Latency Optimized Code Segmentation

mynatix ag *

Abstract
Working in a team can speed-up the completion of a job. However, splitting tasks and coordinating them effectively within a team can be complex. Analogously, adapting code to optimize the use of modern multicore hardware can be challenging: Significant experience and expertise is required to develop code that allows several units to work on the same problem in parallel. While there are tools and libraries available to assist developers in adapting their code for parallel processing, the full automation of an optimal software adaption to more than one computational unit remains challenging, especially since parallelism often depends on runtime variables. In compilers, the fundamental principle to extract parallelism on code level is data dependency analysis, which is used to exploit Instruction Level Parallelism (ILP). Three dependencies, read-after-write, write-after-read and write-after-write, are fundamental to any code compilation and are efficiently used in modern compilers and hardware schemes. The read-after-read (RAR) dependency has been disregarded in the code compilation so far, as it cannot cause any data hazards. This article introduces a novel method to use the additional RAR dependency information contained in any code to enhance automatic parallelization. The novel principle is introduced using a simple example and is tested with data and task parallelism problems: the principle was tested on a multicore-CPU to automatically parallelize the computing of the recursive Fibonacci function, which leads to a speed-up of 3.8-fold compared to a version scheduling each function call as a parallel task. Many physical phenomena can be described by Partial Differential Equations (PDEs) and in a different example, the principle was applied to compute the solution at the discrete points of a mesh describing space and time. Computing the solution for all points can generate large data sets. Solving this discretized two-dimensional diffusion equation with a Finite Difference (FD) scheme can require the computation of a three-times nested loop. Using our novel approach, speed-ups of up to 4.8-fold can be reached without any additional code annotation. State-of-the-art compilers are not able to parallelize such nested loop-section from unannotated code without the proper use of corresponding frameworks.

* info@mynatix.com

Contents

Introduction

Task distribution among multiple workers is a problem known in many use-cases and industries, from logistics over coordinating team work to running software on modern computing platforms. The distribution of tasks to available workers starts by splitting a process into tasks and then optimally distributing these tasks to the available workers. A fundamental principle when distributing work comprises two steps:

a) the communication of all necessary information regarding a task to the workers, and
b) the workers using the provided information to execute their assigned task.

These two steps can be described by a communication and a processing part. To quantify the ratio between doing work $E_{processing}$ and needed communication $E_{communicating}$, this ratio is defined as granularity $G = \frac{E_{processing}}{E_{communicating}}$, where E indicates the effort.

Mynatix introduces a novel technique to segment a process or code into distinct blocks which can be used afterwards to parallelize this process or code. Each block has a granularity G and consists of a sequence of distinct work-steps requiring the same input information. Each block knows from preceding blocks which information is needed. The structure of these blocks builds a base for numerical optimization techniques to e.g. minimizing the execution time or reduce the energy demand of the overall process. This technique is very promising to optimize code to run on modern (parallel) computing platforms. The next chapters introduce how the Latency-Optimized Code Segmentation (LACOS) carves out blocks of arbitrary sequential instructions and potential data transfers between these blocks from a given (sequential) code. LACOS retrieves more information from code than

state-of-the-art methods by using the read-after-read (RAR) dependencies. This enables the compiler to generate a novel, generic dataset from a sequential code, which can be used to enhance the code distribution to a parallelized hardware using numerical methods. LACOS improves the automized adaptation of code to a hardware and:

a) is not sensitive to programming languages and thus has many areas of application.
b) is a physically based and therefore patentable technique, implementable as a software or hardware.
c) enhances static code analysis to impact the compile time and not the execution time which thus doesn't delays the runtime [34].

Mynatix is beginning to explore the limits of this new approach. As numerical optimisation techniques strongly depend on the underlying data and LACOS has been demonstrated to retrieve more information from static code analysis, more automation will be possible to adapt code to hardware by using LACOS. This reduces the requirements of expertise, time and costs in software development.

1. Theory

This section will introduce some fundamental aspects of parallelism regarding both hardware and software.

1.1 Parallelism on circuit level

The time to charge or discharge the capacitance of a circuit, meaning the change of its state by charging or discharging the voltage level, is called the propagation delay and can be seen as the time required to compute a binary operation $t_{compute}$. The signal transfer between hardware components by wire requires a transfer time, $t_{transfer}$. Looking at the parallelism possible on a circuit level, digital circuits can be distinguished between combinational and sequential ones [45]. Sequential circuits depend on past and current inputs, which means they must have a memory system and a clock to keep track of timing. Additional to $t_{compute}$, $t_{transfer}$ is needed to transfer and input past data from other circuit parts. On the other hand, in combinational circuits, the result depends only on the current input and only the computational time $t_{compute}$ is required.

1.2 Parallelism on hardware level

Parallelism on (micro-)processor scale, covers a wide field of architectures. On computing systems with only one CPU (uniprocessors), parallelism is exploited by optimising instructions to the CPU-pipeline and in corresponding hardware schemes [17]. Different uniprocessors exist from single instruction / single data (SISD, a standard uniprocessor) to Single Instruction / Multiple Data stream (SIMD, e.g. SIMD instructions, GPUs, etc.) to multiple instruction / single data streams (MISD) [12]. Multiprocessors can be distinguished by the used memory models. In non-uniform memory access (NUMA) designs, a part of the memory is allocated for each processor. In this case, processors use the same virtual address space and the communication can be implemented via shared memory variables [21]. On the other hand, if each processor uses a different virtual address space, like e.g. in multicomputers or clusters, the communication between processors must be performed by message passing. Each architecture type and communication model has its own characteristic for handling the computation and transfer of data. However, all share the challenge of optimally distributing the computations (instructions) in a code to more than one computing unit. In general, transferring data is a magnitude slower than computing several instructions in a row on the same data.

1.3 Parallelism on code level

The order of the statements in a code is normally given in a sequential form. Depending on the architecture of the hardware, it is therefore necessary to reorder the instructions in the statements and / or distribute them to different cores to exploit the computational power of the hardware optimally. Modern compilers and frameworks optimise the handling of data and instructions and leave developers to optimise their codes to a specific architecture. State-of-the-art compilers efficiently optimise code for a target architecture with one computing core. Uncompiled programming languages, such as interpreted languages, can be by design serial or provide extensions to enable developers to exploit parallelism on different hardwares. One can distinguish between different levels of parallelism:

- `Bit-level parallelism`. The larger the processor's (register) size, the smaller the number of needed instructions (example: sum of a 16-bit integer on a 8-bit processor = 2 instructions compared to a 16-bit processor = 1 instruction).
- `Instruction-level parallelism (ILP)`. This refers to exploiting the simultaneous execution of multiple instructions, such as instruction pipelining, out-of-order execution, etc. [17].
- `Loop-level parallelism (LLP)`. In this case, ILP is exploited between different loop iterations: the loop-carried dependencies are key for this type of parallelism. There are different approaches to parallelise loops, such as DOALL, DOACROSS or DOPIPE parallelism [48, 28, 6]. Compared to ILP on modern hardware, this is most incorporated on source code level by techniques such as unrolling loops, vectorization and other [4].
- `Thread-level parallelism (TLP)`. In contrast to ILP, TLP is rarely covered by hardware schemes. Independent tasks are created by programmers or operating system to exploit running tasks in concurrency.

In general, there are two ways to differentiate how parallelism is exploited in TLP: data and task parallelism [22]. LACOS is able to extract more information from dependency analysis of static code than state-of-the-art methods are. This enables LACOS to enhance both ways of exploiting parallelism, data and task parallelism, in a code automatically.

1.4 Compilers and ILP

Data dependency analysis is well known in compiler theory. The novel segmentation is based on data dependencies and its fundamentals are briefly summarized below, with focus on the history of data dependency and its use to exploit instruction level parallelism (ILP). This is followed by a description of state-of-the art compiler technologies, with focus on how they rely on the fundamental basis of dependence information, which was studied intensively to exploit ILP up to the peak in research around the year 2000. It will be shown that read-after-read (RAR) dependencies have not been used to support automatic parallelization of code. This leads to the introduction of the principle how these dependencies can be used to introduce potential transfers between groups of arbitrary sequential instruction chains. These potential transfers link to the inevitable latency for any binary computation. The approach is introduced with a simple example.

Sections 3 and 4 present two applications on how the segmentation can be used to optimise a code on a multicore platform.

1.5 History of dependency analysis

Segmenting code into basic blocks (BB) in a control-flow graph (CFG) has a long history in Computer Science [2]. Today's state-of-the-art compiler-frameworks have implemented sophisticated steps to analyse the segmented code in their intermediate representations (IR) [25, 7]. Intermediate representations are the compiler's own language the input code from the frontend is translated to. When compiling code to a hardware, the program correctness is one of the fundamental principles of compilers and is maintained by correctly handling control and data dependencies [17]. These dependencies are used in compilers i.e. to preserve the correct order and timing of data operations in a program when exploiting instruction level parallelism (ILP). Therefore, data dependencies are key for exploiting available parallelism in a program. At the level of ILP, the scheduling of instructions is optimized to the latencies of the functional units in a CPU-pipeline, which is the standard in all processors since 1985 [17]. The central aspect of the optimization is to reorder instructions to optimally, respectively in parallel, run on a given processor.

1.5.1 Fundamentals of dependency analysis

A distinction is made between three types of dependencies: data, name and control dependencies [17]. In the following, the focus is set on data dependencies. There are three cases of dependency that can lead to data hazards:

- `Read-after-write (RAW)`: Flow dependence, when an instruction must wait till a preceding instruction is calculated. The instruction depends on the result of a preceding instruction.
- `Write-after-write (WAW)`: Output dependence, when an instruction writes its result to the same register as a previously, but not yet finished operand. A change of this order would change the final output.
- `Write-after-read (WAR)`: Anti-dependence, when an instruction writes to a destination after it is read by another instruction. A change of this order would result in the reading instruction using the wrong data.

Different approaches have been built to exploit ILP while taking these data-, name- and control-dependencies in account. There are two fundamentally different approaches: a) finding parallelism statically at compile time and b) parallelizing from the hardware-side by dynamic scheduling [17, 32, 40], which includes handling potential data hazards [5, 20] and thus focuses on the influence of data, the data-flow execution [17].
An overview of ILP approaches in the 20th century is given in reference [35]. Multiprocessors started to be used in the 20th century and a peak was reached for ILP after 2000 [17].

