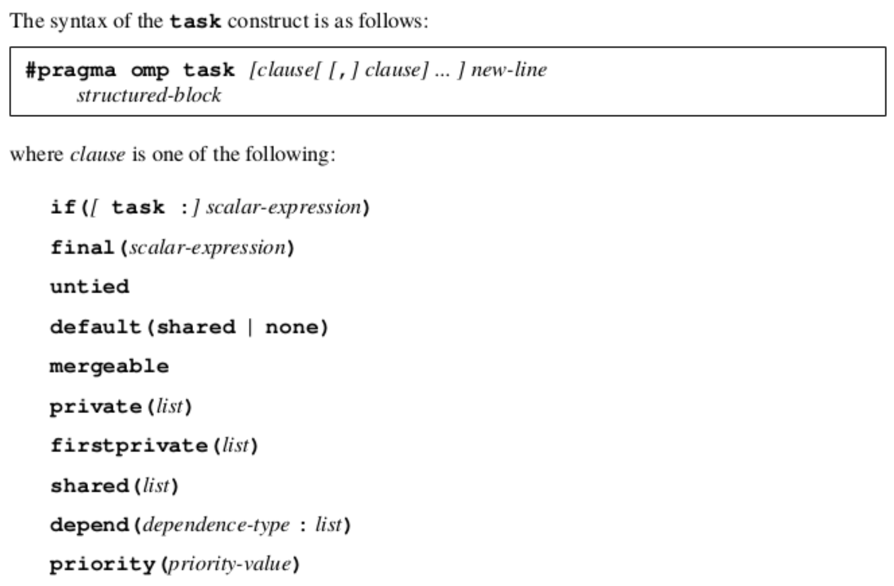
In a taskgroup construct every new tasks binds to the group and has to end before the group statement or block returns. The end of the block is also a task scheduling point where other tasks of the group can be picked up if they are not executed at the time. This construct would introduce a lot of complexity by exposing all internal bookkeeping of the runtime and is thus postponed to a later stage where the runtime is more refined and stable.

4.2 The Task Construct

An OpenMP task is declared by a tasking directive (`#pragma omp task`) and consists of the directive and the associated code section which is either a single statement or a code block. It can be further configured by clauses which are evaluated either at compile time or every time the control flow encounters the task directive.

Task completion can be guaranteed by either ending the parallel section or by a taskwait construct.

In the following paragraphs those clauses and how to apply them on a cluster shall be discussed in the same order they are defined in the OpenMP manual.



The syntax of the **task** construct is as follows:

```
#pragma omp task [clause[ [, ] clause] ... ] new-line
    structured-block
```

where *clause* is one of the following:

```
if([ task :] scalar-expression)
final(scalar-expression)
untied
default(shared | none)
mergeable
private(list)
firstprivate(list)
shared(list)
depend(dependence-type : list)
priority(priority-value)
```

Figure 4.1: The definition of an OpenMP 4.5 Task

[Ope15, p. 84].

The task structure mentioned in the following paragraphs is the one from the tasking header. [1]

4.2.1 `if` clause

When a task with an if clause is encountered during execution, the scalar expression is evaluated and it's truth value computed. If the value is `true` nothing happens, but if the expression evaluates to `false` the current task is suspended until the new task is completed and execution continues with the new task.

As this is directly impacts the control flow of the program it has to be transformed and built into the runtime. The expression is stored and evaluated when the task structure is constructed.

4.2.2 `final` clause

When a task has a final clause and the associated expression evaluates to `true` the task is a final task. A final task only generates final and included tasks if it encounters a task construct. Included tasks are executed immediately by the encountering thread and no task is generated at all. This is especially useful in order not to generate tiny tasks in a recursion where at some point the overhead of creating and scheduling a task becomes the majority of time spent on the task.

Thus the final clause is transformed and stored the same way as the if clause. As the `final` clause is inherited for all child tasks, this information has to be provided by the runtime environment as it is not known during the transpiler phase.

[1]The tasking header can be found in section 9.1.1.

4.2.3 `untied` clause

A task can be suspended when it reaches a task scheduling point such as the creation of another task or a taskwait construct. Tasks which execution already started can only be resumed by the thread which first started them if there is no untied clause present. The untied clause thus allows the task to continue execution by another thread. This might improve overall performance and responsibility of the program because one long running task can no longer block several other ones.

In the current version of the runtime there is no support for suspending tasks, but the value is parsed and stored none the less.

4.2.4 `default` clause

This clause defines the default handling of captured variables, the variables used in the code section of the task. Possible values are shared, which is the default and means that variables are shared between the task and the enclosing scope, and none, which means that every variable which is used within the task has to appear in a private, firstprivate or shared clause. If one is omitted it is a compiler error.

As this clause is evaluated during compile time and, because of this, it is not strictly necessary to parse this clause. But it has some influence if a task has variables for which no sharing clause exists and is thus parsed as `Task.shared_by_default`. This allows shortcuts in the memory sharing logic of the runtime.

