UNIVERSITÄT PASSAU

Fakultät für Informatik und Mathematik

Master's Thesis in Computer Science

Combining the DKU Pattern with Polyhedral Optimization and Tiling

by

Stefan Kronawitter

Supervisor: Dr. Armin Größlinger

Examiner: Prof. Christian Lengauer, Ph.D.

April 30, 2013

Abstract

Performance portability is a huge problem when parallelizing an application. Practically, the source codes for parallel systems are strongly adapted to the available hardware and therefore the performance of the application is tightly coupled to a single machine. This dependency can be weakened by transferring the necessity for strong adaptions from the application code to a framework, which acts as an additional layer between the user application and the hardware. Moving to a new system then only requires the adaption of a single framework and leave all other source code untouched.

For application codes fitting the polyhedron model, this thesis introduces a new way to generate automatically the program code specific for a parallelization framework. A generator was developed, which only requires a sequential version of the application code to create architecture independent methods required by the framework in order to execute the application in parallel. The generated code is also able to exploit the communication required on distributed memory systems in order to apply layout optimizations, which leads to an additional speedup on top of the parallelization itself.

Acknowledgements

This work would not have been possible without the support of many people.

First of all, I would like to thank my advisior Armin Größlinger for his help during all stages of this work; starting from the first decisions for the implementation until the proofreading for this work. No matter if I had problems concerning the development of the DKU generator, or if I needed help to understand the behaviour of the generated code, he had always time to answer my questions.

I am also very grateful to Christian Lengauer, who gave me the time and opportunity to finish this work, although the next project I am participating in already started.

Especially I want to thank Sean Halle for providing me his implementation of the DKU framework, on which this work is based.

In addition, I would like to thank my office mate Andreas Simbürger for his help in every situation and the proofreading of many chapters. Thanks also to all colleagues within LooPo for many very fruitful discussions concerning my master's thesis and the humorous private conversations in between.

Beside all support concerning this work, I want to thank my family and all my friends, who provided me the motivation for not only this work, but also for the whole studies on the one side, and the distance and regeneration on the other side.

Contents

1 Introduction

One of the first decisions when parallelizing an application is to choose a programming language. If the target hardware uses shared memory, OpenMP can be a good choice. But OpenMP is not able to handle communication between different nodes on a cluster. So, for best performance on varying hardware, the algorithm must be ported to different parallel languages and, in practice, it is also closely adapted to a single target machine. In this case, the so-called performance portability problem appears every time the hardware changes.

One solution for this dilemma would be to use an automatic parallelization tool and/or a parallelizing compiler. But such a tool or compiler will be very complex, if it has to generate code suitable to the large number of architectures available.

Another approach is to use a framework as an additional layer between the application code and the parallel hardware, which encapsulates the choice of the used parallel languages and paradigms that fits best to the used machine. Based on this, the programmer has to provide a more abstract and architecture independent code using a special pattern, whereas the framework is the only part which is completely adapted to the actual hardware. When moving to another machine, none of the application codes need to be touched, only the framework must be ported and optimized again.

Given such a framework, a next possible step is to generate the application code for the framework automatically, starting with a simple sequential version. This part – the code generation for the framework based on the polyhedron model – is the main task of this thesis. Such a parallelization framework requires specific methods for, e.g., preparing the input to communicate all required data to the nodes, or to divide the computation in order to execute it in parallel. This information can be computed completely automatically, if the application code can be represented using the polyhedron model.

Therefore, a generator was developed, which only takes a sequential version or a polyhedron description of the application code and automatically generates all required methods for a parallelization framework. Chapter 2 provides a brief introduction into the used libraries, the DKU pattern for the parallelization framework and the polyhedron model, which is used by the generator to perform all transformations and optimizations in. An overview over the usage of both the generator and the DKU framework is given in Chapter 3, while Chapters 4 and 5 describe the internals of both in more detail. Chapter 6 presents and explains some strange effects that appeared during the implementation and testing of the generator. The performance achieved, when using the automatically generated application code with the DKU framework, is also discussed in this chapter. Finally, Chapter 7 summarizes the results and discusses future work.

2 Background

2.1 Polyhedron Model

Many techniques used to optimize and parallelize loop nests modify the source code of the loop structure. Different and more powerful approaches are based on mathematical models and therefore, they do not need the textual structure of the source code. The polyhedron model [FL11] serves as an abstract representation of a loop program. The first usage of this model in a loop parallelization context was presented by Lengauer in 1993 [Len93]. In contrast to previous loop parallelization techniques, it is based on a mathematical model using polyhedra. This allows more powerful transformations than text-based approaches and it has evolved even further since then.

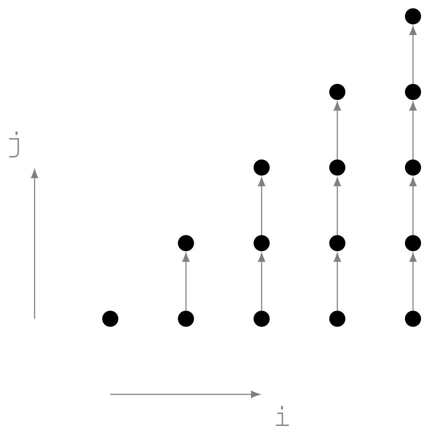
Unlike other models, such as abstract syntax trees, its basic component is a statement instance, i.e., one specific execution of a statement. Each instance is represented by a node in the computation graph, which is associated with a point $\vec{i} \in \mathbb{Z}^d$. The components of this so-called iteration vector correspond to the values of the iterator variables of the d loops surrounding the statement this instance belongs to. The vector $\vec{i}$ is also the position of the statement instance in the computation graph.

In addition to the loop variables, the model also contains structure parameters, which typically correspond to the problem size. For example the numbers of rows and columns for both input arrays of a matrix multiplication are denoted by structure parameters. They are unknown at compile time, as the size typically varies for different invocations, but they are constant at run time.

If each loop bound of a loop nest is an affine inequation in the surrounding loop variables and structure parameters, the set of instances depicts a $\mathbb{Z}$-polyhedron, the so called iteration domain of the loop program. A $\mathbb{Z}$-polyhedron is the intersection of a (rational) polyhedron with $\mathbb{Z}^d$ or a lattice, so it contains only integral points. This enables a compact representation, which is independent from the number of loop iterations or the runtime values for the structure parameters.

```
for (int i = 0; i < n; ++i) {
  for (int j = 0; j <= i; ++j) {
S:  A[i] = .5 * A[i] + B[j];
  }
}
```

(a) Simple input code



(b) Iteration domain and dependences of the input code

Figure 2.1: Polyhedron model of a simple loop nest

Consider the source code of Figure 2.1a. The loop program consists of a single statement `S` which is enclosed by two loops and its iteration vector is therefore $\vec{i} = (\mathtt{i},\mathtt{j})^\top$. According to the loop bounds the iteration domain of this statement is defined by the constraints

$$0 \le \mathtt{i} < \mathtt{n}, \quad 0 \le \mathtt{j} \le \mathtt{i},$$

where n is a constant structure parameter. The computation graph of this iteration domain for $\mathtt{n} = 5$ is depicted in Figure 2.1b.

In addition to the iteration domain, the execution order of the statement instances must also be specified for a complete and well-defined model of the input program. As a statement instance cannot be executed before itself, the ordering must be strict and it is represented as directed edges in the computation graph. In the case of a sequential program, every pair of different instances can be ordered, which leads to a total ordering. If the loop nest should be executed in parallel, the total ordering must be replaced by a partial one, as two parallel instructions can be executed in an arbitrary ordering, or completely simultaneously.

This leads to the new problem of finding an appropriate ordering, which has enough parallelism to utilize as much parallel execution units as possible without changing the result of the loop nest. A common approach to deal with this is to use the dependences between the instances which was first formulated by Bernstein [Ber66].

Definition 1 (Dependence) *Two different instances u and v are said to be in dependence iff both access the same memory cell and at least one of them modifies it.*

Therefore, if u and v are executed in the same sequence as given by the initial total ordering, their result is guaranteed to be identical with the sequential version. Utilizing this, the orderding deduced by the dependencies of the statement instances along with the original iteration domain describe a parallel program that leads to the same result as the sequential version.

Consider again the loop code of Figure 2.1a. The array elements `A[i]` are updateded for each value of `j`, i.e. they depend on the old value and therefore all modifications of one sepcific array cell must not be interchanged. In particular, all instances with the same value for `i` are in dependence. According to the original, sequential version, every instance `(i,j)`$^\top$ depends on and must be executed after the instance `(i,j-1)`$^\top$, if it exists. Although the elements of the array `B` are also accessed multiple times, this need not be considered here as there is no write access to `B`. The dependences are also shown in Figure 2.1b.

As long as both loop bounds and array subscripts are affine relations in the loop variables and additional structure parameters, all dependencies can be efficiently computed using integer linear programming tools [Fea91].

In contrast to work directly with a relation representing the ordering for the program, it is more handy to use a mapping from the iteration domain to a multidimensional virtual time $\mathbb{Z}^t$ for an arbitrary t. The idea of this mapping is to transform the iteration domain in a way such that the resulting virtual time is totally ordered using the lexicographical ordering $<_{lex}$. For all $\vec{a}, \vec{b} \in \mathbb{Z}^t$ the following equivalence holds:

$$\vec{a} <_{lex} \vec{b} \quad \Leftrightarrow \quad \exists m \geq 0 : \; \forall i < m : \; a_i = b_i \wedge a_m < b_m.$$

Such a mapping is called a schedule for a given statement. A valid parallel schedule for the example of Figure 2.1 is `(i,j)`$^\top \to$ `j`, as all values for `i` can be executed simultaneously and the `j`-loop must remain sequential.

2.2 Integer Set Library

The Integer Set Library (isl) [Ver10] is today one of the most frequently used C libraries when working with the polyhedron model. It can efficiently handle integer sets and relations bounded by affine inequalities. Operations for manipulating them are for example set/map union, intersection, or coalescing.

The library is also able to deal with unknown but constant parameters and existentially quantified variables. The latter are e.g. used to model holes in a $\mathbb{Z}$-polyhedron. Consider for example the simple loop:

```
  for (int i = 0; i < n; i += 2)
 T: A[i] = 0;
```

A first attempt to model the iteration domain of statement `T` is with the constraint $0 \leq \mathtt{i} < \mathtt{n}$, but this covers odd values for `i`, too. Actually it is not possible to describe the iteration domain of this example without any extensions to the model. To deal with this, isl uses existentially quantified variables, which do not modify the shape of the polyhedron, but the integral points inside it. A valid iteration domain can then be represented by the constraints: $0 \leq \mathtt{i} < \mathtt{n} \ \wedge \ \exists \mathtt{a} \in \mathbb{Z} : \ 2 \cdot \mathtt{a} = \mathtt{i}$.

Back to the example of Figure 2.1, the iteration domain of statement `S` can be written in isl syntax as follows:

```
[n] -> { S[i, j] : 0 <= i and i < n and 0 <= j and j <= i }
```

In this example, `n` is a structure parameter which denotes the size of both arrays. In isl syntax, structure parameters of a set are placed in front of it. Sets can also be named, which is done by adding a label before the domain vector. In the given example, the tuple is named “`S`”.

Relations are denoted very similiar in isl. The only difference is that there is not only a domain vector, but also a range vector in front of the constraints. Both vectors may be labeled and they are connected by an ASCII arrow. The schedule for the given code of Figure 2.1a can be denoted in isl syntax as follows:

```
[n] -> { S[i, j] -> [i, j] }
```

In addition to some basic methods to inspect and manipulate sets or maps, isl also offers higher level functions used in loop/compiler optimization like performing a dependence analysis or computing an optimal schedule for the given constraints. As all computations are performed with GMP [Gt12], the library can deal with arbitrary sized integers.

isl provides access to extended functionality through the use of other libraries, e.g. barvinok, a library for counting the number of integer points in a polyhedron, or the iscc calculator, which provides an easy to use interface due to its very high abstraction level.

```
 1 [n] -> { S[i, j] : j <= i and i <= -1 + n and j >= 0 }
 2 [n] -> { S[i, j] -> [0, i, 0, j, 0] }
 3 =
 4  [n] -> { S[i, j] -> A[i] }
 5   read: 0
 6   write: 1
 7  +
 8   *
 9    0.5
10    [n] -> { S[i, j] -> A[i] }
11     read: 1
12     write: 0
13   [n] -> { S[i, j] -> B[j] }
14    read: 1
15    write: 0
```

Figure 2.2: Model of statement S from Figure 2.1a, extracted with pet.

2.3 Polyhedral Extraction Tool

The Polyhedral Extraction Tool (pet) [VG12] is designed to extract a polyhedral representation from C source code. In contrast to other tools, pet does not have its own C parser. It uses LLVM's C frontend clang instead and the polyhedral description is extracted directly from the high-level abstract syntax tree. This grants pet full support of C99 and, therefore, it can deal with, e.g., variable-length arrays. The user can also benefit from the very nice warnings and error messages that clang generates.

The extracted polyhedral representation is stored by the isl (cf. Section 2.2). For each statement, pet returns an abstract syntax tree of the statement's code along with the corresponding iteration domain and schedule. Additionally all array accesses are represented by a mapping from the iteration domain to the actual index and the type of the access, which denotes if it as a read or write access (or both). Therefore, transformations of the access relations are also possible, which reduces the effort of, e.g., memory layout optimizations.

Consider again the sample code of Figure 2.1a, the extracted representation of statement `S` is shown in Figure 2.2. The first line shows the iteration domain and is identical to the representation shown in Section 2.2. (The missing constraint `0 < i` can be deduced from `0 < j` and `j <= i` and is therefore not needed.) Line 2

```
S[i, j] -> [0, i, j];
T[i, j] -> [1, i, j];
```

```
for (int i = ...)
  for (int j = ...)
    S(i, j);
for (int i = ...)
  for (int j = ...)
    T(i, j);
```

(a) Constant output dimension in first position.

```
S[i, j] -> [i, 0, j];
T[i, j] -> [i, 1, j];
```

```
for (int i = ...) {
  for (int j = ...)
    S(i, j);
  for (int j = ...)
    T(i, j);
}
```

(b) Constant output dimension between others.

```
S[i, j] -> [i, j, 0];
T[i, j] -> [i, j, 1];
```

```
for (int i = ...) {
  for (int j = ...) {
    S(i, j);
    T(i, j);
  }
}
```

(c) Constant output dimension in last position.

Figure 2.3: Effect of additional constant output dimensions for the schedule.

contains the schedule for statement `S`. As the surrounding loops scan the iteration domain in a lexicographic order, the schedule can be an identity map.

In a general situation of more than one statement, the schedule must not only specify the ordering of the instances for each statement separately but also between all statements. An easy way to achieve this is to add additional output dimensions to the schedule mapping. Consider the examples of Figure 2.3, as the instances are scanned in a lexicographic ordering after applying the schedule map to the iteration domain, different values of an additional constant dimension can be seen as different loop nests for the following inner loops.

The last lines of Figure 2.2 define the structure of the statement using an abstract syntax tree and, as already mentioned, array accesses are represented by an isl map.

```
1 #define S(i,j) A[i] = .5 * A[i] + B[j]
2
3 for (c2=0;c2<=n-1;c2++) {
4   for (c4=0;c4<=c2;c4++) {
5     S(c2,c4);
6   }
7 }
```

Figure 2.4: Loop code generated by CLooG using the model from Figure 2.2.

2.4 Chunky Loop Generator

CLooG is a tool to generate loop code for scanning integer polyhedra [Bas04]. Originally it was designed to resolve the loop generation problem, which was one of the main challenges for compilers when using the polyhedron model. That is, previously known algorithms for generating target code either require severe restrictions on the schedule, which limits the possibility for optimizations, or they are simply too expensive to be included in real-life compilers. Beside preventing these problems, the generated code is also trimmed for efficiency, i.e., control overhead is avoided wherever possible. Therefore, CLooG does not create a loop with only one iteration (if this can be determined statically) and, in general, it can completely unroll loops with only few iterations. The resulting code is also valid C code, assuming methods or preprocessor definitions for `min`, `max`, `floord`, and `ceild` are available.

Lines 3 – 7 in Figure 2.4 show the generated C code when running CLooG with the model from Figure 2.2. CLooG does not know the actual code for the statement, so it adds a call to the statement label. The parameters for this invocation are specified by the iteration domain and any transformation given by the schedule is revoked, i.e. if for example the schedule interchanges both loops (e.g. with `[n] -> { S[i, j] -> [0, j, 0, i, 0] }`) the corresponding call in the generated code would be `S(c4,c2);`.

As already mentioned the statement code must be inserted manually, which can be easily accomplished by using a preprocessor macro as depicted in the first line of Figure 2.4. This works fine, if the macro inserts only a single statement, but in more complex situations, it can lead to invalid C code; encapsulating the statements in a `do { ... } while(0)` block is more fail-safe and should be preferred when automatically generating code.

2.5 DKU Pattern

DKU is the short form for "Divider, Kernel, Undivider", which is the the core part of this pattern [HC09]. As the name suggests, it is a variant of a divide and conquer approach. The programmer has to provide equally named methods for these routines, so the framework is able to encapsulate the code concerning parallelism and all related constructs to deal with performance portability problem. Therefore, the main idea of this pattern is to adapt only the framework to the actual hardware and to keep the application code untouched. That is, when updating the hardware or even when switching to a new architecture only the framework has to be ported and all application code can be reused.

In DKU, computations are represented as *pieces* where each piece contains a user pointer to access all required application specific data, including all information about its computation size. All user methods provided to the framework operate on these.

The divider's task is to split the given piece in a set of disjoint, (ideally) equally sized subpieces, concerning the work load of the computation. The desired number of subpieces is passed along with the piece itself to the user method. The only constraint is that the generated subpieces define computations with no dependencies in between, i.e. with no synchronization or any need for communication.

After splitting the computation using the divider, the kernel is called once for each generated subpiece. Each of them is computed by one thread, so the user code for the kernel should be purely sequential. Therefore, the programmer does not need to know anything about parallel constructs like OpenMP[1] or MPI[2]. The parallelization part is completely hidden within the framework.

Each kernel computes one part of the result and the undivider must ensure the root piece contains the whole result at the end. Therefore, the undivider is called once for each subpiece and its input is the subpiece itself and its corresponding parent. Both pieces are allowed to share the same memory area for the result, so in this case the programmer may provide an empty undivider. (Disregarding the memory management.)

Figure 2.5 shows a complete example of how the DKU Pattern can be used to parallelize a matrix multiplication.

[1] Open Multiprocessing (`http://openmp.org/wp/`)
[2] Message Passing Interface (`https://www.mcs.anl.gov/research/projects/mpi/`)

```c
struct Matrices {
 int acRows, bcCols, aCols_bRows;
 int cBeginRow, cEndRow, cBeginCol, cEndCol; // cEndRow and Col exclusive
 float *A, *B, *C;
};

void Divider(struct DKU_Piece *p, int n,
          struct DKU_Piece **sub_ps, int *nr_sub_ps) {
 // find largest m <= n such that m = 2^b for a natural b
 int b = 0;
 while ((n >>= 1) > 0) ++b;
 *nr_sub_ps = 1 << b;

 struct Matrices *mats = p->user;
 int cRowLen = mats->cEndRow - mats->cBeginRow;
 int cColLen = mats->cEndCol - mats->cBeginCol;

 int x = 1 << (b/2), y = (1 << b) / x;
 for (int i = 0; i < x; ++i)
  for (int j = 0; j < y; ++j) {
   *sub_ps = DKU_Piece_alloc();
   struct Matrices *sub_mats = (*sub_ps)->user = malloc(sizeof(mats));
   sub_ps++;
   memcpy(sub_mats, mats, sizeof(struct Matrices));

   sub_mats->cBeginRow += i * cRowLen / x;
   sub_mats->cEndRow = sub_mats->cBeginRow + cRowLen / x;
   sub_mats->cBeginCol += j * cColLen / y;
   sub_mats->cEndCol = sub_mats->cBeginCol + cColLen / y;
  }
}

void Kernel(struct DKU_Piece *p) {
 struct Matrices *mats = p->user;
 float *A = mats->A, *B = mats->B, *C = mats->C;
 int acRows = mats->acRows, bRows = mats->aCols_bRows;

 for (int i = mats->cBeginRow; i < mats->cEndRow; ++i)
  for (int j = mats->cBeginCol; i < mats->cEndCol; ++j)
   for (int k = 0; k < mats->aCols_bRows; ++k)
    C[i * acRows + j] += A[i * acRows + k] * B[k * bRows + k];
}

void Undivider(struct DKU_Piece *p, struct DKU_Piece *sub) {
 free(sub->user);
 DKU_Piece_free(sub);
}
```

Figure 2.5: Simple example for the methods the user has to provide to compute the matrix multiplication with DKU. (Assuming matrix C can be divided without any remainder.)

```
void schedule(struct DKU_Piece *p) {
 struct DKU_Piece *sub_ps[nr_cpus];
 int nr_sub_ps;

 Divider(p, nr_cpus, &sub_ps, &nr_sub_ps);

#pragma omp parallel for
 for (int i = 0; i < p->nr_sub_ps; ++i) {
  Kernel(sub_ps[i]);
  Undivider(p, sub_ps[i]);
 }
}
```

Figure 2.6: Simple OpenMP scheduler for a DKU framework.