1.5.2 ILP and TLP

The shift to multiprocessors went along with the growing number of data-intensive applications. The demand for data parallelism as well as request-level parallelism (large number of parallel requests) [17], led to thread-level-parallel (TLP) approaches. With thread-level-parallelism 'Multiple Instruction Multiple Data' computer architectures [12] could be exploited. Here, the independent threads are normally identified by the programmer, which stands in contrast to ILP, where parallel operations within a loop or a straight-line code segment are exploited by compilers, hardware-schemes and operating systems [17]. TLP is hardly covered by the hardware [27]. This resulted in a shift from hardware-supported ILP exploitation to programmers exploiting TLP by hand. There was a development from exploiting ILP by software to hardware exploiting ILP and, now, software exploiting TLP became relevant. It is known that ILP must be detected over the boundary of basic blocks. Therefore, the focus was set on loops to enable TLP [27]. In this context, the RAR dependency has not had any relevance [26].

1.5.3 ILP in loops

Instruction level parallelism is especially important for loops [17]. In loops, unrolling is one technique to convert Loop-Level-Parallelism (LLP) into ILP, but it is limited by different factors [17]. With focus on dependencies, the task is to determine dependencies for the statements in the loop body. Distance vectors can be used to define direction vectors, which can be used to identify dependencies between iterations by indicating how far apart accesses to the same memory location are [13]. If the distance vector between two iterations lies outside the scope of the unrolled loop, those two iterations can potentially be executed in parallel. Allen & Kennedy [4] demonstrated, how flow dependencies are the only important kind of dependency and that anti- and output-dependencies are meaningless for vectorization, as they only fix the order [4]. Allen & Kennedy [4] relied on work done by Towle [41] to define dependencies as "[they] must be considered when detecting recurrences that inhibit vectorization" [4]. Testing array-subscript dependencies is crucial for parallel execution

of a program, ensuring different parts do not conflict. This involves verifying, whether one program segment depends on another's output. Solving Linear Diophantine equations seeks integer solutions to polynomial equations [13], which can help to identify the dependencies between array segments but is a NP-hard problem. This has led to today's limits. As will be explained later, using RAR enables to build groups of instructions based on the distance vector and a simple scheme to minimize the inner-loop data dependencies holding novel opportunities for static code analysis and automatic parallelization.

1.5.4 State-of-the-art

Examples of state-of-the-art work exploiting ILP in loops are discussed in references [48] or [28]. Earlier work based on exploiting thread-level parallelism by speculating on multicore chips during runtime to overcome code immunity to static parallelization is shown in reference [47]. These approaches are based on data dependency speculation, meaning that it is guessed, whether the data access is in a different location. In case there is a RAW hazard, this is reversed in an additional step [47]. This thread-level speculation is the origin of the limits reachable by static parallelization [13]. Zaidi et al. investigated the limits of LLP [48], but they base their approach only on three types of data dependencies: RAW, WAR and WAW. Another state-of-the-art paper introduces a compilation layer over the open source compiler infrastructure LLVM [28] and uses the automatic parallelization framework HELIX for loopregions [6]. These authors use the work described in [15] to identify dependencies and focus on the loop-carried data dependencies, which also do not use RAR dependencies. Furthermore, the latest studies on exploitable ILP, e.g. [11], do not indicate that RAR could be used to exploit ILP in any of these schemes.

1.6 Using RAR in compilers

Historically, RAR dependencies are not considered as they do not cause any data hazards. Similarly, when it comes to making use of Instruction Level Parallelism (ILP), compilers do not need to consider RAR dependencies. This is also true for hardware that dynamically schedules tasks, since it can naturally handle RAR situations without any issues. To the best knowledge of the authors, this is still the state-of-the-art in today's compilers and hardware-side dynamic scheduling mechanisms. Nevertheless, RAR dependencies provide information about the parallelism in code and the novel technique Latency Optimized Code Segmentation (LACOS) uses this additional information available in any code. LACOS uses data dependency analysis at instruction level to build blocks with arbitrary serial instructions and introduces potential data transfers to take RAR data dependencies into account. This results in a finer grained code segmentation than with Basic Blocks with two distinct features:

- A physically based segmentation of parallelizable code segments. The information of parallel blocks with sequential instruction chains is identified in the Control Flow Graph (CFG) of a code based on static code analysis. This results in a generic, continuous series of computations followed by a transfer / communication segment, from which parallel code can be derived.
- The series of computational and transfer segments are used to calculate a ratio between the time that is used to compute and transferring data. This ratio was earlier introduced as the granularity G_0. In the optimization step, the granularity is then taken into account to distribute the computing segments to different computational units with minimized transfer latencies. The separate computational units (ALU, cores in a multicore CPU, GPU-threads, etc.) are in the following called localities.

Sequential instructions are in the following grouped into computation blocks (CB), defined by RAW dependencies. The fundamental idea is that in any computer system that processes data in binary form, the time it takes to share data, represented by potential transfers between blocks, is inevitably physically dependent on the latency experienced by accessing data. It does not make sense to granularly segment the computation blocks, as they have a direct write-to-read link. It holds, that this kind of segmentation is the fastest form to compute a binary result in any scale of computing, from circuit, microprocessor, pipelined CPUs through superscalar processors, to multicomputers with distributed memory platforms. This article aims to show the use of LACOS in compilers and for optimizing interpreted languages used on platforms with more than one computing unit. LACOS contrasts today's modern (superscalar) processors, which already include many approaches to exploit ILP in their working schemes [39], but with an expensive overhead in amount of chip area used [34]. There has been a trend towards multicore processors over the last two decades [33]. LACOS is therefore particularly suitable to bridging the gap between the level of ILP that modern processors can utilize and the increasing demand of applications that can take full advantage of multicore platforms, by e.g exploiting Thread-level parallelism.

2. Latency Optimized Code Segmentation

The repeated reading of the same data element in a code (RAR) provides the possibility to run the subsequent instructions in parallel. This indicates that data must be transferred to another locality. A very simple program with one Basic Block and with four statements is shown in Figure 1 to illustrate how LACOS works.

The four statements can be illustrated in the form of a computation graph: triangles represent instructions, circles represent data information, while angular, solid and directed lines indicate data flow, and dotted lines indicate flow dependencies resulting from the data dependencies. The RAR on the variable a indicates the possibility to compute the subsequent instructions in parallel. Therefore, the instructions calculating x and y could potentially be computed in parallel at two different localities. Based on this information, LACOS builds a

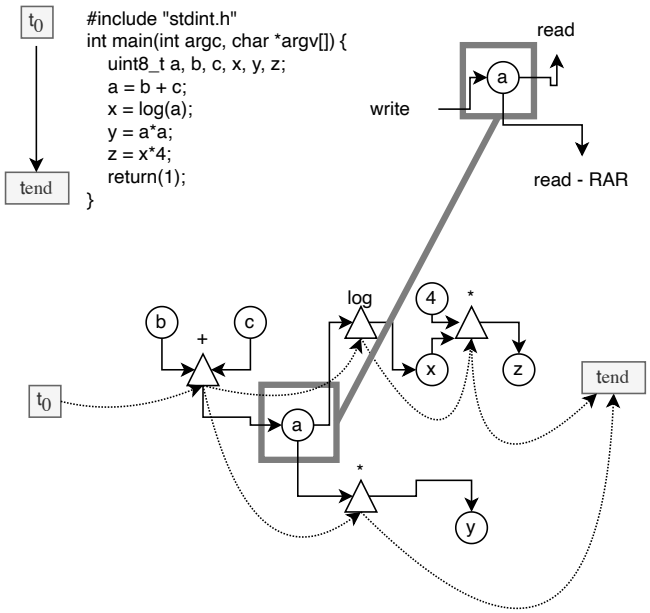

Figure 1. Simple code example with 4 instructions and corresponding computing graph built using the data dependencies of the instructions. Triangles represent operations, circles represent data information, angular, solid and directed lines indicate data flow and dotted lines indicate flow dependencies resulting from the data dependencies.

Figure 2 shows the CB configuration for the example introduced earlier. When adding the statement $y=a*a$, the RAR on the variable a is resolved by adding CB2 and CB4 after the statement, where a is written by $a=b+c$. The introduced potential transfer for the variable a is visible as a dotted line in Figures 2, 3 and 4. It is important to note that it is not yet given whether these transfers are established in a subsequent parallel code. This decision is made in the optimization step, see section 2.3.

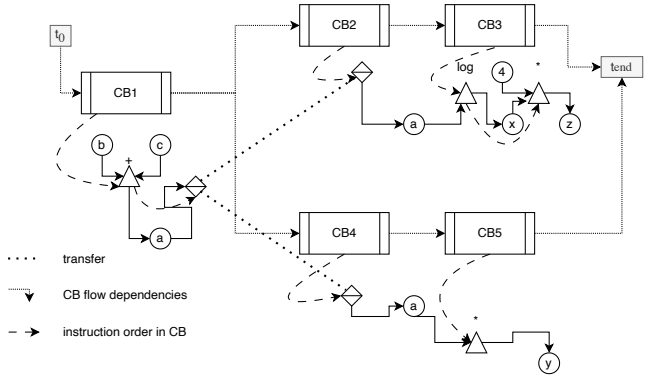

Figure 2. Graph with computation blocks for code in Figure 1. Dotted line indicate data transfer between CB1 to CB2 and CB3, fined dotted the flow dependencies of the computation blocks and dashed instruction order within the CBs.

specific computation flow graph, called LACOS-graph, with novel segments, the computation blocks (CB). One CB consists of a series of arbitrary sequential instructions (RAW). For each CB, potential data transfers at the beginning and end are defined, see Figure 2. The CB can be built with a simple rule to add a new instruction N to the LACOS-graph - first without including control dependencies:

a) If the N is a RAW instruction dependent on instruction M, and all input data needed to execute N is available in the computational block of M, then add instruction N to the chain of the computation block of M after the instruction M.
b) In case an instruction N needs further input data then available in the CB of M (e.g. the data elements read by N are written by two instructions M and K, where M and K are in two different CBs), a new CB is created with N and the corresponding data transfers are added.
c) If N is, again, a RAW instruction dependent on M, but in this case M is not at the end of the instruction chain of its CB, split the original CB of M directly after M and add two new computation blocks, with the needed data transfer. An example can be seen in Figure 2, where M is the statement, where a is written, and N is the statement, where y is calculated.