4.2.5 `mergeable` clause

If a task has a mergeable clause and is an included task [2] the task may be executed within the same data environment as the generating task. This further reduces overhead of an included task.

This clause is also unused at the moment due to lack of support by the runtime but is still parsed and stored, as it will become relevant when special handling of final tasks is improved.

4.2.6 Access clauses

These clauses give detailed information about how variables are shared and they might occur several times with different variable names. They are parsed into an enumeration in the extraction of variable bindings which is described in section 5.3.2.

[2]See section 4.2.2 for the definition.

`private` clause

There is a new variable of the same kind for each task, but it is not specified how or whether it is initialized. According to the specification it should always be considered uninitialized and after the construct it is not defined which of the variables, the original one or one from the tasks, is referenced by the original reference.

`firstprivate` clause

The variable is initialized once and then copied in order to obtain a private reference within each thread. This can pose a problem if the task is untied, in this case access to this variable is non conforming and thus undefined behaviour.

`shared` clause

Variables which are defined to be shared are exactly this, every task gets the original reference to the data and the programmer has to ensure that no race conditions may ever happen. This behaviour is closely related to what happens when the programmer spawns threads on his own, because the references this thread may have behave exactly this way.

4.2.7 `depend` clause

With this clause dependencies between tasks can be defined. If a variable occurs in an ‘in’ or ‘inout’ depend clause and in an ‘out’ or ‘inout’ clause of a sibling task the task can only be started if the sibling has already finished.

The values of this clause are parsed into a list of strings for each type and are then used by the runtime to decide determine whether a task can run already or whether it has to wait for another task to finish.

4.2.8 `priority` clause

This clause is evaluated as a numerical clause and the number is a recommendation for the runtime system which tasks to run first. The higher the number the higher the priority for the task. If this clause is omitted the priority is 0.

It is extracted and then evaluated when the task is constructed, but due to the lack of a proper scheduler it is not used at the time of writing.

4.3 Other transformations

4.3.1 Taskwait Construct transformation

The taskwait construct from section 4.1.5 is transformed to a call to the `taskwait` function provided by the tasking header[3]. This function then uses the runtime in order to find all children and wait for their termination.

4.3.2 Number of threads

In OpenMP it is possible for an application the specify how many threads it wants to have at a certain point in the current region. This is usually done by calling `omp_set_num_threads` or by adding a `num_threads` clause to an OpenMP parallelisation construct. In order to fully utilize a node of a cluster it might be advantageous to allocate several tasks to this node. Therefore it is necessary to know how many cores of the node are currently in use. Furthermore is the number of threads lost when a task is transferred to another node, which might pose some issues, for example if there should only be two threads for interleaving execution of a loop or something like that. In order to deal with the complex mechanics of this, whenever such a function call or clause is discovered it will be replaced by a push and pop mechanism on a claim stack. From there a task on a new node can restore the previous settings and the runtime can take advantage of that. This mechanic is in the early prototype stage so since runtime support is still lacking.

[3]This function is not implemented yet, as it is a feature of the runtime which is not covered by this thesis.

5 Implementation Details

5.1 Data structures

In this thesis there are two major points where data is stored. One point is the transpiler itself, which has to keep the state of the currently processed source file, the other is the time when the transformed program gets executed and needs values computed during the transpiler phase. Those values are stored in structs which have been designed during this thesis and are being introduced in the next sections, they are then written to the source code of the transformed program where they are compiled into the final application.

5.1.1 Task struct

The main data structure within this thesis is the one representing a transformed task, the data structures representing the queue of scheduled tasks and the representation of nodes and running tasks is part of the runtime. In order to store all relevant information about a OpenMP task and allow passing tasks between nodes there has to be a structure representing a task[1]. It contains the id of the code section associated with the task, the values for the clauses and the prepare and schedule method which are used to make sure that the task is prepared for transferring to another node and is scheduled for execution respectively. The id of the code section, which is the hash of the file name, the line number and the location of the OpenMP task directive hashed together, is used as the “code id” in the remainder of this thesis.

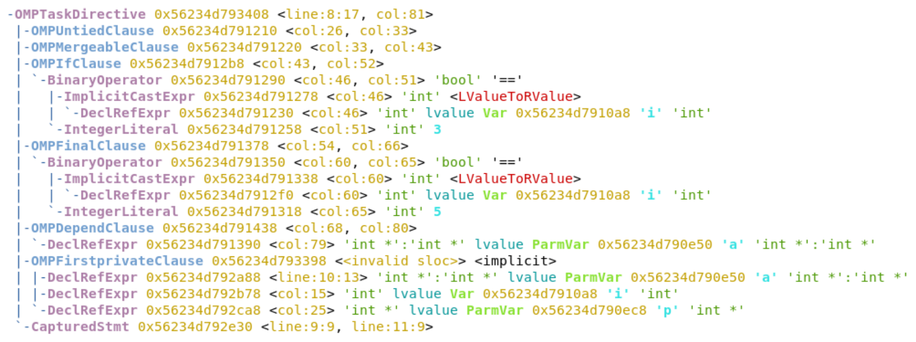
5.1.2 Var struct

Variables used within a task are represented in another data structure, the Var struct[2]. It contains the variable name, it’s address, the access type[3] and the size in bytes of the variable. The size is set to `0` if the size is unknown during the transpiler phase, the size is then calculated during the prepare method of a task by the method explained in section 5.5.2.

