Compiler Assisted Source Transformation of OpenMP Kernels

1st Jannek Squar
Scientific Computing
Universität Hamburg
Hamburg, Germany
squar@informatik.uni-hamburg.de

2nd Tim Jammer
FG Scientific Computing
Technische Universität Darmstadt
Darmstadt, Germany
tim.jammer@sc.tu-darmstadt.de

3rd Michael Blesel
Scientific Computing
Universität Hamburg
Hamburg, Germany
3blesel@informatik.uni-hamburg.de

4th Michael Kuhn
Scientific Computing
Universität Hamburg
Hamburg, Germany
michael.kuhn@informatik.uni-hamburg.de

5th Thomas Ludwig
Deutsches Klimarechenzentrum
Hamburg, Germany
ludwig@dkrz.de

***Abstract*—Many scientific applications use OpenMP as a relatively easy and fast approach to utilise symmetric multiprocessor systems at their full capacity. However, scalability on shared memory systems is limited and thus distributed parallel computing is inevitable if the full potential through horizontal scaling shall be achieved. Additional software layers like MPI must be used, which require further knowledge on the scientific developers' side. This paper presents CATO, a tool prototype using LLVM and Clang, to transform existing OpenMP code to MPI; this enables distributed code execution while keeping OpenMP's relatively low barrier of entry. The main focus lies on increasing the maximum problem size, which a scientific application can work on; converting an intra-node problem into an inter-node problem makes it possible to overcome the limitation of memory of a single node.**

Our tool does not focus on improving the absolute runtime, even though it might improve it by e.g. introducing concurrency during the I/O phase; but we rather focus on increasing the maximal problem size and our benchmark of a *stencil code* shows promising results: The transformation preserves the speedup trend of the code to some extent. Another example demonstrates the capability to increase the maximum problem size while using additional compute nodes.

***Keywords*-OpenMP, MPI, LLVM, Source Transformation, Code Distribution**

I. Introduction

Over the years, larger many-core architectures became an essential part of the modus operandi of scientific computing: To increase the performance of a compute node, the trend goes towards increasing the amount of CPU cores instead of core frequency due to power and thermal constraints [1]. It becomes inevitable to use parallelisation techniques on shared memory to fully utilise a node's performance. One prominent solution is to use *OpenMP*, which allows executing concurrent threads on shared memory within a single node. Due to OpenMP's compiler-based pragma approach, it usually only requires relatively little changes to the source code and allows incremental parallelisation. This lowers the barrier of entry for scientific developers.

But for solving large-scale problems, distributed computing is necessary: While single compute nodes typically feature main memory configurations in the range of several gigabytes, large-scale simulations can require multiple terabytes of main memory. But the use of *MPI* for leveraging distributed memory would require costly code changes or even significant code restructuring.

A. Motivation

The advantage of OpenMP is that it is relatively easy to use: A developer only needs to add simple compiler pragmas into the code. The compiler and runtime take care of the thread creation, the distribution of data and the computation. This is the reason why the majority of scientific software, which makes use of high performance computing (HPC) systems, also makes use of OpenMP. However, the parallelisation through OpenMP is limited to a single node. This directly influences the kind of problem that can be solved by a scientific application because it has to fit in a single node's main memory. For example, an input grid would be limited regarding its area, resolution or number of dimensions, which directly impacts the explanatory power of the scientific application.

For a trivial problem without the need of intercommunication, the memory limitation of a single node can be bypassed by dividing the problem into smaller sub-problems and solve them with a new, independent application process on its own node. But as soon as the sub-problems need to exchange data, a new parallelisation scheme for distributed memory needs to be applied.

If the development team of the scientific application consists mostly of experts of the problem's domain and there is no time or budget to do the MPI parallelisation themselves, our tool – CATO – could offer them a satisfying solution to compute bigger problem sizes. Using CATO makes it

 DOI: 10.1109/ISPDC51135.2020.00016

possible to utilise distributed HPC hardware without having knowledge about MPI, as CATO instruments applications with the necessary MPI operation calls automatically. To preestimate, which memory regions might be worthwhile to be distributed, our tool analyses the used OpenMP directives: shared (multidimensional) variables are a good candidate to be distributed.

There are other possibilities to execute an OpenMP application on distributed hardware (cf. section II) but these usually require substantial code changes. With OpenMP 4.0 [2] offloading capabilities have been added but they require advanced knowledge of OpenMP and are intended to be used with coprocessors or accelerators on a single node.

B. Outline

This paper describes our prototype of CATO, which makes use of a fully functional LLVM pass and allows the distribution and access of memory in compliance with a preset communication pattern. It is structured as follows: At first we give an overview in section II about related work and technologies, which CATO makes use of. In section III we outline the individual components of CATO; The functional prototype allows us to demonstrate and partially evaluate the feasibility of our prototype. Section IV discusses the ongoing development of CATO to automatise and improve certain modules. Finally, section V concludes the review of the current status of CATO.

II. Related Work

There are several possibilities to execute OpenMP code on distributed hardware to benefit from an increased maximum memory or even an increased runtime performance. Two popular practices to tackle these problems are distributed shared memory and MPI.