Following these rules, each block has: a) computations in the form of a chain of arbitrary sequential instructions with True dependencies and, b) known transfer properties at the start and end of each block to other blocks.

2.1 Data and control perspective

The LACOS-graph represents the data flow in a novel form by including the, until now, hidden information in RAR dependencies. In classical data flow analysis, the instructions are data-dependent from more than one predecessor, which is an issue when optimizing [17]. In LACOS-segmented code the RAR are resolved by adding parallel CBs, which represent potential transfers in two flows that are separated by a transfer of the data element causing the RAR dependency. Viewing a program from the control and data perspective has a long history [16]. The control graph shows all paths that can be traversed by a program and the data graph shows the flow of the data between instructions [1]. A LACOS-graph is a control/data flow graph (CDFG) [46], with a finer granularity than a Control Flow Graph (CFG) with Basic Blocks. The RAR dependencies are used to differentiate parallelizable CBs that contain instructions originally grouped by their RAW dependencies.

It is important to distinguish several Basic Block types, where a Basic Block is defined as a subsequent part of code with no jumps in or out of the block, except the entry and exit [3]: BBs in loops (see BB2 in Fig. 3) and BBs with sequential instructions not in loops (BB3 in Fig. 3). Applying LACOS to a code in Basic Block segmentation, CBs can be positioned according to the control dependency of their original Basic Blocks, see Fig. 3. The information of the number of parallel CBs and their order can be preserved over the boundaries of Basic Blocks. This is visible in Figure 3 for the “True”

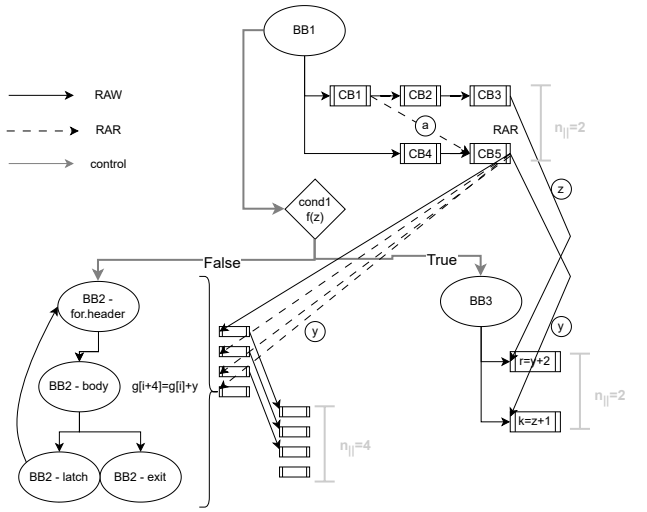

Figure 3. The LACOS-graph: a CDFG with BB1 containing code from Figure 2, BB2 a loop over $g[i+4] = g[i] + z$ and BB3 $r = y + 2; k = z + 1;$

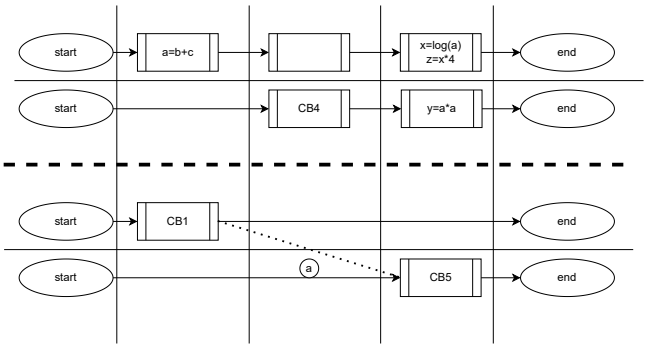

Figure 4. Ordered CBs in a computation matrix (above) and a transfer matrix (below) with two distinct paths to compute the code.

case for condition *cond1* ($BB1 \rightarrow BB3$). The approach to apply LACOS to loop-sections is introduced in more detail in section 4.1. This enables the usage of LACOS in compilers and for optimizing interpreted languages, which provides new opportunities by e.g.:

a) generating tasks with a calculatable granularity as a function of runtime-variables. This simplifies e.g. the scheduling of tasks in the kernel of operating systems.
b) identifying the critical path using the levels of parallel CBs in different basic blocks [42].
c) exploiting loop-level-parallelism using distance and direction vector to form CBs for BBs in loop-sections.

2.2 Retrieving parallel code

Computation and transfer properties can be stored separately in two matrices: a computation matrix and a transfer matrix (Figure 4). In Figure 4 the control dependencies are not shown for simplicity and correspond to the example in Figure 1. Different possible paths can be stored by adding a third dimension to the matrices. Independent flows that allow computing code at different locations are indicated as rows, each with their own *start* and *end* symbols. This form of code segmentation provides a structure to distinct between parallelizable codes for each location with all necessary data transfers.

Based on these two matrices, it is possible to deploy the computation and transfers in a parallel code. Different communication models can be applied to execute the data transfers, depending on the target platform: from using shared memory to data messaging. For example, barriers (synchronization of threads by e.g semaphores) can be added when a transfer occurs (e.g. before CB5 starts) for platforms using shared memories. Alternatively, an active communication model e.g. 'Message Passing Interface'-protocol (MPI) can be used to send a after CB1 to location 2, which expects to receive data during CB4. The resulting parallel paths consist of a continuous series of computations and transfer steps for each location.

This segmentation of code is grounded in the physical principle that the time required to transfer data is an order of magnitude greater than the time needed to continuously compute serial instructions at the same location. Applying LACOS to a code, generates a segmentation with a base granularity G_0 by identifying the smallest sequential segments in a code. The enumeration and placement of the blocks in the two computation and transfer matrices result in each row representing a separated computing path connected with appropriate transfers. In this generic form, the row dimensionality of the matrices shows the maximal parallelizability of a code for the case that the number of localities / processors is unlimited. This level of parallelism only depends on the computation graph (ideal parallelism). In real applications this is obviously not given.

2.3 Optimization complexity / scheduling

After any kind of code segmentation, the resulting segments have to be mapped to localities. The optimization step consists of mapping the segments to available, different localities, which we will refer to in the following as computing units. This step consists of the matching (meaning assigning) and scheduling (meaning ordering) of a series of instructions (tasks) by optimizing their compute and dependency pattern to a target objective function [37]. In LACOS-matrices, each row consists of tasks with different granularities [23], as shown in Fig. 4. This structured form of the code can be used to retrieve code with higher granularities G by combining rows in the matrices. This leads to the summation of computations and elimination of potential transfers (Figure 5).

In the example in Figure 5, a hypothetical runtime for both versions can be defined as:

$$t_{end,2loc} = t_{c,CB1} + \max\left(t_{t,CB1\rightarrow BC5} + t_{c,CB5}, t_{c,CB3}\right) \quad (1)$$

$$t_{end,1loc} = \sum t_{c,i} = t_{c,1} + t_{c,2} \quad (2)$$

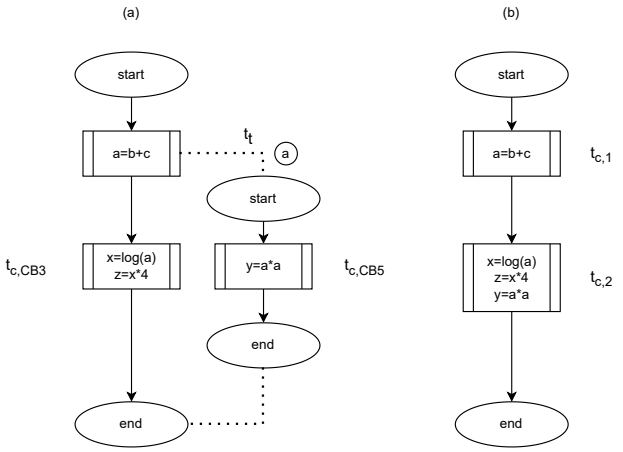

Figure 5. Combining rows in the matrices reduces the level of parallelism which leads to a higher granularity G

See Fig. 5a for the LACOS-graph for Eqn (1), and Fig. 5b for Eqn (2). The minimal / optimal runtime can be then defined by Eqn (3).

$$t_{end,minimal} = \min\left(t_{end,2loc}, t_{end,1loc}\right) \tag{3}$$

This is only a schematic example to introduce the principle for an optimization approach using the proposed novel segmentation.

In general, the distribution of n segments with a brute-force approach would result in an $\mathcal{O}(n!)$ algorithm, if all sequences would have to be evaluated [43]. Optimization by distributing code to different cores has a long history in research [14]. LACOS groups the number of smallest segments (instructions) of the code to physically inseparable groups and orders these groups based on which can run parallel to each other. From a physical perspective it does not make sense to further split the grouped instructions in the computation block since combinational computation (without any transfer) is faster than sequential computation [45]. This can significantly reduce the complexity of mapping the CBs to different units, especially for loop-sections. Using this method, compilers can see distinct groups of instructions to schedule on different locations, thus resolving the problem of having instructions with more than one data dependency, which is an issue when optimizing [17]. It enables the reformation of a code to build tasks (instruction blocks) with a certain target granularity dependent on the used platform. This aproach trivializes the optimal scheduling of the tasks. The tasks are composed to have (approximately) the same granularity, in other words computing and transferring ratios.
LACOS can introduce solutions that significantly reduce the effort required to map the LACOS-segments for the following cases:

a) A distinct number of same, parallel CBs have to be distributed to symmetrical units. In this context, symmetrical means that all localities have the same computing performances and equal transfer-latencies for moving data between each-other (e.g. a multicore-CPU).
b) Building a series of tasks with one, fixed target granularity.

The feasibility of such an optimization approach is supported by the limited parallelism in Basic Blocks [44] - excluded from these limits are Basic Blocks in loop-sections, see section 4.1.