[1]The exact source code can be found in section 9.1.1 at line 93.

[2]The exact source code can be found in section 9.1.1 at line 86.

[3]Either `shared`, `firstprivate`, `private` or `default`, see section 4.2.6 for more information about variable access clauses.



```
-OMPTaskDirective 0x56234d793408 <line:8:17, col:81>
 |-OMPUntiedClause 0x56234d791210 <col:26, col:33>
 |-OMPMergeableClause 0x56234d791220 <col:33, col:43>
 |-OMPIfClause 0x56234d7912b8 <col:43, col:52>
 | `-BinaryOperator 0x56234d791290 <col:46, col:51> 'bool' '=='
 |   |-ImplicitCastExpr 0x56234d791278 <col:46> 'int' <LValueToRValue>
 |   | `-DeclRefExpr 0x56234d791230 <col:46> 'int' lvalue Var 0x56234d7910a8 'i' 'int'
 |   `-IntegerLiteral 0x56234d791258 <col:51> 'int' 3
 |-OMPFinalClause 0x56234d791378 <col:54, col:66>
 | `-BinaryOperator 0x56234d791350 <col:60, col:65> 'bool' '=='
 |   |-ImplicitCastExpr 0x56234d791338 <col:60> 'int' <LValueToRValue>
 |   | `-DeclRefExpr 0x56234d7912f0 <col:60> 'int' lvalue Var 0x56234d7910a8 'i' 'int'
 |   `-IntegerLiteral 0x56234d791318 <col:65> 'int' 5
 |-OMPDependClause 0x56234d791438 <col:68, col:80>
 | `-DeclRefExpr 0x56234d791390 <col:79> 'int *':'int *' lvalue ParmVar 0x56234d790e50 'a' 'int *':'int *'
 |-OMPFirstprivateClause 0x56234d793398 <<invalid sloc>> <implicit>
 | |-DeclRefExpr 0x56234d792a88 <line:10:13> 'int *':'int *' lvalue ParmVar 0x56234d790e50 'a' 'int *':'int *'
 | |-DeclRefExpr 0x56234d792b78 <col:15> 'int' lvalue Var 0x56234d7910a8 'i' 'int'
 | `-DeclRefExpr 0x56234d792ca8 <col:25> 'int *' lvalue ParmVar 0x56234d790ec8 'p' 'int *'
 `-CapturedStmt 0x56234d792e30 <line:9:9, line:11:9>
```

Figure 5.1: The syntax tree representation of the OpenMP task in the test program.

See section 6.1 for the example code.

5.1.3 `tasking_function_map`

In order to use all the extracted tasks, or better the code they contain, they have to be found during the run of the application. Thus all the code is extracted and a global map called `tasking_function_map` is created which associates the code id and the actual instructions. This is done as a easy method of making that code available on other nodes and thus allow the tasks to roam and use an identifier to find the relevant code to execute. The map is populated using static evaluation before the `main` method of the transformed program begins in order to make all functions easily available for the runtime.

5.2 Clang API

Within clang tooling there is an API for source tree traversal using the clang lexer and parser. But first one has to create a `CompilerInstance` and determine the flags with which it was called. This step is currently implemented in a way that the transpiler parses arguments after a double dash as if they were given to the clang compiler. This feature also allows the use of the transpiler in larger projects where not all needed files, e.g. headers, are trivial to find. So if you run `make` with the compiler set to the transpiler created during this thesis followed by a double dash and then run make again with the actual compiler it should be able to transform all the source code. The next thing to do is to find the file which should be transformed. It is either found as the first parameter to the transpiler or pulled from the invocation line mentioned earlier. Afterwards the file is parsed into a syntax tree which is then traversed by two different visitors. The first one is rewriting the source, the second one is extracting and inserting C++ statements. Rewriting is done before extraction because the nodes in the syntax tree are visited from top to bottom so code which is within a changed block could otherwise be extracted before a proper rewriting took place.

5.3 Task extraction

In the clang API there are methods which can be implemented for different kinds of tokens. The main work is done in the `VisitOMPTaskDirective` method of the `ExtractorVisitor`, there all the clauses and variable captures are extracted and a first try to determine the size of those variables is made. In figure 5.1 the textual representation of a task with all clauses can be seen. The `CapturedStmt` represents the code within the task and which should be offloaded to another node. Clang also allows to go back to the source from this syntax tree representation and shows the variables captured within the code block.

5.3.1 Clause extraction

The values for some clauses are evaluated when a OpenMP task is encountered during the run of the program[4]. This means they can not have a value assigned to them during the transpiler phase, so the statement string within the clause is extracted and written into the transformed program in order to be evaluated when the task struct is created during the run of the program. Another clause exists in order to track dependencies between tasks, in the `depend` clause, which was discussed in section 4.2.7, input and output dependencies are added as a list of variable names. These names are used by the runtime to track dependencies between tasks.