As a `DKU_Piece` can hold exactly one pointer for the whole user data, the first part is the definition of a structure to encapsulate all data, which is needed by the user methods.

The most complicated method is the divider. Its task is to divide the computation optimally. In a standard matrix multiplication, all elements of the result matrix can be computed independently and, therefore, both loops iterating over either the rows or the columns of this matrix can be split. If the divider is asked to generate `n` subpieces, it has to find two natural numbers $x, y \in \mathbb{N}$ such that $x \cdot y$ is close to `n`. For the sake of convenience, the shown divider uses a quite simple heuristic to compute these values, assuming `n` is a power of 2.

The next method presented is the kernel. It extracts relevant data from the piece and then computes the result for the specified part of the computation, which is in this example a submatrix of `C`. It can be easily seen that all pieces can be computed simultaneously by invoking the kernel multiple times in parallel.

The last method needed is the undivider. As all pieces share the same memory location for the result matrix, the undivider need not collect any data. Only memory management is left for this method.

Figure 2.6 shows a small scheduler for a DKU framework. After dividing the input the subpieces are executed by multiple OpenMP threads on a shared memory architecture.

An extension to distributed memory systems is a bit more complex and requires the programmer to provide four additional methods, namely `BundleInput`, `Un-`

bundleInput, BundleResult, and UnbundleResult. Distributed memory architectures require data transfer for both input and result. In case of MPI, the communication must be performed explicitly by the scheduler and therefore both input and result must be bundled before and unbundled after the transfer. As the framework itself does not know anything about the actual user data, this part has to be done by the user.

An extension to the aforementioned matrix multiplication example can be seen in Figure 2.7. The BundleInput method first computes the space for the whole input and copies all data to the newly allocated memory. As the kernel requires only a part of the input matrices, an obvious optimization would be to store submatrices of A and B in the generated bundle to reduce the communication overhead, but for lack of space and a better comprehensibility the complete matrices are copied here. The scheduler is then able to transfer the bundle to another node, which can restore the input using the UnbundleInput method. After invoking the kernel, the result must be bundled again using BundleResult and communicated back to the master, which uses the UnbundleResult method to merge all parts of the result. The result bundle can be optimized again by only transferring the computed submatrix and not the whole C.

```
void BundleInput(struct DKU_Piece *p, void **bundle, int *size) {
 struct Matrices *mats = p->user;
 int sizeA = mats->acRows * mats->aCols_bRows * sizeof(float);
 int sizeB = mats->aCols_bRows * mats->bcCols * sizeof(float);
 *size = 7 * sizeof(int) + sizeA + sizeB;

 *bundle = malloc(*size); int *b_int = *bundle;
 b_int[0] = acRows; b_int[1] = bcCols; b_int[2] = aCols_bRows;
 b_int[3] = cBeginRow; b_int[4] = cEndRow;
 b_int[5] = cBeginCol; b_int[6] = cEndCol;

 char *b_char = (char *) (b_int + 7);
 memcpy(b_char, mats->A, sizeA); b_char += sizeA;
 memcpy(b_char, mats->B, sizeB);
}

void UnbundleInput(void *bundle, int size, struct DKU_Piece *p) {
 struct Matrices *mats = p->user = malloc(sizeof(struct Matrices));
 int *b_int = bundle;
 mats->acRows = b_int[0]; mats->bcCols = b_int[1];
 mats->aCols_bRows = b_int[2];
 mats->cBeginRow = b_int[3]; mats->cEndRow = b_int[4];
 mats->cBeginCol = b_int[5]; mats->cEndCol = b_int[6];

 int lenA = mats->acRows * mats->aCols_bRows;
 int lenB = mats->aCols_bRows * mats->bcCols;
 int lenC = mats->acRows * mats->bcCols;

 mats->A = (float *) (b_int + 7);
 mats->B = mats->A + lenA;
 mats->C = malloc(lenC * sizeof(float));
}

void BundleResult(struct DKU_Piece *p, void **bundle, int *size) {
 struct Matrices *mats = p->user;
 *size = mats->acRows * mats->bcCols * sizeof(float); // only C needed
 *bundle = malloc(*size);
 memcpy(*bundle, mats->C, *size);
 free(mats->C);
}

void UnbundleResult(void *bundle, int size, struct DKU_Piece *p) {
 struct Matrices *mats = p->user;
 float *res = bundle;
 for (int i = mats->cBeginRow; i < mats->cEndRow; ++i)
  for (int j = mats->cBeginCol; i < mats->cEndCol; ++j)
   mats->C[i * mats->acRows + j] = res[i * mats->acRows + j];
}
```

Figure 2.7: Bundle and unbundling methods for the matrix multiplication example of Figure 2.5.

3 Overview

This chapter provides a brief overview of the procedure and the data flow when implementing a new program with the DKU framework using the generator. The simplified process is shown in Figure 3.1.

3.1 Generator

Frontend The generator takes two input files:

- a C99 source file, containing the application kernel code,
- a desired memory layout transformations in isl syntax (optional).

The first obligatory input is a source file, which contains a sequential version of the loop program that should be executed in parallel using the DKU framework. As the input file is parsed by LLVM's C-frontend clang, the input must be a valid C99 file. Simplifications for the input, which are not specified by the standard, such as writing only the loop nest and omitting the method signature, are not permitted. In order to specify which part of the input code should be considered, the loop nest must be surrounded by `#pragma scop` and `#pragma endscop`. Variables have to be defined outside this block, apart from the loop iterator definitions in the header of `for` loops.

Beside the input code for the algorithm itself, the user is allowed to pass another file containing only an isl map in isl syntax, which represents a layout transformation for the used arrays. If no transformation is needed, the parameter for the frontend can be omitted, or an empty map, denoted by `{}`, may be passed. As the bundling methods need to copy the arrays anyway, it is easily possible to change their layout here in order to improve the performance of the kernel code. But on a shared memory system, it may be sufficient to simply divide the input and start the actual computation without any communication overhead. Therefore neither of the bundle methods is invoked and the used kernel must be able to deal with

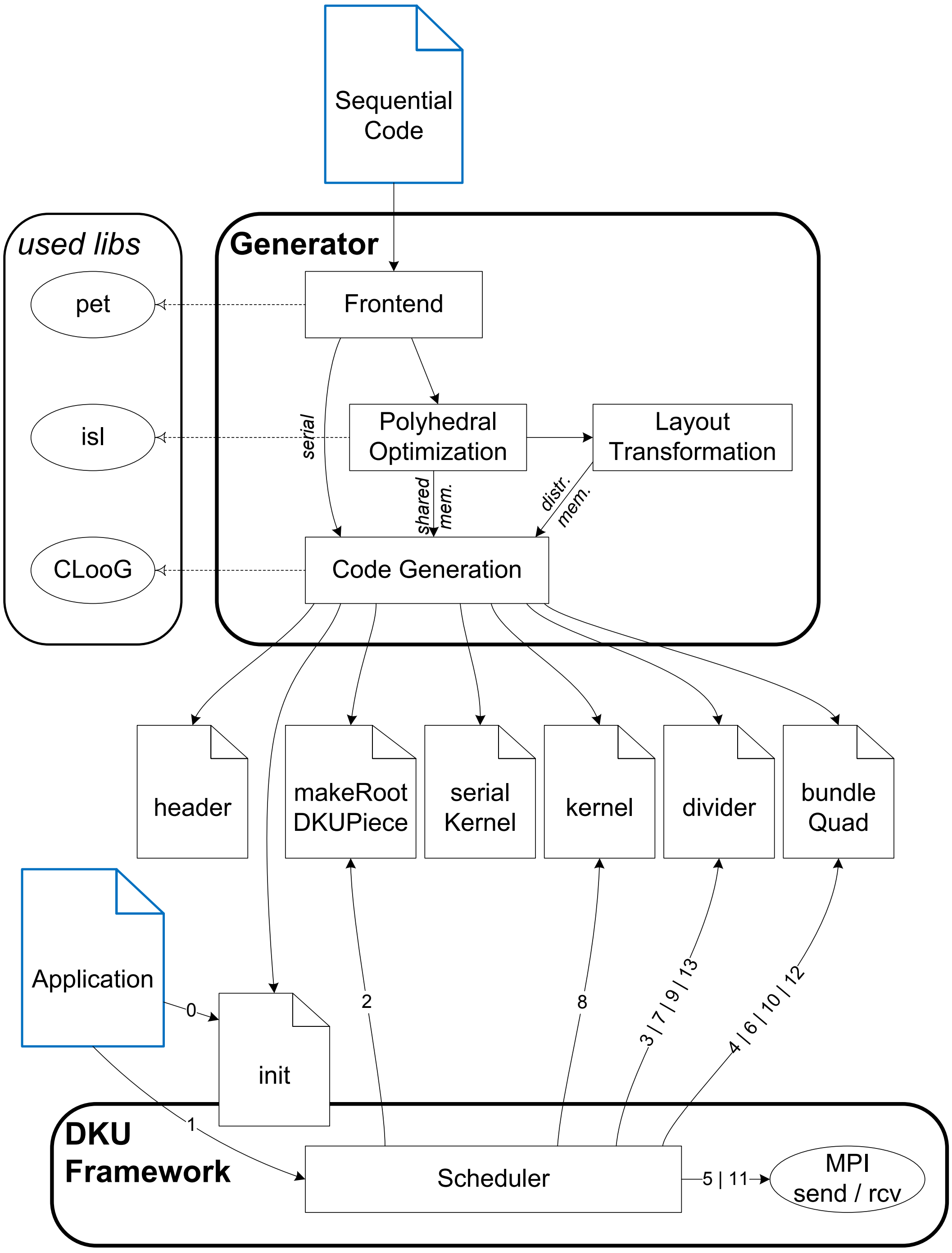


Figure 3.1: Simplified process when using the DKU framework along with the generator. Blue files denote user written code, oval shapes are external libraries.

the original array layout. This enforces the generator to create two different kernel methods, because a layout transformation may change the type of an input array, e.g. when transposing a rectangular, non-squared matrix, but C requires the types of variables to be fixed not later than at compile time.

The extracted polyhedral representation of the input code mainly consists of two parts:

- one array, that contains all used arrays of the input code and
- another array, that holds all statements of the loop nest.

Each statement contains its iteration domain and schedule. The first is represented by an isl set, the latter by an isl map. The actual statement code is represented by an abstract syntax tree, which contains isl maps to represent array accesses. These relations map from the iteration domain to the index vector, which is used to access the memory. The extracted polyhedral description is then transformed to the input for the generator, which is close to the data structure pet returns. A more detailed description of the input can be found in Section 5.1.

Optimizations The next task that the generator executes contains polyhedral optimizations, as depicted in Figure 3.1. The generator uses isl to perform a dependence analysis on the representation. The results are then used to compute an optimal schedule utilizing the built-in scheduler of isl. Based on this schedule, a new iteration domain and a new schedule for the kernel code must be created, which cover only a subset of the original iteration domain. In order to delimit the domain of the isl set, new parameters for the loop bounds have to be introduced. Section 5.2 provides more information about the polyhedral optimizations, that are performed.

An additional part is to apply the layout transformations for all used arrays. This implies to modify the statements itself and also to adjust the bundle methods to ensure they perform the transformations when creating the input bundle and to revert the layout transformation, when collecting all results. A new idea of how to automatically compute a suitable layout transformation is presented in Section 5.3.

Code Generation All computed information is then used to create the source code for the DKU framework. During this process, the polyhedral code generator CLooG is invoked multiple times. The following files are generated here:

header The first one created is the header for the application. It contains declarations for all methods of the following files and two structure definitions. One to encapsulate all user data and another for management information added by the generator and needed by the following files.

init The initializer introduces all user methods to the framework. Therefore, it has to be invoked before executing the scheduler.

makeRootDKUPieces This file contains an equally named method that allocates the first DKU piece based on the user data passed to it.

serialKernel This file contains a fall-back solution for the kernel if only one processor is available. The loop code is generated directly with CLooG using the input model without any transformation or optimization.

kernel Contains two kernel methods: one using the original memory layout, the `localKernel`, and one using the optimized layout, the `kernel`. Analogous to the serial kernel, the loop nest for the application is generated by CLooG's pretty printer.

divider This file contains the methods `divide` and `undivide`. The divider is organized in a way such that all pieces share the same memory for the data. The undivider only need to free the data structure for the piece.

bundleQuad Contains four bundling methods: `bundleInput`, `unbundleInput`, `bundleResult` and `unbundleResult`, can be found in this file. The loop code to copy from or into the bundle memory is also generated by CLooG.

More information on how these methods are generated can be found in Section 5.4.

3.2 DKU Framework

Initialization After all DKU methods are created, the user is able to call the initializer and the scheduler in order to start the computation (c.f. to edges 0 and 1 from Figure 3.1).

A first, architecture-independent step is to create the root pieces for the framework. Therefore, the user method `makeDKUPieces` is invoked (2), which is in general allowed to create more then one root piece. They are used to specify and start several independent computations at once. But as the user code is generated automatically for exactly one given loop nest by the described generator, this method only returns a single root piece. However, an extension of the generator, that allows building more than one root piece is possible.

For parallel systems there are two different architectures that should be discussed here: distributed and shared memory systems.

General Procedure for Distributed Memory When using any kind of distributed memory system, the scheduler divides the given piece into as many subpieces as nodes are available in the network (3). Each subpiece is then bundled using the `bundleInput` method (4), which performs a layout transformation to enhance the cache utilization for the kernel. The created memory blocks are transferred to the corresponding nodes using explicit MPI sends (5) and the receivers restore their piece utilizing the `unbundleInput` method (6). Every node invokes the divider again to generate pieces for each CPU core (7). Bundling is not necessary for distributing pieces to cores as the cores of each node have shared memory. It is also possible to generate more subpieces than computing units are availabe to reduce the computational effort for a single piece. This may also lead to a better utilization of the CPU caches, as smaller blocks of the input arrays are needed at once. After these preparations are done, all `kernel` invocations – one for each subpiece – are performed in parallel using OpenMP (8). Then, every node executes the undivider for each local subpiece (9) and bundles the local result using `bundleResult` (10). Every node transferres its bundle to the master (11), which restores the encapsulated result into the corresponding local piece and reverts its memory layout by calling `unbundleResult` (12). At last, all received pieces must be undivided (13) to ensure the user data corresponding to the root piece contains the whole result, which is now equal to the original, sequential result.

Specialized Scheduler for Shared Memory For an architecture with shared memory only, every thread is able to access the input data. Therefore, no communication and no bundling is needed. In-place access to the input data by the cores implies that memory layout optimizations are not performed, as they are tightly coupled with the bundling methods. However, a bundling and unbundling step may be used anyway to integrate a layout transformation, but this may lead to a performance degradation. In detail, on a shared memory system the scheduler first divides the root piece in at least as many subpieces as physical or logical cores are available and invokes the `localKernel` once for each piece in multiple OpenMP threads. The third and last step then undivides all subpiecesafter the computation has been performed.

Fall-Back Version for Purly Sequential Systems If only one physical processor is available, the scheduler may use the purely sequential kernel to avoid any scheduling overhead. In that case the only required method is the `serialKernel`. Actually, the serial kernel does not need any DKU piece, only a pointer to the user data structure itself is passed to it.

4 DKU Framework

Work on the generator tool started with a prototypical implementation of a framework for the DKU pattern.

Besides the seven methods described in Section 2.5, this implementation of the DKU framework needs another one to generate all root pieces. Each root piece represents a new, independent computation, i.e., in the case of a matrix multiplication example several different multiplications can be specified and scheduled for execution at once. On the one hand, if the user code for DKU is generated automatically for a given loop nest, `makeRootDKUPieces` returns only a single root piece specifing exactly the whole input code. On the other hand, if the user code is modified or even completely written without the generator, the `serialKernel` may provide a version that is optimized on computing several instances at once.

All eight methods must then be introduced to the framework which is done by passing function pointers to a bunch of initialization methods. As this code is very generic, the generator encapsulates it in an initializer method to facilitate the use of the DKU framework.

The first version of the DKU framework was focused on debugging and, therefore, no parallel scheduler was included, i.e., all kernels were executed sequentially. In order to measure the speedup of the generated DKU code, the framework had to be adapted. The most important part of the modifications was to parallelize the scheduler, i.e., to enable it to execute all kernels in parallel. There are three ways to achieve this:

OpenMP First of all, the scheduler could be parallelized using OpenMP. This approach performs very well on a multicore shared memory architecture, as OpenMP has quite low overhead. But as this approach is completeley focused on shared memory, OpenMP provides no mechanism to transfer data using any kind of network.

MPI The Message Passing Interface (MPI) is a standardized approach to deal with distributed memory systems. MPI can handle both point-to-point and collective communication, but in the case of the DKU framework only low-level methods are needed, as the framework itself does not know anything about the data to transfer. Compared to OpenMP it has also a relatively high overhead. Combining both OpenMP and MPI is therefore a good choice to deal with clusters consisting of several multicore machines and to cover a large set of architectures.

MPI requires an initialization before it can be used (e.g. by calling `MPI_Init`). And as the framework is designed to encapsulate even the decision which parallel language should be used, the initializer `void DKU_Init(int *, char ***)` must be invoked before parsing the program parameters and it calls `MPI_Init`, if required. The framework also provides a method `boolean DKU_isMaster()` to protect code that should be executed only by the master. This is required as all MPI nodes are started simultanously when launching the program. A delayed execution of arbitrary nodes is not possible. The presented method enables the user for example to ensure only the master reads or writes files from disk.

GPGPU Beside traditional shared and distributed memory systems, GPGPU is a fairly new domain in high-performance computing. The DKU framework can also be adapted to utilize the graphics processing units, if a CUDA[1] or OpenCL[2] kernel is available. Although this could lead to a higher performance, an automatic code generation for all DKU methods would be more complex and therefore only the composition of OpenMP and MPI was implemented.

Layout Transformations Another contribution is the extension of the DKU pattern by explicitly allowing memory layout transformations. In C, arrays are stored in row-major order, so accessing them row-wise would lead to a much smaller number of cache misses than accessing them column-wise. Consider, for example, the basic matrix muliplication from Figure 4.1. In contrast to the arrays `A` and `C`, `B` is accessed column-wise, which is more complex for the prefetcher to handle effectively. One possible solution to deal with this problem is to transpose matrix `B`. In general an additional rearranging of the memory layout is expensive and may exceed the speedup of the actual computation, but in the special case of using a distributed

[1] Compute Unified Device Architecture (`https://developer.nvidia.com/category/zone/cuda-zone`)
[2] Open Computing Language (`https://www.khronos.org/opencl/`)

```
for (i = 0; i < r; ++i) {
  for (j = 0; j < c; ++j) {
    sum = 0;
    for (k = 0; k < n; ++k) {
      sum += A[i][k] * B[k][j];
    }
    C[i][j] = sum;
  }
}
```

Figure 4.1: Matrix Multiplication Kernel

memory architecture all relevant input data must be transferred to the corresponding node. Therefore, the bundle method is called for each subpiece and all relevant data must be copied anyway, so applying layout transformations here can be done with relatively low overhead.

Beside the possible performance gain this extension also adds new constraints to the framework. First of all, as the bundle methods are optional, they may not be used on a shared memory system, which requires two different kernel methods – the standard in case of the original memory layout and another for the optimized one. The framework then chooses the appropriate version depending on the use of the bundle methods. Another constraint is that the master thread is the only one allowed to bundle the data and also at most once per call, because the layout transformation is tightly coupled with the bundling for performance reasons as aforementioned.

In addition to these explicit improvements the framework was cleaned up. All currently unnecessary preparations for future features were removed in order to reduce the overhead and enhance the readability.

5 Generator

The generator is both a software and a library to generate all DKU methods automatically, starting with a rather simple description of the input program. A brief description of the whole process when using the generator can be found in Chapter 3 and the following sections deliver a deeper insight in all mentioned parts, as well as the problems which came up during the development.

5.1 Input Description

An early question of the project concerns the description of the algorithm that should be executed using the DKU framework. There are two different approches here. First of all, a very flexible one is to read a configuration file which contains all needed information. All parts of the polyhedral description, as for example, iteration domain or schedule, may be written in isl syntax and other parts like statement codes or variable definitions can be given as single C statements. A major disadvantage here is that the user must be capable of directly formulating the iteration domain and the schedule of all statemtents in isl syntax. This approach therefore has a high learning curce and it is rather error-prone.

Second, it is possible to parse a C source file that contains a sequential version of the algorithm that should be parallelized utilizing DKU. On the one hand, this enables the user to write a simple, sequential C code, which is easier for a programmer than the first approach. On the other hand, extracting a polyhedral representation is not as easy as simply parsing a complete polyhedral description, but there are different libraries to solve this problem. The one used in the generator is the polyhedral extraction tool (pet), which is introduced in Section 2.3.