When not all units have the same characteristics, such as, if a platform consists of localities with different performance and transfer latencies, LACOS produces a data set suitable for numerical optimization. It provides a numerical system that can be optimized for an objective function with known (heuristic) optimization methods, e.g. to reduce overall latency times. The optimization of a LACOS-graph has combinatorial complexity as a function of

$$n_{units}{}^{\sum_{i=0}^{kernel} n_{CB,i}} \tag{4}$$

where n_{units} is the number of computing units, n_{CB} the maximal available parallel CBs over a *kernel*-width for a target platform. The *kernel*-width relates the (minimal) computing time to the (maximal) transfer-latency given on a platform.

2.4 Benchmarking LACOS

Measuring the performance of parallel computations on a platform can be complex and must be done carefully [18]. LACOS automates the step of parallelization, respectively distributing a code to more than one unit. Therefore, the improvement gained by applying LACOS to a code must be compared to methods with the same scale of automation. LACOS also extends the application field of compilers by enabling the parallelization of code to more than one unit. Therefore, the output of code transformed by LACOS can for now be compared to state-of-the-art compilers. When using modern development-stacks for software development, the process of parallelization, respectively distributing code to more than one unit is largely explicitly instructed by developers. This manual work is supported and simplified by a wide range of frameworks (e.g. TensorFlow, OpenMP, CUDA), standard protocols (e.g. MPI), tools (e.g. numba for python) or specific languages (e.g. julia, fortran). All these frameworks and tools support developers to write parallel and concurrent applications suitable for running on a distinct platform, e.g. on multicore-CPUs, GPUs or cluster-instances. For interpreted languages, extensions (e.g. numba for python [24]) can improve code performance to reach a state-of-the-art compiler optimization level with only few manual annotations. In such cases, the performance gain reachable by LACOS is again comparable with compiled solutions. To benchmark LACOS against any solution created by developers using established frameworks or writing parallel code directly, two aspects must be compared: a) the reachable level of optimization, b) comparing the development time including needed skills. Such a study has to be performed with many different applications of LACOS and is planned in a next article.

2.5 LACOS in test cases

In general, there are two approaches to break down a code into concurrent parts: data and task parallelism [22]. For a first test of LACOS, it was applied to both. These experiments are presented in sections 3 and 4, showing promising results.

3. Application of LACOS to the Fibonacci Sequence

The computation of the Fibonacci sequence for a given n is a well-studied phenomenon. In principle, the computations cannot be parallelized as the computations base on the first two elements.

```
define dso_local i32 @fib(i32 noundef %n) {
entry:
  %retval = alloca i32, align 4
  %n.addr = alloca i32, align 4
  store i32 %n, ptr %n.addr, align 4
  %0 = load i32, ptr %n.addr, align 4
  %cmp = icmp sle i32 %0, 1
  br i1 %cmp, label %if.then, label %if.else
if.then:
  %1 = load i32, ptr %n.addr, align 4
  store i32 %1, ptr %retval, align 4
  br label %return
if.else:
  %2 = load i32, ptr %n.addr, align 4
  %sub = sub nsw i32 %2, 1
  %call = call i32 @fib(i32 noundef %sub)
  %3 = load i32, ptr %n.addr, align 4
  %sub1 = sub nsw i32 %3, 2
  %call2 = call i32 @fib(i32 noundef %sub1)
  %add = add nsw i32 %call, %call2
  store i32 %add, ptr %retval, align 4
  br label %return
return:
  %4 = load i32, ptr %retval, align 4
  ret i32 %4
}
```

Figure 6. Recursive implementation to compute Fibonacci sequence in LLVM intermediate representation (IR)

Nevertheless, running a recursive implementation in concurrency can improve the speed of the calculation in a multithread environment [36]. Figure 6 illustrates the recursive implementation to compute a Fibonacci number with the function *fib(n)* for a given number n. The function definition visible as LLVM IR is chosen as an example, because by running each recursive function call as a task, the resulting task-graph matches the LACOS-graph. This makes it suitable to demonstrate how to optimize a LACOS-graph to a platform.

Different colors are used for each basic block in Figure 7 given by the branching due to the if-statement. Figure 7 shows the CBs and potential transfers generated with LACOS for a recursive unrolled call *fib(3)*, based on the code in LLVM IR. As each transfer on a platform introduces an overhead, the

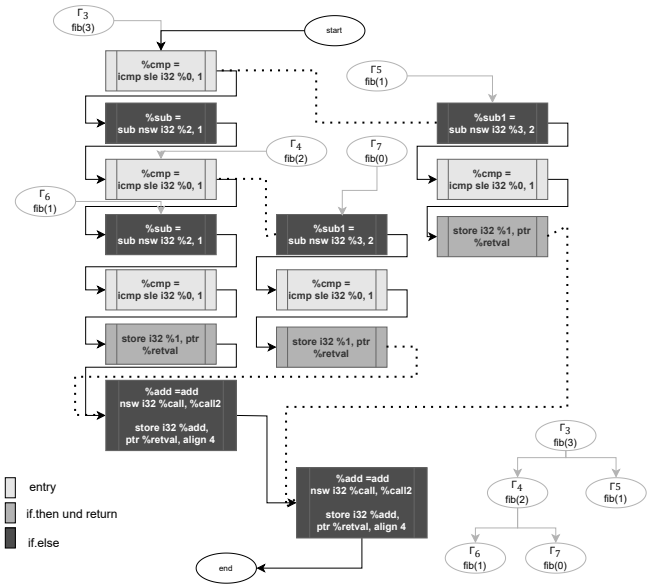

Figure 7. LACOS-graph, which is equivalent to the task-graph, when starting function as separate tasks for code in Figure 6. Fixed lines indicate instructions belong to the same computation segment (task), dotted lines potential transfers in G_0

granularity G_0 (e.g. in Figure 7) can be changed to meet the physical demands of a specific platform. This can be achieved by combining parallel LACOS-nodes until the overhead to run in parallel is lower than the serial execution. Although this approach doesn't guarantee to find the global maxima, this optimization technique is computationally bounded. This can be illustrated for the children nodes indicated with Γ_6 and Γ_7 and their parent node Γ_4:

$$t_{serial,\Gamma 4} = \sum \triangle t_{c,\Gamma} = t_{c,\Gamma 4} + t_{c,\Gamma 6} + t_{c,\Gamma 7} \tag{5}$$

$$t_{parallel,\Gamma 4} = t_{c,\Gamma 4} + t_{ctxsw} + \max\left(t_{c,\Gamma 6}, t_{c,\Gamma 7}\right) \tag{6}$$

As long as $t_{parallel,\Gamma 4} > t_{serial,\Gamma 4}$ it is beneficial to run the group Γ_4, Γ_6 and Γ_7 in series. Otherwise it is beneficial to run the nodes in parallel despite the overhead of starting (and ending) an additional task, t_{ctxsw}. By combining the LACOS graph nodes of the same graph-depth, the computation of the sum and the transfers vanish. By continuously grouping nodes in the LACOS-graph to tasks until the intended granularity is reached, the code can be optimized to fit the latency properties of a target platform. A LACOS-graph can be composed from any code and this form of graph optimization enables to exploit task parallelism in code in a novel, generic form. This approach was tested on a platform, with an Intel®Xeon®CPU E5-2680 V3 @ 2.50GHz with 32GB RAM running ubuntu 22.04.2 and using the oneTBB Task scheduler for task scheduling [8]. Following [38], a context

switch time of $t_{ctxsw} = 3860ns$ was measured on the platform. The code has an IPC of 1.13 at a speed of 254 GHz given by using '*perf*'. The LACOS-graph can be traversed and combined until the cumulated computations of the nodes become larger than the context switch time, (Eqn. 5 and 6). With the measured IPC and context-switch-time, the threshold for the granularity can be forecast during compile-time. For the given platform, the threshold is evaluated $threshold = 15.82$. This corresponds well with the optimal threshold gained by the measurements (Fig. 7).

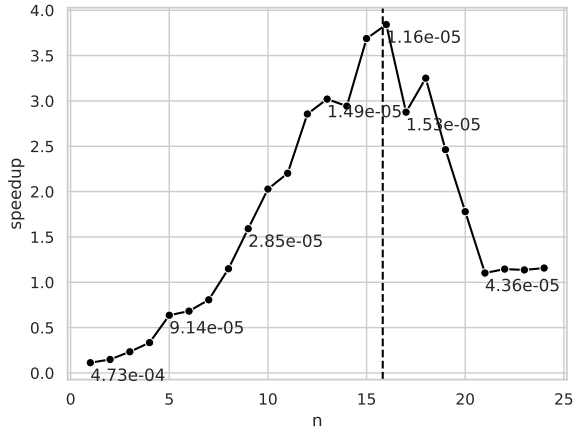

Figure 8. Measured speed-up by combining LACOS-nodes until the parallel execution of tasks is beneficial in regard to the overhead of context-switches t_{ctxsw}. The numbers show the runtime in *ns* using [9].

This threshold can be determined solely by the LACOS graph, which is based on the static code analysis, the IPC and the context-switch-time t_{ctxsw}. This illustrates how code can be adapted to the physical properties of a platform using LACOS. Additionally, this allows code to be optimized depending on runtime variables, which can enable new limits of automatic parallelization using static code analysis methods. The code can be automatically generated:

```
long fib2(int n) {
 if (n < 2) {return n;}
 else {fib2(n-1) + fib2(n-2);}
}
long fib(int n, int threshold) {
 if (n < threshold) {return fib2(n);}
 else {
 int x, y;
 tbb::task_group g;
 g.run([&]{x=fib(n - 1, threshold);});
 g.run([&]{y=fib(n - 2, threshold);});
 g.wait();
 return x + y;
}}
```

While running this code, the tasks are executed, if a certain *threshold*, symbolizing the transfer overhead defined by the hardware-granularity is not reached.

4. Application of LACOS to the two-dimensional diffusion equation

Loop sections can result in a high computational demand. LACOS is able to exploit data parallelism opportunities in nested loop sections by including the RAR dependencies. The implementation of a two-dimensional diffusion equation using the Finite-Difference (FD) method to discretize the equation on a mesh, leads to a three-nested loop over the spatial and temporal solution space. State-of-the-art compilers, such as LLVM or gcc cannot directly parallelize such loop sections. Furthermore, frameworks, such as OpenMP, must be applied correctly to only the two inner loops to achieve a successful parallelization for a multi-threaded platform. Therefore, this example is chosen to demonstrate how LACOS can correctly exploit the data parallelism in this algorithm fully automized.