In the following lines there is a small OpenMP task definition, with which the generation of task structures is shown. Transformation of code and variables within the task is addressed later.

```
#pragma omp task\
    untied\
    mergeable\
    if(i == 3)\
    final(i == 5)\
    depend(in: a)
```

The following code section is the transformation of the original source above.

```
Task t(7574561021973165946ull);
t.if_clause = (i == 3);
t.final = (i == 5);
t.untied = true;
t.mergeable = true;
t.in.push_back("a");
t.schedule();
```

All clauses are reordered according to the order in which they are introduced in the OpenMP specification and then added to the task struct.

5.3.2 Variable extraction

In order to run a task on a different node without a global address space, all memory used by the task has to be transferred to the other node prior to execution of the task. But before the memory can be transferred it has to be located and the size has to be calculated. As already said some part of the calculation is done during the run of the transformed program, this is addressed in section 5.5. During the transpiler phase the only task is to discover all used variables, which clang takes

[4]The `if`, `final` and `priority` clauses.

care of and determining how many dereference operators are needed in order to get the actual data. This is done by undoing all `typdef` type declarations to get the native type and how often a it has to be dereference in order to get the actual data.

```
int* a = new int[AS];
int b[AS];
int i = 0;
int* p = &i;
```

The variables in the example code above are transformed into the following code at task creation.

```
Var a_var = {"a", &(*a), at_firstprivate, 40000};
t.vars.push_back(a_var);

Var i_var = {"i", &(i), at_firstprivate, 4};
t.vars.push_back(i_var);

Var p_var = {"p", &(*p), at_firstprivate, 0};
t.vars.push_back(p_var);
```

The first thing to note here is that b does not get transformed at all. This is expected behaviour, because `b` is not used within the OpenMP task. On top of that one can see the different amount of dereference operators according to the type of the variable. Variables are then wrapped in the correct amount of references again before the extracted code starts executing. The code which unpacks the variables above can be found below.

```
void * a_pointer_1 = arguments[0];
void * a_pointer_0 = &(a_pointer_1);
int * a = *((int **) a_pointer_0);
void * i_pointer_0 = arguments[1];
int i = *((int*) i_pointer_0);
void * p_pointer_1 = arguments[2];
void * p_pointer_0 = &(p_pointer_1);
int * p = *((int **) p_pointer_0);
```

5.3.3 Source code extraction

There is one thing missing from the parsing side, the source code extracted from a task. The code is extracted by the preprocessor from the code during the transpiler phase. The location of this source code also forms the so called code id, the identifier of the code associated with a task. It is hashed and then used to look up a generated function which unpacks the variables and contains the extracted source code of the application. Another important point is that the source code of the task itself might have been changed previously by the rewrite step. It may also contain other OpenMP task definitions, but nested tasks are currently not properly transformed due to the way the source code is traversed by the clang library[5]. This extracted code is then stored in a temporary file in the `/tmp/tasking_functions` directory in one file per processed source file. In

[5]It uses a top-down approach, so the outer task is encountered and transformed before the preprocessor knows about the inner one.

order to include this in the final application there is one header to collect all of it, which is named `/tmp/tasking_functions/all.hpp`. This is in turn included by the tasking header. On the one hand this architecture is simplistic and easy to use, on the other hand it does not scale really well because every source file with a OpenMP task pulls in the transformed code of every task within the whole application. This has to be optimized away by the linker currently, but it might be an easy target for future work on the architecture.

5.4 Add the runtime to the original code base

In order to run the tasks on other nodes the runtime has to be set up on all those nodes and all but one of the nodes have to go into a execution or management role which is not implemented in the original application. This is done by inserting a function call for setup before the first instruction in the `main` method of the original program and one for teardown after the last statement of the `main` method or, if the last statement is the return statement, before the return statement. Those functions should set up the runtime and should only return for one node, the others should change into runtime or worker nodes. Worker nodes are the ones executing tasks, the runtime nodes keep track of the distributed tasks, handle completion and schedule the tasks for execution.

There are two other functions with which the runtime and the transformed program interact, the first is the `schedule` method of the task struct and the `taskwait` function. All those functions are currently not implemented, implementing them would be the task of the runtime programmer because they are only called from the transformed program and integrate much tighter into the runtime than into the transpiler.