A. Distributed Shared Memory

This category contains operating systems and programming libraries, which abstract the memory distribution by simulating a view on a single virtual shared memory. They can be distinguished by their need to change the original source code.

The user needs to adjust the source code if for example *Partitioned Global Address Space* (PGAS) is used, which creates one virtual shared memory. This memory virtualisation is accessible by every processing element. It is not relevant whether the machine really features one large shared memory, a PGAS language like Unified Parallel C or Coarray Fortran will still create the illusion of one large shared memory [3]. Another example for such an abstraction layer was *TreadMarks* [4] from the 1990s, but the source is not publicly available anymore.

Other approaches similar to ours are programming models like *XcalableMP* [5] or *OpenMPD* [6], which offer new pragmas. Users can use them to annotate their code to define which variables shall be distributed amongst several nodes. In contrast to our solution this requires still some code adjustments and focuses primarily on performance but not optimising the memory footprint.

An alternative to those libraries are low-level system layers like *Single System Image* [7] (SSI) and *Cluster OpenMP* [8]. SSI provides a global centralised view of a distributed system. However, software implementations of SSI tend to be impaired by poor performance and scalability. Even if SSI is implemented on the hardware level, the scalability is suffering due to process migration [3]. Intel Cluster OpenMP is based on TreadMarks and offered a proprietary OpenMP implementation, which handled the distributed execution, but has been abandoned in the meantime.

These approaches are not applicable for our target audience because we assumed in section I-A that a re-implementation is out of their scope.

B. MPI Approaches

Several approaches [9]–[11] translate OpenMP code into equivalent MPI code but do only use two-sided communication, which can have a negative impact on performance. The authors of [12] examine the potential use of MPI-2 one-sided communication to replace OpenMP in a simple algorithm, but the replacement performed worse in comparison to OpenMP and was hard to implement.

The authors of [13] make use of LLVM to analyse an application's byte code and generate system profiles to create the final hybrid code via their implementation of the BSML library [14]. Their approach requires that the application user creates an XML file to describe their algorithm. On the other hand, their tool automatically determines a fitting configuration of MPI processes and OpenMP threads. Our approach does not oblige the user to provide information about their algorithm, in exchange we leave the determination of the best configuration of MPI processes and OpenMP threads to the user – we suppose that the user is able to choose the best configuration because of their expert knowledge of the problem domain.

In addition, all these approaches do not focus on increasing the maximum problem size but on improving the runtime performance. It is possible to replace OpenMP with MPI RMA in general but additional effort is necessary to lessen the overhead as much as possible. However, the extension of MPI RMA through MPI-3 provides the user with additional possibilities to achieve this goal.

III. CATO

Our tool, **CATO** (**c**ompiler **a**ssisted source **t**ransformation of **O**penMP kernels), consists of several modules, of which the most important ones have already been implemented as a prototype. Its main component is an LLVM [15] *Transform Pass*, which analyses and transforms the original OpenMP kernel during the optimisation phase (cf. fig. 1). Focusing on existing OpenMP kernels allows CATO to derive valuable information: the part of the code, which should be executed concurrently, and memory, which should be processed concurrently, indicated by the shared variables.

LLVM is an open source compiler infrastructure and is built in a very modular way, consisting of different parts like the LLVM Core, LLVM OpenMP and language frontends and

hardware architecture backends. Besides its modularity, one of LLVM's biggest strengths is its *intermediate representation* (IR), which is a human readable, assembly like representation of the code that is being compiled. Using LLVM IR eases the analysis and transformation of code through the LLVM optimizer and also makes LLVM very suitable for writing individual tools to analyse, optimise and modify code during compilation. By using the well-established LLVM framework, we are independent of the source code language and do not need to worry about changes of the code style. Evaluating the source code directly (for example, through regular expressions) would be more error-prone and unstable than using LLVM's abstraction layers.

Our automatically inserted MPI parallelisation kernel might be less scalable and provide less performance than hand-written, optimised MPI code. But since CATO's focus lies on additional horizontal scaling and targets users without the possibility to implement the MPI parallelisation themselves, this poses only a minor drawback. We make also use of one-sided communication operations from MPI-3 [16], which offer high performance but are quite cumbersome and erroneous to use if done by hand. The authors of [17] have shown that modern interconnect hardware like InfiniBand, Blue Gene and Ethernet RoCE, which allows an efficient execution of RDMA, benefits from this mechanism through shorter latency times and less usage of CPU and memory resources. The automatic aspect of CATO is therefore an opportunity that users without the capability to use MPI-3 RMA may still benefit from its advantages.

A. General Workflow

We aim for an easy to use workflow: The generation of IR code, the execution of our pass and building the binary in the end is handled by the LLVM infrastructure itself. The only steps, which have to be performed by the user, are the first two steps as well as the last one.