4.1 LACOS for loop sections

First, the principle of LACOS in a loop is illustrated using a simple, comprehensible example, as it is tricky to illustrate all data dependencies for complex, nested and unrolled loop sections. It is important to note that the statements that are computed in the loop-body often depend on loop variables, which themselve often depend on runtime variables. One of the possibilities to exploit ILP in loops is by using loop-unrolling and by detecting data dependencies in the statements within loop-bodies [17]. Such a dependency analysis has been known to be limited for a long time [13]. The limitations arise from the increasing complexity of the code analysis with the increasing unrolling depth [10].

In this code example:

```
for (i=0, i<N, i++) {
 a[i+6] = a[i]+a[i+2]+C;
}
```

the LACOS-graph can be derived solely by the distance $\overrightarrow{\Delta rw}$ built by the indices determining the linear access to memory. Using this distance, respectively direction, vector [13], the number of parallel computation blocks in a LACOS graph can be expressed as a function of runtime variables by applying the fundamental principle of LACOS.

For this code sample, the element at index *i+6* is overwritten with the sum of the value in element *i* and element *i+2* and a constant *C* in every iteration. Therefore, *a[6]* is written at *i=0* and is read for the first time from within the loop at iteration *i=4* and read a second time at *i=6*. At *i=6* the RAR dependencies are indicated, which LACOS uses in addition to state-of-the-art methods to exploit ILP. In a *for*-loop, the loop-variable changes in each iteration by a given rule (in the example $i++ : i = i+1$). To illustrate how LACOS handles these statements, one computation block is introduced for each unrolled iteration. Parallel LACOS nodes consist of

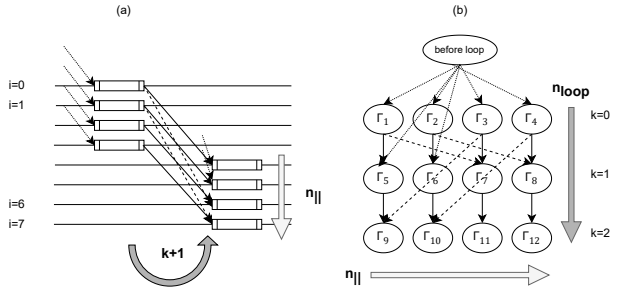

Figure 9. (a) Computation blocks which each read, compute and write the statement $a[i+6] = a[i] + a[i+2] + C$ for each iteration i. (b) LACOS-graph with parallel nodes for each iteration and indicated internal transfers. Solid arrows show RAW, dotted arrows show RAR.

instruction groups, which have no write to read dependencies (RAW) and do not share the same value for the incrementing index i. In Figure 9 a, it is shown how four computation blocks are parallel until the first write to read dependency (RAW) creates a potential data transfer. The solid arrows indicate the transfers triggered by the RAW dependency, the coarse dotted arrows show the RAR dependencies and the fine dotted arrows show the necessary data transfers from the CB(s) preceding the loop section.

The loop-Basic Blocks can be composed in the LACOS-graph by $n_{\|}$ and n_{loop}, where $n_{\|}$ describes the number of parallel CBs and is a function of the distance between the indexed accesses on a, $\overrightarrow{\Delta rw}$, where 'r' stands for 'read', and 'w' stands for 'write'. n_{loop} describes the minimal amount of sequential iteration over the $n_{\|}$ parallel CBs, and cannot be parallelized and is a function of the resulting $n_{\|}$ and the loop variable definitions. Between each read $a[i+o_r]$ and write $a[i+o_w]$ an index distance pair can be derived based on the offset o_r and o_w from the loop-variable i. For the equation $a[i+6] = a[i] + a[i+2] + C$ this results in two distances: $\Delta rw_1 = o_w - o_r = 6-0 = 6$ and $\Delta rw_2 = o_w - o_r = 6-2 = 4$. One inner-loop transfer (RAW) occurs at the minimal index distance, in this case after $\Delta rw_2 = 4$ iterations and the next transfer (RAR) $\Delta rw_1 = 6$ iterations later, respectively $\Delta rr = o_{r2} - o_{r1} = 2-0 = 2$ iteration after the RAW.

Based on the $\overrightarrow{\Delta rw}$ and the loop definition, where $0 \leq i < N$, $n_{\|} = 4$ and $n_{loop} = (N-1)/n_{\|}$. The LACOS-loop graph is shown in Figure 9b for $N = 12$. Each CB consists of the computation in the loop body, in this case $a[i] + a[i+2] + C$, each using a different value for i. n_{loop} is a function on the loop variables in the loop definitions, in this case N. The number of LACOS-nodes often depend on runtime variables, however, this dependency must be linear, as indexes to access array are always natural numbers. The LACOS-nodes from $\Gamma_1 - \Gamma_4$; $\Gamma_5 - \Gamma_8$ and $\Gamma_9 - \Gamma_{12}$ can be run in parallel. For a symmetrical platform (cores with same performance and inter-data-exchange latency) it is possible to distribute the parallel CBs to the available n_{units} units with an analytical step. This is possible as every LACOS-node in G_0 requires the same amount of computing power and each unit holds the same computing power as well. The key step is to minimize the inner-loop-transfers. A first characteristic of the LACOS-graph is that no additional loop-internal transfer occurs if the number of parallelizable nodes is distributed to an even number of $n_{units} = n_{\|}$, see Figure 9b. In all other cases, additional loop-internal transfers occur.

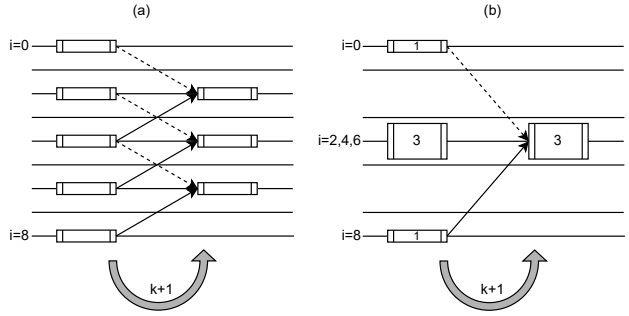

Figure 10. Combining parallel LACOS-graph-nodes by the read-write-pair leads to the vanishing of potential transfers. a) shows computation blocks with transfer for $\Delta rw_1 = 2$ and $\Delta rw_2 = -2$. b) shows the blocks combined by their RAW and RAR distances

The RAR dependency information bears another opportunity to minimize the loop-internal transfers by merging LACOS nodes that share as many read-read pairs as possible. Each merge of LACOS-nodes reduces one potential loop-internal transfer, see Figure 10.

Optimizing the loop-internal transfers along the distances between the read accesses on the same data, leads to the reduction of the amount of potential transfers, as can be seen in Figure 10b. In this simple and illustrative example, the distances are $\Delta rw_{1,2} = \pm 2$. The only type of node merge that minimizes the number of transfers is the one that combines the LACOS-nodes of every second iteration. Any other combination of parallel LACOS-nodes leads to more transfers. As read-read dependencies are also a function of the distance vector $\overrightarrow{\Delta rw}$, they can be used to reduce the amount of parallel LACOS-nodes to fit the number of units n_{units} with a further analytical step, see for example Fig. 11.

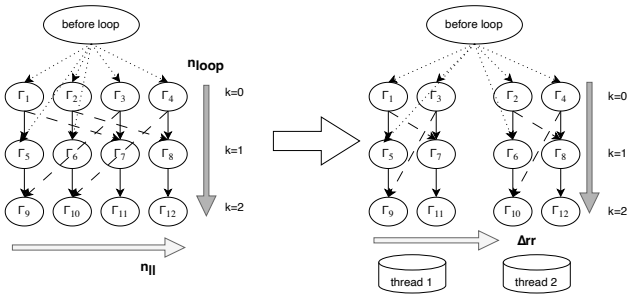

Figure 11. Minimized transfers in sample code for $n_{units} = 2$

Given the presented links between the loop definition, the statements in the loop-bodies and the LACOS-graph, it is

possible to build a LACOS-graph with this information from static-code analysis. Both properties $n_{\parallel}$ and n_{loop} can be derived as a function of the distance vector $\overrightarrow{\Delta rw}$ and runtime variables without an explicit data-analysis step, respectively unrolling the loop. This generates novel limits for the exploitation of LLP in loops and can be enhanced by state-of-the-art methods. Mynatix' method enables new code optimization opportunities and enables the exclusive dependency of parallelization on runtime information at compile-time. Additionally, because of the structured way a LACOS-graph is formed, loop sections can be reformed directly to different parallel frameworks, e.g. to the pthread-model exploiting TLP in the code, creating a CUDA kernel by writing the calculation of the CB as a kernel, or using the MPI-protocol to implement the minimal amount of data transfers into explicit communication patterns. This is shown with the following example.

4.2 Application of LACOS to the two-dimensional heat equation

The two-dimensional diffusion equation:

$$\frac{\delta u}{\delta t} - \alpha \left(\frac{\delta^2 u}{\delta^2 x} + \frac{\delta^2 u}{\delta^2 y} \right) = 0 \tag{7}$$

can be discretized on a mesh with dimensions n_x and n_y. The temporal evolution can be computed iteratively [30, 31]. The factor α summarizes physical properties and u defines the field to compute. The focus of this chapter lies on the corresponding nested loop-section in the code when calculating the two-dimensional heat equation, demonstrated in python code notation:

```
for k = range(0, tk, 1):
 for j = range(1, ny, dx):
  for i = range(1, nx, dy):
   u[k+1,i,j]=gamma*(u[k,i+1,j]
   +u[k,i-1,j]+u[k,i,j+1]
   +u[k,i,j-1]-4*u[k,i,j])+u[k,i,j]
```