As pet uses a real C compiler to parse the input file, it requires valid C99 source code. A simple input file for a matrix multiplication example is shown in Figure 5.1. The model extracted by pet is used to build the data structure for the generator, which is close to pet's representation. The generator can also be used as a library

```
void MM(int r, int c, int n,
        float A[r][n], float B[n][c], float C[r][c]) {

  float sum;

#pragma scop
  for (int i = 0; i < r; ++i) {
    for (int j = 0; j < c; ++j) {
      sum = 0.0;
      for (int k = 0; k < n; ++k)
        sum += A[i][k] * B[k][j];
      C[i][j] = sum;
    }
  }
#pragma endscop
}
```

Figure 5.1: Generator input for the matrix multiplication.

and its input can be created without pet. The internal data structure, which is created by the frontend consists of the following parts:

id The DKU framework uses an id to distinguish different application codes. It is defined in the generator by either an optional argument, or it is derived from the filename of the sequential input code.

variables This list contains all information about the used variables, as the generator has to insert code to declare them. In case of multi-dimensional arrays, the compiler needs – beside the element type – the extent of the outer dimensions to create code for computing the explicit memory address.

statements Every statement is represented by an abstract syntax tree, which is similar to the reprensentation pet uses.

output This optional parameter can be used to define which arrays contain the result of the computation. If no output arrays are defined, the generated code collects and returns all modified data after the computation. In case of the matrix multiplication only the result matrix is bundled in the `bundleResult` method. But, if there are temporary arrays, whose elements are not needed by the caller, they need not be transferred to the master and so the programmer may use the output parameter to specify this.

In addition to the file containing the input code, the user is also allowed to pass a second file containing a single isl map, which denotes layout transformations for all arrays. The frontend uses isl to parse the map and stores every transformation in the structure for the corresponding variable.

5.2 Polyhedral Optimizations

Dependence Analysis Before any optimization or transformation can be found or applied, the generator needs to perform a dependence analysis. The dependencies describe an ordering between statement instances, which must not be interchanged in order to ensure the result of the program is still valid. A dependence analysis can be performed using the isl. It provides a method for computing dependencies, given `sink` and `source` access relations. Dependencies are called flow dependencies, if they denote from write accesses to a later read access of the same memory location and they can be computed by passing write accesses as `source`s and reads as `sink`s. Write accesses for both `source` and `sink` lead to so called output dependencies and an invocation with reads for the `source`s and writes for the `sink`s computes anti dependencies. All three types must be considered for all variables, because each of them involves a memory modification. Memory accesses can easily be aggregated in a single pass through all statements, as they are represented by isl relations within the abstract syntax trees.

In addition to the dependencies, this analysis also returns all `sink` accesses without any `source` access before. In case of computing flow dependencies, all elements that are read, but never written before according to the given read and write accesses, are also returned. The generator uses this information to determine which arrays contains input values and must therefore be transferred to the corresponding node.

Scheduling In order to use the DKU framework, at least the outermost loop of the resulting kernel code must be parallel. Therefore, the given description of the kernel code is passed to the isl scheduler in order to compute an optimal schedule. The scheduler integrated in isl uses an algorithm which is similar to Pluto's [BHRS08], but Feautrier's scheduling algorithm can be selected as well [Fea92].

```
1  void PP(int na, int nb, double A[na], double B[nb],
2        double C[na + nb - 1]) {
3
4  #pragma scop
5    for (int i = 0; i < na + nb - 1; i++)
6  S0: C[i] = 0.0;
7
8    for (int i = 0; i < na; i++)
9      for (int j = 0; j < nb; j++)
10 S1:  C[i+j] += A[i] * B[j];
11 #pragma endscop
12 }
```

Figure 5.2: Generator input for the polynomial product.

Consider for example the input code from Figure 5.2. The extracted iteration domain and schedule are:

```
[na, nb] -> {
    S0[i]: i>=0 and i<na+nb-1;
    S1[i,j]: i>=0 and i<na and j>=0 and j<nb; }
[na, nb] -> { S0[i] -> [0,i,0]; S1[i,j] -> [1,i,j]; }
```

The original input does not contain an asynchronous parallelism, apart from the initialization of the result array `C`, because different iterations of both the `i`- and `j`-loops in lines 8 and 9 modify the same element of `C`. This can also be seen, when inspecting the computed dependencies, as there is a transition from `S1` to itself for different values of the outermost loop iterator `i`:

```
[na, nb] -> {
    S0[i] -> S1[0,i]: i>=0 and i<nb and na>0;
    S0[i] -> S1[i+1-nb,nb-1]: i>=nb and i<na+nb-1 and nb>0;
    S1[i,j] -> S1[i+1,j-1]: i>=0 and i<na-1 and j>0 and j<nb;}
```

A better schedule for this representation can be computed using isl:

```
[na, nb] -> { S0[i] -> S0[i,0,0]; S1[i,j] -> S1[i+j,i,1]; }
```

The iteration domain and the optimized schedule still describe the same computations, but the ordering of the statement instances may have changed. This reordering may help to exploit the parallelism of the computation. This new system can not be used directly to create the kernel code, as for the outer parallel dimensions, which now might exist, the generator must insert new structure parameters, which restrict the computation one kernel call has to perform. In order to be able to delimit individual dimensions, the generator creates a completely new system for the kernel code by applying the schedule to the iteration domain. This ensures every loop of the resulting kernel code corresponds to exactly one dimension of its iteration domain. The new iteration domain and the according schedule for the kernel is:

```
[na, nb] -> {
  S0[i0,0,0]: i0>=0 and i0<na+nb-1;
  S1[i0,i1,1]: i0>=0 and i0<na+nb-1
      and i1>=0 and i1>i0-nb and i1<=i0 and i1<na; }
[na, nb] -> {
  S0[i0,i1,i2] -> S0[i0,i1,i2];
  S1[i0,i1,i2] -> S1[i0,i1,i2]; }
```

The dependencies for this new system can now be used to determine which dimensions can be executed in parallel (which is explained in the next paragraph):

```
[na, nb] -> {
  S0[i0,0,0] -> S1[i0,0,1]: i0>=0 and i0<nb and na>0;
  S0[i0,0,0] -> S1[i0,i0+nb-1,1]:
      i0>=nb and i0<na+nb-1 and nb>0;
  S1[i0,i1,1] -> S1[i0,i1+1,1]:
      i1>=0 and i1>i0-nb and i1<na-1 and i1<i0; }
```

It is now possible to insert new structure parameters, which represent the new loop bounds in the resulting kernel code.

The final iteration domain and schedule for the kernel code are then:

```
[na,nb,_l0,_u0] -> {
  S0[i0,0,0]: i0>=0 and i0>=_l0 and i0<na+nb+1 and i0<=_u0;
  S_1[i0,i1,1]: i0>=0 and i0>=_l0 and i0<na+nb-1 and i0<=_u0
      and i1>=0 and i1>i0-nb and i1<=i0 and i1<na; }
[na,nb,_l0,_u0] -> {
  S0[i0, i1, i2] -> S0[i0, i1, i2]: nb+na>1;
  S1[i0,i1,i2] -> S1[i0,i1,i2]: nb+na>1; }
```

```
if ((na >= 1) && (nb >= 1)) {
  for (c1=max(0,_l0);c1<=min(_u0,na+nb-2);c1++) {
    S_0(c1,0,0);
    for (c2=max(0,c1-nb+1);c2<=min(c1,na-1);c2++)
      S_1(c1,c2,1);
  }
}
if (nb <= 0)
  for (c1=max(0,_l0);c1<=min(_u0,na+nb-2);c1++)
    S_0(c1,0,0);
if (na <= 0)
  for (c1=max(0,_l0);c1<=min(_u0,na+nb-2);c1++)
    S_0(c1,0,0);
```

Figure 5.3: Generated loop nest for kernel using the input as depicted in Figure 5.2.

The resulting loop nest generated by CLooG for this system is shown in Figure 5.3.

Finding Parallelism In a next step, the generator needs to know how many of the outer loops can be split by the divider, or in other words, the number of outer dimensions that do not carry any dependency. But to determine this, not all dependencies are relevant. Consider again the source code of Figure 5.1, the local variable `sum` is only used to avoid several accesses to the same element of array `C`, which may lead to a performance degradation, if the compilation updates the main memory on each access. But as it is the same variable for all iterations of the `i` and `j` loop, there is an anti dependency from line 12 to line 9 of the subsequent iteration for the outer loops. In a naive approach, this dependency prevents all parallelism, but as every kernel invocation has implicitly its own copy of all non-array variables, a shared update problem can not occur, so there is no problem if such dependencies are violated. Therefore, when computing the number of parallel dimensions, anti or output dependencies for non-array variables must not be considered, as only flow dependencies carry information.

Another problem is that the optimized schedule may contain constant dimensions that do not map to an explicit loop of the resulting kernel code, but only effect the structure of the loop nest, as mentioned in Section 2.3 and depicted in Figure 2.3. Therefore the generator categorizes each dimension as follows:

```
Input: Dependency deps[n_dims]
Result: Category dims[n_dims]

for (int i = 0; i < n_dims; ++i)
  dims[i] = parallel_scalar;

for (Dependency dep : deps) {
  for (int i = 0; i < n_dims; ++i) {
    if (dims[i] == parallel_scalar
        && Is_no_scalar_dimension(dep, i))
      dims[i] = parallel_loop;

    if (Dependency_is_carried_by_dimension(dep, i))
      dims[i] = sequential;
  }
}
```

Figure 5.4: Computation of the category for each dimension.

parallel loop Determines that this dimension describes a loop of the resulting kernel code and it can be executed in parallel. Therefore the divider is allowed to split it.

parallel scalar These dimensions are not represented directly in the generated code, as they refer to constant output dimensions of the computed schedule, which are explained in Section 2.3, but they do not limit the parallelism. They must not be considered by the divider.

sequential For sequential dimensions it does not matter if they are represented in the resulting kernel code by a loop, as they carry a dependence. Therefore neither these, nor any of the following dimensions are allowed to be split by the divider.

Figure 5.4 shows an algorithm for the categorization of all dimensions, given all dependencies. In detail, `Is_no_scalar_dimension(dep, i)` is true, if at least domain or range of dimension `i` from the dependency `dep` is not a constant value. `Dependency_is_carried_by_dimension(dep, i)` returns true, if dimension number `i` carries the given dependency `dep`.

Definition 2 (loop-carried dependency) *A loop or dimension* `i` *carries a dependency iff the* `i`*th element of the distance vector for the dependency is non-zero and all previous* $\mathtt{i} - 1$ *elements are zero.*

The distance vector is simply the difference between the target iteration vector and the source iteration vector of the dependency.

The actual number of outer parallel dimensions and also the number of dimensions the divider is allowed to split can now easily be determined by counting all *parallel_loop* entries of the computed category array that appear before any *sequential* entry.

5.3 Layout Transformation

As already mentioned in Section 4, layout transformations may increase the performance of the application code on distributed memory systems and can be realized with almost no overhead, as the input data must be copied anyway. But the computation of an optimal transformation for a given application code is not trivial.

This section presents a new idea for computing an affine layout transformation for a given program. The core part of this approach is to use an unmodified scheduling algorithm to compute the transformation. Both scheduling algorithms implemented in isl require two different types of dependencies:

validity The computed schedule is guaranteed to respect all validity dependencies.

proximity The second input consists of proximity constraints and the algorithm tries to minimize the dependence distances over them.

If the scheduling algorithm is used to compute a layout transformation, there are no hard constraints on the order of the elements and therefore there are no validity dependencies, the algorithm must consider. But the minimization part for proximity dependencies of the scheduling algorithm can be used to minimize the distance between subsequently used array elements.

As the layout transformation is used to optimize the cache utilization, the distance between subsequent accesses to an array must be minimized. A relation over array elements that indicates for every value the element that is accessed consecutively when executing the kernel can be used as a proximity dependence relation.

The computation of these proximity dependencies starts with the access relations for a given array and a polyhedral representation of the kernel code. Consider for example the matrix multiplication from Figure 5.1, if the statement in line 11 is named S, the access relation for array B can be denoted as:

```
[r,c,n] -> { S[i,j,k] -> B[k,j] :
    0<=i<r and 0<=j<c and 0<=k<n }
```

In a next step, new access relations are computed by removing all array dimensions. The resulting relation then represents accesses to the whole array. For the access to array B, the modified relation could be:

```
[r,c,n] -> { S[i,j,k] -> B[] :
    0<=i<r and 0<=j<c and 0<=k<n }
```

Then a flow dependence analysis is performed, using the newly created access relation for both source and sink. For the given example, the schedule of the source code of Figure 5.1 is simply an identity map, so the computed dependencies are:

```
[r,c,n] -> {
  S[i,j,k] -> S[i,j,k+1] : 0<=i<r and 0<=j<c and 0<=k<n-1;
  S[i,j,n-1] -> S[i,j+1,0] : 0<=i<r and 0<=j<c-1 and n>=1;
  S[i,c-1,n-1] -> S[i+1,0,0] : 0<=i<r-1 and n>=1 and c>=1; }
```

These dependencies now denote for every statement instance that accesses the given array, the subsequent instance under the given schedule, which also accesses the same array. A last step is now to transfer both domain and range from the iteration space to the index space of the appropriate array. This can be done by simply applying the original access relation to both domain and range of the dependence maps. For the matrix multiplication example this would lead to the following proximity dependencies:

```
[r,c,n] -> {
  B[k,j] -> B[k+1,j] : 0<=k<n-1 and 0<=j<c and r>=1;
  B[n-1,j] -> B[0,j+1] : 0<=j<c-1 and r>=1 and n>=1;
  B[n-1,c-1] -> B[0,0] : r>=2 and n>=1 and c>=1; }
```

It is easy to verify that the source code from Figure 5.1 accesses the elements of array B in the same ordering as denoted by the dependencies.

In theory, the scheduler should now return an optimal layout transformation for the appropriate array, as it minimizes the distance between subsequent array accesses. But the actual output is in most cases an identity, as the given proximity constraints apear to be too complex for the scheduling algorithm. There are two approaches which might solve this problem. On the one hand, a heuristic can be used to preprocess and simplify the computed dependencies. On the other hand, the scheduling algorithm can be adapted to this problem. The first approach was tested in this work and the adaption of the algorithm remain as future work. Following the example above, the first two mentioned maps contain multiple elements and therefore they describe multiple different dependencies, depending on the value of the structure parameters `n` and `c`. In contrast to this, the third map only contains a single element, which leads to a single dependency, namley from the last element of the matrix `B` to the first one. So the last map can be considered as less important for the layout transformation. Removing it and invoking the isl scheduler then results in the desired transformation:

```
[c,n,r] -> { B[k,j] -> B[j,k]; }
```

A filter for the dependencies, which removes maps, that contain only a single element is implemented, but this is only sufficient for rather simple input codes, as e.g. the matrix multiplication. With an increasing complexity of the input, the requirements for the preprocessing heuristic are also growing and a universal solution could not be found.

In contrast to this, specializing the scheduler is more extensive, but it may lead to more satisfactory results for a larger number of programs.

5.4 Code Generation

For the code generation part, two different approaches are possible, the *templated generation* and the *transformer generation*.

On the one hand, the application code could be created by using a template. That means, the application code is derived from a generalized code file, called template. The template is then modified at previously marked positions to adapt it according to the given model. An advantage of this apporach is that the template can be basically read and understand without inspecting the source code of the generator, but the use of a template needs the generator to be able to parse it in order to find and replace all marks.

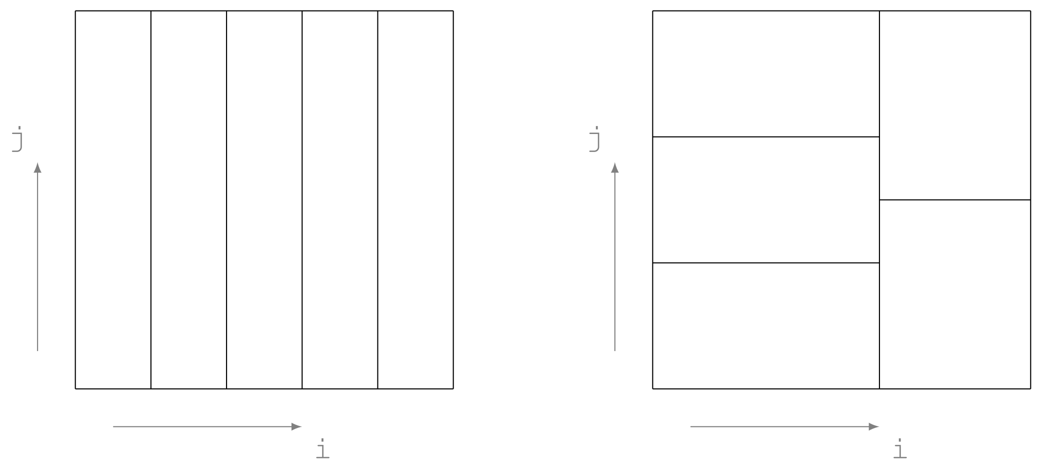
On the other hand, a more lightweight and more flexible approach is to use a transformer. That means, the generator writes the complete application code directly into the output file and therefore it handles a huge amount of strings. Structural modifications of the generated code also require a recompilation of the whole generator and in contrast to a template file, the inspection of the basic structure results in studying the source code, or the generated files for different inputs. But as the transformer is easy to integrate and it does not need additional external libraries, this approach was chosen for the generator.

5.4.1 Divider / Undivider

The divider's task is to split the input piece in as many subpieces as desired, in a way that every subpieces leads to an almost identical amount of work for the kernel. It is allowed to create more or fewer subpieces as requested, but to optimally exploit the hardware, the given number should be respected.

Distribution of Subpieces accross parallel Dimensions In case of multiple parallel loops, the computation can be split in more than one dimension. This may lead to problems if the requested number of subpieces is prime. Using a consistent distribution, i.e. splitting the iteration domain using hyperplanes, the only possibility to respect the given number is to split only a single dimension. This could be a waste of parallelism potential, as splitting multiple dimensions also lead to an indirect tiling of the computation. Consider again the matrix multiplication example from Figure 2.1a, the outer `i` loop iterates over all rows and the `j` loop traverses the columns of the result matrix. If only the first dimension is split by the divider, each of the resulting subpieses tranverses the array `B` multiple times witht he maximum possible reuse distance. That means, before a specific element of the matrix is accessed the second time, all others are required first. The CPU cache can now only increase the performance if the whole matrix `B` can be cached at once. If the divider now splits both the `i` and the `j` loop, each subpiece requires only a subset of the array `B`. This leads to a smaller reuse distance, which may increase the performance, if all accessed data can be cached now.

Consider an application with two parallel loops that should be split in five subpieces. Figure 5.5a shows a naive partitioning of both dimensions. It can easily be seen that all subpieces cover the same area, but this solution does not exploit the second parallel dimension *j*. A better partitioning for the same problem is shown in Figure 5.5b. This can be achieved by dropping the constraint that each dimension has to be split evenly by using hyperplanes, for one single dimension. Furthermore



(a) If all dimensions must be split evenly by hyperplanes, the only partitioning possible does not exploit both dimensions, as the requested number only has the trivial factorization $5 \cdot 1$.

(b) If a single dimension is allowed to be split arbitrarily (here dimension j), a better partitioning is possible.

Figure 5.5: Possible partitioning with five subpieces created by the divider for a two dimensional iteration space.

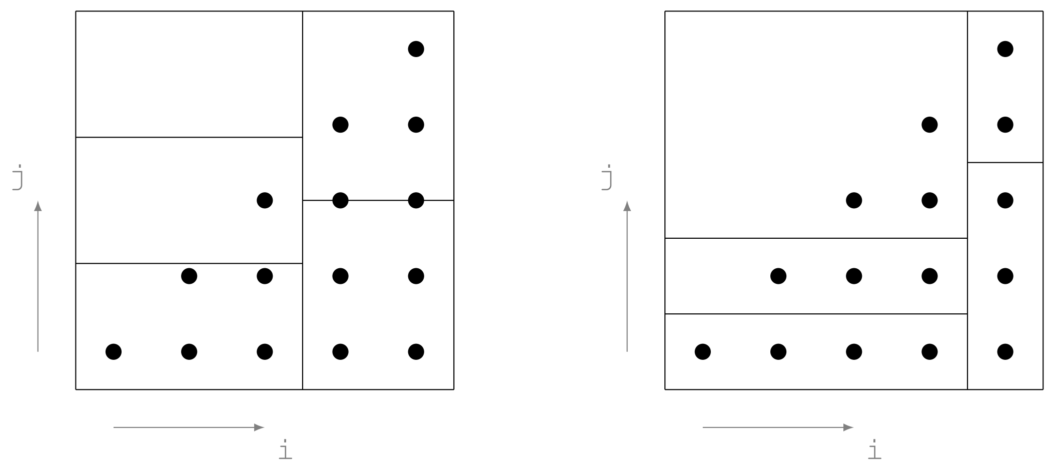
```
Inputs: $n$, $d$
Results: $c_i$, $b$

remaining = $n$;
prod = 1;

for i = 0...$d$-2 {
  root = round( $\sqrt[d-i]{\text{remaining} + \frac{1}{4}}$ );
  remaining = $\frac{\text{remaining}}{\text{root}}$;
  $c_i$ = root;
  prod = prod · root;
}

$c_{d-1}$ = remaining;
prod = prod · remaining;
$b$ = $n$ - prod;
```

Figure 5.6: Algorithm used to compute the number of bars c_i per dimension.