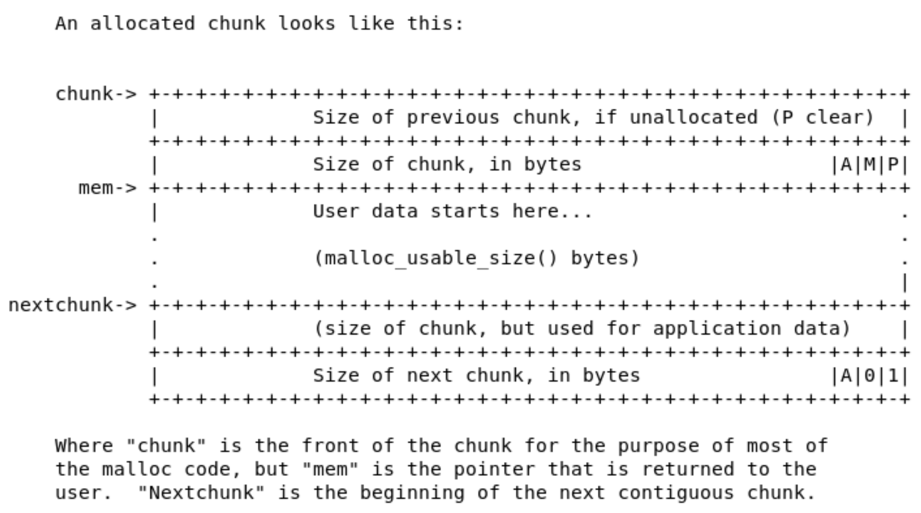
5.5 Variables at runtime

In section 5.3.2 the way variables are extracted from OpenMP tasks is discussed. The largest impact of those variables is during the run of the program though, the address or location of the variable as well as the size can not be determined before execution because some data structures can be allocated according to input to the program and their size can thus not be calculated during the transpiler phase. Furthermore the length of arrays defined in other source files is not known in C++.

5.5.1 Determine the location

During the transpiler phase the variable name is known as is the amount of pointer dereferences needed to find the actual memory. There is code being emitted to calculate the location of the memory during run time and the access type [6] is stored with it. In order to save all this data there is a struct in the tasking header which stores the name of the variable, the pointer to the beginning of the associated memory on the local machine, the access type and the size of the memory location on the local machine. All this data is used by the runtime to transfer memory and for dependency

[6]See section 4.2.6 for further details about access types and how they are used.



```
    An allocated chunk looks like this:


    chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of previous chunk, if unallocated (P clear)  |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of chunk, in bytes                     |A|M|P|
      mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             User data starts here...                          .
            .                                                               .
            .             (malloc_usable_size() bytes)                      .
            .                                                               |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             (size of chunk, but used for application data)    |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of next chunk, in bytes                |A|0|1|
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

    Where "chunk" is the front of the chunk for the purpose of most of
    the malloc code, but "mem" is the pointer that is returned to the
    user.  "Nextchunk" is the beginning of the next contiguous chunk.
```

Figure 5.2: The memory layout of the heap with the default allocator of glibc

[GNU18, malloc/malloc.c l.1088f]

tracking of tasks. The size is either known at the transpiler phase, e.g. for objects, structs and builtin types or calculated at runtime, which will be discussed in the next chapter. To mark variables for runtime size calculation the size should be given as `0`. There is a `prepare` method for the task which iterates over all variables and calculates their size if they are marked that way.

5.5.2 Determine the size

Some sizes can be determined at the transpiler phase as detailed before. But some others can not, especially dynamically allocated arrays, which are an important part of matrices and other data structures often used in high performance computing (HPC). Currently pointers are treated the same way as arrays because it is not certainly known to the transpiler whether or not they refer to a single instance or to an array. Furthermore memory can be allocated on the heap and on the stack of the currently running program.

Heap

Memory allocated on the heap of a program is usually allocated by using some variant of `malloc`. In order to determine the size of the allocation one has to rely on the information provided by the allocator and thus we assume the default `glibc` allocator is used [7]. There is a function within `glibc`[8] to retrieve the usable size of a given pointer, but naturally it only works with pointers allocated on the heap and only with those returned by `malloc`. Other pointers might lead to a return value of `0` or a segmentation fault. So the algorithm first determines whether or not the variable is on the heap

[7]The original `malloc` implementation details can be found at [Lea00].

[8]See [GNU18] for further information about `glibc`.

and if it is, it tries to call `malloc_usable_size`, the function mentioned above, in order to get the actual size. If the pointer points into the middle of an array the return value is `0` and the pointer is decreased by `1` and the call is repeated until either the size is returned or an error occurs.

Stack

If a variable is within the stack the stack size limit of the platform is used as the allocated size. Variables on the stack have no size what so ever and thus the whole stack frame is transferred to the other node. This happens even if some variables from the same stack frame are used and thus the same frame might be transferred several times. The surplus memory should not be an issue because the code should only access valid memory, it has no reference to the surplus memory and if it runs into it, it would also run into problems in a single node scenario.

The actual code which is used in the project can be found in the tasking header in section 9.1.1, the function is called `get_allocated_size`.