1) Replace compiler call with CATO wrapper script
2) (*optional*) Provide expert knowledge by adding CATO pragmas
3) LLVM frontend translates original code into IR
4) CATO analyses and transforms IR
5) LLVM backend translates IR into machine code
6) Execute binary via `mpiexec` or `srun`

The user needs to adjust their build script and decide on the optimal binding of processes and threads to the hierarchical topology of their hardware. Choosing the correct thread/process-data affinity is important to achieve optimal performance [18]. Because the newly built binary involves MPI function calls, it needs to be executed as an MPI application.

Currently, CATO uses a predefined set of rules how memory in OpenMP kernels is handled (cf. section III-B). We denote these rules as *equivalence class* (EC). An EC preserves the semantics of a communication pattern using OpenMP, only memory accesses are adjusted (and distributed if required) while maintaining equivalent behaviour. This modular approach allows us to implement and evaluate different ECs and to gradually expand the variability of communication patterns. In contrast to the core modifications, which are performed on the IR, the ECs are written in C++, so changes can easily be done and tested. In a later version of CATO the user may annotate their application code with pragmas to enforce a communication pattern or replacement strategy.

B. Status of Implementation

The current version of CATO takes care of the identification of OpenMP kernels and replaces it with an EC, which makes use of MPI. A wide variety of OpenMP pragmas can already be transformed to MPI:

- `parallel` and `parallel for`
- `sections`
- `task` and `taskwait`
- `single` and `master`
- `barrier` and `critical`
- `firstprivate` and `lastprivate` clauses
- `reduction` clause

Currently only static scheduling is handled and even though the `task` pragma is included, its implementation is merely a proof of concept and lacks good performance: The key restriction for the OpenMP codes to be transformed is that the usage of some explicit pointer arithmetic is not allowed due to the memory being distributed over multiple processes, each having its own address space. Besides that, most OpenMP codes can be translated to MPI.

1) Memory Handling: In order to share a variable on distributed memory, CATO will insert (one-sided) MPI communication. In the current version, three different usage patterns of a shared variable are considered:

I *Master-based:* Only the master process will hold the data, every other process has to read the data first. This class is intended for smaller variables such as single integer variables, where a distribution of the data is not desired.

II *Duplicated:* Every process will have its local duplicate of the variable. Besides the larger memory consumption, a store operation from one process has to be applied to all duplicates in the other processes as well to preserve the memory consistency. Therefore, this pattern is intended for shared data that is mostly read (e.g. an auxiliary look-up table).

III *Distributed:* Multidimensional arrays are split into independent chunks and distributed amongst the available processes; this is the default class for multidimensional arrays. For a 2D-array, this means that each process will be assigned a subset of the matrix. When a process accesses the distributed data, it checks at runtime if it is already available locally. If communication is needed, a process will load/store a whole (cache-)line of the data from/to the process owning the accessed line.

2) Replacement: The replacement of OpenMP code is handled by inserting calls of the CATO runtime library into the IR of the program. The first step of the transformation inserts the MPI initialisation at the beginning of the main

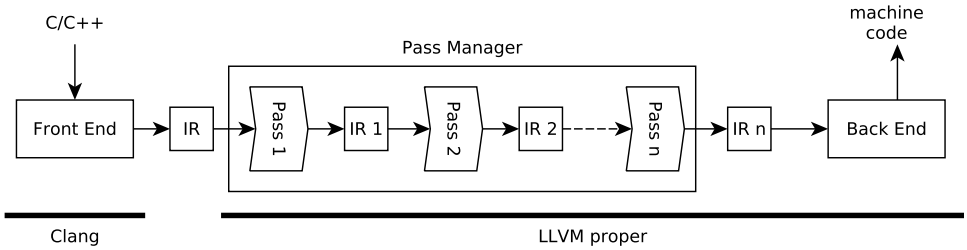


Figure 1: LLVM workflow: Frontend → Optimizer → backend

function and finalisation code in front of all exit points of the program. The next step is the replacement of simple OpenMP functions that can be translated into MPI equivalents. These are functions like `omp_get_num_threads`, `omp_get_thread_num` and `omp_barrier`. Those functions calls are simply replaced in the IR code by their respective MPI equivalents.

After these early steps, all OpenMP parallel regions in the code have to be identified and the memory allocation of shared OpenMP variables have to be replaced. At this point, all `#pragma omp parallel` statements in the original source code have been translated into calls to the OpenMP runtime library by the LLVM frontend. In its IR form, the body of each OpenMP parallel region has been outlined into a new function (a so-called *microtask*), which is passed to OpenMP's runtime library for execution via OpenMP threads. The microtasks take pointers to the shared variables used by them. With this information, CATO identifies all shared variables and all sections of the code that have to be run in parallel. At first, the pass removes the OpenMP runtime library calls and replaces them with direct calls to the microtasks, thereby essentially removing all remaining OpenMP functionality from the program. This creates a working MPI program but still misses the communication regarding the shared variables.

Modifying the memory allocations and adding MPI communication to the shared variables is the biggest task of CATO. Which variables need to communicate via MPI is extracted from the argument lists of the microtasks. The pass then traces the memory allocation points for the shared variables and replaces them with calls to the CATO runtime library, where the memory allocation is suitably replaced with an allocation pattern matching the selected communication pattern for the shared variable. In its current state, CATO is able to replace the standard C/C++ memory allocations functions like `malloc, calloc, new` etc. A later version will also add support for memory handling with some commonly used third-party libraries (cf. section IV-B).