(a) subpieces created independently from the structure of the iteration domain
(b) optimal loop bounds for the subpieces according to the iteration domain

Figure 5.7: Illustration of five subpieces for the code from Figure 2.1a.

it is possible for an arbitrary dimensionality $d \in \mathbb{N}$ to find a $k \in \mathbb{N}$ and a partitioning of the iteration domain into $n \in \mathbb{N}$ subpieces such that all except one dimension are divided into either k or $k+1$ blocks and for the last dimension all blocks are also divided into either k or $k+1$ pieces. This can be formulated as follows:

Conjecture 1
$\forall n, d \in \mathbb{N} : \exists k \in \mathbb{N} : \exists c_0, \ldots, c_{d-2} \in \{k, k+1\} : k \cdot \prod_{i=0}^{d-2} c_i \leq n \leq (k+1) \cdot \prod_{i=0}^{d-2} c_i$

The procedure used to compute such a partitioning is shown in Figure 5.6. Its correctness is not proven mathematically, but was tested empirically for all combinations of $n \in \{1, \ldots, 1.000.000\}$ and $d \in \{1, \ldots, 100\}$. The summand $\frac{1}{4}$ inside the root in line 8 is required to ensure the computed c_i have the desired form. The results $c_0, \ldots, c_{d-2}$ determine the number of blocks the first $d-2$ dimensions are split into. All these blocks are then divided into k or $k+1$ pieces, such that overall n pieces are available.

Shape and Size of the Subpieces Another problem appears at the explicit subpiece creation. The given code only computes the layout of the subpieces, but their explicit shape and size must be determined in a second step. Consider again the source code from Figure 2.1a, a naive creation of the subpieces is shown in Figure 5.7a. It can easily be seen that this does not lead to an appropriate load balance, as for example the upper left subpiece does not contain any computation. Therefore, it is advisable to shift the subpiece boundaries in order to balance the number of statement instances inside them. Figure 5.7b shows an optimal partitioning of the given iteration domain. It can be determined by performing a binary search and the generator is able to create code for this part. In detail, let $\rho(p)$ be the number of integral points inside the iteration domain of the kernel description, which is encapsulated in the subpiece p. The generator uses barvinok [VSB+07], a library which extends isl, to compute a C expression for ρ. The subpieces $p_1, ..., p_n$ then denote an optimal partitioning of the parent piece p, if $\rho(p_1) \approx ... \approx \rho(p_n) \approx \frac{\rho(p)}{n}$. The best loop bounds for a subpiece p_i can be computed by evaluating $\rho(p_i)$ and comparing it to the desired $\frac{\rho(p)}{n}$. If the number of iterations matches, an optimal piece was found, if its too small or too large, the size must be adapted accordingly and tested again. As Figure 5.7b shows, it is in general not possible to find a perfect solution, for which all subpieces are exactly equally sized, because the subpieces are restricted to a rectangular shape. But in practice, if the overall number of iterations is large enough, its impact is quite low.

In contrast to the rather complex divider code, the undivider only has to free the subpiece structure. As the divider does not copy the user data for the subpiece creation, but only references the same structure the original piece contains, the undivider does not need to collect any data for the result.

5.4.2 Bundle / Unbundle

General Specification The main specification for the bundle code is to create a single memory block that contains all data which must be transferred to other nodes. But, as communication and memory transfers produce a huge overhead, another important part for the bundle methods is to limit it as good as possible.

Communication Reduction A fist obvious improvement is to transfer not all arrays entirely to every node, but only the part the corresponding node requires in order to compute its subpiece. The information which array and even what specific elements a single node requires to compute its part can be extracted from the results of the dependence analysis, as suggested in Section 5.2. The flow dependence analysis isl provides, requires in general three different relations, a `schedule`, which indicates at which specific virtual point in time the statement instance is executed and two maps that indicates `source` and `sink` access relations to an arbitrary domain. In addition to the dependencies, the used algorithm computes a relation `no_source`, which maps each point p of the given domain to the according target *t*, which is specified by the `sink` access, if no chronological previous – according to the given `schedule` – point *q* exists, that reads the same target *t*, as indicated by the `source` relation. Applying this mapping to the iteration domain of the kernel description results in a set, describing only the array elements that has to be transferred.

The most aggressive reduction of the communication volume can then be realized by simply using the created set as an iteration domain for a new loop nest, which can easily be generated utilizing CLooG. The loop nest now scans only the needed array elements and therefore the body of this loops must simply copy the element at the index denoted by the iteration vector to the next empty slot of the bundle. In order to allocate enough memory for the bundle, the number of elements for all arrays can be computed by counting all integral points inside the corresponding isl set using barvinok. The same code may also be used to extract the bundle inside the `unbundleInput` method.

A similar optimization can also be performed for the results bundle. As the `unbundleResults` method only has to extract the computed results in order to ensure data from other nodes is not overwritten, all information needed to optimize the communication volume has to be computed anyway. This information can easily be determined by collecting all access relations that represent only array modifications, and applying them to the iteration domain of the kernel representation. The resulting set now represents all array elements, computed for a given subpiece description. The loop code for both `bundleResults` and `unbundleResults` can then be generated analogous to `bundleInput` and `unbundleInput` respectively.

Memory Reuse Another rather simple possibility to reduce the overhead of the memory transfers is to reuse the bundle memory. Therefore, the `unbundleInput` method neither need to allocate new memory for the input data nor extract the bundle explicitly. The only task left is to compute the positions of all input arrays inside the bundle memory and to store the corresponding pointer in the piece structure.

If the whole input is transmitted to the nodes, the offset calculation is quite simple, as only the complete size of every input array must be known and this information is part of the input description.

But if the communication overhead should be minimized as mentioned above, the code generation is a bit more complex. The presented aggressive reduction of the bundle size may lead to an arbitrary new layout of the input arrays, as it is possible that only every second element of an array is needed, or for a two dimensional matrix only elements in the lower triangle are used. Therefore, if the resulting structure from the bundle memory should be used in the kernel code, the access pattern may become arbitrarily complex. In order to prevent this, the generated input and result bundle contain the rectangular hull of all array elements that have to be communicated. Therefore, the index vector for the array accesses in the kernel code must only be shifted according to the position of the computed rectangular hull inside the original array. The amount of memory a single array requires in the bundle is computed by the product of the array extent for each dimension separately, which is defined by the difference between the maximum and the minimum index, the array is accessed with. The extent for each array dimension is also used to define the complete type of the variable which is needed in the kernel code to access the arrays.

If the bundle memory for the results bundle should also be reused, it has to be allocated in the `unbundleInput` method before calling the kernel method. At this point, if a single array is used for both input and output, its content must be copied from the input bundle to the results bundle.

Another important issue when reusing the bundle memory is the alignment of each array. If the base address for the array accesses is no multiple of at least eight, the performance of the resulting code may be worse depending on the processor it executes. Therefore, it is reasonable to align all arrays to a multiple of the word size of the used architecture, which must then also be considered when computing the offsets inside the bundle memory.

5.4.3 Kernel

The generator provides three kernel methods, one for a sequential version to minimize the overhead if only a single processor is available on the target hardware and two versions in the context of DKU for both distributed and shared memory architectures (with and without layout optimizations respectively). The main part of all kernel methods, the loop nest, which contains the actual application code, is generated by CLooG using either directly the extracted polyhedral reprensentation of the input code for the serial kernel, or the optimized one for both DKU kernel methods. As suggested in Section 2.4, the source code for the statements is inserted using preprocessor macros.

Consider again the matrix multiplication input from Figure 5.1, the initialization part of the generated kernel for distributed memory systems is shown in Figure 5.8.

```
1   // restore all user data from DKUPiece
2   struct data_DKU_GEN_MM *_d = _piece->appSpecificPiece;
3
4   // initialize primitive user variables
5   int c = _d->c;
6   int n = _d->n;
7   int r = _d->r;
8   float sum = _d->sum;
9
10  // initialize variables for loop bounds
11  const int _l0 = _d->_internal._l[0];
12  const int _u0 = _d->_internal._u[0];
13  const int _l1 = _d->_internal._l[1];
14  const int _u1 = _d->_internal._u[1];
15
16  // initialize variables for memory access bounds
17  const int __l0 = _d->_memory->_l[0];
18  const int __u0 = _d->_memory->_u[0];
19  const int __l1 = _d->_memory->_l[1];
20  const int __u1 = _d->_memory->_u[1];
21
22  // initialize user arrays
23  float (*restrict A)[n] = _d->A;
24  float (*restrict B)[n] = _d->B;
25  float (*restrict C)[(-__l1 >= 0 && -2 + c - __u1 >= 0) ? (__u1 + 1) :
26     (-1 + __l1 >= 0 && -2 + c - __u1 >= 0) ? (-__l1 + __u1 + 1) :
27     (-__l1 >= 0 && 1 - c + __u1 >= 0) ? (c) : c - __l1]
28    = _d->C;
```

Figure 5.8: Initialization part of the generated kernel code for a distributed memory architecture.

The kernel has to declare and extract all needed variables from the DKU piece first. The primitive variables can easily be copied, but the declaration of the arrays is more complex, as for multi-dimensional ones the compiler needs the extent of the inner dimensions in order to generate code to compute the exact memory location for a given index. As explained in Section 5.4.2, the arrays are not transferred completely if not necessary and as they are still stored in the bundle memory, their extent may differ from the original one. This leads to the rather complex definition for array `C` in lines 25 – 28.

The additional scalars declared in lines 11 – 20 represent the loop and memory bounds for this subpiece. The latter ones are needed as DKU pieces are allowed to be divided several times subsequently and the array extent may depend on the bounds of a super piece.

Figure 5.9 shows the second part of the kernel code following the example from Figure 5.8. The actual code for the computation of the result is generated by CLooG using the newly created iteration domain as explained in Section 5.2. But as the statements depend on the original iteration domain, it is rebuilt using an additional preprocessing level, as depicted in lines 22, 24 and 26.

The statement code must also be modified slightly, as the extent and even the layout of the used arrays might have changed when transferring them to another node in a distributed memory system. In case of the matrix multiplication example, the array `B` is transposed in order to enable a row-wise access. The elements of all access vectors are also adapted in order to compensate the smaller extent of the arrays. The generated offsets are computed before executing the loop nest and they are stored in separate variables in order to ensure they are computed only once and not for each memory access over and over again.

For shared memory systems, the generated kernel method, the `localKernel` is very similar to the presented one. The only difference is that it uses the original arrays passed to the framework by the caller. That means, no layout transformations or modifications of the array sizes must be considered and the access relations must not be adapted.

```
  // calculate array offset only once
  // (compiler does not recognize it is always the same value)
  const int _offset_A0 = (-__l0 >= 0) ? (0) : __l0;
  const int _offset_A1 = 0;
  const int _offset_B0 = (-1 + __l1 >= 0) ? (__l1) : 0;
  const int _offset_B1 = 0;
  const int _offset_C0 = (-__l0 >= 0) ? (0) : __l0;
  const int _offset_C1 = (-__l1 >= 0) ? (0) : __l1;

  // add user statements via cpp
#undef _S_0
#define _S_0(i, j) do { (sum = 0); } while(0)
#undef _S_1
#define _S_1(i, j, k) do { (sum += \
    (A[-_offset_A0 + i][-_offset_A1 + k] * \
    B[-_offset_B0 + j][-_offset_B1 + k])); } while(0)
#undef _S_2
#define _S_2(i, j) do { \
    (C[-_offset_C0 + i][-_offset_C1 + j] = sum); } while(0)

#undef S_1
#define S_1(c1, c2, c3, c4, c5) _S_1(c1, c2, c4)
#undef S_2
#define S_2(c1, c2, c3, c4, c5) _S_2(c1, c2)
#undef S_0
#define S_0(c1, c2, c3, c4, c5) _S_0(c1, c2)

  // declare loop variables
  int c1, c2, c3, c4, c5;

  // loop code, generated by cloog with clast_pprint(..)
  if (n >= 1) {
    for (c1=max(0,_l0);c1<=min(_u0,r-1);c1++) {
      for (c2=max(0,_l1);c2<=min(_u1,c-1);c2++) {
        S_0(c1,c2,0,0,0);
        for (c4=0;c4<=n-1;c4++)
          S_1(c1,c2,1,c4,1);
        S_2(c1,c2,1,n,0);
      }
    }
  }
  if (n <= 0) {
    for (c1=max(0,_l0);c1<=min(_u0,r-1);c1++) {
      for (c2=max(0,_l1);c2<=min(_u1,c-1);c2++) {
        S_0(c1,c2,0,0,0);
        S_2(c1,c2,1,n,0);
      }
    }
  }
```

Figure 5.9: Computational part of the generated kernel code for a distributed memory architecture. (Continuation of Figure 5.8.)

6 Experiments

6.1 Available Hardware

The major part of all benchmarks and experiments run on a non-uniform memory access (NUMA) system, equipped with eight QuadCore AMD Opteron 8356 processors. Non-uniform memory access means, that each one of the eight processors has its own memory and therefore its own link to it, which must not be shared with other sockets. But accessing the memory of another processor is therefore slower than working with the own memory, as all transferes must be routed through another socket. For this machine, each CPU uses 8 GB main memory, which leads to a total amount of 64 GB RAM.

Other computers available for experiments and tests were normal workstations equipped with various processors from AMD and Intel, starting with an Intel Pentium SU4100, up to a Core i5-2400, or a Phenom II X6 1045T.

6.2 Problems and Paradox Behavior

During the evaluation of the results for the initial experiments some strange and at first sight paradoxical results appeared. These are discussed in this section in order to demonstrate different effects, which can invalidate benchmarks.

Influence of Cache Associativity One of the benchmarks tested, was the *gemm* kernel from the PolyBench/C benchmark suite. For 1024 times 1024 matrices the results from the original PolyBench kernel were surprisingly poor compared to a simple one thread DKU version, which additionally copies all input arrays to a new memory location (but without any layout transformation). The original kernel took about two to three times more time to compute the result matrix, although the instructions for the actual loop nests, the processor executes, were completely identical.

Further investigations had shown, that the allocation method of the PolyBench code enforces a memory alignment to at least 32 byte for all three matrices, in contrast to only 8 byte alignment, the generated DKU code uses. The hard alignment in conjunction with the row size for the second input matrix results in a quite disadvantageous situation for the CPU cache.

To understand this problem, the functionality of the cache must be described a bit more detailed first. As the main memory of a computer is extremly slow compared to the CPU, directly accessing all data from the main memory would lead to a poor performance. To prevent this, a small but fast cache is available to store most frequently used main memory locations. This memory location can not be accessed directly by the program, it is completely managed in hardware. When the processor needs to access the main memory, it first searches the cache for the corresponding data by comparing the memory address of the access with all addresses hold by the cache. But for larger cache sizes, this would lead to a high amount of comparisons, which should be performed in parallel in order to retain a fast access. To reduce the overhead for a large amount of comparison units, a given memory address is only allowed to be stored in a subset of the whole cache. AMD Athlon CPUs have for example a 2-way associative (level-1) cache, which means that each memory location can be stored in only two different cache locations. Therefore, only two cache entries have to be checked for a given memory access. To search the cache for a single access, the address is split into three different parts:

1. If one cache entry, which is called cache line, consists of i byte, the least significant i bits of the address are used as an index for the entry and must not be considered when searching for the appropriate cache line.

2. The next n bits of the remaining address are then used to determine which section of the cache must be checked. For a N-way associative cache, the relation between the total number of cache lines L and the number of lines N inside a section can be formulated as $L = 2^n \cdot N$.

3. The remaining bits are then compared to the tag of the N cache lines of the corresponding section in order to check if the needed memory location is available (cache hit), or if the main memory must be accessed (cache miss).

Back to the matrix multiplication example, as the second input matrix `B` (c.f. Figure 5.1) is accessed row-wise in the inner loop with a row size of 1024 double values, the 13 least significant bits of the memory accesses are identical. Therefore, only few sections of the cache are possible to store the elements of this matrix. In addition to this, each cache line can store only two different locations, as the used AMD machine has a 2-way associative level-1 cache. In conjunction with the strong alignment restrictions from polybench, the result matrix `C` is also cached in the same sections as the according column of `B`, which overloads the cache even further.

In contrast to this, the generated bundle methods for the DKU version only enforce an 8 byte alignment, which lead at runtime to slighlty different bits in the lower part of the addresses for `B` and `C`. Therefore, both arrays do not collide according to the cache sections and a larger part of the cache can be used, which results in a higher performance for the DKU code.

The first mentioned problem, the accesses to the array `B` itself, can be avoided by using a slightly different row size, which lead to different sections used to cache the elements of the input array. 1025 rows, instead of only 1024, increases the performance by roughly factor five to seven on AMD CPUs.

Another interesting part is that both demonstrated effects only appear on AMD processors. On an Intel Core i5-2400 neither of these occurs in a comparable way. One explanation might be, that Intel uses an 8-way associative level-1 cache, so for every memory address there are eight different cache locations possible, not only two.

AMDs Prefetcher Other strange results appeared for one of the first experiments during the implementation of the generator. A simple matrix multiplication example as shown in Figure 5.1 was used in a first version to test the speed up of the DKU pattern. The scheduler used for this experiment copies all data to a newly allocated memory location, regardless if it is useful or needed. In this situation, parts of the input data should be available in cache, when invoking the kernel.

Running this test on an AMD processor with a matrix size of 1000 times 1000 and a complete number of 25 subpieces is significantly faster, then the same experiment with only 24 or 26 subpieces. For the fast version of 25 subpieces, each subpiece describes a mtrix multiplication of size 200 times 200. Therefore, the second input matrix is traversed row-wise, which leads to memory accesses of stride 200 times 4 Byte. Further investigations had shown, that for AMD CPUs a stride of 200 single precision floating point values results in a much smaller number of level-1 data cache misses for this example than a stride of 199, or 201 values. This effect occurred on both available AMD processors, the previously mentioned AMD Opteron 8356, as well as an AMD Phenom II X6 1045T. But in contrast to AMD, neither of the Intel

matrix size	600	700	800	900	1000	1100	1200	1300
one socket	0.127	0.203	0.319	0.519	0.672	0.875	1.133	1.325
four sockets	0.127	0.201	0.314	0.516	0.663	0.846	1.068	1.286

matrix size	1400	1500	1600	1700	1800	1900	2000	2500
one socket	2.096	3.508	5.549	7.811	9.652	12.037	14.129	27.621
four sockets	1.592	2.112	3.649	7.601	9.851	12.280	14.829	28.911

Figure 6.1: Comparison of the *gemm* execution time for a different thread allocation on a NUMA architecture. (Lower values are highlighted.)

CPUs available, starting from a Pentium SU4100 up to a Core i5-2400, preferred a stride of 200 values.

Thread Allocations on a NUMA Architecture When benchmarking the PolyBench/C *gemm* kernel, the situation of a single MPI node and four OpenMP threads computing 16 subpieces each showed an unexpected behaviour according to the thread allocation. For this experiment, all threads accesses the same memory to compute the result of the matrix multiplication. Binding all four threads to the four cores of a processor should lead to a better performance, than spreading them among different ones. In the former case, all threads can access the main memory directly, whereas in the latter situation three threads require more time to read the input data, as they must access the main memory located at another socket.

But for special input sizes, the more spreaded version run faster, as depicted in Figure 6.1. One explanation for these results is the increased cache size, because every processor has its own cache hierarchy. For matrix sizes of 700 times 700, one subpiece requires to traverse one submatrix of the second input with size 700 times 88 multiple times, which results in almost 500 KB data. For four threads running in parallel, all data fit in the 2 MB level 3 cache.

If the matrix sizes increase, the required data can not be cached on a single processor at once, but if all four threads run on different CPUs, each with its own 2 MB of level 3 cache, the number of accesses to the main memory can still be held low.

For larger matrices, if the required data does not fit into the level 3 caches anymore, the single socket version is gaining on and even passes the four socket version, as it has a lower latency when accessing the memory. For a matrix size of 1600 times 1600, the required submatrix, which is traversed at a stretch, consumes about 2.4 MB, which exceeds the level 3 cache, but it may fit into the combined level 2 (512 KB) and level 3 (2 MB) caches. For 1700 times 1700, the distance between both versions is quite low again and for the 1800 times 1800 and above, the one socket version is faster.

6.3 Preparations

Experiment Selection During the development of the generator, its correctness was tested using three simple input files containing a matrix multiplication, a polynomial product and a n-body simulation. All three input files are shown in Figures 6.2, 6.3 and 6.4. Their results are discussed first in this section.

In addition to these three examples, for the evaluation of the generator all 30 benchmarks from the PolyBench/C[1] benchmark suite version 3.2 were tested if the generator is able to deal with them. As DKU requires asynchronous parallelism in its current state, only nine benchmarks can be processed directly. One of them, the *reg_detect* benchmark contains only a single parallel loop which consists of at most twelve iterations for the largest suggested input data. Therefore, this benchmark was left out and it will not be discussed in this work.