6 Examples

In order to put this all together here is an example of a full transpiler run. This is only a simple one file program in order to avoid all the shortcomings of the current version.

```
#define AS 10000

void test(int a[AS], int* p) {
    for(int i = 0; i < AS; i++) {
        #pragma omp task untied mergeable if(i == 3) final(i == 5) depend(in: a)
        {
            a[i] = i + *p;
        }
    }
}

int main(int argc, char** argv) {

    int* a = new int[AS];
    int b[AS];
    int c = 0;
    int* p = &c;

    test(a, p);
    return a[0];
}
```

Listing 6.1: The example input

It becomes the following output after processing, note the incorrect indentation. In order to indent the inserted code correctly one would have to determine the current indentation and whether it is one, two, four or eight spaces per indentation level. This effort was not deemed worth the necessary time and thus the indentation is a bit odd.

```
#define AS 10000

#include "processor/tasking.h"
void test(int a[AS], int* p) {
    for(int i = 0; i < AS; i++) {
        #pragma //omp task untied mergeable if(i == 3) final(i == 5) depend(in: a)
Task t(7574561021973165946ull);
t.if_clause = (i == 3);
t.final = (i == 5);
t.untied = true;
t.mergeable = true;
{
        Var a_var = {"a", &(*a), at_firstprivate, 40000};
        t.vars.push_back(a_var);
}
{
        Var i_var = {"i", &(i), at_firstprivate, 4};
        t.vars.push_back(i_var);
}
{
        Var p_var = {"p", &(*p), at_firstprivate, 0};
        t.vars.push_back(p_var);
}
t.in.push_back("a");

        t.schedule();
    }
}

int main(int argc, char** argv) {

    setup_tasking();
int* a = new int[AS];
    int b[AS];
    int c = 0;
    int* p = &c;

    test(a, p);
    teardown_tasking();
return a[0];
}
```

Listing 6.2: The example output

As you can see the a task instance is created, the associated code id is handed to the task and all the clauses are transformed as specified in section 4.2. Furthermore the tasking header is added to the include list of the source file and the setup and teardown functions are added before the first instruction in main and before the return statement, if there is one.

```
#ifndef __test_cpp__
#define __test_cpp__
#ifndef __TASKING_FUNCTION_MAP_GUARD__
#define __TASKING_FUNCTION_MAP_GUARD__
#include <map>
std::map<unsigned long, void (*)(void **)> tasking_function_map;
#endif

void x_7574561021973165946 (void* arguments[]) {
    void * a_pointer_1 = arguments[0];
    void * a_pointer_0 = &(a_pointer_1);
    int * a = *((int **) a_pointer_0);
    void * i_pointer_0 = arguments[1];
    int i = *((int*) i_pointer_0);
    void * p_pointer_1 = arguments[2];
    void * p_pointer_0 = &(p_pointer_1);
    int * p = *((int **) p_pointer_0);
    {
            a[i] = i + *p;
        }
}
int setup_7574561021973165946() {
    tasking_function_map[7574561021973165946ull] = &x_7574561021973165946;
    return 1;
}

int tasking_setup_7574561021973165946 = setup_7574561021973165946();
#endif
```

Listing 6.3: The example output header

In this file the `tasking_function_map` is defined, the associated code block of the task is used inside of a function boilerplate which extracts all the packed variables and then executes the original code. On top pf that the `tasking_function_map` is also populated using static evaluation and thus making the map available to use.

The last file is the header to put it all together, but in this example there is only one file to include.

```
#include "test.cpp.hpp"
```

Listing 6.4: The example output all header

7 Limitations

There are several limitations of this prototype which are mentioned scattered throughout the whole thesis. They shall be collected here for a central reference.

7.1 The runtime is not yet completed

As of this writing the runtime is not completed and thus no performance figures can be shown. This is due to the partner thesis [Erw18] having incomplete code and will most likely be addressed in another thesis. For a code base with high usage of tasks a speedup should be observable.

7.2 Nested pointers

The memory transfer works by identifying the memory which a pointer or a variable name references. If this memory contains other pointers the memory they reference is not transferred. This is caused by the way memory transfer is implemented, a explicit transfer can not include nested pointers, because it is really hard to tell values and pointers apart in C++.

7.3 Nested OpenMP tasks

Currently one OpenMP task per code block is allowed, because only the first one is extracted correctly. So if there is an OpenMP task in the code block of another one, it is encountered after the extraction of the source code happened for the first task. This leads to the situation where the original code is still in use inside the first task and the second task is not scheduled for execution by the runtime at all. This can be fixed by altering the traverse order in clang or changing the logic by which tasks are handled completely. Thus there should not be any directly nested tasks, nesting them via functions is although supported.

7.4 Stack limit might not be correct

The stack limit can be changed during runtime and it is thus not absolutely reliable for a variable size in the transfer. But apparently there is no other way to get a fallback value for the stack size at all. Another problem is that the stack size can be set to unlimited in which case every time all of the node's memory is transferred. One last problem with this approach is, that the page after the stack is usually protected and thus can not be accessed without a segmentation fault, which needs to be handled somehow.

8 Outlook

8.1 Scheduler

Introducing a scheduler will benefit the runtime because tasks which need the same memory can be scheduled together, dependency tracking can be improved and several task which do not use all threads on a node can be efficiently executed concurrently. This is especially relevant for tasks which do not use internal OpenMP because then almost every core of a node is idle. A proper scheduler could also reduce the number of nodes needed for a program if it utilises the remaining nodes better. Furthermore a scheduler could provide metrics and insights into a running program or statistics for optimisation.