The last step of the pass is the insertion of MPI communication for each memory access to a shared variable. This is implemented by analysing which `load` and `store` instructions in the IR code are accessing shared variables and then replacing them with CATO runtime library calls for the corresponding communication pattern. To optimise halo updates, which are for example common in stencil codes, we cache lines of adjacent memory blocks by using bulk communication instead of transmitting only single values. This minimises the amount of necessary inter-node communication.

C. Demonstration of Pass Transformation

To demonstrate the procedure, we used CATO to transform the code shown in listing 1, which allocates two arrays, initialises them and then performs a copy. Since the original IR code and the transformed IR code are quite extensive (170 respectively 266 lines of code), we discuss only an excerpt from the IR code after the transformation (cf. listing 2). The original IR of listing 1 can be obtained by generating it via `clang -emit-llvm -S`.

```
1   int* a = (int*)malloc(sizeof(int)*4);
2   int* b = (int*)malloc(sizeof(int)*4);
3   // [...] Initialisation
4   #pragma omp parallel for
5   {
6       for(int i = 0; i < 4; i++) {
7           b[i] = a[i];
8       }
9   }
10  printf("[%d,%d,%d,%d]",b[0],b[1],b[2],b[3]);
11  free(a);
12  free(b);
```

Listing 1: OpenMP example (Original C code)

According to section III-B2 there are several steps, which are performed during the transformation.

- **Initialisation:** The call of `cato_initialize` in line 1, sets up the MPI environment (initialisation of MPI as well as retrieving rank and communicator size).
- **Allocate memory:** The memory, which is shared within the OpenMP kernel, needs special handling during its lifetime (cf. section III-D). Therefore CATO adds an additional memory management layer and substitutes the original `malloc` calls in lines 3 and 4.
- **Initialise memory:** Complying with the memory handling by CATO, the pass substitutes all read and write operations on affected memory (line 7). The initialisation of the memory in listing 1 is performed in the sequential section of the application, therefore CATO replaces it by a `shared_memory_sequential_store`.
- **Execute microtask:** In line 9 the microtask is executed directly without the original `kmpc_fork_call`, which would have set off the OpenMP thread fork. Within the microtask (lines 29 to 40) the operations of the OpenMP kernel body are executed. Since each process is only responsible for a section of the shared memory, the boundaries are adjusted

in line 37 with respect to the current rank and the total amount of processes. CATO adds an explicit barrier in line 10, because `#pragma omp parallel for` joins all threads with an implicit barrier.

- **Access memory:** Like before, each access on shared memory is handled by CATO, therefore load operations are replaced accordingly in line 15.
- **Termination:** In lines 19 to 24 all memory managed by CATO is freed again and the MPI environment shuts down.

```
1   call void @_Z15cato_initializeb(i1 false)
2 ; [...]
3   %call = call i8* ↩
      @_Z22allocate_shared_memorylii(i64 16, i32 ↩
      1275069445, i32 1)
4   %call1 = call i8* ↩
      @_Z22allocate_shared_memorylii(i64 16, i32 ↩
      1275069445, i32 1)
5 ; [...]
6   %arrayidx = getelementptr inbounds i32, i32* ↩
      %11, i64 0
7   call void (i8*, i8*, i32, ...) ↩
      @_Z30shared_memory_sequential_storePvS_iz(i8* ↩
      %12, i8* %13, i32 1, i64 0)
8 ; [...] More initialisation
9   call void @.omp_outlined.(i32* null, i32* null, ↩
      i32** %b, i32** %a)
10  call void @_Z11mpi_barrierv()
11  %23 = load i32*, i32** %b, align 8
12  %arrayidx5 = getelementptr inbounds i32, i32* ↩
      %23, i64 0
13  %24 = bitcast i32* %23 to i8*
14  %25 = bitcast i32* %4 to i8*
15  call void (i8*, i8*, i32, ...) ↩
      @_Z29shared_memory_sequential_loadPvS_iz(i8* ↩
      %24, i8* %25, i32 1, i64 0)
16 ; [...] More data loading
17  %call9 = call i32 (i8*, ...) @printf(i8* ↩
      getelementptr inbounds ([15 x i8], [15 x i8]* ↩
      @.str.1, i64 0, i64 0), i32 %27, i32 %32, i32 ↩
      %37, i32 %42)
18 ; [...]
19  call void @_Z18shared_memory_freePv(i8* %44)
20  call void @_Z18shared_memory_freePv(i8* %46)
21  br label %cato_finalize
22
23 cato_finalize:
24  call void @_Z13cato_finalizev()
25  %47 = load i32, i32* %8
26  ret i32 %47
27 }
28
29 define internal void @.omp_outlined.(i32* noalias ↩
      %.global_tid., i32* noalias %.bound_tid., ↩
      i32** dereferenceable(8) %b, i32** ↩
      dereferenceable(8) %a) #2 {
30 entry:
31 ; [...]
32  %0 = load i32**, i32*** %b.addr, align 8
33  %1 = load i32**, i32*** %a.addr, align 8
34  store i32 0, i32* %.omp.lb, align 4
35  store i32 3, i32* %.omp.ub, align 4
36 ; [...]
37  call void ↩
      @_Z26modify_parallel_for_boundsPiS_i(i32* ↩
      %.omp.lb, i32* %.omp.ub, i32 1)
38 ; [...] Acessing and writing data arrays a and b
39  ret void
40 }
```