For synchronous parallelism it would be possible to invoke the DKU scheduler multiple times within a loop, which carries all previous dependencies, but on a distributed memory system, all data are then transferred quite often, which may reduce the performance noticable. However, this approach was chosen for the n-body simulation in order to test if it is a usefull method to handle this type of application.

Setup The PolyBench/C code already provides a mechanism for measuring the walltime of the kernel invocation by querying the system time directly before calling the kernel method and straight after returning from it. The same mechanism is also used for measuring the execution time of a DKU call. Therefore, the measured times contain all memory transfers and also the overhead caused by the DKU-pattern, as, for example, the time needed to compute the subpieces, to start and terminate all OpenMP threads or to manage the MPI nodes. For the non-PolyBench examples, the same strategy was chosen to measure the time of both the sequential and the DKU version.

For the DKU framework, four different behaviours were tested on the previously described 32-core machine:

[1] `http://www.cse.ohio-state.edu/~pouchet/software/polybench/`

```c
void MM(int r, int c, int n,
        float A[r][n], float B[n][c], float C[r][c]) {

    float sum;

#pragma scop
    for (int i = 0; i < r; ++i) {
        for (int j = 0; j < c; ++j) {
            sum = 0.0;
            for (int k = 0; k < n; ++k)
                sum += A[i][k] * B[k][j];
            C[i][j] = sum;
        }
    }
#pragma endscop
}
```

Figure 6.2: Input file for the matrix multiplication example.

```c
void PP(int na, int nb, double *A, double *B, double *C) {

    double sum;

#pragma scop
    for (int i = 0; i < na+nb-1; i++) {
      sum = 0.0;
      for (int j = max(0, i-nb+1); j <= min(i, na-1); j++)
        sum += A[i-j] * B[j];
      C[i] = sum;
    }
#pragma endscop
}
```

Figure 6.3: Input file for the polynomial product example.

```
void NB(int n, double dt, double G, double mass[n],
      double posOld[n][2], double posNew[n][2],
      double velOld[n][2], double velNew[n][2]) {

   double accX, accY;
   double dX, dY;
   double tmp;

#pragma scop
   for (int i = 0; i < n; ++i) {

      accX = accY = 0.0;

      for (int j = 0; j < n; ++j) {
        if (i != j) {

         dX = posOld[j][0] - posOld[i][0];
         dY = posOld[j][1] - posOld[i][1];

         tmp = dX * dX + dY * dY;
         tmp = mass[j] / (tmp * sqrt(tmp));

         accX += dX * tmp;
         accY += dY * tmp;
        }
      }

      tmp = dt * G;
      accX *= tmp;
      accY *= tmp;

      posNew[i][0] = posOld[i][0] + dt
                 * (velOld[i][0] + .5 * accX);
      posNew[i][1] = posOld[i][1] + dt
                 * (velOld[i][1] + .5 * accY);

      velNew[i][0] = velOld[i][0] + accX;
      velNew[i][1] = velOld[i][1] + accY;
   }
#pragma endscop
}
```

Figure 6.4: Input file for the n-body simulation example.

OpenMP local In this scenario, only OpenMP threads are used to parallelize the algorithm. No bundling is performed and therefore only the DKU methods `divide`, `localKernel` and `undivide` are called.

OpenMP bundle As for the local version, only OpenMP threads are used, but this time, the `bundle` methods are used as well. That means, input and result data are copied one single time and all layout transformations are performed.

MPI In this version, only MPI nodes are used to parallelize the application. Therefore, the input and result data is bundled and copied once for every node. That means, each node uses only the part of the input it needs to compute the result, which is also transferred to the physical memory associated to the CPU, it is executed by.

OpenMP + MPI The last one is a combination of the two previous described versions. On the first level, the input data is divided for all MPI nodes and each node receives the input data for its subpiece. On a second level, every node redivides its subpiece and computes them in a loop, which is parallelized using OpenMP. As the used machine consists of eight quadcore processors, every node uses four OpenMP threads to compute its subpieces. Every node and all associated OpenMP threads are bound to a single processor, which ensures every physical processor core accesses only its own main memory during the execution.

If the input code allows dividing a DKU piece in more than one dimension, the framework performs an implicit loop tiling for a larger number of subpieces. Therefore, the number of subpieces requested from the divider is increased by the factor 16, which results in 16 smaller subpieces a single thread has to compute, in contrast to only a single one for the other versions. For future work it would be possible to determine the optimal number of subpieces and therefore tiles, instead of using a fixed factor here, to increase the performance even further.

In order to ensure comparable and stable execution times, all tests were executed 21 times and their median was chosen as the final result. As the generator works almost completely automatically, all DKU versions are compared to the unmodified sequential version. Therefore, for the visualization, only the speedup compared to the original version compiled with gcc is shown. The speedup is defined as the ratio of the sequential execution time to the time, the parallelized DKU code needs to compute the same result.

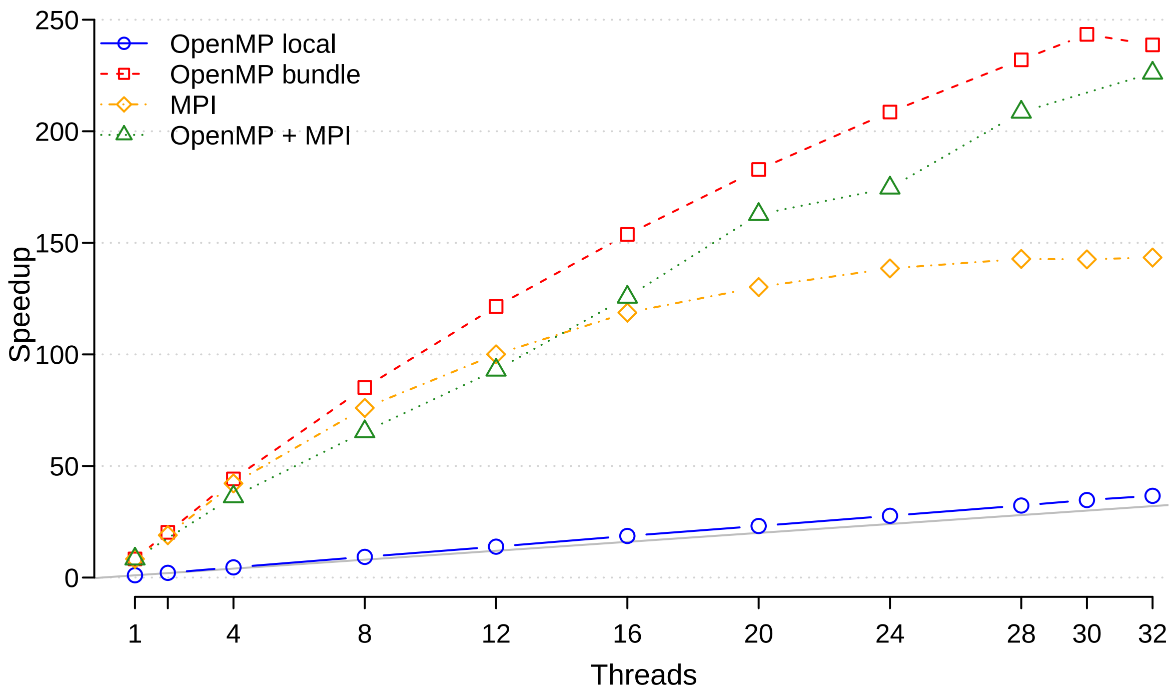


Figure 6.5: Matrix Multiplication using a scalar temporary. 2 parallel dimensions, 16 subpieces per thread.

In order to improve the readability of the graphs, all data points associated to the same version are connected and the identity is also shown as a gray line. For a better comparability of the different graphs among each other, each version has its own unique color, line type and point type.

6.4 Evaluation

Matrix Multiplication The first experiment is a matrix multiplication which computes $\mathtt{C} = \mathtt{A} * \mathtt{B}$ as shown in Figure 6.2. This code uses a primitive variable to reduce the number of array accesses, in order to enhance the performance of the computation if the compiler is not able to optimize this by itself. This also demonstrates, that the generator is able to deal with primitive, non-array variables.

Two different optimizations are performed here:

- On the one hand, both outer loops of the matrix multiplication do not carry any dependency (except the anti-dependency for the scalar variable `sum`, but this must not be considered here, as explained in Section 5.2), which enables the generated divider to split the iteration domain two-dimensionally. Therefore, the DKU pattern implicitly performs a kind of loop tiling, so the number of subpieces created and computed is increased by the factor 16. That means each OpenMP thread computes 16 smaller subpieces instead of one larger piece.

- On the other hand, as the input code accesses the two-dimensional array `B` row-wise, the `bundleInputs` method transposes this matrix in order to enhance the memory accesses, as explained in Section 5.3.

The speedups of the DKU versions compared to the sequential code are shown in Figure 6.5. First of all, as the *OpenMP local* version does not take advantage of the layout transformation, it performs quite bad compared to the others. However this version also shows a super-linear speedup, which is caused by the implicit tiling of the two-dimensional distribution. If the layout optimization is also performed, as for the *OpenMP bundle* version, the performance dramatically increases. The overhead caused by rearranging the memory layout can be neglected here. Both *MPI* and *OpenMP + MPI* versions benefit from the layout transformation, but compared to the *OpenMP bundle* version, the input data must be copied multiple times, as multiple different nodes require the same part of the data, which leads to a slightly worse speedup.

Polynomial Product The second experiment, the polynomial product, computes a vector $\vec{c}$, such that $\vec{c}\vec{x}_{n+m} = (\vec{a}\vec{x}_n) \cdot (\vec{b}\vec{x}_m)$, where $\vec{x}_i = (x^0, x^1, \ldots, x^i)^\top$. The input C code is shown in Figure 6.3. It also demonstrates that the generator is able to work with more complex loop bounds, as long as they are defined as a union or an intersection of affine inequations.

In this experiment there is neither a layout transformation possible, as all three arrays are accessed linearly, nor is any kind of implicit tiling performed, as only the outer loop is divided.

The speedups compared to the serial version are shown in Figure 6.6. Except for the *MPI* version, the performance increases linearly and it almost reaches a perfect speedup, as depicted by the gray line. The *MPI*-only simulation is limited to a speedup of about 14 to 15, due to the overhead for copying large parts of the input data for every node.

N-Body Simulation The last of the non-PolyBench examples is one step of a two dimensional n-body simulation, shown in Figure 6.4, without algorithmic opti-

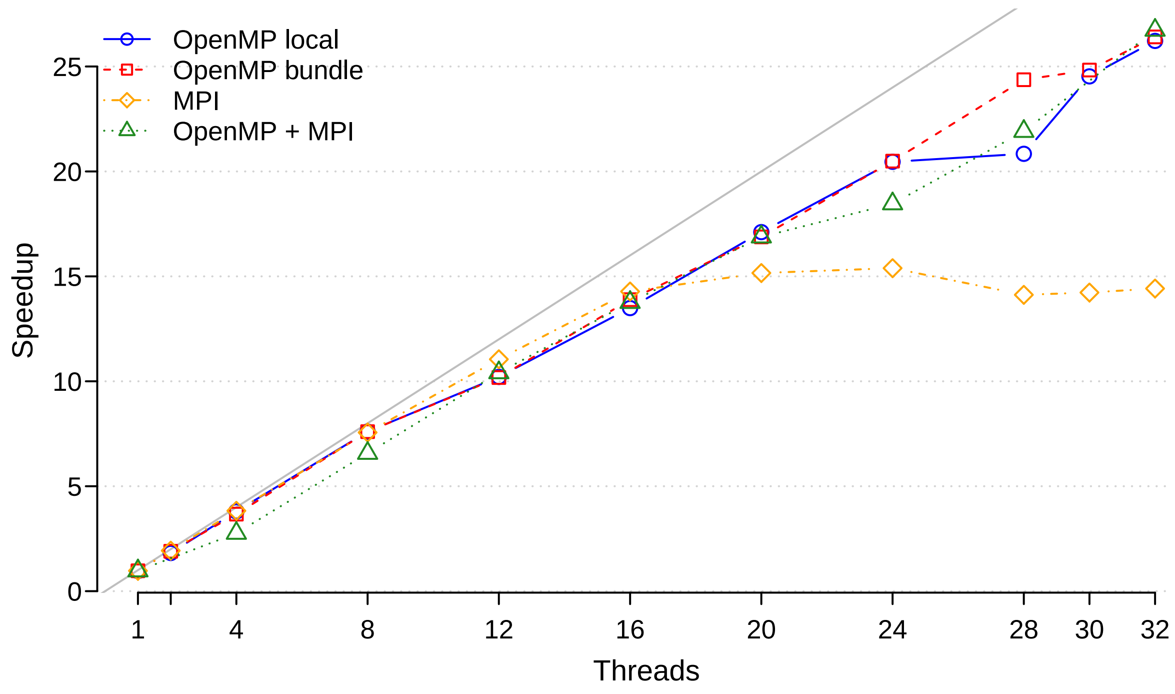


Figure 6.6: Polynomial Product. 1 parallel dimension.

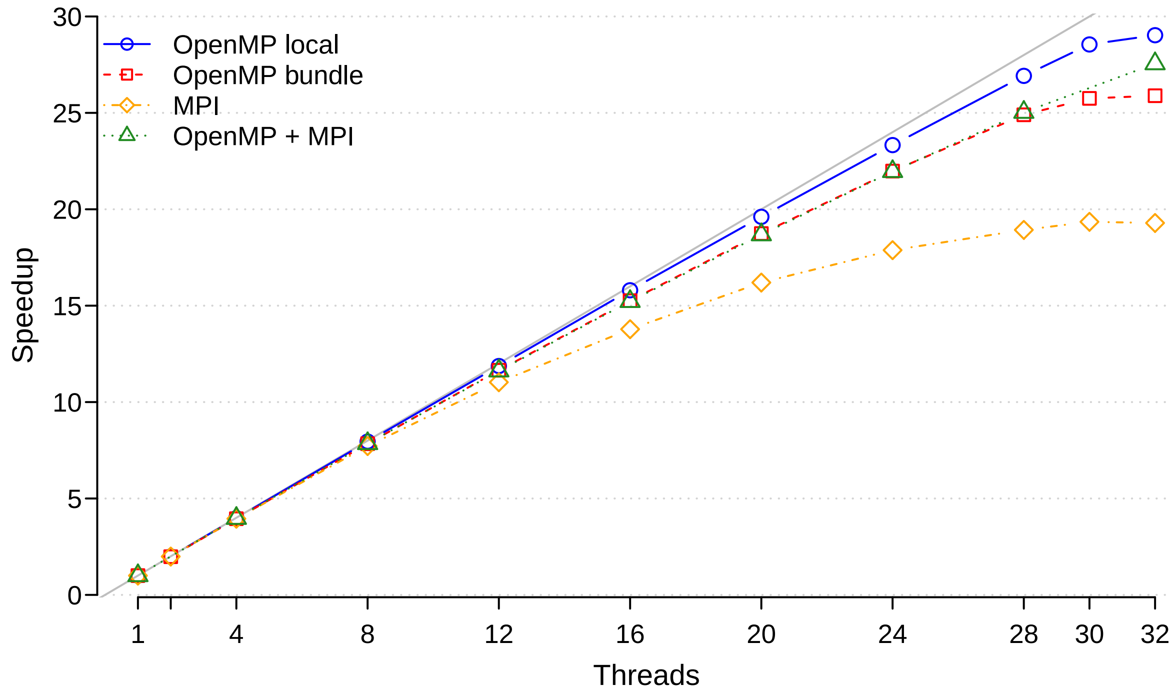


Figure 6.7: N-body simulation. 1 parallel dimension.

mizations, as e.g. using Newton's third law. The DKU scheduler is executed multiple times in a loop to test if it is possible to use the DKU pattern also for synchronous parallelism, which requires a barrier and communication. In the framework there is an implicit synchronization after each invocation of the scheduler, as the framework guarantees all computed data is aggregated on the master node when the method returns. But in most cases, this also leads to a much stricter synchronization as required, because the computed result is collected on one node after each execution and all other nodes loose their data, which must then be copied again for the next step.

For this test, there is also neither a layout transformation, nor an implicit tiling performed.

The results for the n-body simulation are shown in Figure 6.7. The computation for this test is much less data intense then for example for the polynomial product. The input data consists of 4010 planets, where each needs only five double values, compared to the two double arrays with 100.000 elements each for the polynomial product. Therefore, the communication overhead is quite low and an almost perfect linear speedup is possible for all except the *MPI* version, even if the scheduler is invoked 50 times subsequently. For the *MPI*-only runs, each node requires at least the position and the mass of all bodies, so the communication overhead increases linearly with the number of nodes, which limits the maximum speedup here.

PolyBench/C linear-algebra/kernels/2mm In the *2mm* benchmark, two matrix multiplications are performed subsequently. Therefore, in contrast to a single multiplication only one asynchronous parallel loop remains, which restricts the `divider` to split more than one dimension. For the generated kernel code, transposing one of the three input matrices enhences the overall memory layout and therefore leads to a better performance.

The speedup values for this experiment are shown in Figure 6.8. The results show that the memory bandwidth is the limiting factor here. Both *OpenMP* versions accesses the same memory locations in the main memory and as the used hardware is a NUMA architecture, they even share the same link to the RAM. The *OpenMP bundle* version benefits from the memory layout optimization for up to 8 threads here, but when starting more, the overhead for rearranging the layout worsens the maximum performance more than for dealing with a row-wise access instead of a column-wise. For the *MPI* version, the maximum speedup can be achieved by using 16 nodes, which corresponds to the used hardware. The machine consists of

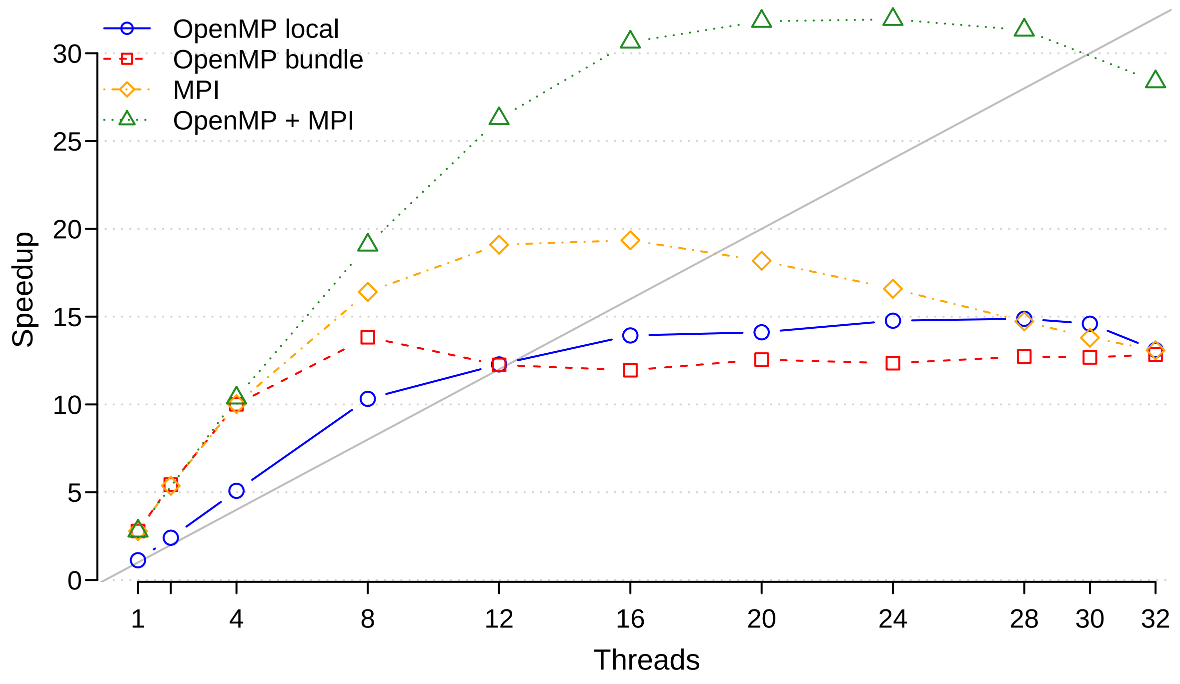


Figure 6.8: PolyBench/C linear-algebra/kernels/2mm. 1 parallel dimension.

eight sockets, each connected with its own memory through a dual channel link. Therefore, the complete computer has 16 channel. For each additional node, the communication amount increases, but the memory bandwidth is already at its limit, which results in a degradation of the performance. The *OpenMP + MPI* simulation can take advantage of both factors. First, the memory layout is optimized, which enables a super-linear speedup and second, the restriction of each MPI node and the corresponding OpenMP threads to a single socket ensures each thread only accesses the main memory according to the socket it is executed on. That means, the available bandwidth is utilized as good as possible, while the communication is reduced to a minimum.

PolyBench/C linear-algebra/kernels/bicg Another PolyBench benchmark tested is the *bicg* kernel. It contains only one asynchronous parallel dimension after transforming the iteration domain and there is no layout transformation possible.