There are five read-write pairs regarding the distance vectors on the array, which enable the extraction of CBs, as shown in Figure 12b. This Figure corresponds to the stencil illustrated in Figure 12a. Each array access can be expressed with indices: the loop variables i, j, k, as well as the mesh / array dimensions n_x and n_y:

$$t_{i,j,k} = u[i][j][k] = (i) + n_x \cdot (j + n_y \cdot k) \tag{8}$$

Each write-to-read pair Δrw in the loop creates a potential transfer, as was indicated in the simple loop example in Figure 9b. For each pair, an index-distance $\Delta rw = t_{write,i,j,k} - t_{read,i,j,k}$ can be computed using the array accesses to build up indexes:

$$\begin{bmatrix} t_a \\ t_b \\ t_c \\ t_d \\ t_e \\ t_f \end{bmatrix} \equiv \begin{bmatrix} u[k,i+1,j] \\ u[k,i-1,j] \\ u[k,i,j+1] \\ u[k,i,j-1] \\ u[k,i,j] \\ u[k+1,i,j] \end{bmatrix} \tag{9}$$

to compute the discretized equation (10) for the PDE shown in equation (7) for each mesh node:

$$t_f = \gamma (t_a + t_b + t_c + t_d - 4 \cdot t_e) + t_e \tag{10}$$

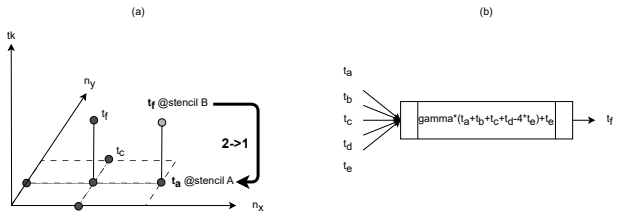

Figure 12. . (a) Stencil (geometrical arrangement of nodes in a mesh related to point of interest - in this case t_f) for computing the discretized diffusion equation and loop-internal transfer between $t_f \rightarrow t_a$. (b) five read and one write dependencies, resulting in four potential transfers between stencils

This results in a list of index-distance pairs, the distance vector $\overrightarrow{\Delta rw}$, as a function of mesh dimensions, respectively runtime variables:

$$\overrightarrow{\Delta rw} = \begin{bmatrix} t_f - t_a \\ t_f - t_b \\ t_f - t_c \\ t_f - t_d \\ t_f - t_e \end{bmatrix} = \begin{bmatrix} (n_x \cdot n_y) - 1 \\ (n_x \cdot n_y) + 1 \\ (n_x \cdot n_y) - n_x \\ (n_x \cdot n_y) + n_x \\ (n_x \cdot n_y + 0) \end{bmatrix} \tag{11}$$

The index-distances represented by $\overrightarrow{\Delta rw}$ can be utilized to represent the transfers with numerical values $(2 \rightarrow 1)$. This can be used to determine the loop-internal transfers numerically, see Figure 12a, where the loop-internal transfer for two stencils is shown ($t_f @ stencilB$ to $t_a @ stencilA$). Only for a small mesh where $n_x = 5$ and $n_y = 4$, the loop-internal transfers between six stencils can be illustrated defined by the index distance vector $\overrightarrow{\Delta rw}$ in equation (11), see Fig. 13. Following the LACOS rules presented in section 2, each computation block consists of one stencil.

Following [13], the information of the $\overrightarrow{\Delta rw}$ can also be used to test which loop-depth exceeds any of the maximal read-write distances that leads to $\max\left(\overrightarrow{\Delta rw}\right) = (n_x \cdot n_y) \pm n_x$ by the direction vector. The number of iterations N_i within a parallelized step can be deduced from the loop definitions, e.g. for a *for*-loop $N_i = \frac{(i_{end} - i_0)}{(i_{step})}$, where i_0 is the start index, i_{end} the end index, and i_{step} the step-width. The number of

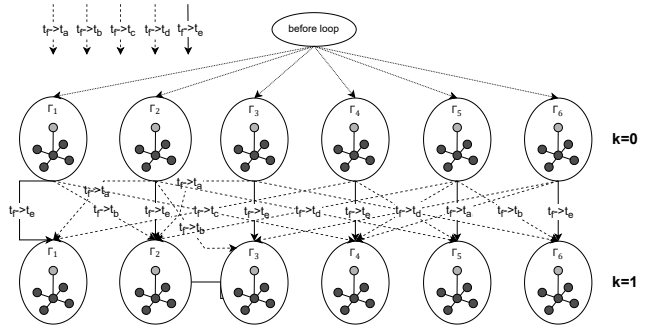

Figure 13. Potential loop-internal transfers for LACOS-graph-nodes (1 node = 1 stencil) for the first two temporal iterations of the (tk-loop)

iterations over the tk-loop is compared to the maximum of the distance vector:

$$N_k(t_k) \cdot N_j(n_y) \cdot N_i(n_x) \leq \max\left(\overrightarrow{\Delta rw}\right) \tag{12}$$

In contrast to the N_j and N_i-loop-levels:

$$N_j(n_y) \cdot N_i(n_x) < \max\left(\overrightarrow{\Delta rw}\right) \tag{13}$$

This enables determining if the tk-loop cannot be further decomposed in the LACOS-graph, as the iterations exceed the maximal read-write distance $\max\left(\overrightarrow{\Delta rw}\right)$ during compile-time. This corresponds to the physical logic, that time cannot be split up. However, the two inner-loops, over nx and ny, can be distributed. The number of parallel LACOS-nodes is defined by $n_{\parallel} = (n_x - 2) \cdot (n_y - 2)$. The mesh with dimensions $n_x = 5$ and $n_y = 4$ is used to illustrate the effect of combining parallel LACOS-nodes for the time model in equation (15). The needed transfers $\vec{T}$ with granularity G_0 visualized in a matrix below, where $\ominus$ indicates no transfer:

$$\vec{T} = \begin{pmatrix} 2 \to 1 & \ominus & 4 \to 1 & \ominus & 1 \to 1 \\ 3 \to 2 & 1 \to 2 & 5 \to 2 & \ominus & 2 \to 2 \\ \ominus & 2 \to 3 & 6 \to 3 & \ominus & 3 \to 3 \\ 5 \to 4 & \ominus & \ominus & 1 \to 4 & 4 \to 4 \\ 6 \to 4 & 4 \to 5 & \ominus & 2 \to 5 & 5 \to 5 \\ \ominus & 5 \to 6 & \ominus & 3 \to 6 & 6 \to 6 \end{pmatrix} \tag{14}$$

Each row of the matrix $\vec{T}$ contains the transfers for one LACOS-node Γ_i and each column represents the index difference $t_f - t_a$, $t_f - t_b$, $t_f - t_c$, $t_f - t_d$, $t_f - t_e$. To calculate one tk-iteration, one has to compute equation (10) for each element and up to four transfers occur for each LACOS-node, respectively stencil (geometrical arrangement of nodes in a mesh related to the point of interest - in this case t_f), at least for nodes not next to boundaries (nodes defining the outer shape of the mesh). This enables to construct the time model for the LACOS-graph with base granularity G_0 assuming each transfer has a length of G_0:

$$\vec{t_{run}} = \vec{T} \cdot \Delta t_t = \begin{pmatrix} 2 \\ 3 \\ 2 \\ 2 \\ 3 \\ 2 \end{pmatrix} \cdot \Delta t_t + \begin{pmatrix} n_{op} \\ n_{op} \\ n_{op} \\ n_{op} \\ n_{op} \\ n_{op} \end{pmatrix} \cdot \Delta t_c \tag{15}$$

With the granularity G_0, the LACOS-nodes could be run on six units in parallel for this example of a very small grid. Larger grids would have proportionally more LACOS-nodes. The LACOS-nodes must be mapped if the computation has to be split between fewer units, e.g. two units. Following the approach described in section 4.1 to use the RAR dependencies to select corresponding Γ-nodes, Δrw_3 and Δrw_4 from equation (11) can be used to build groups of Γ-nodes, reducing the transfers needed to a minimum. First, the transfers can be evaluated numerically for each location. For location 1, merging $(\Gamma 1, \Gamma 2, \Gamma 3)$:

$$\vec{T_1} = \begin{pmatrix} 2 \nrightarrow 1 & \ominus & 4 \to 1 & \ominus & 1 \nrightarrow 1 \\ 3 \nrightarrow 2 & 1 \nrightarrow 2 & 5 \to 2 & \ominus & 2 \nrightarrow 2 \\ \ominus & 1 \nrightarrow 2 & 2 \nrightarrow 3 & 6 \to 3 & 3 \nrightarrow 3 \end{pmatrix} = (3) \tag{16}$$

For location 2, merging $(\Gamma 4, \Gamma 5, \Gamma 6)$:

$$\vec{T_2} = \begin{pmatrix} 5 \nrightarrow 4 & \ominus & \ominus & 1 \to 4 & 4 \nrightarrow 4 \\ 6 \nrightarrow 4 & 4 \nrightarrow 5 & \ominus & 2 \to 5 & 5 \nrightarrow 5 \\ \ominus & 5 \nrightarrow 6 & \ominus & 3 \to 6 & 6 \nrightarrow 6 \end{pmatrix} = (3) \tag{17}$$

Combining the nodes results in the effect that computations are aggregated to run on the same locality and transfers are eliminated, reducing equation (15) to:

$$\vec{t_{run}} = \begin{pmatrix} 3 \\ 3 \end{pmatrix} \cdot \Delta t_t + \begin{pmatrix} 3 \cdot n_{op} \\ 3 \cdot n_{op} \end{pmatrix} \cdot \Delta t_c \tag{18}$$

This results in a higher granularity (more computational effort in relation to communicational effort) with reduced transfers and aggregated computations when running on two computing units. It is possible to retrieve a parallel code from the LACOS graph by using the instructions in the Γ-nodes to write code sections and translating the transfer lines in the graph into suitable transfer and communication models. A Proof-of-Concept implementation successfully demonstrated the fully automatic creation of an optimized code from the LACOS-graph using a) OpenMP, b) the pthread-model, c) a CUDA-kernel with necessary memory copies and d) the MPI-protocol to represent the transfers as explicit communication between computing-nodes. In the following, the focus is set on using multithreading with pthreads. There are $n_{\parallel} = (n_x - 2) \cdot (n_y - 2)$ parallel Γ-nodes to iterate over the tk-loop-depth, leading to $n_{loop} = t_k - 1$ loop iterations. The $n_{\parallel}$ parallel Γ-nodes can be distributed evenly to a given number of threads $n_{threads}$. Since every thread has the same

performance characteristics, it is a simple task to distribute the Γ-nodes evenly. For a multithreaded application, the transfers can be interpreted as necessary barriers to guard the data sharing, respectively prevent race conditions (wrong timing of threads writing data, which can lead to false results) between the threads. Each thread will take care of a subset of the parallel Γ-nodes: *nx_sub*, *ny_sub*. Determining these sub-grids is only an analytical step by dividing the number of $n_{\parallel}$ parallel Γ-nodes by $n_{threads}$. This results in the following pseudo-code-notation:

```
thread() {
 for tk {
  for nx_sub {
   for ny_sub {
    u[k+1,i,j]=gamma*(uold[k][i+1][j]+
    uold[k][i-1][j]+uold[k][i][j+1]+
    uold[k][i][j-1]-4*uold[k][i][j])+
    uold[k][i][j]
  pthread_barrier()
  uold = u
 }}}}
main() {
 for (nthreads) {
  pcreate(thread())}
 for (nunits) {
  pjoin() }
}
```

To test the implementation, the parallel code was run on an Intel®Xeon®CPU E5-2680 V3 with 2.50GHz with 32GB RAM running a ubuntu 22.04.2. The increase in speed for two exemplary meshes are visible in Figure 14.