Listing 2: OpenMP example (IR after pass transformation)

D. Proof of Concept

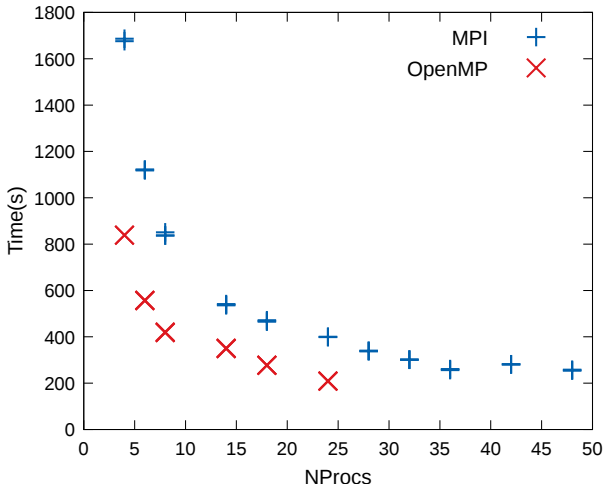
In order to prove the concept, we used CATO on an application that solves a heat dissipation via a 2D PDE using a stencil code. The codes were run on a cluster consisting of up to 10 compute nodes, fig. 2a shows the resulting performance. The used nodes consist of two Intel Xeon X5650 CPUs (2.67GHz), six cores each with hyperthreading. They are equipped with 12 GiB of RAM and operated by Ubuntu 18.04.4 with Linux 4.15. Gigabit Ethernet (1GbE) is used as interconnection, MPICH 3.3.1 provides the MPI implementation.

Scaling: Key property to note is that significant overhead is added when the transformed variant uses as many processes as the original application uses threads. This definitely needs to be addressed in future versions. Nevertheless, the rate of growth of runtimes is strongly correlated and differs only by a constant factor. This is an important outcome regarding the actual use case: It is possible to increase the problem size beyond the single node's memory without a non-linear increase of runtime, which would render this approach impractical.

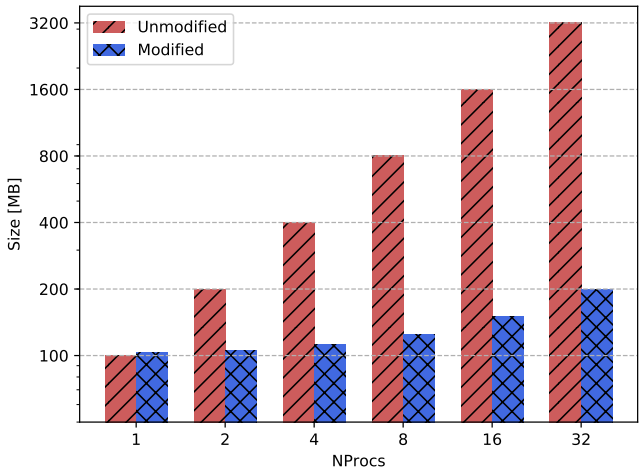
However, the scaling behaviour of the translated MPI version is not always as good as in this example. Another benchmark of a *Fast Fourier Transform* – which combines the data with a butterfly scheme, resulting in all-to-all communication pattern – showed that the overhead imposed by MPI is quite huge and non-linear. Our current implementation makes use of MPI-3 active target communication, which is well-suited for a regular, deterministic communication pattern like the stencil code of our PDE solver. But at the current development stage we use a quite conservative locking so far, which impairs the performance of for example FFT, which makes extensive use of communication operations.

Therefore, the performance achieved by using more cores (that are distributed among several computation nodes) is highly influenced by the need for communication of the translated application. This effect was especially present when using CATO to translate a specific *Branch and Bound* algorithm to MPI. In this particular case, the runtime of the MPI version largely depends on the amount of synchronisation and communication needed (in order to update the current solution, which all processes should bound against).

Initial Data Distribution: Unfortunately, the current implementation of CATO has one limiting factor regarding its memory usage: The parallel part of an OpenMP application usually only covers the actual calculations done on data but not their initialisation. CATO currently matches this behaviour with only one process executing the sequential part and therefore data initialisation. At the beginning of a parallel region all the data initialised in the sequential part needs to be distributed to all other processes, which leads to overhead of runtime and memory consumption. This is limiting the possibility of CATO to enable larger problem sizes, because the initial data has to fit within the master process' memory. To solve this problem,



(a) Strong scaling performance of 2D PDE stencil code of original OpenMP and transformed MPI code (lower is better). NProcs is number of threads (OpenMP) respectively processes (MPI)



(b) Development of heap memory consumption with increasing number of processes (lower is better). NProcs is number of processes

Figure 2: Performance metrics of our proof of concept: Comparison between original and modified version regarding runtime and memory consumption

each node needs to perform parallel I/O for initialisation, which can be quite manifold - ranging from simple data generation at runtime to reading files up to complex I/O backends like NetCDF. See section IV-B for a detailed discussion.

```
1 void* createMemory() {
2     return malloc(100000000);
3 }
4
5 int main() {
6     void *pmem = createMemory();
7     return 0;
8 }
```

Listing 3: Trivial malloc kernel

Figure 2b demonstrates the problem using the example of a trivial `malloc` call allocating a constant number of bytes (cf. listing 3).

- Unmodified: Each process allocates memory for the initial problem, so the total memory consumption increases.
- Modified: Each process only allocates memory for its share. Even though the overhead is crucial for a single process, its effect lessens by using more processes.

Our pass tracks `malloc` calls in the application and adjusts the original argument so that each process only has a part of the requested memory. The pass modifies the malloc kernel code so that each process only allocates a fraction of the requested memory, i.e., not 100 `MB` anymore but $\frac{100\,\mathrm{MB}}{\mathrm{NProcs}}$. So instead of having only one process like in the original version, which has to fulfil the whole memory requirements, we can now split the memory requirement amongst several processes. The modified IR code in listing 4 shows the result of the automatic insertion of MPI methods. Since the MPI environment needs a correct setup, we inject functions into the IR code (line 12 and line 14), which are wrapper calls to the initialisation and finalisation functions of MPI. To adjust the allocation call with respect to the total amount of processes, the original argument, which is passed to `malloc` is divided by the total amount of available processes (cf. lines 2 to 4). The resultant communication of the split memory is omitted in this listing.

```
1  define dso_local i8* @createMemory() #0 {
2    %1 = call i64 @_Z12get_mpi_sizev()
3    %2 = sdiv i64 100000000, %1
4    %3 = call noalias i8* @malloc(i64 %2) #2
5    ret i8* %3
6  }
7
8  define dso_local i32 @main() #0 {
9    %1 = alloca i32, align 4
10   %2 = alloca i8*, align 8
11   store i32 0, i32* %1, align 4
12   call void @_Z15cato_initializev()
13   %3 = call i8* @createMemory()
14   call void @_Z13cato_finalizev()
15   store i8* %3, i8** %2, align 8
16   ret i32 0
17 }
```

Listing 4: IR (extract) of malloc kernel after adjustments by LLVM pass

IV. Further Development Steps

The implementation of CATO has yet to be finished, but the current prototype provides enough functionality to prove the feasibility of the concept. We will continue to extend the feature set of OpenMP, which can be handled by CATO, like dynamic scheduling. Two major topics, which will be worked on next, are improvements on how communication patterns are handled as well as how I/O can be adjusted to allow a larger maximum problem size.

A. Improve Handling of Communication Patterns

At the moment the memory handling by the replacement code is quite limited and needs further improvement and more adaptability. By adding a classification component, CATO could provide specifically modulated ECs for a better fitting match to the detected communication pattern of the original OpenMP kernel.

An automatic detection of access pattern is quite hard, therefore in the final version we need to use heuristics. At first we will focus on the amount of read and write operations on memory of interest as well as on the sequence of their code appearance. This gives already a first estimate of the kind of communication pattern and could allow to preselect a subset of ECs, which are worth to consider. We also plan to use existing frameworks to instrument the application code to log read and write accesses on memory of interest. If then the instrumented application is executed with a problem size fitting into the available memory, we can derive the actual communication pattern from the generated memory access traces. Under the prerequisite that the application behaviour does not change tremendously, if executed with a bigger problem size: This approach allows to choose the right dwarf respectively EC based on the prediction of memory accesses, the data affinity and balancing of induced load.

We focus on an established collection of different communication patterns (e.g., dense matrix algebra or Monte Carlo methods), which are the so-called *dwarfs* [19], [20]. A dwarf describes a class of kernels with similar communication and computation patterns and are considered to represent the most important patterns for science and engineering. Because we are focusing on applications from this background, we are confident to cover most problems of our target audience by using these dwarfs. During the ongoing development, each dwarf will be evaluated if a) it is not worth to be considered and should therefore be merged with another dwarf or b) if it is too complex and should be subdivided. If there are already well-established MPI communication patterns available for a specific dwarf (e.g., a MPI-3 RMA implementation of a 2D stencil solver [21] or an existing framework for multidimensional stencils [22]), they will be reused.

Another possibility for improving the performance is a more extensive use of the possibilities of MPI-3. At the moment we focus on active target communication with fence synchronisation. Using the other synchronisation and communication operations provided by MPI-3 should also benefit the performance, because it allows to improve the replication of the original communication pattern.

Another point is that in the current version the available ECs are a pure MPI replacement. At the moment, the available ECs, which are described in section III-B, remove effectively thread concurrency and use solely processes instead. But it depends on the specific application and underlying hardware if the application would benefit more from a unified MPI parallelisation model or an MPI+OpenMP-hybrid parallelisation model [23].