The results of the experiments are shown in Figure 6.9. In order to understand these extremly bad speedups, consider the original and the parallel version of the *bicg* kernel, which are depicted in Figure 6.10. The original code cannot be parallelized directly, as the inner loop from line 8 to 11 updates all elements of the array `q`, so executing this or the loop starting in line 6 in parallel would cause a

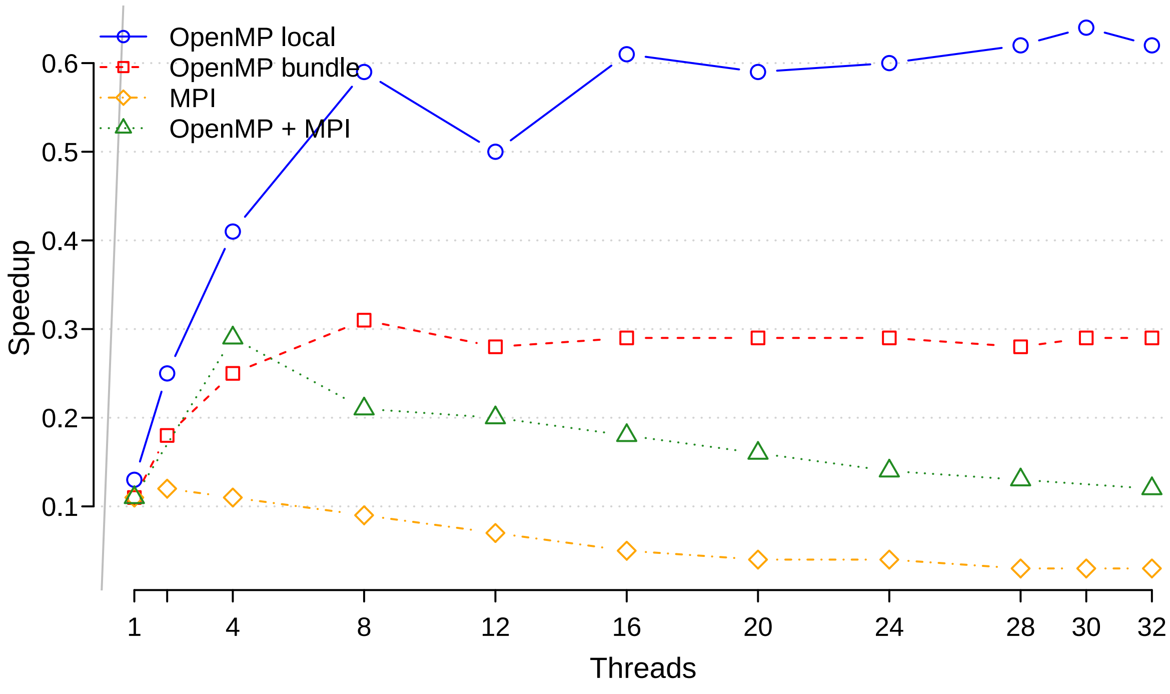


Figure 6.9: PolyBench/C linear-algebra/kernels/bicg. 1 parallel dimension.

```
int i, j;

#pragma scop
for (i = 0; i < ny; i++)
  s[i] = 0;
for (i = 0; i < nx; i++) {
  q[i] = 0;
  for (j = 0; j < ny; j++) {
    s[j] = s[j] + r[i] * A[i][j];
    q[i] = q[i] + A[i][j] * p[j];
  }
}
#pragma endscop
```

```
int i, j;

for (i = 0; i < nx; i++) {
  q[i] = 0;
  for (j = 0; j < ny; j++)
    q[i] = q[i] + A[i][j] * p[j];
}

for (i = 0; i < ny; i++) {
  s[i] = 0;
  for (j = 0; j < nx; j++)
    s[i] = s[i] + r[j] * A[j][i];
}
```

Figure 6.10: Original (left) and parallel (right) version of PolyBench/C linear-algebra/kernels/bicg.

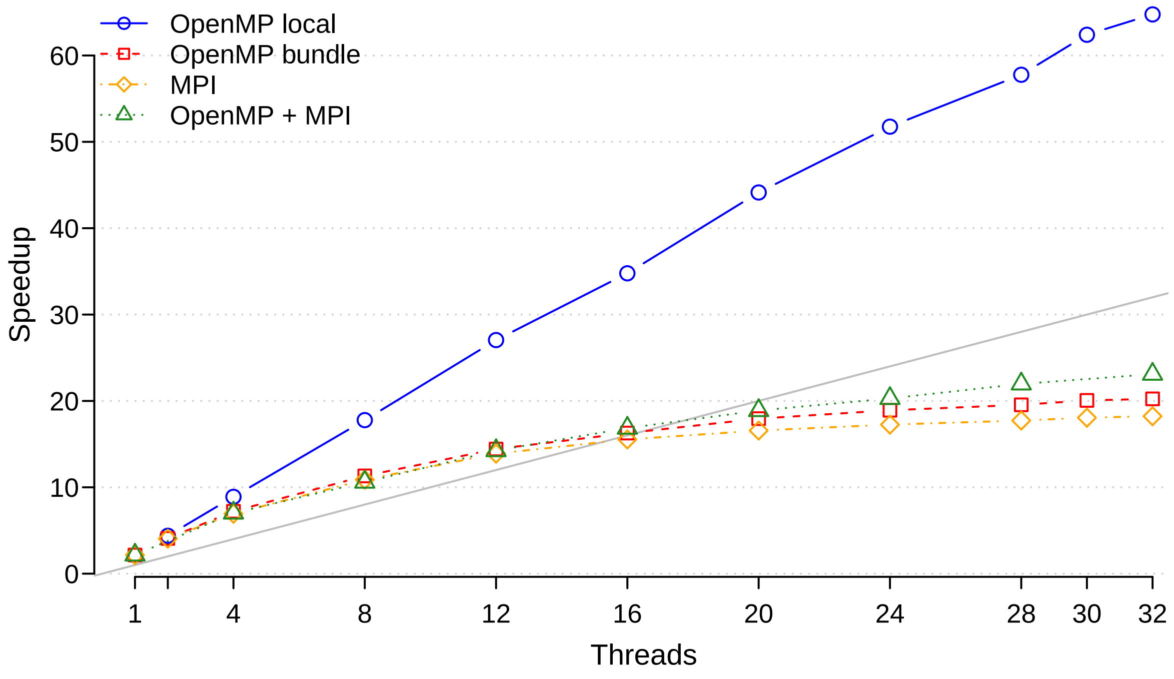


Figure 6.11: PolyBench/C linear-algebra/kernels/doitgen. 2 parallel dimensions, 16 subpieces per thread.

shared-update problem. But as both arrays `q` and `s` can be computed completely independent, the second version depicted in Figure 6.10 can be executed in parallel, or split for the DKU framework. But as for the computation of `s` both surrounding loops are interchanged to allow a parallel execution, the original column-wise access to the array `A` is transformed to a slower row-wise access. Transposing the two-dimensional array is not possible, as the computation of `q` still accesses it column-wise. The DKU version also needs to traverse `A` two times, which increases the total number of accesses to the main memory. Finally, both disadvantages significantly reduce the performance to a level beneath the unmodified sequential version.

PolyBench/C linear-algebra/kernels/doitgen The *doitgen* benchmark is a linear-algebra kernel included in the PolyBench/C benchmark suite. It offers two asynchronous parallel dimensions, which allows tiling, that can be controlled by generating more than one subpiece per thread. It also allows optimizations for the memory layout, but both effects are very limited and they do not influence the performance in a noticable way for a larger number of threads.

The speedup values for this experiment are shown in Figure 6.11. The innermost loop of the kernel accesses three arrays with only 260 subsequent double values.

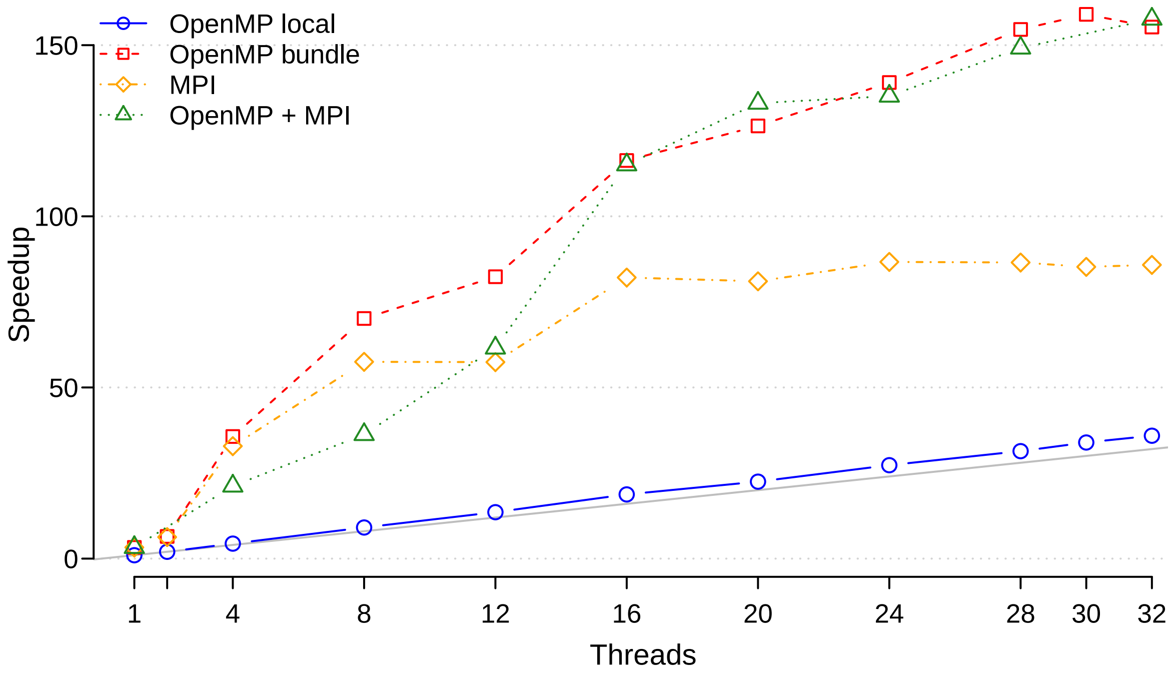


Figure 6.12: PolyBench/C linear-algebra/kernels/gemm. 2 parallel dimensions, 16 subpieces per thread.

Therefore, the computation is able to exploit the cache very well. This leads to the good speedup for the *OpenMP local* version, which reaches a speedup of 64 for 32 threads, resulting in an efficiency of 200%. All other versions need to copy all input and result data, which are about 270 MB, so the execution time converges to the time needed to distribute all data. As the *OpenMP + MPI* version utilizes the hardware best, so that all threads accesses the local part of the memory, this version dominates *MPI* and *OpenMP bundle* here.

PolyBench/C linear-algebra/kernels/gemm The PolyBench/C benchmark suite also contains a matrix multiplication version. In this case, the *gemm* is included, which performs the computation $\mathtt{C} = \alpha \cdot \mathtt{A} * \mathtt{B} + \beta \cdot \mathtt{C}$, where $\mathtt{A}$, $\mathtt{B}$ and $\mathtt{C}$ are matrices and α and β are scalars. This computation can be parallelized very efficiently, as each element of the result matrix $\mathtt{C}$ can be computed independently, which leads to two asynchronous parallel dimensions. This also implies, that the DKU framework performs an implicit tiling and therefore for this experiment 16 subpieces are created for each thread. It is also possible to further optimize the computation by transposing the input matrix $\mathtt{B}$, which leads to a column-wise access.

Consider the results of this benchmark from Figure 6.12, the implicit tiling allows a perfect speedup for the *OpenMP local* version with an efficiency of about 110%. Creating only one subpiece for each thread would have lead to an efficiency of only 50%. If the memory layout transformation is performed, the performance increases dramatically. Even the *MPI* only version, which has a quite high overhead due to the communication and the node management performs good here, but its speedup does not increase for more than 16 nodes. This corresponds to the number of channels the used hardware utilizes when accessing the main memory. Remember that each MPI node accesses its own private copy of the input data, so the accesses of different nodes running on a single socket must be cached separately, even if the same data of the input is needed. Both *OpenMP bundle* and *OpenMP + MPI* are not affected, they are able to use the cache more efficient, which leads to a better performance.

In theory, the four thread runs of the experiments *OpenMP bundle* and *OpenMP + MPI* should be identical, as both uses one MPI node and four OpenMP threads, but the first versions spreads the threads to four different CPUs, whereas the latter binds all to a single processor. A more detailed explanation of this is given in Section 6.2. A similar effect can be seen for two MPI nodes and four OpenMP threads each for the *OpenMP + MPI* version, versus one MPI node and eight OpenMP threads of the *OpenMP bundle* run.

PolyBench/C linear-algebra/kernels/gesummv The *gesummv* kernel computes $\vec{y} = \alpha \cdot A * \vec{x} + \beta \cdot B * \vec{x}$, where A and B denote two matrices, $\vec{x}$ and $\vec{y}$ are vectors and α and β denote scalars. Therefore, the computation is quite simple and it is mainly limited by the main memory. It also consists of only a single asynchronous parallel loop, therefore no implicit tiling effect occurs. As a matrix-vector multiplication accesses all memory linearly and column-wise, the memory layout is already optimal.

The results of this benchmark are shown in Figure 6.13. In order to get a suitable runtime for the computation, the sizes of both matrices and the vector must be high. Both matrices have 8000 times 8000 elements and therefore the vector has also 8000 entries. This leads a memory consumption of almost 980 MB, whose transfer times exceed the computation time. Therefore, all three experiments, which require to copy all input data perform extremly worse here. Only the *OpenMP local* version, which uses the input data is able to speed up the computation. But this speedup

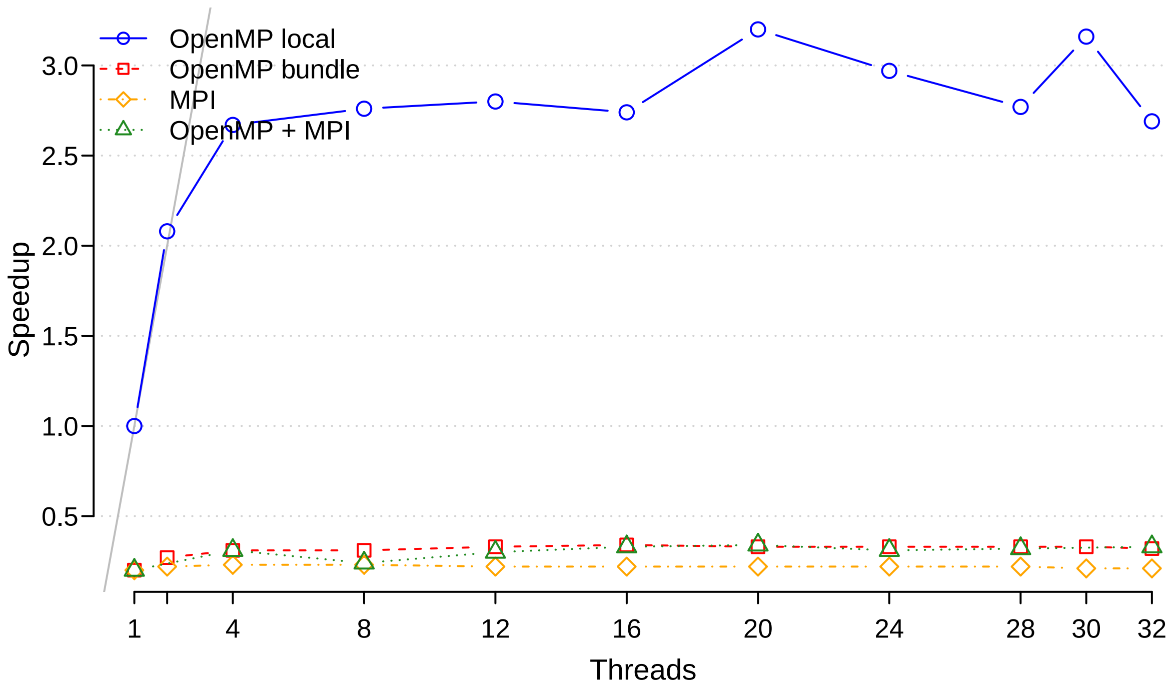


Figure 6.13: PolyBench/C linear-algebra/kernels/gesummv. 1 parallel dimension.

is limited to a factor of about 2.7 to 3.2, because the link to the main memory is exhausted here.

PolyBench/C linear-algebra/kernels/syr2k Another benchmark tested here is the *syr2k* kernel. It computes $C = \alpha \cdot A * B^\top + \alpha \cdot B * A^\top + \beta \cdot C$, where A, B and C denote matrices, whereas α and β are scalars. As both matrix multiplications access the second input transposed, the memory layout can not be optimized any further. The kernel code consists of two asynchronous parallel dimensions, so an implicit tiling is possible here and therefore the number generated subpieces per thread was set to 16.

The benchmark results are visualized in Figure 6.14. The *OpenMP local* version performs best here, as there is no memory layout optimization possible and the input is small enough to benefit from the large level 3 cache. The efficiency of this version ranges from 170% to 240%. All other versions result in a lower speedup, as the communication overhead for copying all input data reduces the overall performance. In case of the *MPI* only version, the worst variant for this benchmark, each used thread increases the amount of communication needed to distribute all data. Therefore, the maximum possible speedup here is limited to about 15, which is still a good value.

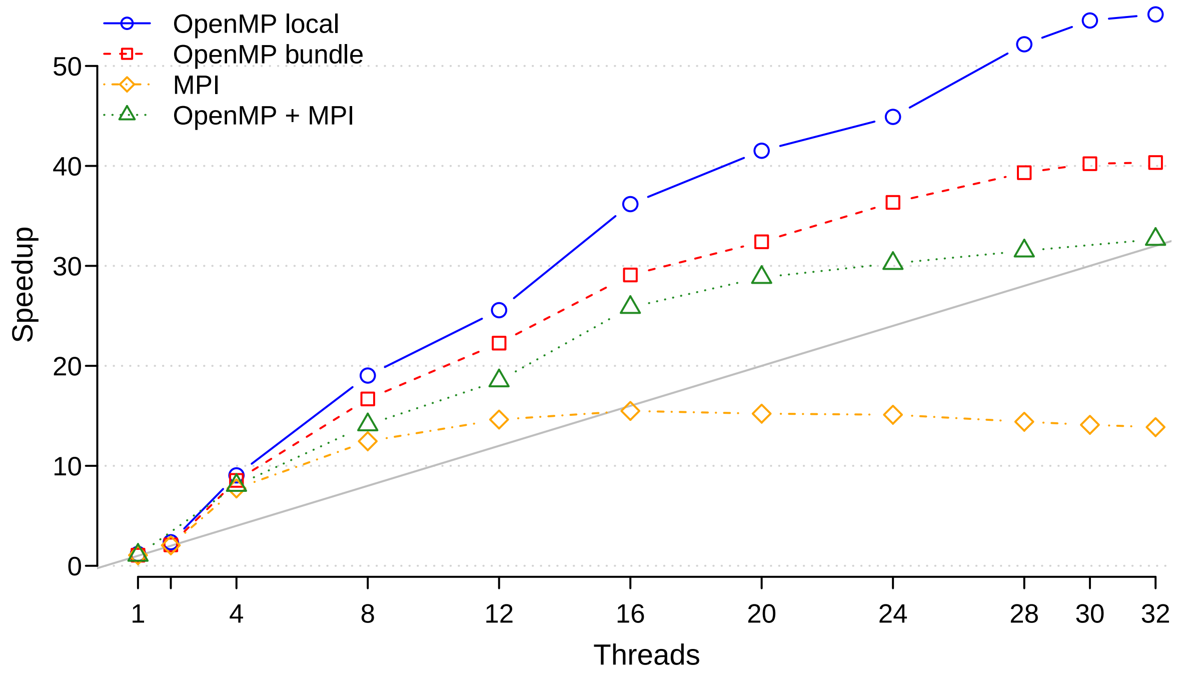


Figure 6.14: PolyBench/C linear-algebra/kernels/syr2k. 2 parallel dimensions, 16 subpieces per thread.

PolyBench/C linear-algebra/kernels/syrk The linear-algebra kernel *syrk* from the PolyBench/C benchmark suite is quite similar to the previous described *syr2k*. This one computes $C = \alpha \cdot A * A^\top + \beta \cdot C$, so it can be seen as a specialization of *syr2k*. The general properties of this code are the same, so there is also no reasonable layout transformation and an implicit tiling is performed by creating 16 times more subpieces than threads are started.

The speedups for this experiment are shown in Figure 6.15. The general evolution of the results is similar to the previous experiment, but as only two matrices are needed, the overall speedup is higher. So for the *OpenMP local* run, a maximum speedup of 93 can be achieved. This corresponds to an efficiency of 310%. The performance degradation for this version with 32 threads cannot be explained satisfactorily. Some tests showed, its effect for smaller input sizes is much higher and it disappears for a larger dataset.