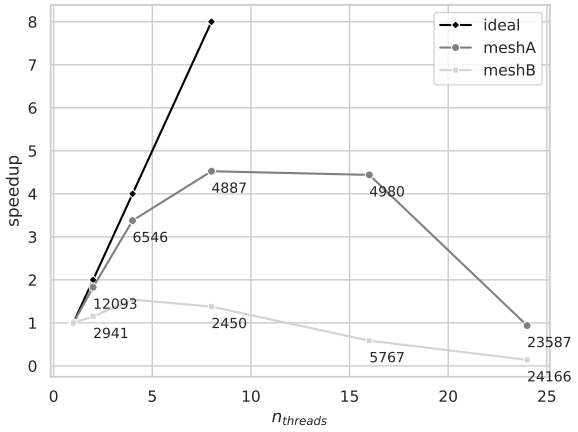

Figure 14. Speed-up for two different meshes: $n_{\parallel,meshA} = 17'464$ with $n_x = 150$ and $n_y = 120$, $n_{\parallel,meshB} = 3'264$ with $n_x = 70$ and $n_y = 50$. The numbers indicate the measured runtime of the parallel computations in *ms* using [19]

Additionally, the LACOS-graph can be used to estimate an optimal number of threads to reach an optimal acceleration. This can be achieved by estimating the ratio between the computational effort and the latency to retrieve data on a specific hardware as a function of $n_{threads}$. It should be emphasized that this is only an approximation, as many factors influence the data access pattern on a modern multicore CPU. Nevertheless, the LACOS-graph contains information about data locality by using the size information used by each Γ-node. The computing performance is therefore approximated by an average IPC value. The latency characteristics for the different data access latency to the corresponding L-caches can be found in Table 1.

Table 1. Intel®Xeon®CPU E5-2680 v3 data locality and latency characteristics [29], IPC by '*perf*'

L3-cache/chip	L2-cache/chip	L1-cache/chip
$30MiB$	$256KiB$	$32KiB$
$2.1 \cdot 10^{-8} ns$	$4.8 \cdot 10^{-9} ns$	$1.6 \cdot 10^{-9} ns$

cycle-time	IPC	$S_{cachelinelength}$
$3.10 \cdot 10^{9} ns$	1.2	$64B$

The data size per node can be approximated with:

$$S_{compute,unit} = \frac{n_{\parallel}}{n_{threads}} \cdot S_{float} \cdot n_{data,\Gamma} \tag{19}$$

$S_{float} = 8$ byte are used for one data-point and each Γ-node had $n_{data,\Gamma} = 5$ data points. The computational effort is estimated with:

$$t_{compute,unit} \approx \frac{n_{\parallel}}{n_{threads}} \cdot n_{op} \cdot IPC \tag{20}$$

Where $n_{op} = 10$ is used to approximate the number of operations needed to compute equation 10. The IPC factor can be found in Table 1. To estimate the latency depending on the amount of data accesses and data sharing, the following relation is used:

$$t_{access,unit} \approx \frac{S_{compute,unit}}{S_{cachelinelength}} \cdot \Delta t_{access} + n_{transfer} \cdot \Delta t_{share} \tag{21}$$

The data access times Δt_{access} can only be approximated by computing the data size per unit with equation (20) and the corresponding values in Table 1. On the platform used, data sharing was possible by L3-caches, therefore the latency to access the L3-cache was used to calculate Δt_{share}. The number of mesh nodes that have to be exchanged between the different threads is $n_{transfer}$. This enables a rough estimation of the difference between computing time and data access latencies for different meshes:

$$D_{latency}(n_{threads}) \approx t_{compute,unit} - t_{access,unit} \tag{22}$$

Figure 15 shows the difference between the approximated access and the computation time given by equation (22). For

mesh A, the data per thread fits in L2-caches and with only $n_{threads} = 24$, the data fits in the L1-caches with a capacity of $32KiB$, see Table 1. For the smaller mesh B, the data sizes from equation (19) indicate for $n_{threads} \leq 4$ L2-cache accesses.

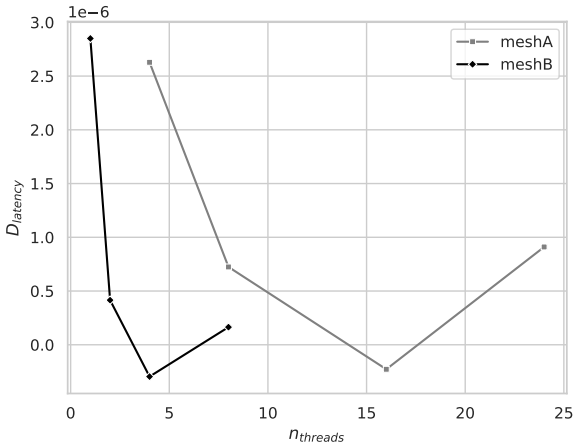

Figure 15. Difference in latency according equation (22) for the different meshes in Fig. 14

Using equation (21), we conclude that a maximal speed-up can be reached with $n_{threads} \cong 16$ threads for mesh A. At this number of threads, the computational acceleration starts to be limited by the latency to access and share data. The calculations for mesh B show an optimal speed-up at around $n_{threads} \cong 4$ threads. This corresponds well with the measured values in Fig. 14.

LACOS can be applied without unrolling the three-time nested loop to compute the finite differences for array u. This enables the retrieval of the LACOS-graph as a function of the runtime variables nx, ny and tk during static code analysis. All information needed can be optained from the code, either from an intermediate representation of a compiler or from a specific programming language. A parallel code can be derived from the LACOS-graph adapted to a target platform fully automated. Different PoCs demonstrated the successful application to different frameworks like OpenMP, pthreads, CUDA and the MPI-protocol. By approximating $D_{latency}$ with equation (22), the optimal number of threads could be approximated during compile time based on the measured platform characteristics in Table 1.

Conclusion

LACOS is a novel technique for segmenting code in a more fine-grained and more suitable way than Basic Blocks to exploit parallelism. The technique uses the RAR dependencies that have not been exploited until now to segment code into computation blocks. LACOS introduces potential transfers between each of the blocks., which are then ordered. Parallelization opportunities can then be realized by using numerical methods to map the segments to a given number of parallel units. In a graph, the blocks can be visualized as nodes and the transfers can be seen as connecting lines between the nodes. Thus, each node represents a set of arbitrary sequential instructions and the connecting lines potential transfers between the nodes. In this form, the code becomes a continuous series of compute and transfer segments for differentiable paths. Each path represents the computation and communication pattern for a potential separate unit, where units can be ALUs, threads, cores or nodes in a cluster. Platform specific parallel code can then be extracted using appropriate frameworks to represent the transfers in the graph, e.g. with pthread or the MPI-protocol. Furthermore, the parallel nodes can be combined to tasks with a fixed target granularity. This enables the mapping of the graph to the characteristics of a platform to compute and transfer or communicate data. It also simplifies the scheduling of tasks, as each task has the same granularity. For symmetrical platforms, in this context meaning platforms consisting of units with the same computational performance and same data transfer latencies between them, LACOS simplifies - at least for our test cases - the mapping of the segments with an analytical step. For three applications to known (complex) problems, the segmentation can be used to generate parallel code and estimate an optimal number of parallel units with only two platform specific properties: IPC and context-switchtime, respectively data-access latencies.

The article shows preliminary results for a novel method to segment code including RAR dependencies. For the well-known recursive computation of the Fibonacci sequence, a speed-up of 3.5 could be reached. For the three-times nested loop-section to calculate the discretized 2d diffusion equation, which is not automatically parallelizable by any known compiler, a speed-up of up to 4.5 could be reached. All speed-ups are directly and automatically caused by the fully automatically LACOS-method and do not require any annotation to the code nor any interaction with a developer. The approach could enable reaching new limits in automatic parallelization of code during compile-time. The work-principle of LACOS can be implemented as a generic method, e.g. as an extension for existing compilers or as a core for an expert system supporting developers to parallelize their codes. LACOS can improve the functionality for compilers and be utilized in cases where compilers cannot be used - e.g. for interpreted languages.

LACOS can save time during development and opens new doors to improve the capabilities of state-of-the-art compilers to auto parallelize code. The available PoCs demonstrated show promising results, but further studies are necessary to test and benchmark LACOS with more applications and platforms to investigate the new limits reachable in automatic parallelization using RAR dependencies.

References

[1] F. E. Allen and J. Cocke. “A Program Data Flow Analysis Procedure”. In: *Commun. ACM* 19.3 (Mar. 1976), p. 137. ISSN: 0001-0782. DOI: 10.1145/360018.

360025. URL: https://doi.org/10.1145/360018.360025.

[2] Frances E. Allen. "Control Flow Analysis". In: *SIGPLAN Not.* 5.7 (July 1970), pp. 1–19. ISSN: 0362-1340. DOI: 10.1145/390013.808479. URL: https://doi.org/10.1145/390013.808479.

[3] J. R. Allen et al. "Conversion of Control Dependence to Data Dependence". In: *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '83. Austin, Texas: Association for Computing Machinery, 1983, pp. 177–189. ISBN: 0897910907. DOI: 10.1145/567067.567085. URL: https://doi.org/10.1145/567067.567085.

[4] Randy Allen and Ken Kennedy. "Automatic Translation of FORTRAN Programs to Vector Form". In: *ACM Trans. Program. Lang. Syst.* 9.4 (Oct. 1987), pp. 491–542. ISSN: 0164-0925. DOI: 10.1145/29873.29875. URL: https://doi.org/10.1145/29873.29875.