There are three parallel programming models using MPI [18]:

1) Pure MPI (inter-node and intra-node communication)
2) Hybrid of MPI (inter-node communication) and OpenMP (intra-node communication)
   - Masteronly: No MPI calls in OpenMP kernel
   - Overlap: MPI calls from (some) OpenMP threads

Therefore, some ECs will make use of a hybrid programming model by reinserting the original OpenMP kernel again but this time with consideration of distributed memory (e.g. by adjusting the ranges), which should result in a significantly improved performance.

B. I/O Backends

The major goal of CATO is to enable the execution of the application with problem sizes, which would actually be too large to be executed on a single node. We look at two scenarios, in which CATO would allow the execution of bigger input problems:

In the simple case only the main process loads the initial data P_{ini} and distributes it afterwards. It is still possible that the total memory demand M increases over the time of execution because of additional temporary memory allocations. For example in case of a Gauss-Seidel iteration scheme (cf. application in section III-D) it could be implemented by allocating a temporary second matrix with size $P_{tmp} = P_{ini}$, in which the results of a single iteration are stored - according to (1) this would roughly double the needed memory space.

$$M = P_{ini} + P_{tmp} + const = 2 \cdot P_{ini} + const \tag{1}$$

Using p nodes to distribute the second auxiliary matrix, this would allow at most to double the possible size of the input problem (2).

$$\lim_{p\to\infty} M = \lim_{p\to\infty} \left(P_{ini} + \frac{P_{tmp}}{p} + const \right) = P_{ini} + const \tag{2}$$

But if each process could directly load its share of the initial data, this would allow to use bigger problem sizes than the memory capacity of a single node.

$$\lim_{p\to\infty} M = \lim_{p\to\infty} \left(\frac{P_{ini}}{p} + \frac{P_{tmp}}{p} + const \right) = const \tag{3}$$

Even though (3) is just a rough simplification and does not consider that in fact the constant overhead depends on the process count, it nevertheless demonstrates the potential of this approach. But this requires to interfere with the load operation of the initial problem. In the best case this just means to adjust and distribute allocation calls in the application itself (cf. section III-D). But in the worst case the application makes a fall back on an external library like NetCDF to load and store its input problem. Since we do not want to apply CATO on the external library, this makes it necessary to handle the external library calls inside the application. CATO needs to adjust the arguments passed to them so that each process only

reads or stores a subset of the input problem, from which also the performance could benefit from. We are currently working on an implementation of a NetCDF backend for CATO. If an external library does not receive a pointer to the memory buffer but does the memory allocation itself, our approach is not feasible.

V. Conclusion

We have implemented a functional prototype of CATO, which allows to distribute the execution of OpenMP kernels. Applications, which were limited to shared memory can now be executed on distributed memory. We already cover several OpenMP pragmas and runtime functions, especially those which are quite popular. It depends on the communication pattern, how well the original speedup can be reproduced. After the application has been built with CATO, the results of the evaluation of the horizontal scaling behaviour in section III-D are promising: Even though the absolute runtime becomes worse, the trend of speedup stays the same and there is ongoing work to minimise the gap. And since we did not exhaust our options to optimise our currently available ECs yet, there is much more potential for further improvements.

In combination with the results of the reduced memory consumption per node, this leads to the conclusion that users may benefit from CATO through the increase of the maximum problem size if they cope with a longer runtime. CATO allows to reach problem sizes beyond the possibility of OpenMP-only code.

As soon as CATO has reached a fairly robust state and feature set, we will make it available to the public with an open source license.

Acknowledgements

This work was supported in part by BASF SE as part of the i_SSS (**i**ntegrated **S**upport **S**ystem for **S**ustainability) project and by the Hessian Ministry for Higher Education, Research and the Arts through the Hessian Competence Center for High-Performance Computing.

References

[1] H. Esmaeilzadeh, E. R. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," *IEEE Micro*, vol. 32, no. 3, pp. 122–134, 2012. [Online]. Available: https://doi.org/10.1109/MM.2012.17

[2] OpenMP Architecture Review Board , "OpenMP Application Programming Interface 4.0," 2013.

[3] D. A. Padua, Ed., *Encyclopedia of Parallel Computing*. Springer, 2011.

[4] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "Treadmarks: Shared memory computing on networks of workstations," *Computer*, vol. 29, no. 2, pp. 18–28, 1996.

[5] XcalableMP Specification Working Group, "Xcalablemp language specification." [Online]. Available: https://xcalablemp.org/download/spec/xmp-spec-1.4.pdf

[6] J. Lee, M. Sato, and T. Boku, "Design and implementation of openmpd: An openmp-like programming language for distributed memory systems," in *A Practical Programming Model for the Multi-Core Era, 3rd International Workshop on OpenMP, IWOMP 2007, Beijing, China, June 3-7, 2007, Proceedings*, ser. Lecture Notes in Computer Science, B. M. Chapman, W. Zheng, G. R. Gao, M. Sato, E. Ayguadé, and D. Wang, Eds., vol. 4935. Springer, 2007, pp. 143–147. [Online]. Available: https://doi.org/10.1007/978-3-540-69303-1_15

[7] R. Buyya, T. Cortes, and H. Jin, "Single system image," *Int. J. High Perform. Comput. Appl.*, vol. 15, no. 2, pp. 124–135, 2001. [Online]. Available: https://doi.org/10.1177/109434200101500205

[8] J. P. Hoeflinger, "Extending openmp to clusters," *White Paper, Intel Corporation*, 2006.