8.2 Internal OpenMP

Using OpenMP within a task is possible but there is no support for it at the moment. There are several points here, the first one is reserving cores which are used by OpenMP directives in order not to have more than one task which tries to execute on those cores concurrently. One more is to report the used cores to the scheduler in case the code for the task is executed several times. This would further increase the level of parallelism and improve the utilisation of the nodes used to run the program. It would also allow for a more OpenMP like use and feel for the programmer because he is no longer restricted to tasks only. In the end this idea combined with the scheduler can lead to large performance gains if the hardware utilisation increases.

8.3 Memory alignment

Currently the assumption is that memory is always allocated in 32 byte chunks and because of this the memory is also aligned that way after it is transferred. If there are pointers which point to the middle of an alignment the alignment is currently destroyed because the transferred memory starts at the pointer and is aligned. This is important for vector instructions and inline assembler because they require memory to be loaded in chunks at the right borders otherwise the program crashes or delivers incorrect results. There was no solution on how to get the current alignment of the data, but there might be a one in the future.

8.4 Transfer structures with pointers

Currently only one level of pointers can be transferred because there is no difference between data and pointers once the capture is calculated. This also leads to problems with multidimensional arrays or structs and classes which contain pointers. In order to solve this issue there has to be something similar to a virtual global address space, otherwise those stored pointers would have to be rewritten or become dangling. A virtual global address space introduces many more issues like where to store the stack of the currently executing task and how to keep track of changed memory. A solution similar to the one in Cluster OpenMP might work but is out of scope for this thesis.

8.5 Handle open files and other system resources

Because a task can be offloaded to another node with another kernel, kernel resources like open files can not be transferred. Any attempt to access a file or window handle on a different node will lead to a segmentation fault or undefined behaviour. Currently these operations are not supported and prohibited for a correct program. Future implementations might support this in order to allow a more natural look and feel for a programmer. This is a very complex topic because some resources might not be available on other nodes or have a different name. Because of this it is the last item on this list and might not be implemented at all.

Bibliography

[18] *Top 500 - The List - June 2018.* 2018. URL: https://www.top500.org/lists/2018/06/ (cit. on p. 17).

[BEKS14] J. Bezanson, A. Edelman, S. Karpinski, V. B. Shah. *Julia: A Fresh Approach to Numerical Computing*. 2014. URL: https://julialang.org/publications/julia-fresh-approach-BEKS.pdf (cit. on p. 14).

[Bur62] Burroughs Corporation. *THE BURROUGHS 0825 MODULAR DATA PROCESSING SYSTEM - PROGRAMMING MANUAL*. 1962. URL: http://bitsavers.informatik.uni-stuttgart.de/pdf/burroughs/military/D8xx/Burroughs_D825_Programming_Manual_Jan62.pdf (cit. on p. 17).

[Erw18] J. Erwerle. "Execution of Task Parallel OpenMP Programs in Distributed Memory Environments - Runtime." Universität Stuttgart, 2018 (cit. on pp. 3, 43).

[GNU18] GNU C Library Project. *The GNU C Library*. 2018. URL: https://www.gnu.org/software/libc/ (cit. on p. 36).

[HB06] L. Hochstein, V. R. Basili. "An Empirical Study to Compare Two Parallel Programming Models." In: *Proceedings of the Eighteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '06. Cambridge, Massachusetts, USA: ACM, 2006, pp. 114–114. ISBN: 1-59593-452-9. DOI: 10.1145/1148109.1148127. URL: ftp://ftp.isi.edu/isi-pubs/tr-676.pdf (cit. on p. 18).

[Hed15] J. B. Hedo. "Run-time support for multi-level disjoint memory address spaces." PhD thesis. Universitat Politècnica de Catalunya, 2015 (cit. on pp. 18, 21).

[Hoe06] J. P. Hoeflinger. "Extending OpenMP to Clusters." In: (2006) (cit. on pp. 13, 18, 19).

[Int07] Intel Corporation. *Cluster OpenMP* User's Guide*. 2007. URL: https://software.intel.com/sites/default/files/m/b/5/9/UsersGuide.pdf (cit. on p. 19).

[KPH61] T. Kilburn, R. B. Payne, D. J. Howarth. "The Atlas Supervisor." In: *Proceedings of the December 12-14, 1961, Eastern Joint Computer Conference: Computers - Key to Total Systems Control*. AFIPS '61 (Eastern). Washington, D.C.: ACM, 1961, pp. 279–294. DOI: 10.1145/1460764.1460786. URL: http://doi.acm.org/10.1145/1460764.1460786 (cit. on p. 17).

[LA04] C. Lattner, V. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation." In: *CGO*. San Jose, CA, USA, Mar. 2004, pp. 75–88 (cit. on p. 24).

[Lea00] D. Lea. *A Memory Allocator*. 2000. URL: http://gee.cs.oswego.edu/dl/html/malloc.html (cit. on p. 36).