[5] James E Bennett and Michael J Flynn. *Performance factors for superscalar processors*. Computer Systems Laboratory, Stanford University, 1995.

[6] Simone Campanoni et al. "HELIX: Automatic Parallelization of Irregular Programs for Chip Multiprocessing". In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. CGO '12. San Jose, California: Association for Computing Machinery, 2012, pp. 84–93. ISBN: 9781450312066. DOI: 10.1145/2259016.2259028. URL: https://doi.org/10.1145/2259016.2259028.

[7] Gcc Developer Community. *GCC 8.0 GNU Compiler Collection Internals*. Ed. by Richard M. Stallman. 12th Media Services, 2018.

[8] Intel Corporation. *oneAPI Threading Building Blocks (oneTBB)*. 2023. URL: https://oneapi-src.github.io/oneTBB/ (visited on 11/25/2023).

[9] cppreference.com. *Date and time utilities*. 2023. URL: https://en.cppreference.com/w/cpp/chrono (visited on 11/25/2023).

[10] George B. Dantzig. *Linear Programming and Extensions*. United States Air Force Project RAND. The RAND Corporation, Aug. 1963. URL: http://www.rand.org/pubs/reports/R366.html.

[11] Taposh Dutta Roy. *Instructional Level Parallelism*. July 2015. DOI: 10.13140/RG.2.1.3338.9920.

[12] Michael Flynn. "Some computer organizations and their effectiveness. IEEE Trans Comput C-21:948". In: *Computers, IEEE Transactions on* C-21 (Oct. 1972), pp. 948–960. DOI: 10.1109/TC.1972.5009071.

[13] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. "Practical Dependence Testing". In: *SIGPLAN Not.* 26.6 (May 1991), pp. 15–29. ISSN: 0362-1340. DOI: 10.1145/113446.113448. URL: https://doi.org/10.1145/113446.113448.

[14] Ronald Lewis Graham et al. "Optimization and approximation in deterministic sequencing and scheduling: a survey". In: *Annals of discrete mathematics*. Vol. 5. Elsevier, 1979, pp. 287–326.

[15] Bolei Guo et al. "Practical and Accurate Low-Level Pointer Analysis". In: *Proceedings of the International Symposium on Code Generation and Optimization*. CGO '05. USA: IEEE Computer Society, 2005, pp. 291–302. ISBN: 076952298X. DOI: 10.1109/CGO.2005.27. URL: https://doi.org/10.1109/CGO.2005.27.

[16] David A. Gustafson. "Control Flow, Data Flow & Data Independence". In: *SIGPLAN Not.* 16.10 (Oct. 1981), pp. 13–19. ISSN: 0362-1340. DOI: 10.1145/954255.954256. URL: https://doi.org/10.1145/954255.954256.

[17] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN: 0128119055.

[18] Torsten Hoefler and Roberto Belli. "Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses When Reporting Performance Results". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '15. Austin, Texas: Association for Computing Machinery, 2015. ISBN: 9781450337236. DOI: 10.1145/2807591.2807644. URL: https://doi.org/10.1145/2807591.2807644.

[19] IBM. *time.h*. 2023. URL: https://www.ibm.com/docs/en/i/7.5?topic=extensions-standard-c-library-functions-table-by-name (visited on 11/25/2023).

[20] Toni Juan, Juan J. Navarro, and Olivier Temam. "Data Caches for Superscalar Processors". In: *Proceedings of the 11th International Conference on Supercomputing*. ICS '97. Vienna, Austria: Association for Computing Machinery, 1997, pp. 60–67. ISBN: 0897919025. DOI: 10.1145/263580.263595. URL: https://doi.org/10.1145/263580.263595.

[21] Krishna Kavi et al. "Shared memory and distributed shared memory systems: A survey". In: *Emphasizing Distributed Systems*. Ed. by Marvin V. Zelkowits. Vol. 53. Advances in Computers. Elsevier, 2000, pp. 55–108. DOI: https://doi.org/10.1016/S0065-2458(00)80004-2. URL: https://www.sciencedirect.com/science/article/pii/S0065245800800042.

[22] Dounia Khaldi et al. “Task parallelism and data distribution: An overview of explicit parallel programming languages”. In: *Languages and Compilers for Parallel Computing: 25th International Workshop, LCPC 2012, Tokyo, Japan, September 11-13, 2012, Revised Selected Papers 25*. Springer. 2013, pp. 174–189.

[23] Jan Kwiatkowski. “Evaluation of Parallel Programs by Measurement of Its Granularity”. In: vol. 2328. Sept. 2001, pp. 145–153. ISBN: 978-3-540-43792-5. DOI: 10.1007/3-540-48086-2_16.

[24] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. “Numba: A llvm-based python jit compiler”. In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. 2015, pp. 1–6.

[25] Chris Lattner and Vikram Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE. 2004, pp. 75–86.

[26] M.H. Lipasti and J.P. Shen. “Exceeding the dataflow limit via value prediction”. In: *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*. 1996, pp. 226–237. DOI: 10.1109/MICRO.1996.566464.

[27] Iván Martel et al. “Increasing Effective IPC by Exploiting Distant Parallelism”. In: *Proceedings of the 13th International Conference on Supercomputing*. ICS ’99. Rhodes, Greece: Association for Computing Machinery, 1999, pp. 348–355. ISBN: 158113164X. DOI: 10.1145/305138.305212. URL: https://doi.org/10.1145/305138.305212.

[28] Angelo Matni et al. “NOELLE Offers Empowering LLVM Extensions”. In: *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2022, pp. 179–192. DOI: 10.1109/CGO53902.2022.9741276.

[29] Daniel Molka. “Performance Analysis of Complex Shared Memory Systems”. In: 2016. URL: https://api.semanticscholar.org/CorpusID:5080429.

[30] K. W. Morton and D. F. Mayers. *Numerical Solution of Partial Differential Equations: An Introduction*. 2nd ed. Cambridge University Press, 2005. DOI: 10.1017/CBO9780511812248.

[31] G. Nervadof. *Solving 2D Heat Equation Numerically using Python*. 2020. URL: https://levelup.gitconnected.com/solving-2d-heat-equation-numerically-using-python-3334004aa01a (visited on 11/25/2023).

[32] Vojin G. Oklobdzija. “Reduced Instruction Set Computing”. In: *Wiley Encyclopedia of Computer Science and Engineering*. John Wiley & Sons, Ltd, 2009. ISBN: 9780470050118. DOI: https://doi.org/10.1002/9780470050118.ecse357. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9780470050118.ecse357. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470050118.ecse357.

[33] Kunle Olukotun et al. “The Case for a Single-Chip Multiprocessor”. In: *SIGPLAN Not.* 31.9 (Sept. 1996), pp. 2–11. ISSN: 0362-1340. DOI: 10.1145/248209.237140. URL: https://doi.org/10.1145/248209.237140.

[34] Laura Pozzi. “Compilation Techniques for Exploiting Instruction Level Parallelism , a Survey”. In: 1999. URL: https://api.semanticscholar.org/CorpusID:11461879.

[35] B. Ramakrishna Rau and Joseph A. Fisher. “Instruction-Level Parallel Processing: History, Overview, and Perspective”. In: *J. Supercomput.* 7.1–2 (May 1993), pp. 9–50. ISSN: 0920-8542. DOI: 10.1007/BF01205181. URL: https://doi.org/10.1007/BF01205181.

[36] Vivek Sarkar and John Hennessy. “Compile-Time Partitioning and Scheduling of Parallel Programs”. In: *SIGPLAN Not.* 21.7 (July 1986), pp. 17–26. ISSN: 0362-1340. DOI: 10.1145/13310.13313. URL: https://doi.org/10.1145/13310.13313.

[37] Zhiao Shi. “Scheduling Tasks with Precedence Constraints on Heterogeneous Distributed Computing Systems”. AAI3251161. PhD thesis. USA, 2006.

[38] Benoit Sigoure. *Little micro-benchmarks to assess the performance overhead of context switching*. 2010. URL: https://github.com/tsuna/contextswitch/blob/master/timetctxsw.c (visited on 11/25/2023).

[39] James E. Smith and Gurindar S. Sohi. “The microarchitecture of superscalar processors”. In: *Proc. IEEE* 83 (1995), pp. 1609–1624. URL: https://api.semanticscholar.org/CorpusID:221940988.

[40] G.S. Sohi, S.E. Breach, and T.N. Vijaykumar. “Multiscalar processors”. In: *Proceedings 22nd Annual International Symposium on Computer Architecture*. 1995, pp. 414–425.

[41] Ross Albert Towle. “Control and Data Dependence for Program Transformations.” AAI7624191. PhD thesis. USA, 1976.

[42] Eric Tune et al. “Dynamic prediction of critical path instructions”. In: *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture* (2001), pp. 185–195. URL: https://api.semanticscholar.org/CorpusID:2646253.

[43] J.J. van Hoorn. “Dynamic Programming for Routing and Scheduling: Optimizing Sequences of Decisions”. English. Naam instelling promotie: VU Vrije Universiteit Naam instelling onderzoek: VU Vrije Universiteit. PhD-Thesis - Research and graduation internal. Vrije Universiteit Amsterdam, 2016.

[44] David W. Wall. “Limits of instruction-level parallelism”. In: *ASPLOS IV*. 1991. URL: https://api.semanticscholar.org/CorpusID:7490723.

[45] Neil Weste and David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. 4th. USA: Addison-Wesley Publishing Company, 2010. ISBN: 0321547748.

[46] Wayne Wolf. *Computers as Components, Second Edition: Principles of Embedded Computing System Design*. 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN: 0123743974.

[47] Paraskevas Yiapanis, Gavin Brown, and Mikel Luján. “Compiler-Driven Software Speculation for Thread-Level Parallelism”. In: *ACM Transactions on Programming Languages and Systems* 38 (Jan. 2016), pp. 1–45. DOI: 10.1145/2821505.

[48] Ali Mustafa Zaidi et al. “Loopapalooza: Investigating Limits of Loop-Level Parallelism with a Compiler-Driven Approach”. In: *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2021, pp. 128–138. DOI: 10.1109/ISPASS51385.2021.00030.