[9] A. Basumallik and R. Eigenmann, "Towards automatic translation of openmp to MPI," in *Proceedings of the 19th Annual International Conference on Supercomputing, ICS 2005, Cambridge, Massachusetts, USA, June 20-22, 2005*, Arvind and L. Rudolph, Eds. ACM, 2005, pp. 189–198. [Online]. Available: https://doi.org/10.1145/1088149.1088174

[10] A. Saà-Garriga, D. Castells-Rufas, and J. Carrabina, "OMP2MPI: automatic MPI code generation from openmp programs," *CoRR*, vol. abs/1502.02921, 2015. [Online]. Available: http://arxiv.org/abs/1502.02921

[11] D. Millot, A. Muller, C. Parrot, and F. Silber-Chaussumier, "STEP: A distributed openmp for coarse-grain parallelism tool," in *OpenMP in a New Era of Parallelism, 4th International Workshop, IWOMP 2008, West Lafayette, IN, USA, May 12-14, 2008, Proceedings*, ser. Lecture Notes in Computer Science, R. Eigenmann and B. R. de Supinski, Eds., vol. 5004. Springer, 2008, pp. 83–99. [Online]. Available: https://doi.org/10.1007/978-3-540-79561-2_8

[12] D. An Mey and I. Tedjo, "Adaptive Integration - Form OpenMP to MPI," 2006.

[13] K. Hamidouche, J. Falcou, and D. Etiemble, "A framework for an automatic hybrid mpi+openmp code generation," in *2011 Spring Simulation Multi-conference, SpringSim '11, Boston, MA, USA, April 03-07, 2011. Volume 6: Proceedings of the 19th High Performance Computing Symposia (HPC)*, L. T. Watson, G. W. Howell, W. I. Thacker, and S. Seidel, Eds. SCS/ACM, 2011, pp. 48–55. [Online]. Available: http://dl.acm.org/citation.cfm?id=2048584

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

[15] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 2004, pp. 75–88. [Online]. Available: https://doi.org/10.1109/CGO.2004.1281665

[16] R. Gerstenberger, M. Besta, and T. Hoefler, "Enabling highly-scalable remote memory access programming with MPI-3 one sided," *CoRR*, vol. abs/2001.07747, 2020. [Online]. Available: https://arxiv.org/abs/2001.07747

[17] P. Balaji, H. V. Shah, and D. K. Panda, "Sockets vs rdma interface over 10-gigabit networks: An in-depth analysis of the memory traffic bottleneck," in *In RAIT workshop*, vol. 4, 2004, p. 2004.

[18] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid mpi/openmp parallel programming on clusters of multi-core SMP nodes," in *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2009, Weimar, Germany, 18-20 Febuary 2009*, D. E. Baz, F. Spies, and T. Gross, Eds. IEEE Computer Society, 2009, pp. 427–436. [Online]. Available: https://doi.org/10.1109/PDP.2009.43

[19] P. Colella, "Defining software requirements for scientific computing," 2004.

[20] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams *et al.*, "The landscape of parallel computing research: A view from berkeley," 2006.

[21] S. Kumar and M. Blocksome, "Scalable MPI-3.0 RMA on the blue gene/q supercomputer," in *21st European MPI Users' Group Meeting, EuroMPI/ASIA '14, Kyoto, Japan - September 09 - 12, 2014*, J. J. Dongarra, Y. Ishikawa, and A. Hori, Eds. ACM, 2014, p. 7. [Online]. Available: https://doi.org/10.1145/2642769.2642778

[22] H. Dursun, K. Nomura, L. Peng, R. Seymour, W. Wang, R. K. Kalia, A. Nakano, and P. Vashishta, "A multilevel parallelization framework for high-order stencil computations," in *Euro-Par 2009 Parallel Processing, 15th International Euro-Par Conference, Delft, The Netherlands, August 25-28, 2009. Proceedings*, ser. Lecture Notes in Computer Science, H. J. Sips, D. H. J. Epema, and H. Lin, Eds., vol. 5704. Springer, 2009, pp. 642–653. [Online]. Available: https://doi.org/10.1007/978-3-642-03869-3_61

[23] F. Cappello and D. Etiemble, "MPI versus mpi+openmp on IBM SP for the NAS benchmarks," in *Proceedings Supercomputing 2000, November*

4-10, 2000, Dallas, Texas, USA. IEEE Computer Society, CD-ROM, J. Donnelley, Ed. IEEE Computer Society, 2000, p. 12. [Online]. Available: https://doi.org/10.1109/SC.2000.10001