PolyBench/C stencils/fdtd-apml The last benchmark from the suite the generator can currently deal with is the stencil *fdtd-apml*. This code performs only a single step of the stencil and for this reason, the loop nest starts with a single asynchronous parallel loop. That means, no implicit tiling is performed. The code is also orga-

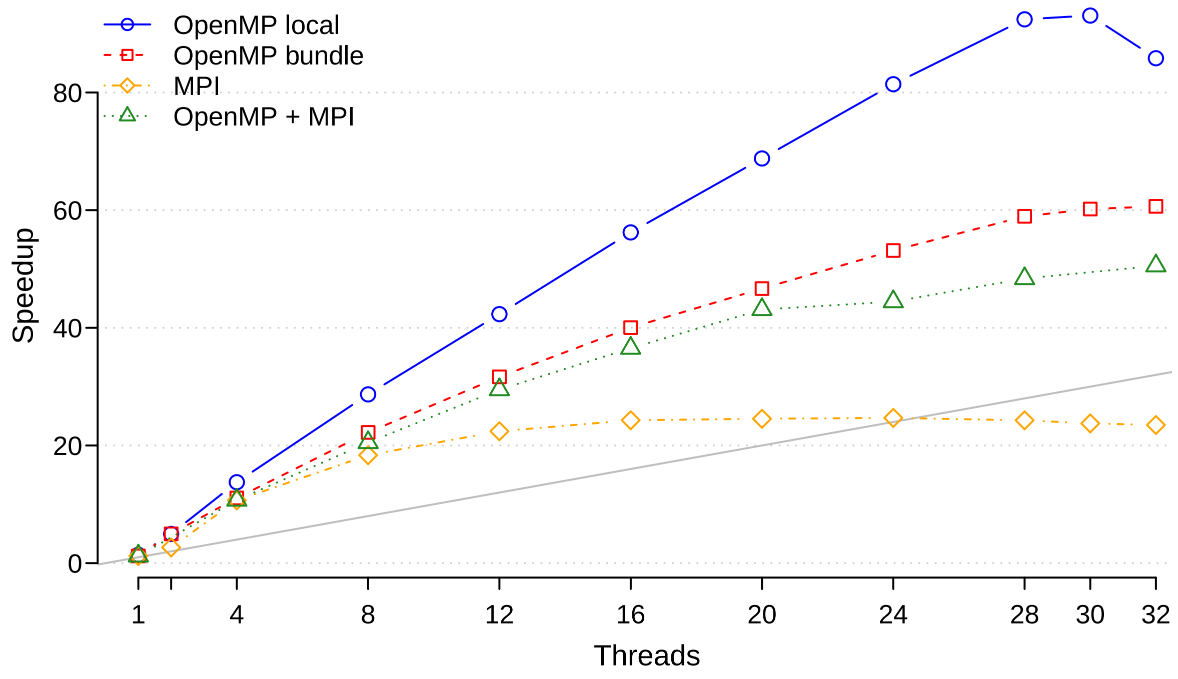


Figure 6.15: PolyBench/C linear-algebra/kernels/syrk. 2 parallel dimensions, 16 subpieces per thread.

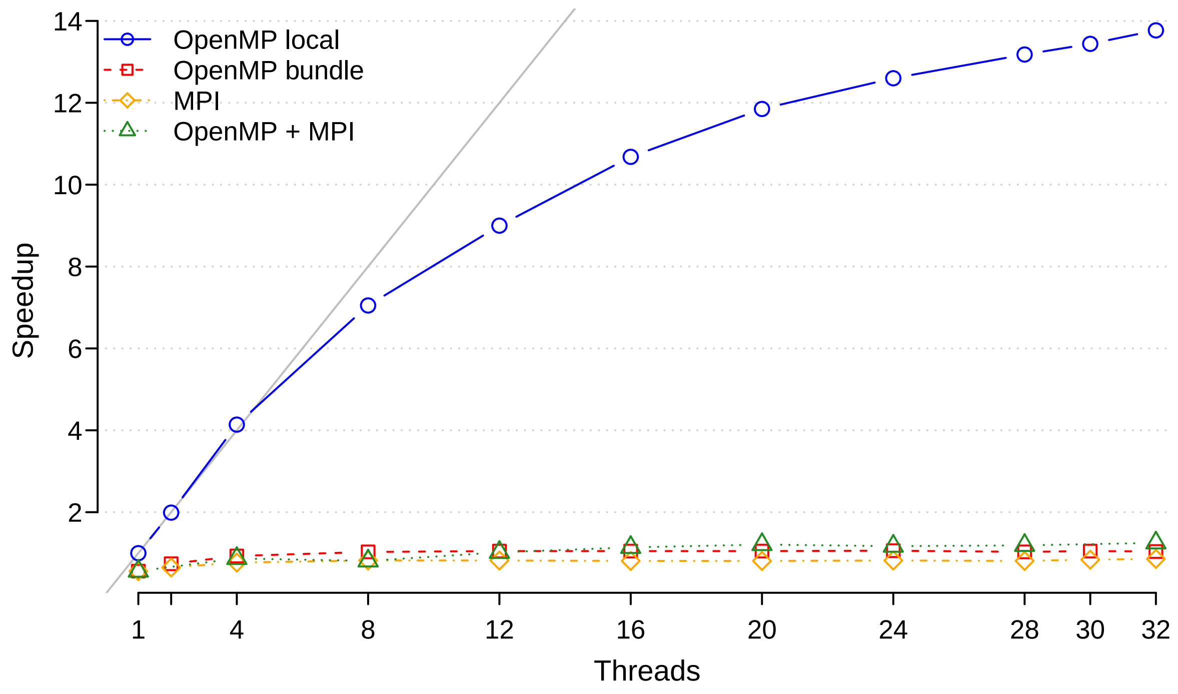


Figure 6.16: PolyBench/C stencils/fdtd-apml. 1 parallel dimension.

nized in a way, such that all arrays are accessed column-wise and due to this, no suitable layout transformation is possible.

The performance measurements for this stencil are shown in Figure 6.16. As the kernel code works with four three-dimensional arrays of size 401^3 each, its memory consumption of almost 2 GB is quite high. Due to the overhead for copying all input data at least once, the execution times for all three *OpenMP bundle*, *MPI* and *OpenMP + MPI* versions converge to the time required to distribute all data. In contrast to this, the *OpenMP local* version need not copy anything, which leads to better performance. But a linear speedup cannot be achieved here, as the memory bandwidth is limiting for a larger number of threads.

6.5 Comparison with Pluto

In this section, the generated DKU code is compared with an optimized version created by Pluto [BHRS08]. Pluto is an automatic parallelization tool based on the polyhedron model. Plutos advantage is its ability in automatically tiling the given loop nest to enhance the cache usage. On the one hand, it can process more general codes as the DKU generator, because Pluto is not restricted to asynchronous parallelism. On the other hand, the DKU framework utilizes distributed memory architectures, whereas Pluto is only able to parallelize the loop nest using OpenMP, which is restricted to shared memory.

The usage of Pluto is quite simple. First, embrace the loop nest, that should be optimzed with both `#pragma scop` and `#pragma endscop`, pass the source code to the polycc tool and translate it with suitable C compiler.

By default, Pluto uses a built-in heuristic to determine the tile size, but the user is also allowed to pass explicit tiling information, in order to utilize both level 1 and level 2 caches best. But as the DKU generator does not require any user optimization, the default behaviour of Pluto was used for the following experiments and all previously mentioned PolyBench/C benchmarks were optimized using only the parameters `--tile --parallel` for polycc. That means, a fixed tile size of 32 for all loops was used by the current version of Pluto and as only one single loop is parallelized, the remaining number of iterations is quite low for most experiments. This results in a stagnancy of the speedup if the number of threads is higher than the number of remaining iterations for the parallel loop. The DKU framework is

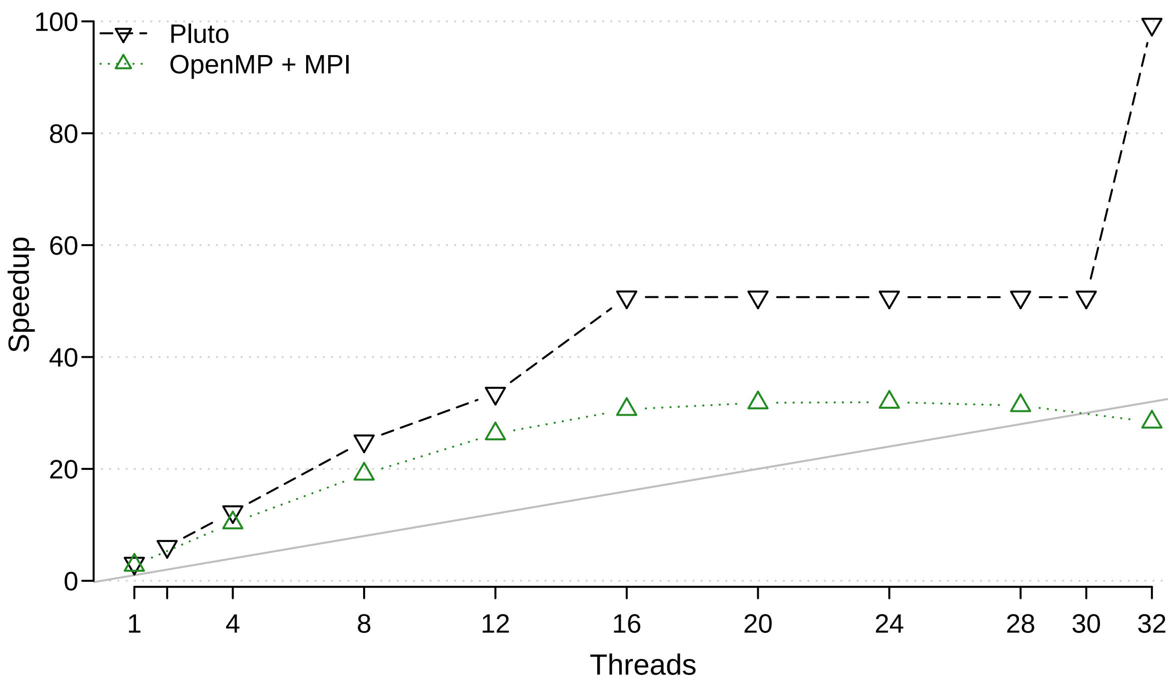


Figure 6.17: PolyBench/C linear-algebra/kernels/2mm. 1 parallel dimension.

not affected by this, as it does not perform explicit tiling and it also uses multiple loops to create exactly the requested number of pieces.

For the following paragraphs, the results of Pluto are compared with at most two versions of DKU. First of all, the best DKU version is shown, as it demonstrates the potentiality of the DKU framework best. In addition to this, the version, which can be chosen by a simple heuristic without knowing the exact baviour of the others is also opposed to Pluto. If the kernel requires much data and the computational effort is comparatively low, the heuristic chooses the *OpenMP local* version, in order to avoid a slowdown due to a high communication overhead. If this does not hold, the *OpenMP + MPI* is chosen, as this one fits best to the actual hardware.

PolyBench/C linear-algebra/kernels/2mm The first benchmark compared with Pluto is the *2mm* kernel. The fastest DKU version for this code uses an *OpenMP + MPI* combination, which is also the preferred version for this code, as the matrix multiplication requires relatively few data, compared to the required computational effort.

The results for the code optimized by Pluto are shown in Figure 6.17. The first interesing point here is the performance stagnancy for 16 to 31 threads. This is caused by the tiling, as all loops are tiled in blocks of 32 iterations. Therefore, the

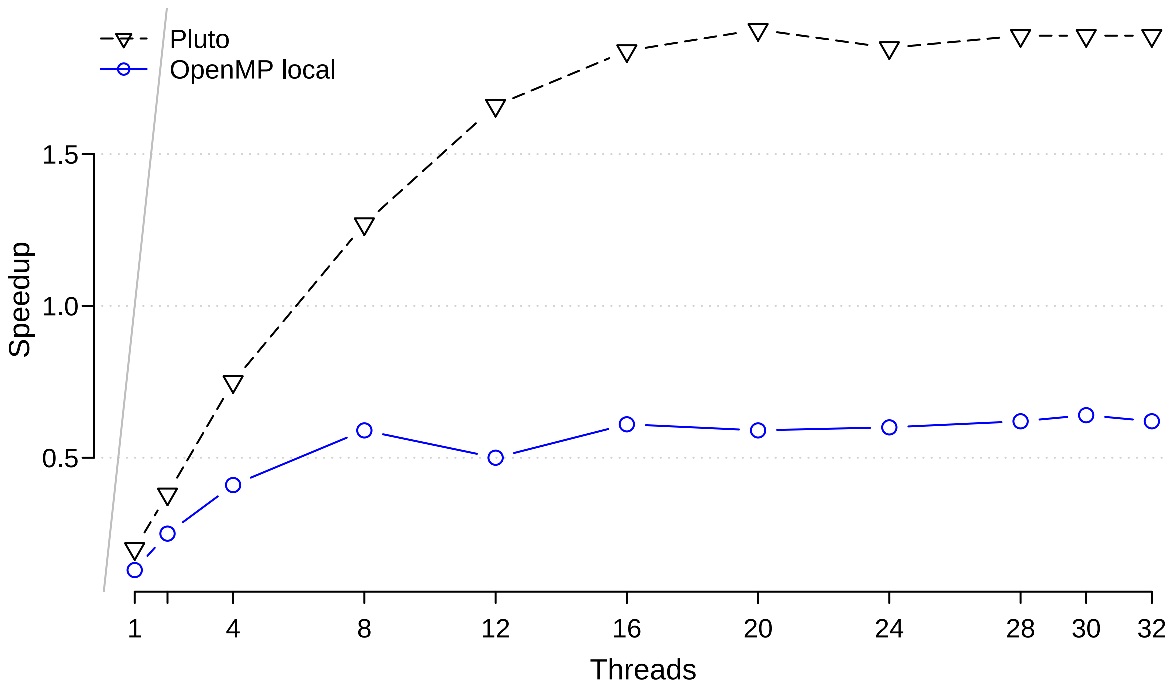


Figure 6.18: PolyBench/C linear-algebra/kernels/bicg. 1 parallel dimension.

outer loop, which is parallelized using OpenMP, iterates over all 32-element blocks and as the original number of iterations were 1000, only 32 iterations remain for the parallelized loop. That means, for each thread number between 16 and 31, at least one single thread has to compute two different blocks, independently of all others and therefore this thread slows the complete execution down.

If the input is large enough, the speedup raises continuously until reaching the maximum speedup for 32 threads. As this code provides only a single asynchronous parallel loop, the DKU framework does not perform an implicit tiling, so Pluto generates faster code here.

PolyBench/C linear-algebra/kernels/bicg The next benchmark used to compare the DKU framework with Pluto is the *bicg* kernel.

Consider Figure 6.9, which shows the results for both the Pluto version and the DKU *OpenMP local* one. As already mentioned in the previous section, due to the structure of the input code, the DKU framework performs bad here. The code optimized by Pluto performs better, but is nevertheless far from a satisfactory speedup, as it runs into the same problems as the DKU generator does.

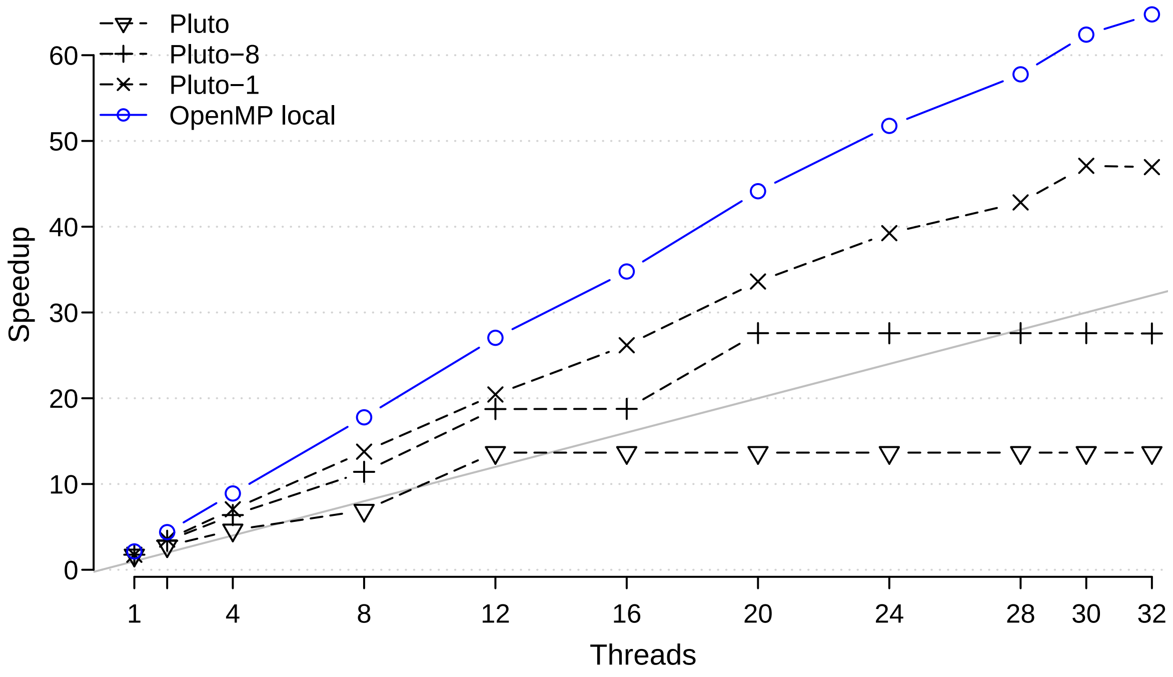


Figure 6.19: PolyBench/C linear-algebra/kernels/doitgen. 2 parallel dimensions, 16 subpieces per thread.

PolyBench/C linear-algebra/kernels/doitgen Another benchmark used to compare DKU and Pluto is the *doitgen* kernel.

As shown in Figure 6.19, the DKU version *OpenMP local* provides good results, and it has an efficiency of about 200%. The stagnancy of the speedup for the Pluto version is caused by a too small number of iterations for the parallelized loop. In this experiment, the original computation consists of three nested loops with 260 iterations each, which results in only $\lceil \frac{260}{32} \rceil = 9$ iterations of the outermost loop, because Pluto always generates tiles of size 32 by default. And as this one is the only loop, which is parallelized, the speedup is limited to the value for using 9 threads. Therefore, the parameters for Pluto were adapated by hand in order to test, how the resulting code behaves for a larger number of threads if it is not limited. The two additional experiments `Pluto-8` and `Pluto-1` were tiled less aggressive, using a tilesize of 8, or 1 respectively for the outermost loop. Both inner loops are treated identically to the default version, which means a tilesize of 32 was used again. `Pluto-1` performs best for this code and it provides also a linear speedup, as the number of iterations executed in parallel is large enough, but it was necessary to manually adapt the optimizations, whereas DKU works automatically. A tile size of 1 practically results in not tiling the associated loop, which may be worse

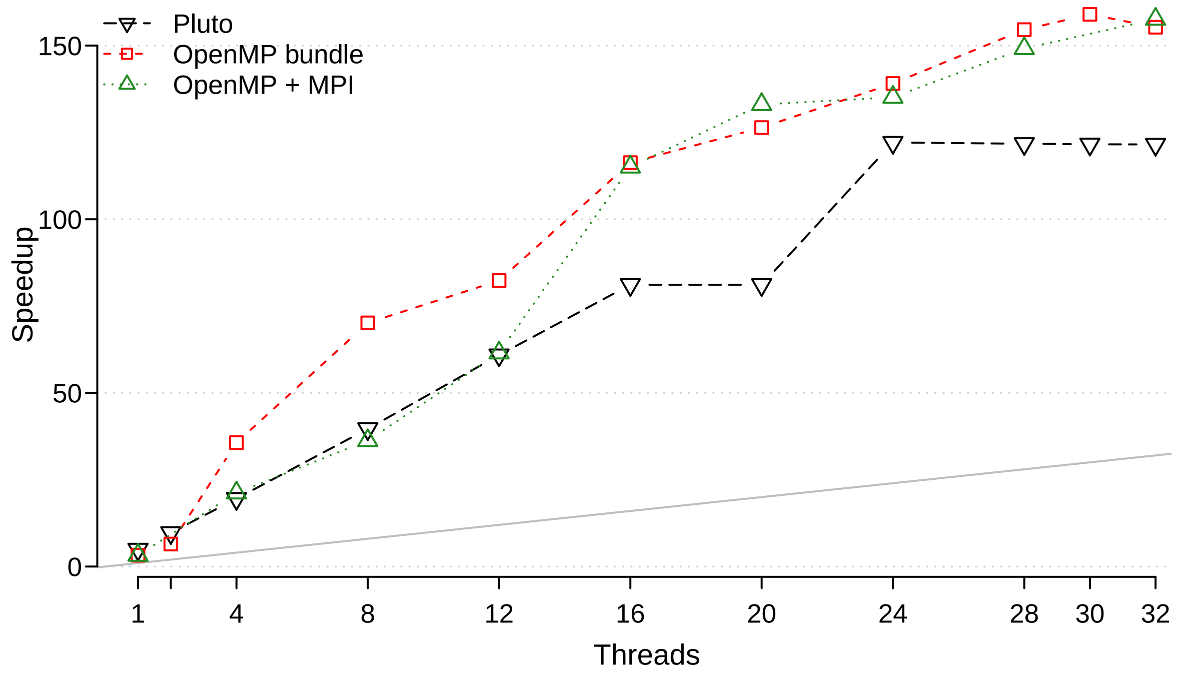


Figure 6.20: PolyBench/C linear-algebra/kernels/gemm. 2 parallel dimensions, 16 subpieces per thread.

for other loop nests. So each application requires its own, individual tiling for an optimal performance.

Combining all information, the best speedup for the default Pluto version is approximately 13.7 for 9 threads used. This leads to an efficiency of 150% - 160%, compared to the roughly 200% for the DKU code.

Note that Pluto is able to perform better for optimized tiling parameters, as inidicated by the results from the `Pluto-1` runs, but this requires tuning the compiler parameters by hand.

Due to the idea of the DKU framework, the implicit tiling performed by splitting the computation in more than one dimension does not result in a statically fixed tile size. That means, the extent of each subpiece depends on the size of the original piece for the whole computation. Therefore, the DKU framework leads to a better work balance in some cases, than a simple tiling into statically fixed chunks is able to.

PolyBench/C linear-algebra/kernels/gemm Figure 6.20 shows the results, when comparing Pluto to DKU using the *gemm* kernel.

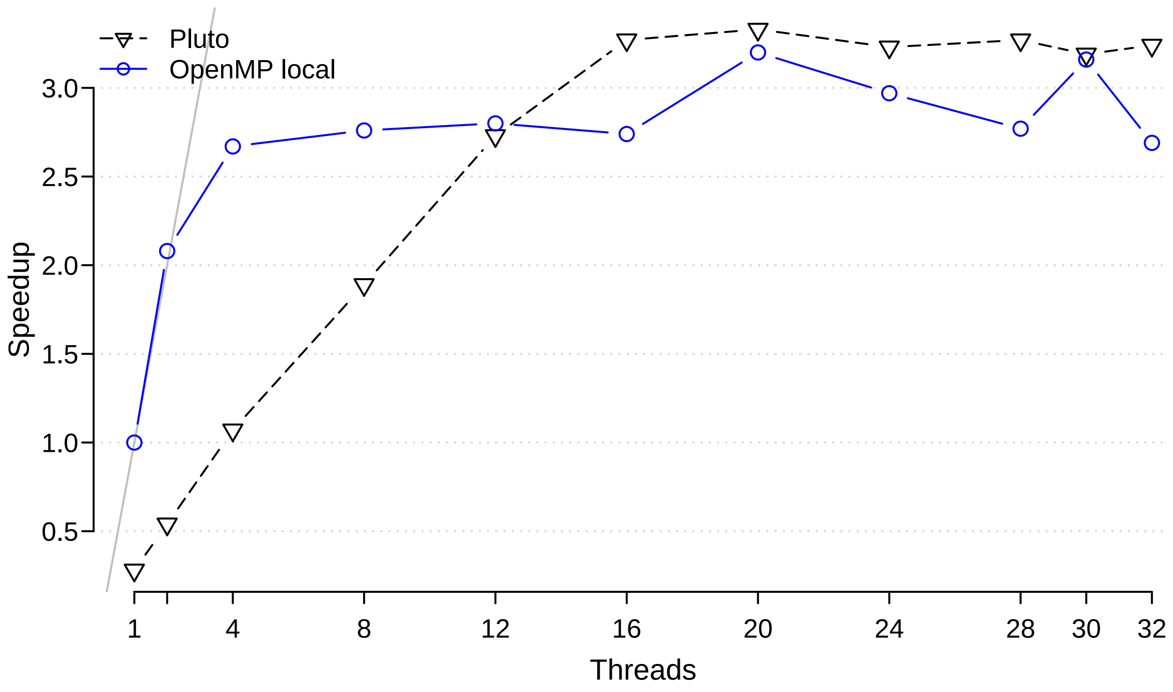


Figure 6.21: PolyBench/C linear-algebra/kernels/gesummv. 1 parallel dimension.

The overall best results for DKU are provided by the *OpenMP bundle* version, but as the combination *OpenMP + MPI* maps best to the used hardware, its speedup values are also shown. For a lower number of threads, the *OpenMP bundle* setup leads to the best results, due to the better utilization of the cache, as explained in Section 6.2. Because of the low number of iterations for the parallel loop, the Pluto optimized versions does not profit from more than 24 threads on the tested hardware. For larger input matrices, the speedup would be linearily increasing until 32 threads.

PolyBench/C linear-algebra/kernels/gesummv The results for the linear algebra kernel *gesummv* from the PolyBench/C benchmark suite are shown in Figure 6.21.

Its kernel code has a quite disadvantageous loop nest, as it requires a comparatively large amount of input data for the actual computation performed. There is also no reuse of the input, so tiling in general has no effect, as the input data is read linearily by the original code. Therefore, the Pluto code performs even worse than DKU.

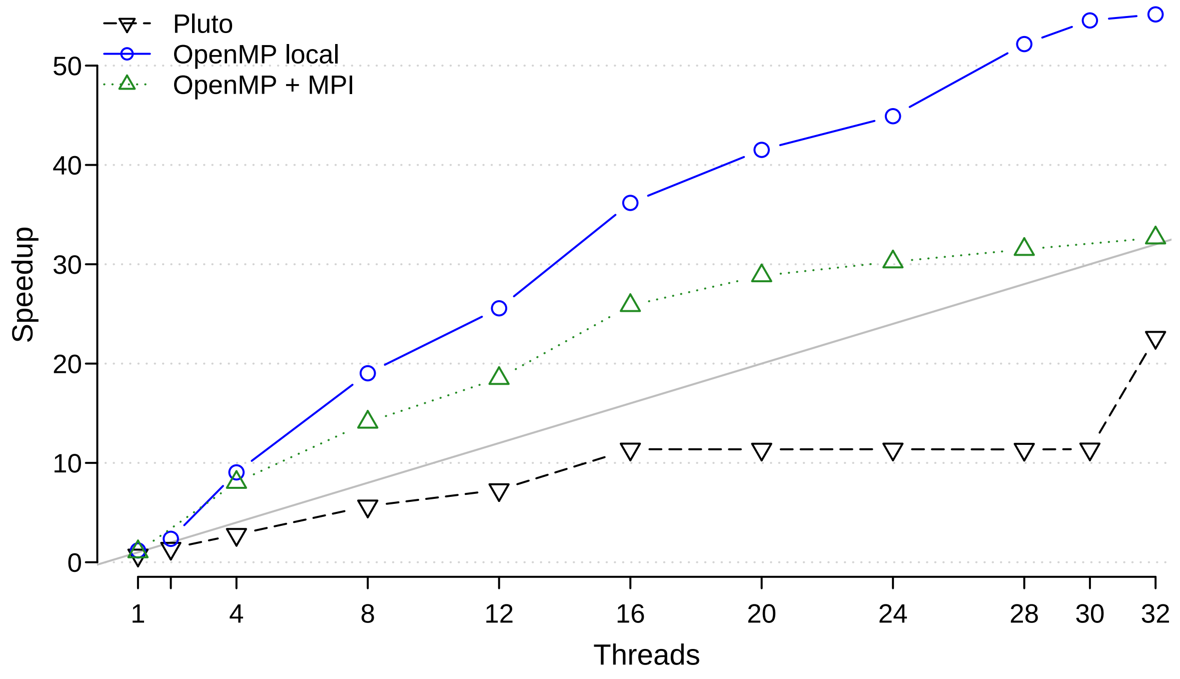


Figure 6.22: PolyBench/C linear-algebra/kernels/syr2k. 2 parallel dimensions, 16 subpieces per thread.

PolyBench/C linear-algebra/kernels/syr2k The results for the next benchmark, the *syr2k* kernel are depicted in Figure 6.22.

The kernel code for this benchmark computes two matrix multiplications. The scheduler used in DKU is able to merge all computations into a single loop nest. Therefore, all accesses to the same elements of the result array are as close as possible. In contrast to this, Pluto splits the whole computation into two subsequent loop nests, which are parallelized independently. This also results in two separate passes through the output array, which leads to a worse performance compared to the DKU versions.

The input size for these experiments is also not large enough to ensure, the work load can be balanced for any number of threads. This results in a stagnancy of the speedup for 16 up to 31 threads. For sufficiently large inputs, the speedup should be more or less linear here.

The *OpenMP local* version performs better than the *OpenMP + MPI* combination here, due to the already optimal memory layout of the unmodified code and the communication overhead for using multiple MPI nodes. But without that knowledge, one would use the latter verson, as it fits better to the hardware. Therefore, the *OpenMP + MPI* version is plotted alongside the *OpenMP local* and Pluto code.

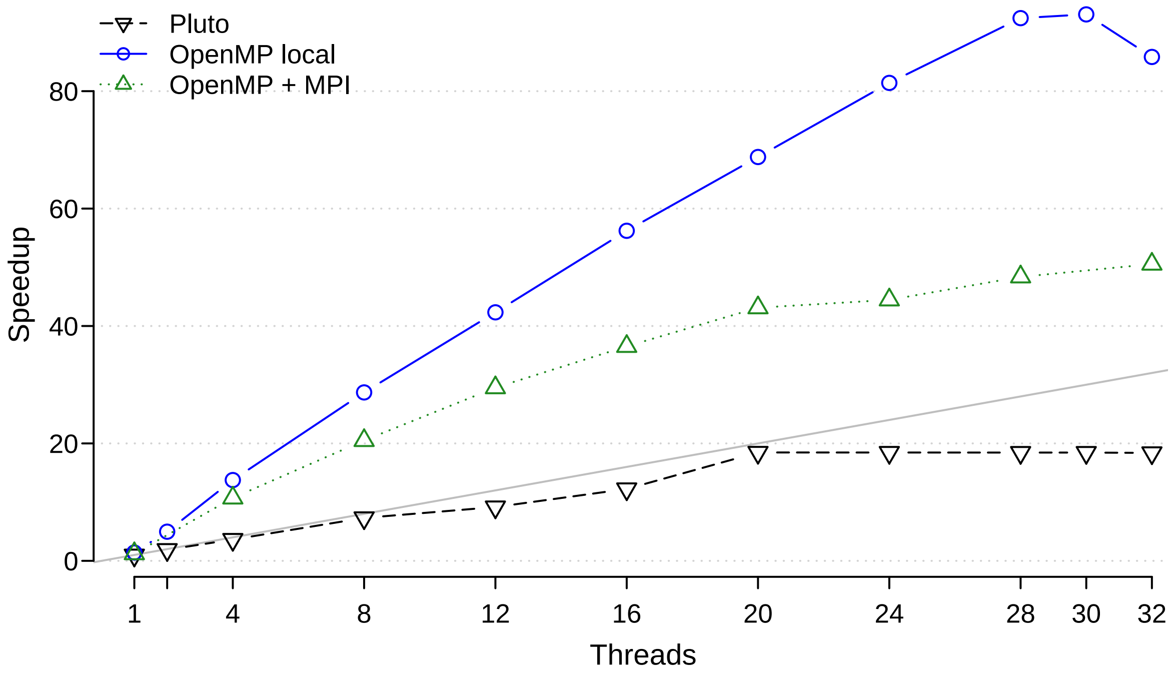


Figure 6.23: PolyBench/C linear-algebra/kernels/syrk. 2 parallel dimensions, 16 subpieces per thread.

PolyBench/C linear-algebra/kernels/syrk The results for a specialized version of the *syr2k* kernel, the *syrk* benchmark are depicted in Figure 6.23.

The general discussion for the results is identical to the one for the previous benchmark. The DKU generator was able to merge all computations into a single loop nest, whereas Pluto creates two parallel loops, which are executed subsequently.

The stagnancy for more than 20 threads can also be explained by the tiling and the resulting low number of iterations for the outermoust loop, which is parallelized using OpenMP.

PolyBench/C stencils/fdtd-apml The results for the last benchmark using the *fdtd-apml* stencil are shown in Figure 6.24.

As the corresponding code computes a three dimensional stencil, the extent of the outermost loop is relatively small. Regardless of that, Pluto tiles this dimension into chunks of 32 iterations, before parallelizing the code. That means, the outermost loop, which was annoted with `#pragma omp parallel for` consists of only few iterations, which enumerates the creates tiles. Therefore, the benefit

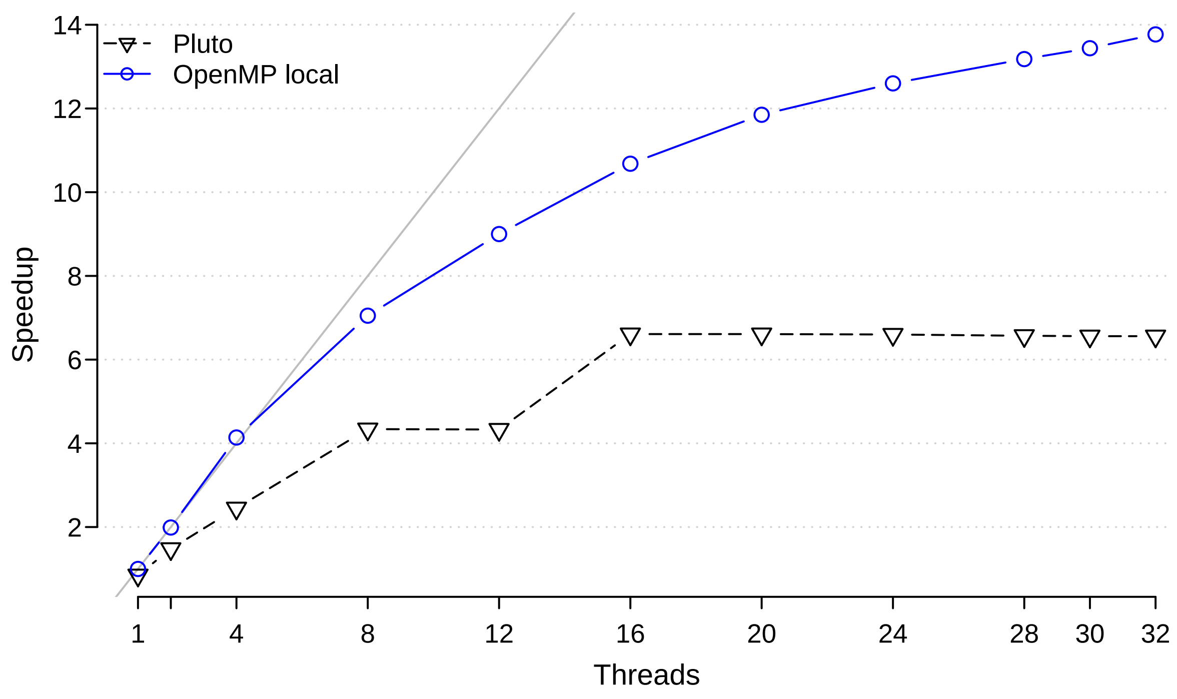


Figure 6.24: PolyBench/C stencils/fdtd-apml. 1 parallel dimension.

from parallelizing this loop is very limited, as the performance does not increase for more than 16 threads.

In addition to this, Pluto also splits the single loop nest from the original code into four different, subsequent loop nests and parallelizes all of them separately. The DKU generator conserves the single loop nest, which results in an improved data locality and a better speedup compared to Plutos version.

7 Conclusion

Summary Multicore systems are widely spreaded today, almost every available personal computer and even most smartphones are equipped with at least a dual-core processor. The wide range of different parallel systems require more advanced tools to parallelize applications and to enhance the portability to a new system.

One technique to deal with this, is to introduce a new layer between the hardware and the application code, which encapsulates all parallelization related code and all further target optimizations. Following this approach, all application codes are architecture independent, but the programmer also need to figure out a way, to adapt the source code for the framework. But if the application can be represented in the polyhedron model, it is possible to extract all required information from the abstract description and generate the source code using the framework automatically. For this purpose, the DKU generator was designed and implemented.

The DKU framework in combination with the generator introduced in this work is a suitable method to automatically parallelize programms, fitting in the polyhedron model, for both distributed and shared memory architectures.

In its current state, the generator requires input code, or a polyhedral description, that can be transformed in a way, such that at least the outermost loop can be executed in parallel. If this holds, the generator is able to create architecture independent code, which can be executed in parallel, if the DKU framework is available and adapted to the hardware. That means, the actual work to adapt the program to a specific hardware is transferred from the application to a more general framework.

As described in Chapter 6, this approach can compete with other tools, such as Pluto, as long as the tile size is not optimzed by hand, but the generator has more restrictions for the input. If the application code can be split in more than one dimension, the framework also performs an implicit tiling. But in contrast to classical tiling methods, the tile size depends on the size of the input and the requested number of pieces. Therefore, it is computed dynamically and not statically fixed, which allows creating as many tiles as required, that also define almost equally sized computations, even if the iteration domain is not rectangular. And as the DKU framework divides the computation in more than one dimension, if possible, the computational effort can be balanced very efficiently. In contrast to this, Pluto parallelizes only the outermost parallel loop. In conjunction with the tiling, this may lead to only few iterations which can be spreaded to the available cores.

Future Work Although the presented results are quite good, the generator and the used version of the DKU framework suffer from some limitations.

First of all, the current version of the generator is not able to automatically compute sufficient layout transformations for the input data. It requires the user to find an optimization here, and pass it to the generator. And as the loop nest may be transformed during the generation, the access pattern of the original input code may also change. So for the current version, the user needs to create the kernel code for the framework once without any layout optimizations, inspect it manually and extract an optimal layout. If the original layout should be changed, the programmer has to formulate a transformation in isl syntax and pass it to a second call of the generator.

Another useful feature would be to support not only conventional architectures, but also graphics processing units for example. As modern GPUs are massively parallel computing units, a single GeForce GTX Titan for example can provide up to 4.5 TFLOPS, while consuming only 250 Watts. But adding support for GPGPU is not as easy as it sounds first. The kernel code for GPUs must be structured in a different way, then for CPUs, as, e.g. the former require cyclic access to the memory, whereas the latter prefer block-wise accesses. Therefore also another kernel method must be provided and the framework must be able in setting up and managing the graphics processing unit. In addition to this, it may be sufficient, if multiple GPUs, or also both CPU and GPU work on the same computation in order to reduce the execution time even further.

A third, major improvement would be to remove the limitation to asynchronous parallelism. In order to implement this, the framework requires more constructs to define synchronization points and to allow communication between different nodes. Currently it is possible to invoke the DKU scheduler multiple times, as already tested for the n-body simulation explained in Chapter 6. But this apporach leads to a much higher synchronization overhead than required, as for each invocation of the scheduler all nodes start from scratch and therefore all input data must be transferred again, no matter if it was modified or not.

Bibliography

[Bas04] Cédric Bastoul. *Code Generation in the Polyhedral Model Is Easier Than You Think*. IEEE PACT, pages 7–16. IEEE Computer Society, 2004.

[Ber66] A. J. Bernstein. *Analysis of Programs for Parallel Processing*. IEEE Transactions on Electronic Computers, volume EC-15, number 5, pages 757 –763. Oct 1966.

[BHRS08] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. *A practical automatic polyhedral parallelizer and locality optimizer*. PLDI, pages 101–113. ACM, 2008.

[Fea91] Paul Feautrier. *Dataflow analysis of array and scalar references*. International Journal of Parallel Programming, volume 20, number 1, pages 23–53. 1991.

[Fea92] Paul Feautrier. *Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time*. International Journal of Parallel Programming, volume 21, number 6, pages 389–420. 1992.

[FL11] Paul Feautrier and Christian Lengauer. *Polyhedron Model*. Encyclopedia of Parallel Computing, pages 1581–1592. Springer, 2011.

[Gt12] Torbjörn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 5.0.2 edition, 2012. `http://gmplib.org/`.

[HC09] Sean Halle and Albert Cohen. *The DKU Pattern for Performance Portable Parallel Programming*. Technical Report UCSC-SOE-09-06, University of California Santa Cruz, Dec 2009.

[Len93] Christian Lengauer. *Loop Parallelization in the Polytope Model*. CONCUR, volume 715 of *Lecture Notes in Computer Science*, pages 398–416. Springer, 1993.

[Ver10] Sven Verdoolaege. *isl: An Integer Set Library for the Polyhedral Model*. ICMS, volume 6327 of *Lecture Notes in Computer Science*, pages 299–302. Springer, 2010.

[VG12] Sven Verdoolaege and Tobias Grosser. *Polyhedral Extraction Tool*. 2nd International Workshop on Polyhedral Compilation Techniques (IMPACT'12), Paris, France, Jan 2012.

[VSB$^+$07] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. *Counting integer points in parametric polytopes using Barvinok's rational functions*. Algorithmica, volume 48, number 1, pages 37–66. June 2007. URL: http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=41970, DOI: 10.1007/s00453-006-1231-0.

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese Masterarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich die Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, den 30. April 2013

Stefan Kronawitter