[Mic18] Microsoft Corporation. *About Processes and Threads*. 2018. URL: https://docs.microsoft.com/en-us/windows/desktop/ProcThread/about-processes-and-threads (cit. on p. 17).

[Ope15] OpenMP Architecture Review Board. *OpenMP Application Programming Interface*. 2015 (cit. on pp. 13, 25, 27).

[SLT+15] E. Slaughter, W. Lee, S. Treichler, M. Bauer, A. Aiken. *Regent: A High-Productivity Programming Language for HPC with Logical Regions*. 2015. URL: http://regent-lang.org/images/regent2015.pdf (cit. on p. 14).

[TFG+11] E. Tejedor, M. Farreras, D. Grove, R. M. Badia, G. Almasi, J. Labarta. "ClusterSs: A Task-Based Programming Model for Clusters." In: (2011). URL: https://www.ac.upc.edu/app/research-reports/html/2011/14/pap200s1.pdf (cit. on pp. 18, 20).

[Xca17] XcalableMP Specification Working Group. *XcalableMP Language Specification*. 2017. URL: http://www.xcalablemp.org/download/spec/xmp-spec-1.3.pdf (cit. on pp. 18, 22).

All links were last followed on November 8, 2018.

9 Appendix

9.1 Source Code

9.1.1 Tasking Header

```
//
// Created by markus on 1/11/18.
//

#ifndef PROCESSOR_TASKING_H
#define PROCESSOR_TASKING_H

#include <vector>
#include <string>
#include <fstream>
#include <sstream>
#include <malloc.h>
#include <sys/resource.h>

#include "/tmp/tasking_functions/all.hpp"


void setup_tasking() {};
void teardown_tasking() {};

void taskwait() {};

// get the number of bytes which should be transmitted to the other node for the given pointer
size_t get_allocated_size(void* pointer) {
    auto t = (size_t) pointer;

    std::ifstream maps("/proc/self/maps");
    bool isHeap = false;
    std::string heap("heap"), stack("stack");

    size_t start_of_heap = 0;

    for (std::string line; std::getline(maps, line); ) {
        size_t start, end;

        // this char reads the '-' between the 2 values in the file, otherwise the return value
            is incorrect
        char discard;

        if (line.find(heap) != std::string::npos) {

            std::istringstream iss(line);
            iss >> std::hex >> start >> discard >> end;


            if (start < t && t < end) {
                isHeap = true;
                start_of_heap = start;
                break;
            }
        }
        if (line.find(stack) != std::string::npos) {
            std::istringstream iss(line);

            iss >> std::hex >> start >> discard >> end;

            if (start < t && t < end) {
                // return the maximum stack size, this should be low enough to not be that much
                     of a burden
                // TODO look wether the relevant memory is already transmitted
                rlimit lim;
                getrlimit(RLIMIT_STACK, &lim);
                return lim.rlim_cur;
            }
        }
    }

    if (!isHeap) {
        throw std::domain_error("unable to determine the size of a pointee");
    }

    size_t loops = 0;
    size_t size = 0;
    do {
        size = malloc_usable_size((void*)t);
        t--;
        loops++;
        if ((size_t) pointer <= start_of_heap) {
            throw std::domain_error("unable to determine the size of a pointee");
        }
    } while (size == 0);

    return size - loops;
}

enum access_type {at_shared, at_firstprivate, at_private, at_default};

struct Var {
    std::string name;
    void * pointer;
    access_type access;
    size_t size;
};

class Task {
public:
    Task(unsigned long long code_id)
            : code_id(code_id), if_clause(true), final(false), untied(false),
                shared_by_default(true), mergeable(true),
              priority(0)
    {}

    void prepare() {
        for (auto& var : vars) {
            if (var.size == 0) {
                var.size = get_allocated_size(var.pointer);
            }
        }
    }

    void schedule() {
        this->prepare();
        void * arguments[this->vars.size()];
        for (int i = 0; i < vars.size(); i++) {
            arguments[i] = vars[i].pointer;
        }
        tasking_function_map[code_id](arguments);
    }

    unsigned long long code_id;

    bool if_clause; // if false, parent may not continue until this is finished
    bool final; // if true: sequential, also for children
    bool untied; // ignore, continue on same node, schedule on any
    bool shared_by_default; // ignore: syntax: default(shared | none) - are values shared by default or do they have to have a clause for it
    bool mergeable;// ignore
    std::vector<Var> vars; //all variables, in order
    //std::vector<void *> vars_private; //ignore: all variables by default, no write back necessary
    //std::vector<void *> firstprivate; //as above, but initialized with value before, no write back necessary
    //std::vector<void *> shared; //ignore, copy in/out, dev responsible for sync
    std::vector<std::string> in; //list of strings, runtime ordering for siblings, array sections?
    std::vector<std::string> out;
    std::vector<std::string> inout;
    int priority; //later

};

#endif //PROCESSOR_TASKING_H
```

Listing 9.1: The tasking header

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature